Acknowledgements

I would like to thank my advisors Abhiram Ranade and Jerry Feldman for providing the advice, support and encouragement throughout the years, and for directing me to the Sather project in the first place.

I would like to acknowledgement the financial support from the International Computer Science Institute (ICSI) during the last three years. The pSather research reported here is done on a CM-5 supported under National Science Foundation Infrastructure Grant number CDA-8722788.

It has been a privilege and an experience for me to participate in the Sather/pSather project at ICSI. This is especially so because it is not everyday that a graduate student gets to learn from and interact with researchers in a dynamic and stimulating research institution. The congenial environment is more than what a graduate student could hope for and has helped me tremendously.

It would be outright blasphemy if this thesis gives an impression that it is solely my work. Therefore, I would like to acknowledge the contributions of the direct participants of the Sather/pSather project — Jeff Bilmes, Ben Gomes, Franco Mazzanti, Stephan Murer, Steve Omohundro, Thomas Rauber, Hans Rohnert, Heinz Schmidt, David Stoutamire and Jerry Feldman. I would also like to thank Stephan Murer, Ben Gomes, Jeff Bilmes and David Stoutamire for suffering through early drafts of this thesis. Steve and Heinz has continually provided feedback on various pSather designs. Steve's design of Sather has helped to shape pSather. The revised Sather 1.0 language has also stretched the functionalities of the parallel constructs not previously possible. Furthermore, the various parallel programs described here have made use of his sequential class libraries. And this is not to forget the help of my "big boss" and advisor Jerry Feldman who heads the pSather group. His faith and trust has allowed me to keep my own pace and to enjoy my work.

And of course, I have to thank my friends from Stanford and Berkeley — Lubna Alsagoff, Ang Boon Seong, Kinson Ho, Lim Chong Hai, Loh Wei Liem, Poh Hean Lee and Yue Ming Bao, for their support and lunch/dinner reprieves. For those friends who do not find their name here, I haven't forgotten about you but am only trying to protect the privacy of your name.

Finally, I would like to thank my parents for their patience and support, and for putting their children's interests before anything else at all times. It has indeed been a long time to be away from home, considering that I was ineligible to vote when I came to the States for my undergrad studies and now I've missed my chance to vote twice. I'm grateful to my brothers for keeping my parents company during this period. The emotional support has remained strong in spite of the physical distance of home (unlike Newton's laws of gravitation).

This thesis is dedicated to everyone who has made it possible.

Contents

List of Figures

List of Tables

Chapter 1

Motivation and Background

It is not a new idea to design and build multiprocessors[1] to achieve higher levels of performance than that of their contemporary sequential counterparts. Over the decades, the multiprocessors which have been built include NBS PILOT [160] in the late 1950's, Burroughs D825 [11], ILLIAC IV [22] in the 1960's, S-1 [228], Cm* [211] in the 1970's, BBN Butterfly [80], Sequent Symmetry [166] in the 1980's, and KSR-1 [199], CM-5 [76] in the 1990's. Despite the variety of multiprocessors, such machines have never gained a foothold in the general computing community which has historically remained dominated by sequential machines. This is true regardless of whether the latter are in the form of maxicomputers (e.g. System/360 family [39, 208]), minicomputers (e.g. DEC PDP-11 [29]), workstations (e.g. Sun workstations [100]) or personal computers (e.g. PCs based on Intel's 486 processors [125]).

One way of using the multiprocessors is to make use of the huge investment in sequential software and build automatic parallelizing compilers for various languages, e.g. Fortran [52, 162] and Lisp [158]. But this approach relies mostly on detecting loop parallelism and is not able to make use of parallel algorithms. A parallelizing/vectorizing compiler's task is further complicated for languages with side-effects and aliasing mechanisms [102]. This has resulted in a situation in which programmers make a conscious effort to change their style of sequential programming in order for the compiler to detect parallelism (and there are guides for writing "good vectorizable Fortran" [92]).

To overcome the limitations of parallelizing compilers, some researchers (e.g. [53, 17]) advocate the use of functional languages (e.g. SISAL [42, 186]). One view [53] of functional languages is that they have implicit parallelism in the language semantics. Our view is that they are also sequential languages in which the obstacles to parallelizing compilers (i.e. side-effects, aliasing) have been removed. This makes functional languages more amenable to parallelizing compilers. This also explains a main advantage of implicitly parallel functional languages: that the programmer does not have to be concerned about managing parallelism and synchronization. However removing side-

[1] We define a multiprocessor as one with multiple functionally-equivalent general-purpose processors that are independent of how they work together.

effects makes certain notions (e.g. imperative data structures with complex state changes) difficult to capture in functional languages. That is why some functional languages, e.g. ML and Scheme are extended with state. Barth et al. [24, 25] adds a data structure (M-structures) to Id to make the language more expressive. But the indeterminacy of Id with M-structures means that the language is no longer side-effect free. Programmers now have to manage synchronizations explicitly (e.g. to avoid deadlocks) and are doing explicit parallel programming.

On the other hand, *parallel programming* allows the programmer to write exactly the intended parallel algorithms and make full use of a multiprocessor's performance. But the *difficulty* of parallel programming is one of the main reasons why parallel computing has remained an "exotic art". In fact, one of the challenges in the use of multiprocessors is to find an easy-to-understand and easy-to-use model that will allow programmers to *efficiently* and *easily* program multiprocessors.

Our goal is to examine how an explicitly parallel object-oriented language can play a role in parallel programming. We choose an object-oriented paradigm because its mechanisms for encapsulation and software reuse support our vision of how it may be possible to make parallel programming easier for the general users. We envision two categories of people programming on multiprocessors. *Users* are the people writing general application codes while *class library writers/designers* provide the class libraries tuned to a particular architecture. For example, a class library designer writes a parallel parser module to be used by different users writing their own parallel compilers. We will refer to both users and class library writers as *programmers*. Object-oriented language features will provide a well-disciplined interface between the two classes of programmers.

The kinds of programming tasks we are trying to accomplish can be best understood when we make the distinction between *parallel* and *distributed programming*. The main characteristic of parallel programming is to make many resources (processors, I/O-devices, etc.) *cooperate* on one large task. Distributed programming, on the other hand, focuses on supporting multiple parallel tasks which *compete* for shared resources. Both parallel and distributed programming share certain concurrency issues (e.g. synchronization). But some important issues in distributed programming (e.g. authentication/protection and fault-tolerance) are relegated to the operating system level for parallel programmers. Distributed programming is usually done on networks of computers (e.g. workstations on a wide-area or local-area network) and has to deal with latencies which are 4 or more orders of magnitude slower than the fastest local memory access. The executing components of a parallel program are usually more tightly coupled and therefore require more communication among processors, and so the remote memory access latencies have to be tolerable ($\leq$ 2 orders of magnitude slower than the fastest local memory access). An example of parallel programming involves writing a computational fluid dynamics simulation to run on multiprocessors, while examples of distributed programming include distributed file systems (e.g. Andrew file system [203]), and mail delivery/user authentication systems (e.g. Grapevine [36]).

Distributed programming involves competing components; message-passing is thus a natural programming model because it avoids *sharing* and protects the components against one another.

Parallel programs have been written in both message-passing and shared-memory models. Our design philosophy (further elaborated in Section 2.1) is that the cooperative nature of parallel programs is best captured by a shared-memory-like model because programmers need to easily express the information-sharing among the executing components. In fact, one of our findings is that although multiprocessors have high remote access latencies, if the latencies are ≤ 2 orders of magnitude slower than the fastest local access, it is practical to implement a parallel object-oriented language with a shared-memory-like model and still achieve relatively good performance.

In the rest of this introduction, Section 1.1 gives a summary of pSather's development. Section 1.2 explains some object-oriented terminology, especially terms related to pSather. Section 1.3 describes our contributions and the structure of this thesis. (Appendix A describes the presentation conventions used in the rest of this thesis.)

1.1 Relation of Project History to Thesis

In order to get a proper perspective of the work that is described here, we need a brief description of the Sather/pSather project history.

Sather [188] was designed in early 1990 by Steve Omohundro and researchers at the International Computer Science Institute (ICSI). It was borne out of a need for an object-oriented language that would combine the cleanliness of Eiffel [175] and the efficiency of C++ [210]. An initial version of the compiler was completed in late 1990. After gaining more experience with the implementation, a preliminary version of Sather (called Sather 0.1) was released to the public domain in summer 1991. At the same time (late 1990), work began on a parallel extension of Sather. This culminated in a first design and implementation of pSather on Sequent Symmetry and SPARCstation in mid 1991. This parallel implementation has allowed us to gain experience on parallel programming using an object-oriented language on a shared-memory machine.

We understood that for pSather to become more generally useful, the language would have to deal with distributed-memory machines because scalable architectures require distributed memory. A preliminary version of pSather was ported to a CM-5 [76] in early 1992. Because of this initial experience of parallel programming on a distributed-memory machine, pSather evolved to support a more general machine model (Section 2.4), and the definitions of both old and new parallel constructs were refined to be suitable in a distributed-memory, shared-address programming model. These refinements of pSather design and implementation continued from 1992 to 1993.

During pSather's development, Sather was not left alone either. Heinz Schmidt and a number of researchers from over the world worked on porting it to various platforms, and newer implementations (versions 0.2g and 0.2i) were released. The 0.2 versions also introduced a few additional language constructs.

In mid 1992, work also began on a major revision of Sather. This version is called Sather 1.0 and its design is finalized and released in 1993. Implementation work on Sather 1.0 is going on. The designs of Sather 1.0 and pSather on distributed-memory machines met in 1993 and the combined design (pSather 1.0) is described in [181].

We have adopted the following approach because of the different versions of Sather/pSather, in order to maintain clarity and consistency in this thesis. The language, implementation, and application chapters will mainly describe pSather as it is now in the CM-5 prototype implementation. This is based on Sather 0.1 and includes a few additional constructs from Sather 0.2. ("pSather" therefore will refer to pSather version 0.1.) This ensures that the implementation and library/application chapters refer to the version of pSather described in the language chapter.

In addition to a description of the status quo, we also want to give a broader understanding of where the project is going and how pSather 1.0's design is an improvement over pSather 0.1. Therefore at the end of each chapter, we will devote some time to the implications of pSather 1.0, e.g. slight changes to the definitions of the parallel extensions (Section 2.7) and how class library/application codes can be improved (Section 4.9).

1.2 Object-Oriented Programming

The concept of object-oriented programming first appeared in Simula [82]. In this section, we describe several object-oriented programming concepts that will come up in the following chapters.

An object-oriented program is typically organized in units called *classes*. During its execution, its state is modeled by *objects* which are instances of classes. A class is analogous to a record definition in languages such as Pascal and also encapsulates operations performed on objects of this class. The operations are called *methods* (or in Sather parlance, *routines* or *functions*). For example, if variable `x` holds an object from class `A` and `f` is a routine in `A`, `x.f` invokes the routine `f` on the object held by `x`. Within the routine definition, the "invoked object" is referred to as a *self* object (and in Sather's case, held by a predefined `self` variable).

A class also specifies the data components of an object (analogous to the fields of a record). In Sather, such components are called *attributes*.[2]

In some languages (e.g. Smalltalk [109]), classes are also considered as objects. Each class definition therefore corresponds to a *class object* which can be modified at execution time. In other languages (e.g. SELF [222]), there is no distinction between classes and objects. Every object is a prototype from which new objects can be cloned (created), and each object's behavior is accessible by objects which inherit from it.

The main difference between class and ordinary record definitions is that a class `A`[3] may also *inherit* from other classes. In most object-oriented languages, when `A` inherits from `B`, it means

[2] They are called instance variables in Smalltalk terminology.

[3] Sather syntax requires class names to be uppercase (e.g. `ARRAY{INT}`, `QUEUE{T}`), and we will follow this syntax in later discussions.

that `A` is a subtype of `B` and also gets the implementation code of `B`. Names of classes are also used in the type declarations of variables. For example, the declaration "`x:A`" (in Eiffel) means that `x` can hold objects from class `A` or any of `A`'s subtypes. (The exact relation of types and classes depend on the specific language.)

A Sather class contains four kinds of *feature definitions*: *routines*, *attributes*, *shared features* and *constant features* (the latter are sometimes called *shareds* and *constants* when the context is clear). Routines specify operations on objects, while attributes describe the layout of objects. A shared or constant feature is a variable that is shared by all objects of that class. The difference is that a constant feature is assigned once during initialization (before the user's program starts executing), and assignments to constant features are disallowed.

1.2.1 Sather-Related Concepts

Sather 1.0 [189] was designed by Steve Omohundro. This section[4] describes the more unique aspects of Sather 1.0 language that we take for granted in the description of pSather 1.0 (Sections 2.7, 4.9).

Separation of Subtyping and Code Inheritance

In Sather 0.1, there are two categories of classes – *value* (or *basic*) and *reference* class. There are five predefined value classes: `BOOL`, `CHAR`, `DOUBLE`, `REAL`, `INT`. Objects from this class are called value (or basic) objects and are passed by value. The only way to define another value class is to inherit (directly or indirectly) from one of the predefined value classes. There are also several predefined reference classes (e.g. `ARRAY{T}`, `ARRAY2{T}`). These and other programmer-defined classes are descendents of `OB` by default. (`OB` is an ancestor of all reference classes.) Objects from reference classes (referred to as "reference objects") are referenced by pointers and have to be created explicitly using the predefined `new` routine:

```
x := ARRAY{INT}::new;
```

Value and reference class *definitions* may be parametrized (e.g. `ARRAY{T}`).

A class may inherit from other classes. The inheritance rule restricts the inheritance tree of a predefined value class to be separate from all other predefined value and reference classes. For example, class `A` cannot inherit from both `CHAR` and `INT`. Inheritance in Sather 0.1 implies both *subtyping* and *code inheritance*. So if `A` inherits from `ARRAY{INT}`,

```
class A is
  ARRAY{INT};
```

[4] I would like to thank Steve for allowing me to extract sections of the 1.0 manual, often almost verbatim, for this introduction section.

`A` is a *subtype* of `ARRAY{INT}`, and also inherits the code and structure of the array.

In this subtyping system, a class `A` can be used as the type of a variable in both *dispatched* ("`x:$A;`") and *non-dispatched* ("`x:A;`") forms. The dispatched variable declaration "`x:$A;`" says that `x` will hold objects from `A` or its descendents. The declaration "`x:A;`" says that `x` will hold only `A` objects.

After some experience with Sather 0.1, we found that the language would be more expressive if subtyping and code inheritance are treated separately. (Similarly, America [7] have argued that inheritance is an implementation mechanism and should not be confused with subtyping, but none of the popular object-oriented languages adopts this separation.)

Sather 1.0 introduces *abstract* classes for subtyping.[5] Abstract classes are used to specify the type of variables. They define a set of interface routines and may or may not have actual implementation code for the routines. A class `A` is a subtype of the abstract class `$B` iff it inherits from `$B`:

```
class A < $B is ...
```

`A` will additionally inherit `$B`'s code if it is stated explicitly:

```
class A < $B is
  include $B ...
```

Variables may be declared using abstract types but there are no objects of abstract type.

Only abstract classes may be used in a dispatched manner for variable declarations. So if we have a reference class definition `A`, but not `$A`, it is illegal to declare a variable of type `$A`. A variable may be declared to have any one of five kinds of types: *value types* describe value objects, *reference types* describe reference objects, *external types* describe interfaces to other languages, *abstract types* describe sets of object types and *bound types* describe bound objects. (We will describe bound objects – iterators and bound routines later.) In 1.0's type system, a variable with an abstract type `$B` may hold any objects whose type is a subtype of `$B`. We refer to value and reference type (or class) collectively as *concrete* type (or class).

There are other changes in 1.0. All classes are now subtypes of `$OB`, so a variable of type `$OB` may hold any kind of object. Programmers may also define their own value class using the predefined tuple type or `BITS` class:

```
value class A1 is
  include {INT,INT}; ...


value class A2 is
  include BITS;
  constant bsize:INT := 8;
```

[5] Sather 1.0 also has *external* classes which define the interface between Sather and other languages; this is a generalization of the (predefined) `C` class in 0.1 which defines the interface of C routines used in Sather.

`A1` define value objects with two integer components while `A2` define 8-bit value objects. The type systems in 0.1 and 1.0 are described in detail in [188] and [189] respectively. 0.1 has only value and reference objects: a value object has a *value* class as its type while a reference object has a *reference* class as its type. 1.0 further introduces bound objects which have predefined bound types (not associated with any class).

Bound Routines and Iterators

Sather 1.0 also introduce another kind of feature called *iterators* (or simply *iters*). Iters are similar to routines but encapsulate iteration abstractions. Iter names end with an exclamation point "`!`". This symbol is part of the name and may not be separated from the rest of it. Iters may only be called within `loop` statements. The type specifiers declaring iter arguments may be followed by "`!`" to indicate that they will be re-evaluated on each iteration.

Unlike routine bodies, iter bodies may include `yield` and `quit` statements. Iter bodies may not contain `return` statements. In a loop, storage is associated with each iter call to keep track of its execution state.

```
loop
  x:=arr.elts!; ...
end;
```

When a loop is first entered, the execution state of all enclosed iter calls is initialized. The first time each iter call is executed in a loop, the expressions defining `self` and each of the arguments are evaluated left to right. On subsequent calls, however, only the expressions for arguments that are marked with a "`!`" are re-evaluated. `self` and any arguments not marked with a "`!`" retain their earlier values.

When an iter is called, it executes the statments in its body in order. If it executes a `yield` statement, control is returned to the caller and the current value of `res`, if any, is returned. Subsequent calls on the iter resume execution with the statement following this `yield` statement. If an iter executes `quit` or reaches the end of its body, control passes immediately to the end of the enclosing loop in the caller. In this case no value is returned from the iter.

Corresponding to routine and iter definitions are the bound objects – bound routines and bound iters. Bound routines and iters generalize the "function pointer" and "closure" constructs of other languages. They bind together a reference to a routine or iter and zero or more argument values (possibly including `self`). Bound objects are constructed by expressions of the form `#ROUT(...)` (for bound routines) and `#ITER(...)` (for bound iters). These surround an ordinary routine or iter call in which any of the arguments or `self` may be replaced by the underscore character "`_`". These arguments will be specified when the bound routine or iter is eventually called.

Each bound routine defines a routine named `call` and each bound iter defines an iter named `call!`. These have argument and return value types that correspond to the bound type descriptor. Invocation of this feature behaves like a call on the original routine or iter with the arguments specified by a combination of the bound values and those provided to `call` or `call!`. The arguments to `call` or `call!` match the underscores positionally from left to right (e.g. `i::=#ROUT(2.plus(_)).call(3)` is equivalent to `i::=2.plus(3)`).

Attribute Access and Other Syntactic Sugar

Object attributes are variables which are part of the internal state of reference objects. In Sather 1.0, only abstract and reference classes may define object attributes. Value objects do not have named attributes.

Another difference in 1.0 is that each object attribute definition causes the definition of two routines with the same name.[6] The "reader" routine returns the value of the attribute. It has a return type which is the attribute's type and no arguments. The "writer" routine sets the value of the attribute and has a single argument whose type is the attribute's type and has no return value. Thus the assignment statement which updates an object's attribute:

```
x.attrib := <expr>;
```

is really syntactic sugar for:

```
x.attrib(<expr>);
```

There are other forms of syntactic sugar, e.g. "`x[1,2] := 3;`" is sugar for "`x.aset(1,2,3);`". Appendix B gives a list of syntactic sugar expressions in Sather/pSather 1.0.

1.3 Contributions

The contributions of the pSather project so far can be categorized into three aspects — language,[7] implementation and use. This also reflects the thesis organization. Chapters 2 and Chapter 3 look at the design and implementation respectively of a parallel object-oriented language: pSather. Chapter 4 describes a few small and medium-sized abstractions and applications which we have programmed using pSather. The performance and clarity of the programs support our optimism that the object-oriented class library approach is a practical one for parallel programming. Chapter 5 also gives a high-level summary of the work, based on our experience and findings in Chapters 2– 4. The language implementation is only one aspect of a parallel programming environment; Chapter 5 describes future work that needs to be done for parallel programming environments to be as easily usable as commercial sequential environments.

[6] Sather 1.0 allows overloading and hence features of the same name as long as they can be disambiguated by their type signatures.

[7] Sather 1.0 also introduces new powerful encapsulation mechanisms, e.g. iters [182], but these are not part of the thesis

Language

PSather has a *cluster* machine model (Section 2.4) which includes non-uniform memory access (NUMA) as part of the shared-address space model. The cluster model applies to both shared-memory and distributed-memory multiprocessors, and is supported by several parallel constructs. We are not aware of any parallel language that supports a shared-address model for both shared-memory and distributed-memory multiprocessors, and makes NUMA an integral part of the language design.

In addition to the usual control-parallel constructs (e.g. threads [35]), pSather also supports a form of data-parallelism [131] (Section 2.6) which is usable for both irregular, dynamic data structures (e.g. trees) and the regular arrays. It decouples the execution model from the execution mode of single-instruction-multiple-data (SIMD) machines. The semantics of data-parallel and control-parallel constructs are orthogonal to each other, and they co-exist within a single framework.

PSather has been designed with real and future foreseeable target systems in mind. Therefore the language provides an interesting data point in the space of parallel object-oriented languages in its goals for practicality and future usefulness, and its integration of several concepts (e.g. NUMA, data-parallelism) within a language design.

Implementation

The current prototype implements the full set of pSather 0.1 language features (Chapter 3). We demonstrate that it is practical for a parallel object-oriented language to achieve reasonably good absolute performance (not just speedups). Furthermore, our CM-5 implementation shows that even if a distributed-memory machine does not support a shared-address space (at the hardware/operating systems level), the compiler can equally well provide the shared-address space abstraction when remote latencies are tolerable.

Although we have not explored in full details the implementation of an optimizing compiler for a parallel object-oriented language on distributed-memory machines, we do explore several optimization strategies (e.g. improving access of remote dispatched variables). One of the promising threads for more research appears to be the application of inlining to improve interprocedural analysis for distributed-memory implementations.

Our experience on a distributed-memory machine has also helped to identify architectural characteristics which we feel will be important in the implementation of high-level parallel languages; these issues are discussed in Section 3.8.

Use

The chapter on abstractions and applications (Chapter 4) show the implementations of several useful abstractions (e.g. a workbag and a replicated hash table with caching). The reuse of the same library classes in several applications shows how the object-oriented paradigm supports software reuse (particularly in a parallel context). The reuse/encapsulation mechanisms are not restricted to class inheritance, but also include class parametrization and the use of abstract classes. We do find several weak points in the current pSather 0.1 design, but Section 4.9 will describe how they are remedied in pSather 1.0.

We also show the pSather code used in the implementation of various abstractions and applications. They are examples of code written with a shared-address space model in mind, but actually executed on distributed-memory machines. They also show how programmers can express their algorithms cleanly (in pSather's cluster model), and yet produce relatively efficient programs that take into account important architectural features (e.g. remote vs. local memory accesses) that impact performance.

Chapter 2

Language Design

This chapter discusses the design of pSather which is based on parallel extensions of the existing object-oriented language Sather[1] ([188]).

Section 2.1 describes the design objectives in pSather. Then Section 2.2 discusses the design space of parallel object-oriented languages, with a survey of where various parallel object-oriented languages, including pSather, lies in the design space. The next sections then describe the three categories of parallel extensions in pSather. Section 2.3 describes the control-parallel constructs (i.e. thread creation and synchronization), assuming a simple machine and execution model. This simple model is generalized in Section 2.4 to a *cluster* machine model. This machine model balances the tradeoffs between a simple programming model with shared-address space and making the memory hierarchy of a multiprocessor visible. It is particularly relevant for scalable multiprocessors with distributed memory modules. In support of the execution model, a placement (or `@`) operator is introduced in Section 2.4. The description of the cluster model is relevant to understanding the next two categories of parallel extensions. Additional constructs that support the cluster model are covered in Section 2.5. Section 2.6 switches attention to the third category of constructs which supports a high-level data-parallel style of programming within the cluster model. In Section 2.7, we describe how the parallel constructs become more expressive in pSather 1.0. Finally in Section 2.8 we compare pSather with other parallel object-oriented languages to highlight the strengths and weaknesses of pSather.

2.1 Design Objectives

We start by looking at the desired characteristics of a high-level parallel programming language and then describe how pSather is designed to meet the objectives (Section 2.1.1).

[1] Section 1.1 describes the history of the project and clarifies the version of pSather described in this thesis.

Architecture Independence

The language must insulate the programmer from the underlying architecture (e.g. the hardware support for message-passing). This is a desirable characteristics of all general high-level programming languages in order for programs to be portable. It is particularly important for parallel languages because of the wide range of multiprocessor architectures.

There are two widely-suggested machine-independent models in parallel programming — message-passing and shared-memory.[2] A shared-memory model is simpler to study and use, because it is an extension of the familiar von Neumann model in which data is directly accessible to the thread of control. Because it makes the design and analysis of parallel algorithms simpler than would be the case with a message-passing model, there is a large amount of research on parallel algorithms based on the PRAM model of computation. Furthermore, the many research efforts to support shared-memory systems, whether via distributed shared-memory ([32, 130, 196, 95, 180]) or parallel macros/library packages ([44, 133, 134] are also evidence that a shared-memory model is desirable.

Although a shared-memory model independent of the underlying architecture is desirable, shielding the programmer completely from the architecture also makes the performance characteristics of the machine invisible. A uniform shared-memory model therefore is too restrictive. At the systems level, it is generally acknowledged that scalable multiprocessors will have distributed memory [142]. This is the case whether they provide virtual (non-uniform) shared-memory in hardware (e.g. KSR1 [201], Dash [161]) or rely on the programmer to write message-passing programs (e.g. on CM-5 [76]) or data-parallel programs (e.g. on MasPar MP-1 [70]).

This recognition of distributed memory is also found in theoretical research. There are efforts to extend the restrictive PRAM model and come up with more realistic models of parallel computation that take various practical parameters (e.g. communication latency) into account. Because of this trend, for a parallel language to be generally useful, it should provide constructs which let programmers express algorithms developed under these realistic computation models ([107, 1, 149, 81]).

To summarize, an architecture-independent model should be easy to use (like shared-memory) and reflects realistic memory latencies (like distributed-memory).

Support Software Reuse

For sequential programming, various software engineering techniques, including the object-oriented paradigm [151], have evolved to develop and maintain complex systems. One of the goals in software engineering techniques is to make software development more productive with reuse, i.e. by having software components which can be reused in various applications. Software reuse is well-

[2] The W2 language on the Warp machine [152] built in the mid 1980's at Carnegie Mellon University is an example of a language with a machine-dependent model. The model is a pipeline (or systolic) form of message passing on a fixed processor lattice. As a result, although the W2 compiler [152] implemented various optimization techniques (such as software pipelining) to generate efficient code, W2 exposes the underlying array architecture to the programmer and is difficult to use. This led to AL, a high-level shared-memory language on Warp, and to the work on a parallelizing compiler for systolic array [219, 220].

supported in the object-oriented approach because central object-oriented concepts like the ideas of *classes* and *class inheritance relations* encourage encapsulation and abstraction.

For parallel programs, sofeware reuse is especially relevant because of the difficulty of writing efficient, parallel code. This is why efforts are spent on building libraries (e.g. LAPACK [37, 91], CMSSL [75]). Such efforts however focus mainly on numerical algorithms and not on symbolic algorithms and irregular data structures.

One might argue that if a program can be ported to different architectures with relatively little effort, one is already achieving software reuse. But porting a parallel program to different multiprocessors is not an easy task. This is why there are tools to help programmers port applications from one class of architectures to another. An example is Adaptor [45] which helps to transform data-parallel programs to programs with explicit message passing for distributed-memory MIMD machines.

We think that the language must make it easy to reuse code libraries for a wide range of algorithms and data structures. The language must encourage modular structures in parallel programs, making it easier to port to different multiprocessors.

Expressiveness of Parallel Constructs

In the theoretical literature, certain common paradigms of parallel algorithms (e.g. divide-and-conquer [122, 72], pipeline [121], normal algorithms [221] and master-worker [59]) have been recognized in various works to parallelize sequential algorithms or develop new parallel algorithms. One finds that it is more straight-forward to program algorithms in the above paradigms when certain kinds of language constructs are available. Examples of such *control-parallel* language constructs include the ability to create parallel threads of control and synchronization mechanisms such as lock, barrier etc.

On the other hand, the data-parallel model has also commonly served as a basis for description of algorithms (such as those for graph and matrix problems [192]).

While some parallel languages (e.g. Multilisp [119], Spur Lisp [234]) have focused on providing control-parallel constructs, others (e.g. C* [127], Fortran D [132]) have concentrated on supporting data-parallel constructs. It is unnatural, if not difficult, to express data-parallel algorithms in control-parallel languages and vice-versa. We would like to support both these styles of computation in a coherent framework.

Efficient Implementation

The efficiency goal is obvious, but it has been overlooked sometimes in favor of language constructs which are too high-level and general to be implemented efficiently. The usefulness of multiprocessors is that their raw performance exceeds that of sequential machines. Even with the considerable performance improvement of sequential processors via superscalar and superpipelining techniques,[3] the combined raw performance of many fast sequential processors will still exceed the performance of a single fast processor. Our goal is for programmers to tap this performance. Therefore the parallel language must be efficiently implementable to make parallel programming practically attractive. Although our resource constraints do not permit a production quality compiler, we will provide timing figures in later chapters to show that even a stable prototype implementation can be relatively efficient.

2.1.1 Meeting the Objectives

PSather attempts to meet the design objectives in the following ways.

To achieve architecture independence, we distinguish between shared-memory and shared-address space. The essential characteristic of the von Neumann model captured by a shared-memory model is actually the globally unqiue addresses within an executing program, i.e. a shared-address space. PSather supports a distributed-memory, shared-address machine model which is simple and at the same time reflects the non-uniform memory hierarchy of the new, scalable multiprocessors. Our model has a 2-level distinction between local and remote memory (Section 2.4). There are also language constructs that let a programmer take into account both data locality and the longer latencies of remote operations (Section 2.6). PSather is an object-oriented language and the notion of objects is particularly relevant in using memory hierarchy in an abstract way. This is because objects provide a level of granularity at which programmers can manage local/remote memory in a natural fashion.

Software portability/reuse is a critical issue in parallel programming. By adopting an object-oriented approach like pSather, we hope that the techniques that promote software reuse and the building of parallel code libraries will also help in making parallel programs more portable. The libraries hide the complexity of parallel data structures/algorithms and may be re-written from one architecture to another for efficiency reasons. But the user programs should remain unchanged. Any rewrite of the code libraries should, however, still be relatively straight-forward using the language facilities.

We adopt a "design by contract" philosophy, similar to Eiffel [174]. The interface of a class will specify what it provides in terms of efficiency, safety and functionality. There are tradeoffs between efficiency and safety in parallel programming; for example, data structures which are not shared by parallel threads can be more efficient because there is no synchronization overhead. The user understands best what the client classes require of the library classes. He can then make the efficiency-safety tradeoffs and rely on the class interface as a contract with the library designer to provide the specified efficiency/safety/functionality.

In terms of expressiveness, pSather has both control-parallel (Section 2.3) and data-parallel constructs (Section 2.6). They have orthogonal semantics, and therefore a data-parallel execution

[3] Hennessy and Jouppi [129] give a description of superscalar and superpipelining techniques which are used in the Motorola 88110 [87] and MIPS R4000[178] processors respectively.

may contain thread-based parallelism and vice versa.

Last but not least, our design is based on Sather because we want pSather to leverage off Sather's efficient implementation. Performance measurements on Sather [164] have shown that a Sather program incurs small overhead (in the range of 10%) relative to a corresponding C program. Although this efficiency is not a sufficient condition for pSather to be efficient, it is a necessary condition and supports our optimism for an efficient implementation of pSather.

2.2 Survey of Parallel Object-Oriented Language Designs

This section discusses the design space of parallel object-oriented languages. We discuss how the major parallel programming issues — creation of processes (for parallel execution), synchronization/communication and efficient memory accesses (or data locality) — are generally handled in object-oriented languages. We treat communication and synchronization together because processes need to communicate in order to synchronize and synchronization mechanisms also serve as ways for processes to communicate. Data locality is important if the language is to be used and implemented on distributed-memory machines. Then we look at how inheritance interacts with synchronization code.

We cite some examples in the discussion of each design dimension, and a summary of the design choices of some of the more well-known languages is given in Section 2.2.2.

Processes/Threads

In order for parallelism to exist, independent processes (or *threads*[4] of control) must be created to execute concurrently. There are three main views of threads in an object-oriented language.

A thread may be treated as a first-class object in the system (e.g. PRESTO [33]), just like any other data objects. In this case, routines (or methods in traditional OO terminology) are defined in the threads class to activate, suspend and perform other synchronization operations (e.g. fork-join) on the threads. New types of threads can be defined via class inheritance. Since the thread objects are independent of data objects, multiple threads can execute on an object in parallel.

A second approach is to have actors (also called active objects in some languages), each with its own message queue and thread of control. The thread of control may not exist in actual implementations but from the programmer's point of view, each object has a thread that receives and services incoming requests. In this case, a thread is not a first-class object and is only associated with an object in the system. There is no way for a programmer to directly manipulate a thread. This is the approach taken by the actor class of languages ([4],[3]). For example, in ABCL [231] and POOL2 (Parallel Object-Oriented Language [8]), when an object is created, it also becomes active

[4] As we are preparing for a description of pSather in the next sections, we will try to rephrase ideas in our own terminology when appropriate.

with a thread acting as a server. An actor (data object with thread) receives messages from others and processes them.

In the third approach, objects are passive entities while threads are loci of execution control independent of objects (e.g. Hybrid [184], COOL [62]). The programmer does not have any explicit handle to the thread, so that he cannot perform operations like moving the thread from one scheduler object to another. Most languages in this model allow multiple threads to execute in an object at the same time. Other languages (e.g. Hybrid [184]) group objects into protection domains such that at most one thread can be active in a domain.

Other than the three main approaches above, a combination of the approaches is possible and used in some languages. For example, Rosette has both actors and passive objects [217]. PC++ [103] presents another variation by supporting the creation of multiple parallel threads, each working on a passive object, resulting in a data-parallel model of computation.

Each approach leads to a different programmer's view of how threads are created and executed. For example, if threads are explicit objects, a new thread is created just like any other object. To create and schedule a thread, one will invoke routines in a `THREAD` class. This approach also requires the language to support routines or some form of closures as first-class objects, so that a created thread can use the value of the routine/closure to decide what to start executing. In the actor model, a new thread is implicitly created and becomes active whenever an object is created. So instead of introducing a thread creation construct, such a language needs a way to handle incoming requests and decide which messages can be received. In POOL2, each class definition has a *body* code (e.g. POOL2). When an actor is created, it starts executing the body code that decides which incoming messages can be received and invokes the routine corresponding to a received message. In the third model (threads/passive objects), there are separate mechanisms to create threads and objects, so languages with this model have constructs that support explicit thread creation (e.g. the reflex operation in Hybrid [184] or invocation of *parallel* function in COOL [62]).

Synchronization/Communication

With multiple threads of control, there must be some ways for the threads to communicate and synchronize. We first list several common synchronization patterns found in parallel programs, so that we can later use them to illustrate the synchronization mechanisms in particular parallel object-oriented languages.

Lock protection. One common pattern is when two or more competing threads access shared data. To maintain data consistency, the variable(s) holding the data may have to be protected so that the updates are serialized.

Barrier synchronization. Another situation is when the threads wait for one another at a certain point before all can proceed on. An example in which this happens is a program with distinct computation phases such that no thread may start a phase before the others. The point where the threads must wait before they can all continue on, is called a barrier [44].

Conditional wait. Sometimes we want a thread to suspend and wait for certain conditions to become true before it is allowed to resume execution. An example is the bounded buffer[5] for which we want a thread performing a `get` (`put`) routine to wait until the buffer is non-empty (non-full).

There are two general approaches to achieving synchronization among threads [14] — shared data or message-passing. An example of the shared data approach is the use of monitors ([136, 227]) in Concurrent Pascal ([120, 123]) and Mesa [154]. In this approach, the mechanisms to achieve lock protection include semaphores [88] and protection of critical sections by Dekker's algorithm [30], while a construct which provides conditional wait is the *condition variables* (originally proposed together with the monitor concept [136]). On the other hand, the message-passing approach is characterized by the rendezvous in Ada ([23, 223]), channels in CSP (Communicating Sequential Processes [137], of which OCCAM [195] is an implementation) and the send/receive constructs in PLITS [96].

In an object-oriented language, conceptually, objects interact among themselves by message passing. An object invokes a *routine* (or method) of another object or itself by sending a message to the destination object. It would therefore seem natural that object-oriented languages should adopt a message-passing approach for synchronization. This is indeed the approach adopted by the actor class of languages such as POOL2 and ABCL. Synchronization is achieved by controlling the receipt of messages.

In the actor languages, because there is only one thread within an actor that handles incoming requests and accesses the internal data sequentially, the access/update of an object's data is naturally protected. The actor languages therefore automatically provides lock protection. An actor can further select a set of messages it is ready to receive; it waits until a message in the set is received and is then re-activated. The ability for selective message-receipt serves as a form of conditional wait. In this approach, to do barrier synchronization among a set of actors, the programmer has to construct a barrier object. Every actor which is ready to perform a barrier-sync sends a message to the barrier object and then wait for a `Barrier-Ready` reply. When the barrier finds that all the objects are ready, it sends a `Barrier-Ready` message to the waiting objects which then proceed on.

Message-passing is not the only way to achieve synchronization in parallel object-oriented languages. The model with explicit thread objects (e.g. PRESTO) uses the shared data approach. The synchronization mechanisms are more in the tradition of constructs used in concurrent programming, such as fork-join [86] and semaphores [88], except that these mechanisms are defined

[5] The bounded buffer has been a traditional example used in the discussion of various high-level parallel programming constructs and can be found in many references e.g. [136].

as in classes and are not embodied as fixed language constructs. For example, certain forms of synchronization are performed via routines in the thread class, like the *join* in PRESTO which allows a thread object to wait for the completion of another thread and to receive the latter's return value. Other forms of synchronization are supported by defining classes with synchronization functionalities (e.g. spinlocks and suspending locks in PRESTO). These synchronization classes are in turn used in the definition of data classes in order to protect the data and/or synchronize the execution of multiple threads invoking routines on the same object. They can also be used to build higher-level synchronization classes (e.g. barrier). In this approach, synchronization such as lock protection, barrier and condition wait are implemented as class definitions (e.g. lock, condition and barrier classes in PRESTO).

The synchronization mechanisms in the threads/passive objects model depend on whether multiple parallel threads can execute in an object.

- When multiple threads cannot execute in an object (e.g. Hybrid) locking is automatically available. But it is not clear how the condition-wait and barrier synchronization can be easily achieved in Hybrid.
- When multiple threads can execute in an object, the shared data approach is used and explicit synchronization mechanisms are provided. The difference from the PRESTO-like model is that the synchronization constructs are normally provided via additional language extensions. We consider each of the synchronization patterns (lock protection, barrier and conditional wait) in COOL [62]. To achieve locking, attributes and functions can be qualified as mutex at declaration. To do a barrier, there is a predefined `binc` construct which encloses a block of statements. The current thread suspends until all threads created during the execution of the block terminate. The functionalities of a condition variable is provided by objects of a predefined class `cond`.

Object placement

One attractive feature of object-oriented programming is that objects provide a high-level abstraction for programmers to deal with memory (shared or distributed). Because (local/remote) memory latency costs are often more critical than costs of computation cycles, the programmer's view of objects is especially relevant to program efficiency. We will treat the two aspects of placement of objects — allocation and relocation — together.

The languages cited earlier (PRESTO, POOL2, COOL, actor languages) do not include the placement of objects as part of the language semantics. As discussed in Section 2.1, a uniform shared-memory model (that treats objects uniformly in a shared object space) hides the communication costs of remote accesses from the programmer. The objects (actor or passive) are location transparent. As a result, when implementing these languages (e.g. POOL2, COOL) on a non-uniform memory

```
CLASS Queue
-- An unbounded queue; definitions of insert_back,
-- get_front omitted.
BODY
  WHILE <condition satisfied>
  DO
    IF empty
    THEN ANSWER(insert_back)
    ELSE ANSWER(insert_back, get_front)
  OD;
YDOB
```

Figure 2.1: Textually contiguous next-message-set specification.

access (NUMA) or distributed-memory multiprocessor, a runtime system (e.g. Tarmac [168]) that automatically performs load balancing and maps allocated objects on different processors is needed.

There are however parallel object-oriented languages which support a non-uniform shared-address space in a high-level manner. For example, Sloop [167] lets the programmer specify *alignment* relationships among objects via calls to an *align* routine. The alignment relationship specifies the "spatial" relationship of objects; for example, two strongly aligned objects are always located on the same processor so that whenever one is moved to a different processor, the other has to move with it. Calls to the align routine may cause objects to be moved. Sloop objects are allocated without any location information. The placement of an object is given indirectly via its "spatial" relationships relative to other objects. An object's location may be linked to other objects' locations, but never to specific processors.

Emerald [146] is another example which supports non-uniform treatment of objects and will be discussed later (in Section 2.2.2) when we survey specific languages.

Inheritance

The class inheritance relationship is an important aspect of object-oriented programming. Several research efforts point out the problems that may arise because of interference between inheritance and concurrency ([47, 147, 173]). This inheritance-concurrency interference is called *inheritance anomaly* in the literature. Because of inheritance anomaly, some parallel object-oriented classes use prototypes and delegation instead of inheritance for code-sharing.

We look at actor languages because this problem has been examined in more detail for actor languages. In the earlier discussion of synchronization mechanisms, we noted that the actor languages use the set of receivable messages to control synchronization. There are two possible ways to implement the control of message receipt and inheritance anomaly arises in both cases.

The first approach is to specify the next set of messages which can be received. This next-message-set specification can be textually contiguous or textually distributed. The second approach is to associate a condition with each routine; an object only handles message for a routine whose condition is satisfied.

In a textually contiguous next-message-set specification (e.g. POOL2, Figure 2.1) there is a central body code which uses constructs like the guarded alternatives in CSP to specify the messages which can be received. The example in Figure 2.1 is an unbounded queue. When the queue is empty, we can only insert but not remove items. In this approach when a subclass adds a new routine `r`, the central body code has to be rewritten to take `r` into account. This prevents a subclass from inheriting the body code from its superclass. If there is multiple inheritance, it becomes more complicated as to how body codes from different superclasses should be combined.

Actor languages may also have textually distributed next-message-set specification. For example, in Act3 [2] each routine uses the *become* operation to give a replacement behaviour; the new behaviour controls the next set of messages which can be received.[6] Routines therefore have two parts — a computation and a synchronization part (giving the next set of receivable messages). Now suppose a subclass has a new routine `r`. Many inherited routines may be invalidated because they have to include `r` in the next set of receivable messages (even if the computation parts work correctly in the subclass).

Figure 2.2 shows inheritance anomaly in a language with distributed next-message-set specification. The example (written in pseudo-Sather) again uses an unbounded queue (`QUEUE{T}`), now with an additional subclass `DEQUE{T}`. A `QUEUE{T}` class has routines: `insert_back`, `get_front`. The `insert_back` routine allows both `insert_back` and `get_front` messages to be received next. The `get_front` routine allows both messages to be received next when the queue does not become empty; otherwise, it allows only the `insert_back` message to be received next. In a sequential object-oriented language, it is relatively straight-forward to implement `DEQUE{T}` as a subclass of `QUEUE{T}` by adding new routines: `insert_front`, `get_back`. The inherited routines can be reused without change. But in an actor language with textually distributed next-message-set specification, both inherited routines have to be redefined, e.g. when `insert_back` specifies the next set of receivable messages, it has to include the new routines.

Kafura et al. [147] suggests a solution that treats the next set of receivable messages as an abstract entity. Descendent classes only need to redefine these abstract entities and not the computation parts of inherited routines. Tomlinson et al. [217] propose a similar augmentation to the next-message-set approach.

The inheritance anomaly in the `QUEUE{T}/DEQUE{T}` example however does not arise in approaches which associate a condition with each routine. Decouchant et al. [85] describes how a boolean *activation condition* is specified for each routine. Only when the boolean condition is true, is the corresponding routine enabled and a message for that routine receivable. Activation conditions

[6] Although Act3 does not have inheritance, the following interference problem will still arise in an Act3-like language with inheritance.

```
class QUEUE{T} is

  insert_back(v:T) is
    ...
    become(insert_back, get_front);
  end;

  get_front:T is
    ...
    if (empty) then become(insert_back);
    else become(insert_back, get_front) end;
  end;


class DEQUE{T} is

  insert_back(v:T) is
    ...
    become(insert_back, insert_front,
      get_front, get_back);
  end;

  get_front:T is
    ...
    if (empty) then
      become(insert_back, insert_front);
    else
      become(insert_back, insert_front,
        get_front, get_back) end;
  end;

  -- Plus insert_front, get_back routines.
```

Figure 2.2: Textually distributed next-message-set specification interferes with inheritance, making redefinition necessary.

are separated from the routine definitions so that they can be inherited and redefined independently. Variations of this approach to decouple the synchronization constraints from computation have been suggested to solve the inheritance anomaly problem. For example Frølund [101] argues that synchronization constraints should disable rather than enable routines for incremental specification.

An exhaustive study of inheritance anomaly is beyond our scope, but such a study for actor languages is given by Matsuoka [171]. His thesis describes the problem in detail, discusses various proposals in the literature and the situations when they would fail, and presents language constructs in actor languages to overcome this problem.

The problem of reusing inherited synchronization code also occurs in non-actor languages as demonstrated by an example in Figure 2.3. In the class `QUEUE{T}`, a thread must execute synchronization code in `insert_back`, to protect against other threads executing `insert_back` or `get_front` in parallel. Now suppose we want a subclass `QUEUE_R{T}` with a `read_front` routine which reads the front element of the queue without removing it. `QUEUE_R{T}` has a readers-writer synchronization (as opposed to a writer-only synchronization in `QUEUE{T}`). Multiple threads can execute `read_front` in parallel on a `QUEUE_R{T}` object (multiple readers), but only one thread is allowed to operate on the queue when `insert_back` or `get_front` is executed (single writer). Because the synchronization has changed from writer-only in `QUEUE{T}` to readers-writer in `QUEUE_R{T}`, the `insert_back` and `get_front` routines have to be rewritten. This again shows the difficulty of reusing inherited synchronization code.

2.2.1 PSather's Design Choices

We now look at where pSather is situated in each of the design dimensions. We have described three main approaches to incorporate threads into object-oriented languages. The approach with first-class thread objects is flexible because it does not require any language extensions and allows different thread and synchronization objects, and scheduling strategies to be implemented according to the programmer's needs. The actor approach offers a well-studied theoretical concurrent object-oriented model [71, 2], but is not without its problems (e.g. inheritance anomaly and efficient implementation). PSather follows the model with passive objects and threads as independent loci of control, and has thread creation constructs (Section 2.3). It also supports data-parallel computation with a `dist`-statement (Section 2.6) which implicitly creates parallel threads for distributed data structures.

PSather does not treat threads as first-class objects because the initial version of Sather [188] does not support any form of closure or routine as first-class objects. The new language specification has a form of closure called bound routines (Section 1.2). As pSather evolves, one might imagine eliminating the thread-related constructs and having a predefined `THREAD{T}` class instead (Figure 2.4). On the other hand, the advantage of a distinct thread creation construct is that it helps to clarify programs and lets someone reading a program to easily pick out the code

```
class QUEUE{T} is

  insert_back(v:T) is
    <Protect non-full queue against any parallel
    "insert_back", "get_front" call while "v" is
    inserted at the end of the queue.>
  end;

  get_front:T is
    <Protect non-empty queue against any parallel
    "insert_back", "get_front" call while the last
    element of the queue is retrieved.>
  end;



class QUEUE_R{T} is

  insert_back(v:T) is
    <Protect non-full queue against any parallel
    "insert_back", "get_front", "read_front" call
    while "v" is inserted at the end of the queue.>
  end;

  get_front:T is
    <Protect non-empty queue against any parallel
    "insert_back", "get_front", "read_front" call
    while the last element of the queue is retrieved.>
  end;

  read_front:T is
    <Protect non-empty queue against any parallel
    "insert_back", "get_front" call, but allow
    parallel "read_front" calls.>
  end;
```

Figure 2.3: An example of inheritance anomaly in non-actor languages.

```
class THREAD{T} is
  abstract fork(ROUT);
  < Plus other thread related operations >
end;
```

Figure 2.4: THREAD{T} class

that departs from normal sequential execution.

We also did not adopt the actor model [2] because of the performance costs of maintaining a message queue for each object, and disallowing parallel operations (e.g. reads) on an object. These costs conflict with pSather's design goal to be suitable for efficient implementations. Another reason for not adopting the actor model is that, for many applications, the granularity of parallelism need not correspond to object granularity. For example, in a matrix computation, the matrix and matrix elements are natural choices for objects, but an algorithm may work best on sub-matrices in parallel. If the object granularity is coarser than thread granularity, the actor model restricts the amount of parallelism;[7] if the object granularity is too fine, extra overhead is wasted to maintain the actors' message queue.

In this design, the object-oriented term "message passing" in pSather does not involve communication between threads but has procedure invocation semantics instead; we might view it as message-passing between passive objects rather than between threads. PSather does not need message-passing forms of parallel constructs like actors in POOL2 [8], broadcast in Orca [21], or asynchronous reply in Natasha [78] and ConcurrentSmalltalk [230]. In pSather, sequential routine calls are viewed as the default synchronous mode of message-passing while the thread creation corresponds to asynchronous message-passing.

In terms of synchronization/communication, we have seen that the approach (whether shared data or message-passing) taken by a parallel object-oriented language is related to how it incorporates threads. PSather has a threads/passive objects model. Because "message-passing" is between objects, not between threads, we cannot use message-passing to achieve synchronization. PSather uses the shared data approach for synchronization, and introduces a predefined **GATE** class which will be described in more detail in Section 2.3.

PSather also deals with object placement issues, by supporting a general machine model that is applicable to both shared- and distributed-memory machines. The machine model, described in Section 2.4, is augmented by language constructs that explicitly dictate where objects are allocated and where threads execute (Sections 2.4, 2.5). PSather does not support automatic object migration like Sloop or Emerald. Although this might be viewed as a drawback, we feel that this is in fact an advantage because explicit control over object/thread locations makes it easy for class library designers to experiment with various load balancing and object placement strategies. The optimal strategies are then hidden in classes from the general users.

On the question of inheritance anomaly, our view is that inheritance is not the only means of reusing code. Hence the inheritance anomaly problem which is in large part caused by breaking the class encapsulation, should not be over-emphasized. Even in a sequential object-oriented language, a subclass can break the encapsulation of its superclass when it inherits code from the latter as

[7] Concurrent Aggregates [67, 68] tries to solve this problem by introducing *aggregates*, such that the objects in an aggregate match the granularity of parallelism. Programming aggregates, however, requires additional programming effort to undo the restriction caused by "one thread per object".

described by Snyder [207]. Although the ideas to decouple synchronization and computation codes are useful, when such a general mechanism is implemented for all classes, the runtime costs are high. The ideas also do not overcome the general problems that arise from breaking class encapsulation via inheritance. Taking the earlier `QUEUE{T}` and `DEQUE{T}` examples (of which variations appear in various work [147, 47]), we note that even in sequential programming, the implementation of new routines in `DEQUE{T}` requires knowledge about the actual implementation of `QUEUE{T}` and already breaks the encapsulation in `QUEUE{T}`. The pitfalls in inheriting (computation and synchronization) code may be better circumvented by following certain disciplines in programming (such as suggested by Sakkinen [202]), and designing flexible encapsulation mechanisms that work well in general and not just in parallel programming. Section 2.7.5 shows how general encapsulation mechanisms in pSather (e.g. bound routines) help to separate synchronization and computation codes, and alleviate the inheritance anomaly problem.

2.2.2 Design Choices of Other Languages

We now select some of the more distinct parallel object-oriented languages and describe where they are in the design space. While describing the various systems, we will also try to translate their terminology into ours so that a more consistent picture can be obtained. Because of the variety of languages, this survey is by no means exhaustive.[8] We focus our attention on languages which (1) are parallel extensions of major sequential object-oriented languages (e.g. C++ [210], Smalltalk [109], Eiffel [175]), (2) or embody a unique model or concept. The following categorization is not mutually exclusive — a language may suitably fall into more than one category.

To reduce the amount of survey, we give the design characteristics of the following languages in Table 2.1, but omit any description — Natasha [78], Amber [64], Mentat [115, 116, 183], PROCOL [224], CLIX [141], Dragoon [19], Sloop [167], Hybrid [184], PRESTO (and other systems which provides threads and synchronization libraries [94, 33, 27, 90]), SOS [206, 169], Rosette [217], POOL2 [8, 9, 10]. We have however described some of their more interesting characteristics earlier.

Parallel Versions of Smalltalk

We first look at some parallel implementations of Smalltalk.

Distributed Smalltalk

Bennett [31] describes an implementation of Distributed Smalltalk on a network of Sun 2 workstations. It aims to support a multi-user environment and to allow interaction among multiple processes on different workstations. It does not contain any language extension to Smalltalk. The implementation does not support a shared object space; each program retains a logically distinct

[8] Further survey of concurrent object-oriented programming can be found in [216, 74].

Language	Thread	Synchronization	Object Placement	Inheritance
pSather	P/DP	SD	Y	I
Distributed Smalltalk	P	SD	Y	I
Multiprocessor Smalltalk	P	SD	N	I
Concurrent Smalltalk	P	SD	N	I
Eiffel //	A/P	SD	N	I
Parallel Eiffel	L	SD	N	I
CEiffel	P	SD	N	I
COOL	P	SD	Y	I
μC++	A/P	SD/MP	N	I
PC++	DP	SD	Y	I
ABCL/1	A	MP	N	D
Emerald	P	SD	Y	N/A
Orca	P	MP	Y	N/A
Concurrent Aggregates	A	MP	N	D
Natasha	P	MP	Y	N/A
Amber	L	SD	Y	I
Mentat	A	MP	N	I
PROCOL	A	MP	N	–
CLIX	A	MP	Y	D
Dragoon	A/P	SD	Y	I
Sloop	P	SD	Y	–
Hybrid	P	MP	N	D
PRESTO	L	SD	N	I
SOS	L	MP	Y	I
Rosette	A	MP	N	I
POOL2	A	MP	N	N/A

- Thread: A=Actor language, L=Library of threads and synchronization classes, P=Threads and passive objects, DP=Data-parallel model with threads and passive objects (We view DP as a variation of P, in which the language supports the creation of multiple parallel threads, each working on a passive object.)
- Synchronization: SD=Shared data, MP=Message passing (Languages with only a data-parallel model are marked with "SD".)
- Object Placement: Y=Includes placement semantics, N=Does not include placement semantics
- Inheritance: N/A=No inheritance, D=Delegation, I=Inheritance (Since there is a wide variety of solutions trying to solve inheritance anomaly in actor languages, there is no neat classification.)
- "–" means that the information is not available from the references.

Table 2.1: Summary of the design choices of the more well-known parallel object-oriented languages.

address space. Because it follows a traditional remote operation model, it is possible to invoke routines on remote objects in a transparent manner. Synchronization is provided by Smalltalk classes (i.e. shared data approach) and there is support to move and copy objects to different processors.

Multiprocessor Smalltalk

MP Smalltalk [191, 190] is an implementation of Smalltalk-80 on a multiprocessor without any language extensions. Unlike Distributed Smalltalk, it presents a shared-address space to the programmer. Parallelism and synchronization are derived from the `Process` and `Semaphore` classes. The system therefore treats `Processes` (threads) like any first-class objects. `Semaphore` objects can be used to build synchronization mechanisms such as barrier. Objects are not actors and multiple processes may access an object in parallel. But it is possible to use semaphores to build atomic objects which behave like actors.[9]

A simple form of data-parallel (or object-parallel) computation is supported by creating a process for each object in a `ParallelCollection`. Since the runtime environment (such as a scheduler) is visible, a programmer can also explicitly manage process priorities.

A parallel implementation of Smalltalk, however, requires modifications to the system because some design and implementation aspects of Smalltalk inherently assume a concurrent but not truly parallel environment.

ConcurrentSmalltalk

ConcurrentSmalltalk [230] extends Smalltalk-80 by introducing asynchronous calls and `CBox` objects. (This approach is unlike MP Smalltalk which does not extend Smalltalk-80.) An asynchronous call starts a new thread of execution for the receiver object. The caller does not wait for the result. Instead it gets a `CBox` object which acts as a *future* [119]; the result computed by the receiver is deposited in the `CBox` object. The receiver may return the result asynchronously, thus allowing its thread to continue execution. There is no language construct to distinguish between local and remote objects.

Parallel Versions of Eiffel

Since Eiffel is one of the more well-known object-oriented languages and Sather was derived originally from Eiffel, it is therefore instructive to survey various efforts in extending Eiffel for parallel execution. We have omitted the works in [118, 148, 176] and focus on three efforts that illustrate quite different approaches.

[9] This is done by "wrapping" a semaphore object around the object which is to be the actor. A message targeted to the enclosed object is sent to the semaphore. The semaphore is acquired and the message is relegated to the enclosed object. This prevents multiple threads from accessing the enclosed object.

Eiffel //

Eiffel // [55, 54, 56, 57, 58] extends the Eiffel language and supports both passive and active objects (actors). A call to a passive object is synchronous, just like any procedure call. A call to an active object (process object) is an asynchronous message send and relies on the destination process to explicitly receive the request. For example, after the asynchronous call "`res:=obj!function(parameters)`", the sender continues execution until it needs the value of the result `res`. Then it suspends until the destination process returns a result.

Active objects are instances of the `PROCESS_POWER` class and its descendents. When the `Live` routine[10] is invoked on an active object, a new thread starts executing the code in the `Live` routine. The new thread services request messages for the active object in a FIFO order. Calls on active objects are therefore serialized because only one message can be handled at a time.

To handle the inheritance anomaly problem, Eiffel // also extends the Eiffel language to allow routine as first-class objects. It is therefore possible to build a table that explicitly associates a routine with a function that evaluates the condition for which the routine is blocked. The `Live` routine of an active object may then use this table to decide which messages to receive.

Parallel Eiffel

Parallel Eiffel [113, 73] has a different approach from Eiffel // as it is more a tool for exploring concurrent object-oriented languages. It therefore only introduces new Eiffel classes to deal with parallelism and does not extend the language.

Its approach resembles PRESTO's, but Parallel Eiffel makes a finer distinction among several classes used to create first-class thread objects. For example, it distinguishes between a `THREAD` object (which gives the programmer a handle to the lightweight thread and provides operations such as `suspend`, `resume` etc.), and a `CALL` object (which contains the object and routine that a thread is executing). Parallel Eiffel has a `BOX{T}` class which is used to store the result of a thread when it terminates. This is almost like the `CBox` in ConcurrentSmalltalk, except that a `BOX{T}` object must be explicitly supplied by the caller and passed to a thread object.

While C++ (and thus PRESTO) allows a variable number of arguments, Eiffel does not. As a result, an awkwardness of this thread library approach in Eiffel is that additional parametrized classes are needed to build threads which invoke an object's routine with additional arguments.

Parallel Eiffel provides several synchronization classes (e.g. `MONITOR`, `CONDITION`, `SYNCHRO`) which are used by other classes for protection and synchronization. It is also possible to build actor classes that serialize object operations using these synchronization mechanisms.

[10] The `Live` routine is similar to the body code of POOL2, but unlike POOL2, `Live` has to be explicitly invoked for a new thread to start executing.

CEiffel

We have seen that Eiffel // extends Eiffel and Parallel Eiffel sticks to using the class mechanism for providing parallelism. The approach of CEiffel [165] is to add annotations that would be used by a compiler for parallel machines but otherwise ignored. For example, annotating a routine definition with "–v–" indicates that a new thread is forked when this routine is called. When a routine `r` in class `A` is annotated with "–>–" a new thread is created and executes `r` autonomously when an `A` object is created and initialized. This creates a thread like that of an actor, but unlike POOL2, multiple such threads may execute in an object.

Synchronization is via atomic objects such that at most one call is handled. This default condition however can be relaxed by the "–||. . . –" annotation. When a routine is so annotated, the ". . . " give names of routines which are compatible and can execute in an object in parallel. An annotation "–||–" marks a concurrent class in which all routines are compatible. In addition, the precondition of a routine provides a form of event synchronization by delaying the execution of a routine until the precondition is satisfied. The "–@–" annotation is used to mark preconditions which may cause delays. (One might argue that the last synchronization mechanism is a misuse of preconditions. When a caller invokes a routine whose preconditions is annotated by "–@–", the caller no longer needs to check whether the precondition is satisfied, defeating the original intended use of preconditions.)

Parallel Versions of C++

In this section we look at several parallel versions of C++. It is interesting to note how different approaches have been adopted with the same sequential language, resulting in languages/systems with widely different characteristics.

COOL

COOL [62] supports parallelism by allowing class routines to be declared as `parallel`. New threads are created when such routines are invoked (although the caller has the option of making it a serial call). There are constructs for different types of synchronization. For mutual exclusion (i.e. exclusive access to an object), a routine can be declared as `mutex`. (A set of non-`mutex` routines can execute in parallel on an object.) For event synchronization, condition variables can be declared using a predefined type `condH` with `wait`, `signal` and `broadcast` operations. A `condH` object is also used to hold the result of a parallel thread. To synchronize threads, there is a `binc` statement which suspends the current thread until all threads created within the scope of the `binc` statement have terminated.[11]

An interesting feature of COOL are *affinity hints* to improve locality, load balancing and cache usage for `parallel` routines.

[11] The `binc` statement resembles the `cobegin-end` statement (Section 2.3.3) in pSather.

```
parallel void f(...)
  affinity(x, OBJECT);
  affinity(y, TASK);
  affinity(p, PROCESSOR);
```

An affinity declaration using `OBJECT` gives an object `x` with which a thread should be collocated to improve locality. A `TASK` declaration identifies tasks (threads) that share an object `y` and that are scheduled together to improve cache use. Finally a `PROCESSOR` declaration schedules a thread on a particular processor `p` and lets the programmer perform load balancing.

μC++

μC++ [51] distinguishes among three concurrency notions that might be associated with an object. They are thread, execution state and implicit lock.

Class definitions can have additional specifications `uCoroutine`, `uTask` and `uMutex` each of which reflects a certain valid combination of presence or absence of thread, execution state and lock associated with an object. For example, a `uMutex` class provides passive objects which allows at most one thread to execute in the object at any time, thus acting like a monitor in Concurrent Pascal. A `uTask` class defines actor objects (like those of POOL2, Eiffel //) which have their own thread and execution state; the thread starts executing a distinguished routine `main` after the object is created and initialised.

There is a `uAccept` statement that acts like the guarded alternatives in CSP [137]. This allows a `uMutex` object to select the routines to be executed based on the results of the guard expressions. There is also a `uCondition` class on which threads can be suspended and resumed.

Unlike COOL, μC++ does not have any language constructs for doing thread or object placement.

PC++

A major concern of PC++ [103, 159] is to be able to build complex distributed data structures (e.g. arrays, lists, sets) in distributed-memory machines. PC++ extends C++ by introducing the notion of a *collection* which is a homogeneous group of elements. If `coll` is a pointer to a collection of elements (of class `E`) and `r` is a routine in class `E`, then `coll`$\rightarrow$`r` logically calls `r` on all the elements in parallel. Effectively, each routine call on a collection object is a data-parallel operation with an implicit barrier synchronization for all the logical threads.

In addition to supporting a data-parallel model of computation with object, PC++ has constructs to partition a collection among the processors. The distribution construct is similar to the alignment constructs in Fortran D [99] and High Performance Fortran [98]. For example, the declaration:

```
DistributedList<element> G([MAXPROC], [M], [Block]);
```

allocates a collection with `M` elements and distributes them in a block fashion among a logical 1-D array of `MAXPROC` processors. Thus processor 0 gets the 0,. . .,(M/MAXPROC)-1 elements, processor 1 gets the (M/MAXPROC),. . .,2*(M/MAXPROC)-1 elements etc. PC++ builds irregular data structures (e.g. binary tree) on top of C static arrays and thus the data structures are not dynamic.

An Actor Language

ABCL/1

ABCL/1 [231, 232] is an actor language. Every object (actor) in ABCL/1 has its own thread of control, and is always in one of three modes: dormant, active or waiting. An actor starts out as dormant; it becomes active when it receives a message and executes the routine corresponding to the message. After an active actor finishes executing a routine and there is no more message, it returns to the dormant mode. An active actor may be suspended during the execution of a routine because it expects to receive a specific message. It then goes into a waiting mode. A waiting actor is resumed when the expected message arrives.

Synchronization is achieved by message-passing among the actors. There is a `select`-construct which, when executed, changes an object's mode from active to waiting. This construct specifies the patterns and constraints of messages that will reactivate the object, and is similar to the guarded alternatives in CSP,

Although ABCL/1 does not have a machine model with distributed memory, the language has been implemented on distributed memory machines [233]. ABCL/1 does not have class inheritance, but supports delegation using a form of continuation-passing [15]. The "continuation-passing" works as follows. When an object O1 receives a message, it also gets an object R (called the *reply destination*) to which O1 sends its reply. If O1 is unable to service the message, it can delegate the message (plus the reply destination R) to another object O2. If O2 is again unable to service the message, the message can be delegated again to a third object, etc. Eventually when the message is serviced, the reply is sent back to R directly.

Languages for Distributed Memory

Emerald

Emerald [146, 40] is an object-based system for writing distributed programs. The main targeted systems are networks with about 100 nodes. Emerald is object-based, rather than object-oriented. There is no mechanism to define a class which will serve as the definition for multiple object instances. There is instead an *object constructor* expression which can include attribute and routine definitions. Because Emerald does not have any inheritance, the inheritance anomaly problem does not arise.

Object mobility is one of its design goals and affects the language design (e.g. parameter passing). Object location and mobility are part of the language semantics.

Emerald supports a shared object space. There are process objects and passive objects. A process object is not an actor; its creation simply results in a new thread that can invoke routines on other objects. For synchronization, an object constructor can include a monitor keyword to specify that at most one thread can execute in the object. The references [146, 40] however do not describe how more complex synchronization such as barrier and condition-wait can be provided.

Because object mobility is a central concept in Emerald, we will describe it in more details. Emerald distinguishes between objects which can be moved (*global objects*) and those which cannot be moved (*local objects*), and implements them differently. A global object is not referenced directly by any program variable. Program variables point directly only to local objects or local *object descriptors*. An object descriptor either holds a pointer to a resident global object or a forwarding address which gives the processor on which the global object is resident. When an object is moved, its object descriptors on the source and destination processors are updated accordingly.

Emerald has operations to move an object from one processor to another (`move`), to locate where an object is (`locate`), to make an object immobile (`fix`) or mobile (`unfix`) and to move an object and make it immobile (`refix`). Node objects provide an abstraction to physical processors. A location is specified by a node object or any object. In the latter case, the location of the object gives the desired processor.

Programmers can use an `attached` keyword in variable declarations to specify objects to be attached to other objects. When the latter move, their attached objects automatically move with them (but not vice-versa).

When a routine is invoked on an object, it is executed at the object's processor. Therefore when an object moves, activation records of its routines have to be relocated as well. This means that the runtime system has to keep track of the activation records of every movable object; this might entail high runtime costs. Jul et al. [146] describes how to reduce such costs and what to update when activation records are moved.

Other Languages

Orca

Orca [21, 212] is targeted to deal with network latency in distributed systems. It provides a shared data-object model with passive objects and explicit threads (processes). Processes are created using the `fork` statement. Shared objects are explicitly passed to the new child processes via *shared* parameters, so there are no global objects.

Objects are implicitly protected and operations on them are therefore indivisible. Other than synchronizing and communicating via shared objects, Orca also provides guarded statements (like CSP). There is no inheritance mechanism so that considerations of inheritance anomaly do not arise.

Orca is implemented on top of a reliable broadcasting layer that supports serial updates to objects, but does not guarantee sequential consistency for performance reasons. Object placement is handled by the runtime system.

Concurrent Aggregates

Concurrent Aggregates (CA) [67, 68] adopts an actor model so that invocations are serialised. A main concern in its design is that this serialisation leads to bottlenecks and gets worse with more layers of abstractions. This leads to the idea of *aggregates* which is a homogeneous, distributed collection of active objects (referred to as *representatives*). This collection co-operates as a single entity to the client, but is able to receive more than one message, because a message can be handled by any representative.

Message-passing in CA is synchronous and concurrent execution is specified explicitly using the conc- and forall-constructs:

```
(conc <expr>+)
(forall <varname> from <expr0> below <expr1>
<expr>+)
```

CA also supports continuations as first-class objects; this is used to implement various forms of synchronizations, message reordering, asynchronous reply and data-parallel operations like the `ParallelCollection` in Multiprocessor Smalltalk. Object placement is not visible at the language level. Since CA supports delegation instead of inheritance, it is not necessary to handle any inheritance anomaly.

The interesting aspect of aggregates is that an aggregate has a unique identity to the client but it internally consists of more than one object instance. At one point in the design of language support for distributed data structures, pSather had an analogous notion called replicated objects but this was dropped in favor for a `SPREAD{T}` class. The tradeoffs between replicated objects and `SPREAD{T}` are discussed in Section 2.5.3.

2.3 PSather – Control-Parallel Constructs

Now that we have seen where pSather lies in the design space of parallel object-oriented languages, this and the next three sections will describe pSather's parallel extensions in detail.[12] The extensions are based on an older version of Sather (Section 1.1). In Section 2.7.1, we will describe how the parallel constructs become even more "powerful" in pSather 1.0.

In this section, we start with a simplified machine model with uniform shared memory. This allows us to focus on the language constructs that provide threads and synchronization. Then

[12] This is joint work with Jerry Feldman, Franco Mazzanti and Stephan Murer, also described in [97] (pSather 0.1) and [181] (pSather 1.0).

in Section 2.4, we generalize the machine model to a *cluster* model that is applicable to both shared and distributed memory machines. Section 2.5 describes language constructs for programming in the cluster model's non-uniform shared address space. Last but not least, in Section 2.6, we look at how pSather supports a high-level form of single-program multiple data (SPMD) data-parallelism. PSather's data-parallel constructs are unique because they are designed to be used for both *dynamic* regular and irregular data structures, and in a distributed memory environment.

The central synchronization construct is based on two predefined `GATE`[13] classes with a set of predefined attributes and routines. These two reference classes — `GATE{T}` and `GATE0`, have the same basic functionalities. They do not alter the syntax or semantics of Sather 0.1's existing type system, and can be used like other library classes.

The `GATE` classes use a relatively small number of routines to support a variety of synchronization functions, including locking, conditional wait and atomic access. `GATE` is therefore a higher level synchronization mechanism than constructs like spinlocks, semaphores and critical sections. There are several reasons for this approach.

First the `GATE` classes fit naturally into Sather's type system. We avoid a proliferation of predefined classes, by combining the functionalities in the `GATE` classes. Higher level synchronization constructs also helps to make parallel programming safer. For example a common problem in the use of semaphores is deadlock caused by releasing semaphores in the wrong order. PSather prevents this problem by having a `lock` statement (Section 2.3.4) that locks multiple gates atomically. A third reason is that programmers often combine low level mechanisms to get high level synchronizations (e.g. using semaphores to build barriers). Sometimes the low level mechanisms may not be at the right level of abstraction. For example, if only condition variables [154] and locks are provided, the programmer will find it difficult to atomically wait on a conditional variable and release a lock.

The main disadvantage of the integrated functionalities in `GATE` is that some efficiency may be lost. A program may also become less clear because the declaration of a gate object does not immediately tell us whether the gate is used as a lock, a barrier or something else.

There were many other design alternatives which we have considered. Discussions of the design process and rationale is given in [97].

There are two ways to use a `GATE` (ie. `GATE{T}` or `GATE0`) class. A class can be a descendent of a `GATE` class and inherit the predefined routines. Another way is to be a client of `GATE` classes, ie. a class contains a gate in one of its attributes for synchronization purposes. From now on, we will use the term "`GATE` class(es)" to refer to either of the two classes or any of their descendents. Also any description of a `GATE{T}` (or `GATE0`) also applies to its descendents. A *gate object* or more simply *gate* is an object whose type is a `GATE` class.

We first describe the attributes and functionalities of the `GATE{T}` class in Sections 2.3.1

[13] The previous name for this construct is `MONITOR`. We abandonned "`MONITOR`" because it is associated with well-known constructs in Mesa, Concurrent Pascal etc. Since our construct is quite different, using "`GATE`" avoids any preconceived misconceptions. A unique term also allows us to discuss more clearly the differences and similarities of our construct with other synchronization constructs, such as M-structures [24, 25] and monitors [136, 227].

to 2.3.4. Then in Section 2.3.5 we point out the (minor) differences between `GATE{T}` and `GATE0`. To illustrate the usefulness of the `GATE` class, in Section 2.3.6 we show how it can be used to implement the synchronization mechanisms proposed in other systems.

2.3.1 Predefined GATE{T} Class

The abstract view of a gate is that it contains the following unnamed attributes:

- a *lock* counter,
- a queue of values whose types must all conform to `T` (the type parameter of `GATE{T}` class), and
- a set of associated threads.

Sometimes we are only interested to know whether the lock counter is 0 or non-zero. Therefore, we say that a gate is *unlocked* (*locked*) when its lock counter is 0 (non-zero). Also, to make it simpler to say whether the queue of values is empty or not, we treat a gate as having a *bind* status which is set to *bound* (*unbound*) when the queue of values is non-empty (empty).

Testing GATE's Internal State

A `GATE{T}` class has four predicates — `is_bound`, `is_unbound`, `has_thread`, `no_threads` — that test a gate's internal state. When there is an empty queue of values, the gate is considered to be in an unbound status and the predicate `is_unbound` returns true. When the queue of values is non-empty, the gate is in a bound status and the predicate `is_unbound` returns false. The `is_bound` predicate is simply the negation of `is_unbound`.

The predicate `has_thread` returns true when the set of associated threads[14] is non-empty; otherwise, it returns false. The predicate `no_threads` is the negation of `has_thread`.

The predicates to test a gate's lock counter are not predefined because they can be implemented on top of a locking statement (Section 2.3.4).

All the predicates only test for the properties at a certain instant in time. There is no guarantee that the property will continue to hold after the predicate returns. To get such a guarantee, a thread can lock and test the gate atomically by using the locking statements with any of the four predefined predicates (Section 2.3.4).

2.3.2 Modifying GATE{T}'s Internal State

The unnamed attributes of a gate are not directly modifiable by the programmer. Instead the programmer acts on these attributes by invoking one of the routines in the `GATE{T}` class's

[14] A thread is associated with a `GATE` object using the deferred assignment statement described in Section 2.3.3.

interface, or using a gate in a predefined language construct (Sections 2.3.3, 2.3.4). The semantics of these routines and predefined constructs are fixed for `GATE{T}` and its descendent classes.

`GATE`'s Lock Status

There are two locking statements (Section 2.3.4) that can increment a gate's lock counter and an `unlock` statement that can decrement a gate's lock counter. A locked gate also records the id of the *locking thread* (ie. the thread that executed the locking statement). After a gate becomes locked (ie. its lock counter is > 0), only the locking thread can further increment its counter.

`GATE`'s Queue of Values

The queue of values may be modified using one of the four routines in a `GATE{T}`'s interface — `set`, `enqueue`, `read` and `take`.

```
class GATE{T} is
  set(v:T) is ...;
  enqueue(v:T) is ...;
  read:T is ...;
  take:T is ...;
end;
```

All four operations modify the queue (of values) atomically. Any one of these operations can only proceed if and only if the gate is unlocked or locked by the executing thread; otherwise, the executing thread is suspended until the gate becomes available.

The `set(v:T)` operation accepts a value `v` (whose type conforms to `T`, the type parameter of the `GATE{T}` class) and places it at the front of the queue (replacing any value if the queue is non-empty). The `enqueue(v:T)` operation inserts `v` at the end of the queue.

To retrieve values from the queue the `read` and `take` operations are used. An executing thread performing the `read` operation is suspended if the gate has an empty queue. When the gate's queue becomes non-empty, `read` returns (but does not remove) the value from the front of the queue. The `take` operation also suspends the executing thread if the gate's queue is empty. When the gate's queue becomes non-empty, the `take` operation removes and returns the value from the front of the queue.

`GATE`'s Set of Associated Threads

The last internal state of a gate is a set of associated threads. When a thread is forked in the program, it may be associated with a gate. The gate will hold the result of the thread when it terminates. The creation of threads is described in more details in Section 2.3.3.

2.3.3 Thread Creation and `GATE{T}`

We mentioned that a gate has a set of associated threads. There is a *deferred assignment statement* which allows the programmer to associate a thread with a gate:

```
<gate-expr> :- <routine-call>;
```

This statement works as follows. First the left-hand-side expression `<gate-expr>` is evaluated. The result must be a non-void gate.[15] The right-hand-side expression `<routine-call>` must be a routine call which may use a dispatched or non-dispatched variable:

```
g:GATE{$POLYGON}; ...
a:SQUARE;
g :- a.compute_area;
b:$POLYGON;
g :- b.compute_area;
```

If `<gate-expr>` evaluates to a `GATE{T}` object, then the return type of the right-hand-side routine call must conform to `T`. Examples of illegal right-hand-side expressions include local variables, parameters, reading/writing object attributes, reading/writing shared and constant features.

The object (if any) and argument values in `<routine-call>` are evaluated left to right.[16] Any embedded routine calls are evaluated as well. Finally if the gate is unlocked or locked by the executing thread, a new thread is created to execute the specified routine. Otherwise, the only other possibility is that the gate is locked by some other thread, in which case the executing thread is suspended until the gate becomes unlocked.

We can view the deferred assignment statement as an *asynchronous* routine invocation while normal routine calls are *synchronous*. The forked thread is put into the set of threads associated with the gate (evaluated on the left-hand-side). Therefore, from the programmer's point of view, a thread is not a first-class object and cannot be directly manipulated. When the thread terminates, the result is enqueued in the queue of values and the thread is removed from the set of associated threads.

This association of a forked thread to a gate allows us to use gates as *future*s [119].

```
g:GATE{INT} := GATE{INT}::new;
-- Create a gate with queue of INT's.
...
g :- compute;
...
result := g.read;
...
```

[15] A runtime error due to a void gate can be caught if the program has been compiled with a *runtime-check* option. Without the runtime-check option, the program behavior is undefined. When runtime-check is off, we can imagine an exception being raised when a runtime error occurs. But since exception handling is not well-defined in pSather, so we will just leave the behavior as undefined, when runtime-check is off.

[16] The evaluation order is the same as that defined in Sather 1.0.

The statement "`g :- compute;`" creates a new thread to do some computation. Since the current thread does not need the result immediately, it can continue to execute. It is suspended only if the result is needed (`g.read`) and the other thread has not finished its computation. The gate `g` acts as a first-class future object and can be passed as an argument or stored in other objects. The differences between using a gate as a future and a Multilisp *future* are as follows.

- In pSather, a value is explicitly retrieved from a gate using the `read` or `take` routines. A Multilisp future value does not need any explicit `read` operation; when we try to read the value of variable and it is not yet available, the executing thread is suspended.
- A gate can queue return values from more than one thread, while a Multilisp future only holds one return value from a parallel thread.

The association of a thread with a gate is however not mandatory. A thread can be forked without being associated with any gate.

```
:- <routine-call>;
```

A gate `g` can also be dissociated from its threads by `g.clear`. The result of a dissociated thread is not enqueued in `g`. A thread can call a boolean function `CONFIG::clear_request` to check whether it has been dissociated from its gate. The `clear` operation provides an indirect way for threads to self-abort when they find that their results are no longer needed. An example is when writing search programs with alternative competitive strategies.

```
g :- perform(strategy1);
g :- perform(strategy2);
g :- perform(strategy3);
result := g.take;
g.clear;
```

When a thread succeeds, the result is enqueued in `g`. After retrieving the result, `g.clear` is called. A thread that finds that `CONFIG::clear_request` is "true" can then self-abort.[17]

We have so far only assumed a uniform shared memory model. In such a model, various runtime systems [215, 33] have demonstrated the feasibility of good load balancing techniques. Therefore the deferred assignment statement only provides the functionality of thread creation and leaves load balancing to the runtime system. But when we generalize the model to a more sophisticated one (Section 2.4.3) in which there is no effective runtime load balancing strategy, we also extend the deferred assignment statement such that a programmer can specify where a new thread executes and do explicit load balancing. Section 2.7.1 describes the changes in pSather 1.0 that affects the deferred assignment statement.

[17] There is no mechanism in the language for a thread to kill another thread.

`cobegin-end` Statement

One common way of writing parallel programs is to fork a number of threads for one phase of the computation, wait for them to finish and fork another set of thread for the next phase, and so on. PSather therefore provides a `cobegin-end` statement to support this common form of synchronization.

```
cobegin
  <statements>
end;
```

A thread `T0` that executes this statement will suspend until all the descendent threads created during the execution of `<statements>` have terminated. Then `T0` continues on with the next statement. Note that the descendent threads of `T0` are defined in a transitive manner. A thread `T1` is a descendent thread of `T0` if (1) `T1` is created when `T0` executes a deferred assignment statement, or (2) there exists a thread `T'` such that `T'` is a descendent of `T0` and `T1` is a descendent of `T'`.

Because `cobegin-end` statements can be nested, we also define the *(dynamic) scope* of a `cobegin-end` statement. Suppose a thread executes:

```
cobegin -- S1
  :- f;
  cobegin -- S2
    :- g;
  end; end;
```

In the example, the thread for `f` is considered to be in S_1's dynamic scope, while that for `g` is in S_2's dynamic scope. (To be more precise, the dynamic scope is associated with the execution instance of the `cobegin-end` statement, not with its syntactic occurrence.) Any thread created during `f`'s execution is also considered to be in S_1's dynamic scope (unless `f` executes a `cobegin-end`). We note that although `g`'s thread is created within S_1, S_1 need not wait for `g`'s thread to terminate because synchronization with S_2 guarantees that `g`'s thread must have terminated.

When a `return` statement is executed, nested within K `cobegin-end` statements, we do not exit the routine until all K `cobegin-end` synchronizations have completed. Similarly, suppose a `break` statement is to transfer control out of L `cobegin-end` statements which are contained in a loop:

```
loop
  ...
  cobegin ...cobegin ...
    ...break; ...
  end; ...end;
end;
```

the control is transferred out of the loop only after completion of all L `cobegin-end` synchronizations.

To summarize, the `cobegin-end` statement performs a barrier on all the threads created directly or indirectly (ie. all descendent threads) within the dynamic scope of its body.

2.3.4 `lock`, `try` and `unlock` Statements

There are three statements which modify a gate's internal lock counter. The first is a `lock` statement with blocking semantics.

```
lock <expression-list> then
  <statement-list-1>
end;
```

The second is a non-blocking `try` statement.

```
try <expression-list> then
  <statement-list-1>
[else <statement-list-2>]
end;
```

We refer to them as *locking statements.* The third is an `unlock` statement that is the inverse of the *locking statements.*

The `<expression-list>` in the `lock` statement is a list of expressions, each of which either evaluates to a non-void gate or is of the form `<gate-expr>.<gate-predicate>`. In the latter case `<gate-expr>` evaluates to a non-void gate and the predicate is one of the four predefined predicates described in Section 2.3.1. A runtime error occurs if any of the evaluated gate is void (and this can be caught with the runtime-check option).

The executing thread can lock all the specified gates only when the following conditions hold.

- Every gate is lockable, ie. its lock counter is 0 or it is already locked by the executing thread.
- If the expression is of the form `<gate-expr>.<gate-predicate>`, the gate must also satisfy the predicate.

As long as any of the above criteria does not hold, the thread remains suspended and all the gates' lock counters remain unchanged. When all the gates satisfy the conditions for locking, their lock counters are incremented by 1. The increments appear as an atomic transaction to the programmer. Then `<statement-list-1>` is executed, at the end of which the gates' lock counters are decremented by 1. Thus the status of the gates are restored to that just before the `lock` statement was executed.

In the `try` statement, the gates must satisfy the same conditions in order for the lock counters to be incremented. The difference is that when any of the criteria does not hold, the thread is not suspended. Instead the thread continues to execute `<statement-list-2>` (if it is present) or the statements after the try-statement (if `<statement-list-2>` is not present). The status of the gates remain unchanged. If all the gates satisfy their required conditions, they are locked in a manner which appears atomic to the programmer. The statements in `<statement-list-1>` are executed after which all the gates are restored to their original status.

In both locking statements, every expression in <`expression-list`> that evaluates to a gate is computed exactly once.

The `unlock` statement has the form:

```
unlock <gate-expr>;
```

where `<gate-expr>` evaluates to a non-void gate object.[18] An `unlock` statement must be syntactically enclosed in a `lock` statement or the `then` branch of a `try` statement. It must not occur within a `loop` statement unless it has a *syntactically enclosing locking statement* which is in the `loop` statement. The following therefore constitutes a compile-time error.

```
until test loop
  unlock g;
end;
```

But this next piece of code is acceptable.

```
until test loop
  lock g then
    unlock g;
  end;
end;
```

The `unlock` statement also dictates that the gate object G evaluated by `<gate-expr>` must be one of the gates locked by a syntactically enclosing locking statement. This latter condition is checked during run-time and the implementation is described in Section 3.4.2.

Intuitively, what we want is for an `unlock` statement to perform early unlocking, ie. to decrement a gate's lock counter by 1 before execution reaches the end of its locking statement. So if G is locked by two or more syntactically enclosing locking statements, we want the `unlock` statement to undo the effect of the *innermost* one. This condition is also checked during run-time (Section 3.4.2).

An important aspect of locking in pSather is that each gate has a lock counter and a thread id, to count how many times it is locked by the thread. It is possible for a thread to lock a gate more than once, whether with a statically nested locking statement or one found in some later routine calls. The lockable-multiple-times and locked-by-thread properties are important because it prevents deadlocks in many common situations.

It is common for an object to invoke a routine on itself:

```
class A is
  r0 is
    lock g then ...end;

  r1 is
    lock g then ...; r0; ...end;
  end;
```

[18] A runtime error occurs if the evaluated gate is void.

```
is_locked:BOOL is
  -- Returns "true" if the gate is locked.
  try self then
    res := false;
  else
    res := true;
  end;
end;
```

Figure 2.5: Definition of `is_locked` routine in `GATE{T}` or `GATE0` classes.

```
diner(id:INT) is
  until done loop
    lock chopstick[left(id)], chopstick[right(id)] then
      -- Grab chopsticks.
      eat(id);
    end;
    think(id);
  end;
end;
```

Figure 2.6: A "dining philosopher" example to illustrate usage of `lock` statement.

This could cause deadlock if a gate were not locked with respect to a particular thread. For example, when `r0` is invoked from within `r1`, `r0`'s `lock` statement would be suspended and never resumed. A similar problem would occur if a thread could not lock a gate multiple times.

Using `lock`, `try` and `unlock` Statements

In Section 2.3.1, we mentioned that there are no predefined predicates to test a gate's lock counter. This is because such predicates can be defined using the `try` statement (e.g. the `is_locked` predicate in Figure 2.5). The predicate `is_locked` returns true when the gate is in a locked status; otherwise it returns false when the gate is in an unlocked status.

Figure 2.6 shows how to use a `lock` statement in the solution of a "dining philosophers" program. In the code, a philosopher (thread) can grab two chopsticks atomically using multiple-locking facility in the `lock` statement. The statement semantics prevent any deadlock, which might otherwise result if the programmer uses a nested locking schema instead [97].

Another use of the `lock` statement is illustrated by a `join` routine (Figure 2.7) defined within the `GATE` classes. To express a fork-join computation, forked threads are attached to a gate `g`. When the parent thread is ready to wait for the termination of child threads, it calls "`g.join`". This suspends the parent thread until all the forked threads have terminated (and the `no_threads` predicate becomes true). The use of `g.join` however is less attractive than the

```
join is
  -- Wait till is no thread on the gate.
  lock self.no_threads then end;
end; -- join


g :- worker1;
g :- worker2;
...
g.join;
```

Figure 2.7: Using the lock-statement, we can define a `join` routine which allows us to express fork-join computations.

```
lock g0 then
  hash_table0.insert(key, entry);
  lock g1 then
    unlock g0;
    hash_table1.insert(key, entry); ...
```

Figure 2.8: A use of the `unlock` statement.

`cobegin-end` statement (Section 2.3.3) because it may require the synchronizing gate to be passed to a deferred assignment statement in some other contexts, possibly making the code less modular.

Next we show the usefulness of the `unlock` statement. Suppose a data structure consists of two duplicated hash tables (e.g. one per processor to improve data locality) which must have consistent entries. We can lock both hash tables using one gate, but this lock granularity may be too coarse. If we use one gate per hash table, it is incorrect to write the code as follows:

```
lock g0 then hash_table0.insert(key, entry) end;
lock g1 then hash_table1.insert(key, entry) end;
```

because a race condition may result (Figure 2.9). Suppose the hash table replaces an old entry by a new one (with the new key). After the race, table 0 contains entry (`key0`, `entry0`) while table 1 contains entry (`key0`, `entry1`). The tables become inconsistent because thread T2 is able to overtake

T1	T2
hash_table0.insert(key0, entry1);	
	hash_table0.insert(key0, entry0);
	hash_table1.insert(key0, entry0);
hash_table1.insert(key0, entry1);	

Figure 2.9: Race condition in a duplicated hash table.

T1 when accessing table 1. Figure 2.8 shows how the `unlock` statement prevents this overtaking. The gate `g1` is locked before explicitly unlocking `g0` using the `unlock` statement, thus ensuring that if a thread T' accesses table 0 before T", T' will also access table 1 before T".

Semantics of `lock` Statement for Multiple Gates

In this section, we define more precisely the semantics when there are more than one gate in the `lock` statement. The semantics of the `lock` statement involves tradeoffs among the following criteria:

- Deadlock-freedom.
- Maximal concurrency.
- Efficiency.
- Fairness.

We will discuss their implications on implementation in Section 3.4.1; here we describe how they affect the language semantics.

Freedom from Deadlock

PSather does not prevent the user from writing code which may result in deadlock. Consider the examples in Figure 2.10. In Figure 2.10 (a), a deadlock may occur because pSather does not restrict nested locking statements. Even restricting nested locking statements statically is not sufficient, as illustrated by Figure 2.10 (b). In a sense, (b) is an example of dynamic nested locking statements. Furthermore, as illustrated in Figure 2.10 (c), the ability to specify locking a gate based on its state may also result in deadlock even if there is no nested (static or dynamic) locking-statements.

In spite of the possibility of deadlock due to programmer error, the semantics of the `lock` statement guarantees that no deadlock will occur due to two or more `lock` statements with multiple gates such as:

```
lock g1, g2, g3 then ...end; -- Thread 1

lock g3, g2, g1 then ...end; -- Thread 2
```

The operation of `lock` statement guarantees that either all the gates are locked atomically or none at all.

Maximal Concurrency

The all-or-none semantics also allows as many threads as possible to proceed, at any time. Suppose we have:

```
[Thread 1]                      [Thread 2]

lock g1 then ...                lock g2 then ...
   lock g2 then                    lock g1 then
      ...                             ...
   end;                            end;
end;                            end;
```

a. Nested `lock` statements.

```
[Thread 1]                      [Thread 2]

lock g1 then ...                lock g2 then ...
   f1;                             f2;
   ...                             ...
end;                            end;

f1 is                           f2 is
   lock g2 then                    lock g1 then
      ...                             ...
   end;                            end;
end;                            end;
```

b. "Dynamically-nested" `lock` statements.

```
[Thread 1]                      [Thread 2]

lock g1.is_bound then ...       lock g2.is_bound then ...
   g2.set;                         g1.set;
   ...                             ...
end;                            end;
```

c. Locking with condition on `GATE` state.

Figure 2.10: (a), (b) and (c) illustrate possible reasons for deadlock.

```
lock g1 then ...end; -- Thread 1

lock g2 then ...end; -- Thread 2

lock g1, g2 then ...end; -- Thread 3
```

and thread 2 has acquired `g2`. When thread 3 tries to lock both `g1` and `g2`, it cannot succeed. However, if thread 3 keeps `g1` locked while waiting for `g2`, then thread 1 is prevented from locking `g1`. As a result, only thread 2 is executing, even though both threads 1 and 2 could be executing. To allow for maximal concurrency, the `lock` statement guarantees that a thread that is waiting to lock one or more gates does not have priority over later threads trying to lock the same gate(s).

Efficiency

A straight-forward implementation of the all-or-none locking semantics releases any locked gates if not all gates can be locked atomically. This assumes that other threads may need the locked gates – a conservative assumption. But if a thread assumes optimistically that the locked gates will not be needed by others, it can avoid unlocking them and reduce the costs of multiple-locking. Section 3.4.1 describes a protocol for doing this on shared-memory machines.

On distributed-memory machines, the costs of unlocking gates due to an unsuccessful `lock` statement may be compounded by message latency. But since most uses of `lock` statement have only one gate and this is a special case which can be optimized, we retain the all-or-none semantics. This ensures freedom from deadlock and maximal concurrency, which we think, are more important criteria than *possible* performance degradation[19] (which may actually be insignificant).

Fairness

The following code illustrates that fairness and maximal concurrency are conflicting objectives.

```
lock g1 then ...end; -- Thread 1

lock g2 then ...end; -- Thread 2

lock g1, g2 then ...end; -- Thread 3
```

Suppose each thread repeatedly executes its `lock` statement. If we want maximal concurrency, thread 3 may be prevented from entering its critical section by threads 1 and 2. On the other hand, if we want fairness, then concurrency must necessarily be reduced, because at some point, thread 1 (or 2) has to refrain from locking and give precedence to thread 3 even though it may actually proceed. We choose maximal concurrency over fairness in this tradeoff.

To summarize, the two major properties of the `lock` statement are as follows.

[19] We stress the word "possible" because the amount of performance degradation ultimately depends on how often multiple-locking is used and the actual contention for gates.

- Either all the gates are locked or none at all.
- A thread that is waiting to lock one or more gates does not have priority over later threads trying to lock the same gate(s).

2.3.5 Difference Between GATE{T} and GATE0

Sections 2.3.1 – 2.3.4 describe how the **GATE{T}** class is used. In part of the description, the type parameter **T** does not play any role (e.g. locking/unlocking) and a **GATE0** object works in the same way. So the **GATE0** class is almost similar to the **GATE{T}** class except for the following differences.

From the programmer's point of view, the main difference is that while a **GATE{T}** object has a queue of values of type **T**, a **GATE0** object only has a queue counter without any values. The bind status of a **GATE0** object is bound (unbound) when the queue counter is greater than (equal to) 0.

For any operation `op` that operates on the queue of a **GATE{T}** object, when `op` inserts (deletes) a value into (from) the queue of a **GATE{T}** class, the corresponding effect of `op` on a **GATE0** is to increment (decrement) the queue counter. The interface of the routines **set**, **enqueue**, **read** and **take** in the **GATE0** class is given by:

```
class GATE0 is
  set is ...;
  enqueue is ...;
  read is ...;
  take is ...;
end;
```

Unlike **GATE{T}**, the **set** and **enqueue** routines do not take any argument while the **read** and **take** routines do not have any result type. These routines are normally invoked on a **GATE0** object to get the synchronization effects.

A thread executing any of the four operations is allowed to proceed if and only if the gate is unlocked or locked by the executing thread. Otherwise, the thread is suspended until the gate is unlocked. **set** atomically increments the queue counter to 1 if it is 0; otherwise, it leaves the queue counter unchanged. **enqueue** always increments the queue counter by 1. A thread performing **read** or **take** is suspended when the gate's queue counter is 0. It is resumed when the counter becomes non-zero (e.g. via a **set** or **enqueue** operation by another thread). **read** does not change the queue counter; **take** decrements the queue counter by 1.

In addition to the four predefined operations, the deferred assignment statement may also modify the queue counter of a **GATE0** object. We have noted that if the left-hand-side of a deferred-assignment statement evaluates to a **GATE{T}** object, the return type of the right-hand-side expression must conform to **T**. This restriction on the return type and a gate's type parameter **T** does not apply if the left-hand-side of a deferred assignment statement evaluates to a GATE0 object. The right-hand-side expression may or may not have any return value. If there is a return value, it is discarded when the thread completes.

2.3.6 Examples

In this section, we consider several more complex examples which use gates. Some of these synchronization paradigms will arise later in our code (Chapter 4).

If we consider the three synchronizations (lock, barrier, condition-wait) discussed in Section 2.2, it is obvious that a gate can be used as a blocking lock or a condition variable directly. To treat a gate as a condition variable, a thread waiting on a condition performs a `read` or `take` operation on a gate. When the condition is true, the gate's bind status is set to bound; this resumes any suspended thread.

Barrier

To implement a barrier, a BARRIER class can be defined using gates (Figure 2.11). The `create` routine takes a parameter `n` and allocates a barrier object that synchronizes n threads. There are four attributes in the barrier. `level` is the number of threads that must reach the barrier before they are allowed to proceed beyond the barrier. The `counter` attribute keeps track of how many threads have reached the barrier and is kept in a gate. The atomic `take` and `set` operations (called in the `hit` routine) ensure that the counter is incremented atomically. The `all_done` attribute contains a signalling gate which acts as a condition variable; it is `set` when the last thread arrives at the barrier, thus resuming threads which arrived earlier and were suspended on it. A thread arriving at the barrier calls the `hit` routine. If it is the last arrival (i.e. the test "`c = level`" is true), then it sets the gate in `all_done`, and reset the counter to 0. `all_done` gets a new gate for the next round of threads. When the thread is not the last one, it simply sets the new counter value and waits for the signalling gate to be `set`.

Signal Arrival of Value

A GATE0 object can be used as a condition variable, by setting the bind status to bound when the condition is satisfied. The queue in GATE{T} further allows the programmer to associate a value with the satisfied condition. For example, if we have a binary tree whose nodes are of type NODE and each node has a `local_field` attribute. Suppose each node can compute its local field almost independently and the only dependency is that a child needs its parent's field to complete its calculation. To signal the readiness of the parent's field, a node stores its field in a GATE{T}, thus providing a signal to its children that the value is ready. Figure 2.12 shows a way to write the code. The thread for each node computes the partial field and then wait for the parent's result. Its

```
class BARRIER is
  -- Supports barrier function for 'n' threads.
  private counter:GATE{INT};
  private level:INT;
  private all_done:GATE0;

  create(n:INT):SELF_TYPE is
    -- A barrier object for 'n' threads.
    res:=new;
    res.counter:=GATE{INT}::new
    res.counter.set(0);
    res.level:=n;
    res.all_done:=GATE0::new;
  end; -- create

  hit is
    -- Thread is suspend until all other threads have hit the
    -- barrier.
    wait_needed:BOOL := true;
    c:INT := counter.take;
    broadcast:GATE0 := all_done;
    c := c + 1;
    if c = level then
      broadcast.set; -- Resume all previous hits.
      all_done := GATE0::new; -- Get another gate.
      counter.set(0); -- Reset counter for next round.
      wait_needed := false; -- This call does not need to wait.
    else
      counter.set(c);
    end;
    if wait_needed then
      broadcast.read; -- Wait until a 'set' is done.
    end;
  end; -- hit

end; -- class BARRIER
```

Figure 2.11: Definition of a simple BARRIER class.

```
compute_partial_field;
parent_field := parent.local_field.read;
local_field.set(f(partial_field, parent_field));
```

Figure 2.12: Using GATE{T} to signal when value is ready.

```
get(k:INT):T is
  lock gate then
    gate.enqueue; -- Increment number of readers.
  end;
  <Perform the actual operation>
  gate.take; -- Decrease counter before exiting.
end;


double_size is
  lock gate.is_unbound then
    -- Unbound status guarantees no reader.
    <Perform the actual operation>
  end;
end;
```

Figure 2.13: Adding readers-writer synchronization for the routines of a class.

own result `f(partial_field, parent_field)` is stored in a local `GATE{T}` object. This signalling mechanism is used in an N-body application (Section 4.8).

Readers-Writer

Another common synchronization paradigm is to allow multiple threads to read a data structure, but only one thread to update (write) it. Suppose we convert a sequential hash table to be shared by multiple threads. Two of the routines in a hash table class are:

```
get(k:INT):T is ...end;
double_size is ...end;
```

The `get` routine returns the entry associated with key `k` in the table, while `double_size` allocates more space for the hash table. Since `get` only reads the internal attribute, multiple threads can execute `get` in the hash table. But a thread executing `double_size` must have exclusive access to the hash table. To achieve this synchronization, we use `GATE0` attribute which serves both as a lock and an implicit reader counter (Figure 2.13). The queue counter in this `GATE0` object keeps track of the number of readers (implicitly). The `get` routine first attempts to lock the gate; if there is a writer, this routine will have to wait. When locking succeeds, the `enqueue` operation records the presence of a reader thread and the gate is unlocked, allowing further readers. The `double_size` routine locks the table and waits for the no-readers condition atomically by using the `lock` statement together with the `is_unbound` predicate (which becomes true when there is no reader). A slight variation of this readers-writer synchronization is used to implement the local tables of a replicated hash table (Section 4.3).

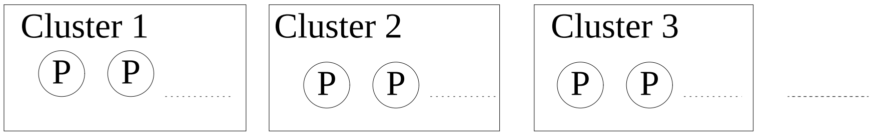


Figure 2.14: Abstract machine model in pSather.

2.4 PSather – Machine and Execution Model

In this section, we describe a high-level machine model which tries to balance the tradeoffs between architecture independence and accessibility to a machine's performance. We also give an execution model (Section 2.4.3) for parallel object-oriented programming. This model (adopted by pSather) is fairly intuitive and allows for straight-forward and efficient implementation. This section also introduces a predefined `where` method that gives an object's location and a placement (or `@`) operator that controls where a thread executes.

2.4.1 Machine Model

We introduce an abstract *cluster* model (Figure 2.14) that tries to achieve the following objectives.

- We want to the model to unify many multiprocessor and multicomputer configurations (e.g. shared-memory, distributed-memory machines) in a single framework.
- We want the long latencies of remote operations on large-scale multiprocessors to be made visible to the programmer, so that these costs can be taken into account when writing programs.
- We want pSather programs to continue to be compatible with the forseeable new multiprocessor architectures like Dash [161] and other cluster-based systems like Cray T3D.

The abstract machine model has P asynchronous processors grouped in C clusters. The model presents a shared logical address-space to the programmers, but divides the address space such that each cluster has its own physical address-space. Clusters may have one or more processors and each cluster may have a different number of processors. A processor belongs to exactly one cluster. If two processors p and q are in the same cluster, p is said to be *near* to q; otherwise, p is *remote* with respect to q. Processors and memory locations belonging to the same cluster are *near* to each other. Consequently, when a thread running on a processor in cluster c accesses an address in c, it is considered a near memory access. Processors and memory locations belonging to different clusters are *far* (or remote) from each other. Far accesses are consistently less efficient than near accesses. Accessing the same far address twice does not cause the hardware to cache the far accesses. Similarly an object or a thread is remote if it is in a remote processor with respect to the executing thread.

An alternative way of looking at clusters is to partition the set of processors into equivalence sets as follows. We first define the *address set* of a processor p, address-set(p) such that an address $x \in$ address-set(p) iff access latency for x approaches optimal hardware limit when p makes a sufficient number of accesses to x. We assume that for any two processors p, q, their address sets are either equal or disjoint. This is justified from our observations of current and foreseeable machine configurations. Two processors p, q are in the same cluster iff address-set(p) = address-set(q).

This model unifies many multiprocessor configurations. In one extreme, for shared-memory machines (e.g. Sequent Symmetry), there is only one cluster; at the other extreme, for distributed-memory machines (e.g. CM-5), each cluster consists of only one processor. As an example of multiple processors per cluster, Cray T3D will have two processors in a cluster. The distinction between near and far addresses allows a programmer to take into account the *(cluster) locations* of a thread and the objects that it use.

The concept of clusters provide a high-level view of the NUMA characteristic of distributed-memory machines. It serves the basis of an architecture-independent programming model and at the same time, allows a programmer to write a relatively efficient program (using the language constructs designed with this model in mind). This concept plays a role in the execution model to be described next, and in other aspects of the language (e.g. placement of objects in Section 2.5.2).

2.4.2 Programming Model

PSather adopts an MIMD programming model. Before the user program starts executing, the shared and constant features of all classes are initialized. In the cluster model, the shared and constant features are replicated on all clusters (Section 2.5.4). Thus, these initializations are executed (possibly in parallel) on all clusters. After the initializations are completed, a thread starts executing the `main` routine of a class, in cluster 0. This main thread may create more threads (e.g. using the deferred assignment statement) and may finish executing before its child threads. The completion of the main thread does not imply program termination. The program terminates only after all the threads in the system complete; the runtime support performs this termination detection.

2.4.3 Execution Model

We describe how the cluster model affects the programmer's view of objects and threads which are the fundamental entities in a parallel object-oriented language.

Objects

In the cluster model, an object has a *cluster location*[20] which is the cluster from which access to the object is most efficient. There are various categories of objects in a pSather program

[20]The word "location" will henceforth refer to the cluster location, unless specified otherwise.

and we consider the definition of *location* for each.

Basic/value objects are allocated when declared. Their locations are undefined and it is a compile-time error to compute the location of a value object. External (non-Sather) objects are managed by an environment outside of pSather. Their locations are undefined. Any information about the location of external objects is provided by an external interface which is not in the definition of pSather.

The location of a reference object `x` is given by `x.where`, and does not change during its lifetime. Reference objects can be allocated on a cluster specified explicitly by a programmer. This is done using an `@`-operator (e.g. "`x := ARRAY{INT}::new @ cluster_id`"), whose use will become clear when we later describe thread execution in the cluster model. A reference object exist as a single block within one cluster and does not have pieces distributed on different clusters. (Section 2.5.3 describes one class that does not follow this single-block rule.) We note that a reference object is accessible from any cluster, even though it resides in exactly one. The access latency however is non-uniform; it is fastest on the object's cluster and slowest from other clusters. To improve data locality, sometimes objects have to be copied or component objects of a data structure have to be allocated on different clusters. Section 2.5.2 describes the mechanisms to allocate objects and copy them to a specific cluster.

Relationship of Threads and Clusters

Section 2.3.3 describes how to fork off a new thread ("`:- <routine-call>`") to execute a routine, but where the new thread executes is determined by the runtime scheduler in a shared-memory model. The deferred assignment statement introduces the notion of user-level threads, but does not fully explain how threads and clusters interact in pSather. To do so, we introduce an additional idea — *subthread*. A subthread handles the flow of control during the execution of a routine in exactly one object. There is no limit to the number of subthreads executing in an object at the same time.

During the execution of a subthread, further invocations on `self` or other objects can be made. A routine invocation causes the current subthread to suspend and starts a new subthread to execute the invoked routine. When the new subthread eventually terminates, the suspended subthread is resumed. A subthread may be on a different cluster from the previous subthread, but like ordinary pSather objects, its location is fixed.

Note that because only the top subthread is active and can make progress, the subthreads of a thread happen in a last-in-first-out (LIFO) order. If we take a snapshot of a user thread at a particular instant, it is composed of a stack of subthreads and only the top one is active. Figure 2.15 shows two user level threads T1 and T2, each consisting of a stack of subthreads. For example, when S2 makes a routine call, it is suspended and a new active subthread S3 is pushed on top of the stack. When S3 terminates, it is popped off the stack, and S2 is resumed.

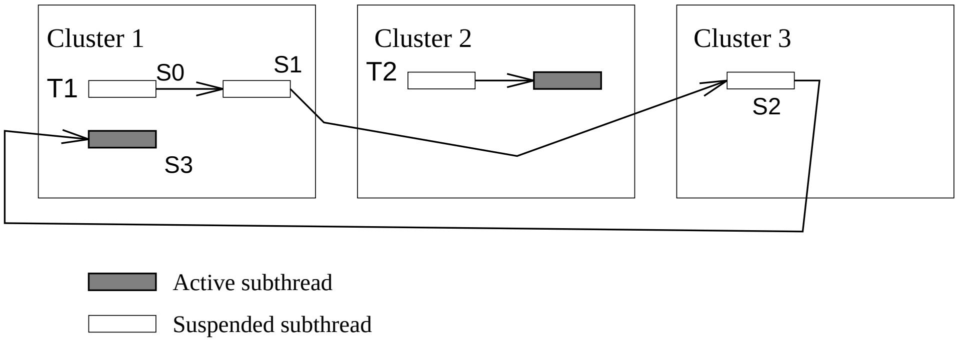


Figure 2.15: Execution of subthreads in pSather cluster model.

The sequential analog is a sequential thread and its activation frames. In fact, we view a subthread as an activation frame with a *cluster location*. Thus like objects, a subthread also has a location and this location is the cluster in which the subthread is executing. In this model, when we say "the current cluster of a thread", we are referring to the cluster of the currently active subthread.

A subthread's cluster location is unchanged during its execution. However, unlike objects, a subthread is accessible only from within its cluster. This means that any processor in a subthread's cluster may execute it, but no other processor may do so.

The location of subthread can be specified by using the `@`-operator.

```
hash_table.insert(e); -- (a.1)
hash_table.insert(e) @ cluster_id; -- (a.2)
```

In both (a.1) and (a.2), a subthread is created to execute the `insert` routine on a hash-table object. In (a.1), the subthread's location is the same as the current thread, so any processor within the current cluster can execute it. In (a.2), the subthread is located on `cluster_id` and can be executed by any processor in `cluster_id`.

A routine `r` may impose certain preconditions on the location of its subthread using other language constructs. Some examples are given in Section 2.5.

Because a thread is made up of many subthreads (possibly located on different clusters) during its lifetime, we cannot fix a thread's location. But when we view a thread as a stack of subthreads, the location of each subthread is determined by the code of the previous subthread, except the bottom-most. We therefore extend the deferred assignment statement to have an optional `@`-operator so that we can specify the cluster location of the initial subthread of the child threads.

```
:- hash_table.insert(e); -- (b.1)
:- hash_table.insert(e) @ cluster_id; -- (b.2)
```

In both (b.1) and (b.2), a user-level thread is created. The first subthread of this user-level thread executes the `insert` routine on a hash-table. In (b.1), the first subthread of the new thread is located in the current cluster and can be executed by any processor in the current cluster. In (b.2), the first subthread is located in `cluster_id` and any processor in `cluster_id` may execute it.

In all cases, an implementation may choose to schedule a subthread on any processor within the subthread's cluster, depending on the resource constraints.

Default Semantics When @-Operator Is Omitted

In the above discussion, we have informally defined the use of the @-operator in relation to the execution of a subthread. In particular, we have defined the absence of @-operator in both synchronous and asynchronous invocations to mean that the new subthread is located on the same cluster. The consequence is that the location of threads are independent of the objects' location. This is in contrast to the approaches in some languages, e.g. Emerald [146] always executes the routine at the object's processor, and Fortran D [132] is implemented using an owner computes rule so that a processor computes the results of its own array elements.

A possible alternative semantics of an invocation without @-operator would be that the routine is executed on the reference object's cluster. Implementation wise, what this means is that without the @-operator, every routine invocation (e.g. "`x.insert(e)`") has to perform an extra check on the location of the object:[21]

```
if (<x is in current cluster>) {
  <Execute "x.insert(e)" locally.>
else {
  <Make a remote call "x.insert(e)".>
}
```

Adopting this alternative default would therefore incur extra overhead. The current approach, on the other hand, means that only an invocation with @-operator has to be translated into a form with an extra check that decides between local or remote invocation. [22]

This definition also does not work for (1) value objects because their locations are undefined, (2) void objects and (3) external objects. To use an object's location as the default for deciding where to execute a routine, when not all objects have a precisely defined location, would introduce special cases and does not lead to a clean, well-understood definition.

Our current (albeit limited) experience (which we will describe in Chapter 4) also shows that most routine calls are local and do not use the @-operator. The intuitive reason is that an efficient class library will try to achieve good data locality by arranging for a data structure and most (if not all) of its associated objects to be in the same cluster. If a data structure and its component objects are on the same cluster most of the time, most of the internal routine calls will

[21] It does not matter whether this check is executed by the caller or in the `insert` routine.

[22] Of course, one might imagine an implementation which eliminates the check and allows the call to proceed ahead, invoking a trap handler to perform the correct remote call if the object is found to be remote. When such an implementation is possible, this reason against the alternative semantics for omitted @-operator becomes less compelling.

not need the @-operator. Therefore, ideally we would rarely need the @-operator to explicitly invoke a routine at a remote object's location.

Another reason for our design is that our target architectures are different from those of Emerald. We assume the remote access latencies will be within two orders of magnitude of local access time, while Emerald is designed to run on networks whose remote latencies are in the order of 1000's or 10,000's of cycles. This means that in pSather, there might be occasions when it is more efficient to execute a routine locally even though the object is remote (e.g. when there are few accesses to object attributes and remote subthreads are expensive due to lack of good thread support in the architecture). But in Emerald, the costs to set up remote subthreads are always smaller than the network latency.

Although pSather's definition does not execute a routine at the object's location by default, the current semantics still allows a programmer to use the @-operator for specifying that a routine must always execute at the cluster of its object.

```
hash_table.insert(e) @ hash_table.where;
```

Our approach gives a programmer fine-grained control over where objects are placed and where routines are executed so that he can adjust for tradeoffs between locality and load-balancing. With well-designed library classes, the @-operator will not be visible to the general user but remains a flexible construct for the library designer. Without such a fine-grained control, pSather will not be as useful for experimenting with parallel algorithms.

Section 2.5.2 further describes how our defaults achieve what we intuitively want for object-allocation and copying on different clusters (which is to allocate/copy an object on the local cluster unless otherwise specified by the @-operator).

2.4.4 Atomicity and Consistency of Memory Operations

Atomicity

Since memory reads/writes are an important (though often implicit) part of any programming language, a parallel language must specify a clear semantics for them. PSather defines the atomicity and consistency of its memory operations. The basic atomicity rule in pSather is: every read or write of a variable of a *built-in* value type or of any reference type is atomic. This means that during a write/read operation, the executing thread has exclusive access to the affected part of the memory and the memory operation is performed as an atomic transaction. In pSather 0.1, the type of any variable is a reference type, an abstract type or one of the *built-in* value types[23] and a variable of an abstract type can only hold reference objects. Thus, the reads/writes of all variables are atomic. Section 2.7.2 discusses how the different type system in pSather 1.0 might affect the atomicity rule.

[23] We ignore external types here.

Memory Consistency

PSather allows threads to freely read and write variables on remote clusters. Multiple threads on different clusters may write to the same variable simultaneously. The atomicity rule guarantees that the variable will have a value written by one of the threads. Another independent question concerns when various threads reading a variable will see the newly written value. This problem arises in modern cache-based processors and is called the memory consistency problem (for which a concise introduction is given in [128]). If every processor cache in a shared-memory machine needs to be synchronized on each write instruction, performance can be greatly reduced.[24] Various levels of consistency among processors have been defined and studied [106]. The strongest of these is sequential consistency [153] which guarantees that every read operation sees the most recent write to the same location.

One condition of sequential consistency is that within a process(or), memory accesses respect the *program order*, i.e. the access order specified by the control and data dependences in the program code for that particular process(or) when no reordering takes place. In fact, this condition is also found in other consistency models e.g. processor consistency and weak consistency.

Since this is essential for reasoning about the semantics of sequential programs, the semantics of pSather adopts this requirement. The intra-thread consistency requirement specifies that accesses issued from a single thread always obey the program order (based on the code executed by the thread). This does place some constraints on an implementation as we will discuss at the end of this section.

However if we require the program order of a thread to be similarly observed by all other threads, we may not be able to take advantage of the parallelism available in the multiprocessor memory system. For example, if a thread T1 executes:

```
x := 3;
y := 4;
f(x, y);
```

and, `x` and `y` are located on distinct physical memory modules, the writes may happen in parallel without violating the intra-thread order. But the write to `y` may complete before `x`, so that another thread sees the old value of `x` and new value of `y`. In order for this update order to be observed globally, T1 must make sure that the write to `x` is definitely completed before issuing the write to `y`.

We therefore relax the consistency requirements across threads as follows. The update order in thread T1 is observable by other threads when T1 performs a *consistency-ensuring* operation. The following pSather operations are consistency-ensuring: the forking of a child thread, the termination of a thread, and any gate operation. The inter-thread consistency rule states that all writes executed by T1 before an ensuring operation will be seen by other threads. For example:

[24] [105] gives performance measurements for several levels of relaxed consistency.

```
[Thread T1]                      [Thread T2]

-- "flag" has value false.       if (flag = true) then
x := 3;                             -- "x" must be 3
g.set;                           end;
flag := true;
```

Figure 2.16: An example of consistency-ensuring operation.

- Consider Figure 2.16. If thread T1 executes an operation on a gate `g`, all the updates performed by T1 before its access of `g` are observable by T2. We note that T2 does not need to perform a consistency-ensuring operation.
- A child thread will see the updates done by its parent before its creation.
- Suppose a parent thread executes:

  cobegin ...`:- f;` ...**end**;

  After the `cobegin-end` statement, the parent thread is guaranteed to see the updates done by the child thread before its termination. (The crucial operation here is the child thread's termination; the updates are observable by other threads which may not be its parent.)

The other consistency conditions are that all processors in the machine will eventually see any update of memory, and that all broadcasts (Section 2.5.4) are guaranteed to complete before the next operation is executed.

Both the intra-thread and inter-thread consistency rules place constraints on pSather implementations. The serial intra-thread rule is just the conventional requirement so long as execution is confined to a single processor. Even this entails restrictions on remote operations for systems (such as CM-5) that do not preserve message order. In addition, pSather allows a thread to continue execution on a different processor by invoking a remote procedure. There is also the possibility in some implementations that a thread can be interrupted and later resumed on a different processor. The serial intra-thread rule specifies that preemption and the starting or ending of a subthread must include establishing memory consistency between the old and new processors executing the thread.

The weaker inter-thread rule imposes similar implementation requirements. The forking or termination of a child thread and any gate operation forces the completion of outstanding write operations.

We believe that our consistency model, while retaining a simple semantics, will enable future implementations to take advantage of efficient memory mechanisms in large-scale distributed-memory multiprocessors. Our prototype implementation on CM-5 (Chapter 3) however retains a sequentially consistent model (Section 3.3.1) because (i) it is simpler to implement, and (ii) the architecture instruction set does not provide any way to use the caches and/or other hardware memory mechanisms. The usefulness of the weaker consistency model, therefore, remains to be verified.

2.5 PSather – Language Support for NUMA

Section 2.4 introduced the @-operator which allows the programmer to control execution within a cluster model. This is one of the constructs that help the programmer deal with NUMA nature of many multiprocessors in a high-level fashion. In this section we describe other constructs in the same category.

2.5.1 Distinguishing Local vs. Remote References – `with-near` Statement

One possible way of implementing global pointers on a distributed-memory machine is to use a <cluster-id, address> pair as a global pointer. An obvious cost in such an implementation is the extra checks on the global pointers for local or remote objects. Since the programmer may have knowledge about distribution of objects (e.g. certain components of an object are allocated locally), it would be useful if the programmer can specify that a variable holds a reference to a local object[25] and need not be checked for remote reference.

We therefore add an assertion-like statement for the programmer to specify a set of variables to be dynamically *near* to an executing thread. This near-assertion statement has the following syntax.

```
with <ident-list> near
  <statement-list-1>
[else <statement-list-2>]
end;
```

The list of identifiers `<ident-list>` may contain the following kinds of variables:

- local variables
- parameters
- predefined variables: `res`, `self`

These *near* variables must be of reference types. This criterion is checked during compilation. Henceforth, we will refer to variables which are listed in `<ident-list>` as near variables when they occur in `<statement-list-1>`.

[25] An object is local with respect to an executing subthread if it is located on the same cluster as the subthread.

When the near-assertion statement is encountered, every near variable is tested to check that it either is void or has a near pointer (i.e. references an object which is near to the executing subthread). If the condition does not hold and an else-part is available, the thread continues to execute <`statement-list-2`>. If the condition does not hold and there is no else-part, a runtime error occurs. If the condition holds, the thread continues to execute <`statement-list-1`>. The `with-near` statement allows the programmer to assert that in <`statement-list-1`>, the near variables (locals, parameters and predefined variables) only reference objects local to the executing subthread.

A near variable may be updated but it is a runtime error to update a near variable to reference a remote object. The handling of this runtime error is implementation-dependent. To help the programmer in debugging, a compiler option may be specified to generate code to check that updates of near variables are valid during program execution.

We illustrate one possible use of the near-assertion statement. The implementation of a routine may require that the subthread executes on the cluster where the object is located. This requirement may be documented in the program. It would be better if this requirement can be tested during program execution. The following code shows how the near-assertion statement can be used to check that the hash table and the subthread executing `insert` are located on the same cluster:

```
insert(e:T) is
  with self near ...end;
  -- We want the subthread to execute on the same
  -- cluster as the hash table.
```

To support the handling of remote and local objects, there are two additional predicates applicable on any object.

```
<obj-expr>.is_far
<obj-expr>.is_near
```

In both cases <`obj-expr`> evaluates to a reference object during execution. The predicate `is_far` returns true if the object evaluated by <`obj-expr`> is not on the same cluster as the subthread executing this test. The predicate `is_near` returns true if the object is on the same cluster as the subthread. If the object is void, both `is_far` and `is_near` return false.

We note the slight difference in the treatment of void pointer: while the `with-near` statement allows void pointers, a void pointer returns false for the `is_near` predicate. This is because we expect `with-near` statement to be commonly used in situations in which a local variable is declared and used to hold pointers to various local objects within a loop:

```
p:POLYNOMIAL;
with p near
  until test loop
    p := <Get a local polynomial>
    < Work on p >
  end; end;
```

Since the `with-near` statement allows void pointers, we can execute the near assertion only once before all the loop iterations. Otherwise, we would have to execute the `with-near` assertion within the loop:

```
p:POLYNOMIAL;
  until test loop
    p := <Get a local polynomial>
    with p near
      < Work on p >
    end; end;
```

This increases execution costs. The current semantics allow programmers to move loop-invariant near assertions (which the compiler does not normally recognize) out of loops.

On the other hand, the definitions of `is_far` and `is_near` allows us to integrate the tests for void pointers, and near vs. remote pointers. Without `is_far` and `is_near`, the only way to get an object's location is to use `<obj>.where`. Since `<obj>.where` is invalid for void `<obj>`, in general, the code has to do a separate check for void pointer. This may result in the following common pattern:

```
if (x = void) then
  S1
elsif (x.where /= CONFIG::current_cluster) then
  S2
else
  S1
end;
```

Allowing `is_near` and `is_far` to return false on void pointers, simplifies the code:

```
if (x.is_far) then
  S2
else
  S1
end;
```

A Design Decision of `with-near` Statement

One design decision was whether the variables in the `with-near` statement can include the attributes, shared and constant features of the current class (i.e. the class in which the `with-near` statement is found). We decided to exclude them.

The reason is that it then becomes possible to enforce the semantics of the `with-near` statement during execution. Attributes, shared and constant features are accessible by multiple threads. Although an attribute may contain a near pointer when the `with-near` assertion is checked, another thread may update the attribute with a remote pointer, violating the assertion that the variable holds only near pointers. Because of the non-determinism in pSather programs, it is impossible to catch the violations even if we checks all reads/writes of attributes when executing <`statement-list-1`> in the `with-near` statement. For example, after a thread T1 has checked that an attribute contains a near pointer, another thread T2 may update the attribute to a remote pointer and back again to a near pointer (both relative to T1) before T1 reads/writes the attribute.

Local variables and parameters[26] however are not shared by multiple threads and any violation of the `with-near` assertion can be caught during program execution. Because the only way to update local variables/parameters is by the assignment statement, the compiler can generate code to verify that, when variables are updated, their new pointer values are still near. Programs therefore become safer.

This restriction does not make the language less expressive, since it is always possible to assign the value of an attribute (or shared or constant) to a local variable and use the local variable within the `with-near` statement.

2.5.2 Copying/Migration of Objects

In a NUMA model such as pSather's, objects often need to be moved or copied to improve data locality. We demonstrate how `@`-operator works with the copy/move mechanisms.

Regular, Deep and Near Copies

We first look at how the object-copy operations in Sather (`copy` and `deep_copy`) work in pSather. Since these operations are routine calls, our use of the `@`-operator applies to them as well. We can have:

```
x.copy;
x.copy @ cluster_id;
```

"`x.copy`" returns a local copy of `x` wherever the current thread executes.[27] The routine call for "`x.copy @ cluster_id`" executes at `cluster_id`, and the result is a copy of `x` at `cluster_id`. The orthogonality of the object-creation operations and the use of `@`-operator therefore works out nicely. The `@`-operator allows the programmer to specify the final object's location independent of where the original object is located.

The routine `deep_copy(ob:$OB):$OB` (in `SYS` class) copies the graph of objects rooted at `ob`. Since the pSather semantics dictate that the location of the deep-copied object is where

[26] `res` is a predefined local variable while `self` is a predefined parameter.

[27] The current cluster of a thread is given by `CONFIG::current_cluster`.

deep_copy is executed, the final object graph is located on one cluster even if the object graph was originally scattered over multiple clusters. Calling deep_copy with the appropriate cluster id using the @-operator allows us to decide where we want the deep-copied object to be.

Because of the distinction between remote and local objects, we feel that another copy routine (near_copy) is more useful in many cases. The routine near_copy(ob:$OB):$OB is defined in the SYS class. It copies the structure of all objects reachable from the root object ob directly via near pointers. The nearness is with respect to the root object and not with respect to the current locus of control. Suppose we have a root object O_1 which points to a remote object O_2 and O_2 points to another object O_3 which is near with respect to O_1. O_3 is not copied because it is not reachable directly via local pointers from O_1. The call SYS::near_copy(x) @ cluster_id copies all objects directly connected via near pointers to x, to cluster_id.

Implementation note: Since deep_copy and near_copy are part of a standard library, we expect that these operations will be implemented in an intelligent manner. When message startup cost is high, one possible strategy is to first pack the entire structure into a compact form, copying the compact form to the new cluster and then expanding it.

Migration of Objects

To support programmer-managed object migration, SYS class also defines a routine move_to:

```
move_to(ob:$OB; id:INT):$OB is
  res := ob.copy @ id;
  ob.invalidate;
end;
```

It makes a copy of the object ob at cluster id and mark the original object as no longer referenced. In a similar vein, we define the move-counterparts of near_copy and deep_copy:

```
near_move_to(ob:$OB; id:INT):$OB is
  res := near_copy(ob) @ id;
  -- And invalidate all objects which have been copied.
end;

deep_move_to(ob:$OB; id:INT):$OB is
  res := deep_copy(ob) @ id;
  -- And invalidate all objects which have been copied.
end;
```

The move operations return a new object identity. pSather in contrast to other approaches (e.g. Emerald [146]) does not support migrating objects which retain their identity, because the identity of an object is its address (including the cluster location) in our simple yet efficient model. The code for a class MOVEABLE{T} shows how the programmer can define a class for more transparently migratable objects.

```
class MOVEABLE{T} is
  obj:T;
  move_to(cid:INT) is
    if (obj.where /= cid) then
      obj := SYS::move_to(obj, cid);
    end;
  end;
end;
```

It is not totally transparent because instead of:

```
x:POLYGON;
x.move_to(cid);
x.draw;
```

the programmer has to explicitly retrieve the object before invoking `draw`.

```
x:MOVEABLE{POLYGON};
x.move_to(cid);
x.obj.draw;
```

Thus far all the object allocation routines are either predefined or provided as part of a standard class (`SYS`). We expect these to handle most of the distributed cases, but we also need general mechanism that allows sophisticated library writers to define their own copying mechanisms. The `PACKET` class provides such a mechanism.

General Copy/Move via `PACKET`

In general, one might want to have complex rules for deciding which parts of a data structure should be copied/moved to another cluster and which parts should remain uncopied/unmoved. The copy/move of a large structure should also execute efficiently. Due to high startup costs of communication, for the forseeable future this means that the most effective way of copying/moving data structures is to send packed representations. For this purpose, pSather provides a system class `PACKET` with the following interface:

```
class PACKET is
  -- Standard class for packing structures.
  mark(ob:$OB) is end;
  pack(ob:$OB; invalid_flag:BOOL):SELF_TYPE is end;
  unpack:$OB is end;
  is_empty:BOOL is end;
end;
```

A programmer builds a `PACKET` object `p` which contains (in a packed form) the data structure to be copied/moved. The `PACKET` object (instead of the data structure) is copied/moved, and then it is unpacked at the destination to rebuild the desired data structure. The objects to be packed are marked using `p.mark(ob)`. Note that after calling `mark`, the object remains unchanged, the packet `p` that makes a note and remembers the object `ob`. The `p.pack(ob, flag)` function applied to an object `ob` will traverse all reachable objects from `ob`. The object attributes are treated differently according to their type.

Value types. Value-typed attributes (e.g. `INT`, `CHAR`) are copied directly into the packet.

External type. Foreign attributes are not packed, and are replaced by the appropriate void value.

Reference type. If the pointer references a marked object, the object is recursively packed. Otherwise, the contents (pointer) is just copied into the packet.[28]

The packed objects are invalidated when the `invalid_flag` parameter is set to `true`. Therefore, the `invalid_flag` parameter gives the option of whether to retain or to invalidate the original data-structure.

`p.pack(ob, flag)` returns a `PACKET` object containing the packed objects that we want to copy or move. The objects are now stored in a single block of memory, so that copying/moving a packet can make use of the machine's bulk communication mechanisms. We can make use of the same packet to `copy` to several destinations. A moved packet is invalidated. These collapsed packets can be reconstituted by executing the `p.unpack` operation.

A packet may be unpacked more than once. Thus `p.pack` can be called multiple times, and each call to `p.unpack` then returns a copy of the corresponding graph of reference objects. `p.unpack` returns void when there is no more object to be extracted. `p.is_empty` returns true before any `pack` operations are called and after all objects are unpacked; otherwise it returns false. We do not expect the general user to do any of this, but the mechanism is needed for building distributed data classes.

Possible Implementation: `PACKET` can be an `ARRAY{CHAR}` or `ARRAY{INT}` with a private hash-table for remembering objects. The `mark` operation first creates a hash-table if none exists, and records `ob` into the table. After a `pack` operation, we might deallocate the hash-table.) The `unpack` operation just looks at the array part to rebuild the data-structure and ignores the presence/absence of hash-table. If we copy a packet `p` without first calling `p.pack`, an empty packet (that does not contain any objects) is copied.

Using `PACKET` – An Example

A class for distributed binary trees might need a way of copying/moving a local subtree to another cluster, but leaving any pointers to other clusters unchanged. This would require routines that copy/move exactly those objects that are dynamically near (Figure 2.17). Every recursive call of `tree_pack` marks the subtree below, and eventually after the root has been marked, we pack everything that has been marked.

[28] In some implementations (not the current CM-5 one), this may require transforming a near pointer to a far pointer.

```
private tree_pack(ch:PACKET) is
  if lt.is_near then lt.tree_pack(ch) end;
  if rt.is_near then rt.tree_pack(ch) end;
  ch.mark(self);
end;

tree_copy_to(id:INT):TREE is
  -- Suppose 'self' is the root of sub-tree.  Return
  -- pointer to remote copy of sub-tree at cluster
  -- given by 'id'.
  p:PACKET := PACKET::new;
  tree_pack(p);
  res := (p.pack(self, false).copy @ id).unpack;
end;

tree_move_to(id:INT):TREE is
  p:PACKET := PACKET::new;
  tree_pack(p);
  p := SYS::move_to(p.pack(self, true), id);
  res := p.unpack;
end;
```

Figure 2.17: Using `PACKET` on a subtree.

2.5.3 Predefined `SPREAD{T}` class

Normal objects in pSather reside completely in the memory of the cluster they were created on. To support distributed objects we introduce the class `SPREAD{T}` as a base class for objects whose space is allocated by spreading over all clusters.

We may allocate an object of class `SPREAD{T}` exactly like any other reference object in Sather:

```
s := SPREAD{T}::new;
```

If `T` is a reference type this statement allocates space for a pointer on each cluster of the machine. Otherwise, if `T` is a value type, space is reserved according to the size of an object of type `T` on each cluster.

The use of `SPREAD{T}` is syntactically similar to the class `ARRAY{T}` in sequential Sather. The difference is that the index range is mapped on clusters instead of consecutive memory locations with the classic array. Thus, the statement:

```
local := s[CONFIG::current_cluster];
```

reads the local instance of a `T`-typed object on the current cluster, while the next statement:

```
s[i] := local.copy @ i;
```

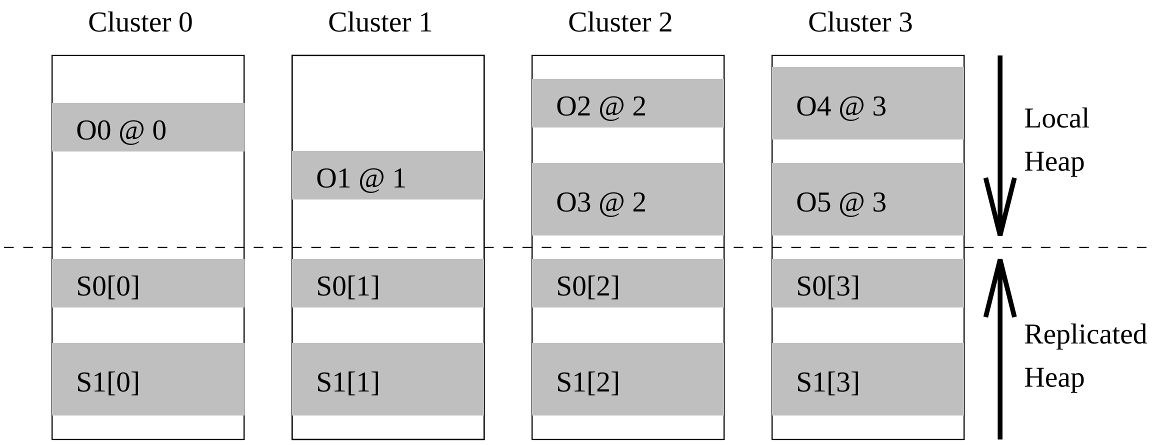


Figure 2.18: Memory organization for `SPREAD` and ordinary objects.

makes a copy of `local` on cluster `i` and assigns the copy to the element of `s` on cluster `i`.

While the index range of a regular pSather array is determined by the size of the array, the indices of `SPREAD{T}` always range from 0 to `CONFIG::num_clusters` - 1.

Spread objects are a low-level concept in pSather and it is important that remote parts of a spread object can be accessed directly without going through a centralized directory. Otherwise, a directory that maps cluster indices to addresses of local chunks of memory, will result in access bottlenecks at the directory. Therefore, spread objects need to share a single object identification on all clusters.

Implementation Note: Since object identification means memory address in pSather, it is helpful to look at one possible implementation in order to understand the semantics of spread objects. Ordinary reference objects are allocated from a heap that is managed for each cluster individually. In addition to this heap there is a second heap for spread objects, located in the same address range on each cluster. This heap may be managed in a centralized or distributed fashion. For example, on the CM-5, this heap is managed by a central agent for the whole machine. Thus, if new memory is requested for a replicated object a chunk of memory is reserved by the central agent in the replicated heap on the same address for each cluster. Figure 2.18 shows the memory map on each cluster after allocating six local (O0 – O5) and two spread (S0, S1) objects. Note that the replicated heap may also hold other replicated entities like code, shared attributes and constants, whereas the stack segments for the threads are allocated in the local heaps. Section 3.4.6 describes the implementation in more details.

2.5.4 Replicated variables

The class SPREAD{T} provides a mechanism to support distribution of an object over multiple clusters. We have seen how a spread object is allocated from a replicated heap. A reference object is however only accessible via a pointer stored in a variable (e.g. parameter, local variable, shared feature). If a reference object is referenced from multiple clusters, it is inefficient to read the object pointer from a single variable. We therefore extend the use of replicated heap to allocate replicated variables as well.

The kinds of variables in pSather include parameters, local variables, attributes, shared and constant features. Only shared and constant features are replicated (allocated on every cluster). Parameters and local variables are not replicated because most of the time,[29] they are only accessible from a single thread. Object attributes are contained in objects with fixed location and are also not replicated.

Shared and constant features serve as a form of global variables and are accessible by multiple threads; they are therefore replicated to speed up access. The declaration

```
shared x:POLYGON := POLYGON::create;
```

allocates `x` on every cluster. Each `x` references a local `POLYGON` object.

Using a shared/constant feature in an assignment, expression or actual parameter always refers to the local instance. There are also ways to read and update remote instances. In pSather 1.0, a read (write) access of an attribute, shared or constant feature is semantically equivalent to invoking a routine that reads (writes) the feature. Using the `@`-operator, we can therefore specify that the read (write) routine be executed at a remote cluster. This allows us to read (write) the remote instance of a shared/constant feature.

```
-- "r" is a shared/constant feature; "c" is a valid cluster id.
c:INT;
r@c := <expr> -- Syntactic sugar for "r(<expr>) @ c".
...:= ...r@c ...
f(..., r@c, ...);
```

Besides accessing a single instance of a replicated variable, there are also predefined operations to manipulate all the instances of a replicated variable. For every shared feature `x`, there are three predefined routines `scatter_x`, `gather_x` and `broadcast_x`, in addition to the normal read and write routines. (A constant feature only has `gather_x` additionally defined because it cannot be updated.)

The interface of scatter, gather and broadcast routines are:

```
scatter_x(vals:$ARRAY{T})
gather_x(vals:$ARRAY{T})
broadcast_x(v:T)
```

[29] The exception is `dist` statement (Section 2.6.2) which allows multiple threads to read/write a common local variable.

Both the scatter and gather routines get an array of values. Each array element is of type `T` which conforms to the type of `x`. The scatter (gather) routine writes (reads) the variable at cluster i from (into) the ith array element. It is a runtime error if the length of the array is not equal to the number of clusters. The broadcast routine gets an argument of type `T` and updates `x` on every cluster to the same value (given by `v`). It is a runtime error to perform two or more simultaneous broadcasts on the same shared feature `x`.

2.5.5 Discussion

In order to support a shared memory programming model, many systems (e.g. Munin [32, 60], Midway [34]) provide a set of common consistency protocols to support distributed shared memory. These consistency protocols are predefined and built into the system. The advantage of an object-oriented system is that users can build their own set of consistency protocols based on the needs of the application. For example, Section 4.3 describes a class that combines the functionality of a hash table and a cache for read-only objects. This is used in both the prime finding and Gröbner basis programs (Sections 4.4 and 4.7 respectively). Section 4.5 further describes how a distributed matrix class is extended to provide a consistency protocol suitable for the successive over-relaxation (SOR) algorithm. These examples show that the class library is a flexible approach to build various consistency protocols based on the application's needs, and that the language constructs make it simple to write/extend the class libraries.

2.6 PSather – Data-Parallel Extensions

The term "data-parallel" can be interpreted in slightly different ways. "Data-parallelism" in the object-oriented model generally refers to invoking routines on a collection of objects in parallel. For example, multiprocessor Smalltalk [191] has a `Collection` class which supports this notion of data-parallelism. When a collection object receives a `parallelDo:` message, it invokes a block in parallel for all the objects in the collection. In Section 2.7.6 we illustrate how this model of data-parallelism can be supported using the control-parallel constructs (Section 2.3) in pSather 1.0.

On the other hand, we are also concerned with the idea of "data-parallelism" associated with executing loops or manipulating arrays in parallel. This idea is a result of the association of early data-parallel languages (e.g. CM-Fortran) with the execution mode of SIMD machines. When data-parallelism is associated with the SIMD execution mode, the user effectively sees only one stream of control. In SIMD data-parallel languages, parallelism is only supported by constructs such as parallel for-statement. For example, when a CM-Fortran (or Lisp) program executes on a CM-2, the processors execute the instruction in lock-step on different data in a parallel for-statement.

In our view, the essential notion of data-parallelism is the ability to automatically fork multiple threads working in parallel on different data (using the same code) when a special construct

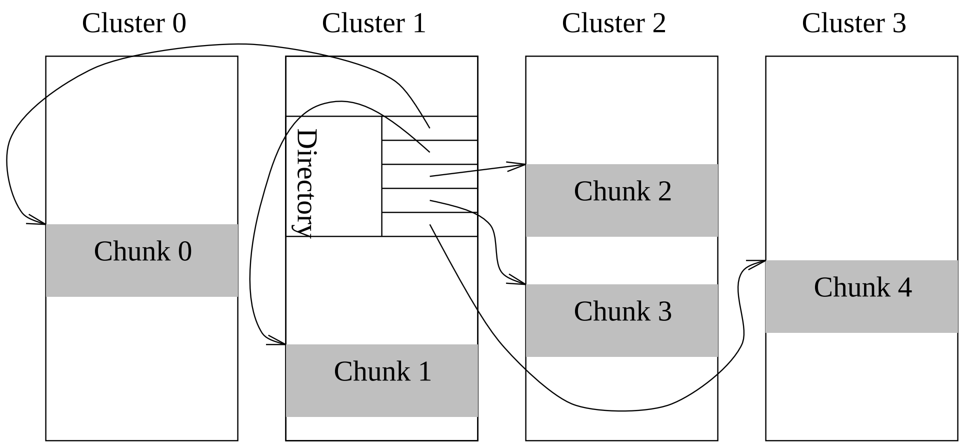


Figure 2.19: Memory organization for a `DIST` object (directory) with its chunks.

(e.g. a parallel for-statement) is encountered. The forked threads need not be working in lock-step. With this view of data-parallelism, it becomes conceptually straight-forward to integrate both data-parallel and control-parallel constructs in a framework. A thread automatically forks multiple threads when it encounters a special data-parallel construct. It then suspends; the multiple threads join back to the original thread of control when they terminate. Any of the forked threads might have created more threads or synchronized with other threads (e.g. one of the forked threads from another data-parallel construct).

But our goal is not restricted to providing a clean, efficiently implementable construct that handles the general semantics above. We also want to handle data-parallelism in *dynamic, distributed data structures*, whether they are regular (e.g. arrays) or irregular (e.g. sets). Section 2.6.1 describes a `DIST{T}` class that is used to build a *directory* for distributed data structures. Section 2.6.2 then introduces a `dist`-statement that works on directory objects from `DIST{T}` or its descendents. This statement supports data-parallelism in distributed structures, while maintaining data locality.

2.6.1 `DIST{T}` class

In general a distributed data structure consists of a directory object and a number of *chunks*. The directory object keeps track of the location and addresses of all the chunks. A chunk is an object that serves as a part of the distributed data structure, and can be built from other Sather classes (sequential or parallel). For example, a distributed array consists of a directory of sequential, non-distributed arrays. There may be more than one chunk per cluster. Figure 2.19 shows the memory organization for a distributed data structure consisting of a directory and five chunks on a machine with four clusters.

For good data locality, the class library writers will arrange each chunk and its data to be on the same cluster. However a chunk may still contain pointers to remote objects. Figure 2.20

```
class DIST{T} is
  dir:ARRAY{T};

  create(nc:INT):SELF_TYPE is
    res := new;
    res.dir := ARRAY{T}::new(nc);
  end; -- create

  num_chunks:INT is res := dir.asize end;

  set_chunk(i:INT; c:T) is dir[i] := c end;

  chunk(i:INT):T is res := dir[i] end;

  local_copy:SELF_TYPE is
    res := copy;
    res.dir := res.dir.copy;
  end; -- local_copy

  is_aligned(d:$DIST{$OB}):BOOL is
    i:INT;
    d_num_chunks:INT := d.num_chunks;
    if (d_num_chunks /= num_chunks) then return;
    else res := true end;
    until (i >= d_num_chunks) loop
      if (d.chunk(i).where /= chunk(i).where) then
        res := false; return;
      end; -- if
      i := i+1;
    end; -- loop
  end; -- is_aligned

  -- Following routines are used by dist-statement
  -- (Section 2.6.2).

  init_dist:INT is res:=0 end;

  is_done(iter_index:INT):BOOL is
    res:=(iter_index>=dir.asize);
  end; -- is_done

  curr_chunk(iter_index:INT):T is res:=dir[iter_index] end;

  curr_c_index(iter_index:INT):INT is res:=iter_index end;

  next_dist(iter_index:INT):INT is res:=iter_index+1 end;

end; -- class DIST{T}
```

Figure 2.20: Definition of DIST{T} class in pSather 0.1.

shows the definition of DIST{T}. In the simple case, a dist-object (i.e. objects of type DIST{T} or its descendents) contains a directory (`dir` attribute) which is an array of pointers to the chunks (of type T). The `create` routine allocates a dist-object with its directory. There are routines to find out number of chunks (`num_chunks`), set a chunk within the distributed data structure (`set_chunk`), retrieve a chunk (`chunk`), and make a copy of the dist-object and directory (`local_copy`). The `is_aligned` predicate checks the alignment with another dist-object. Two dist-objects are aligned iff they have the same number of chunks and their corresponding chunks are in the same cluster. The routines `init_dist`, `is_done`, `curr_chunk`, `curr_c_index` and `next_dist` serve as a way to index through the array of chunks. They are used by the `dist` statement (next Section) and their functionalities are replaced by a single iterator in pSather 1.0 [182, 181].

2.6.2 dist statement

PSather has a `dist`-statement for data-parallel computation on objects inheriting from DIST{T}. We introduce the syntax and the semantics of the `dist`-statement and in the next section, show examples for the `dist`-statement. Syntactically the `dist`-statement is another block statement in pSather:

$$
\begin{array}{l}
\textbf{dist} < \texttt{expr}_0 > \textbf{ as } < \texttt{chunk\_id}_0 >, \\
\quad < \texttt{expr}_1 > \textbf{ as } < \texttt{chunk\_id}_1 >, \\
\quad \vdots \\
\quad < \texttt{expr}_n > \textbf{ as } < \texttt{chunk\_id}_n > \textbf{ do} \\
\;\; <\texttt{Body}> \\
\textbf{end;}
\end{array}
$$

Each $< \texttt{expr}_i >$ evaluates to an object whose type is DIST{T} or a descendent of DIST{T}. Each *chunk variable* ($< \texttt{chunk\_id}_i >$, $0 \leq i \leq n$) is a single-name identifier visible only within the body of the `dist` statement. If the identifiers are not unique, the later one will overshadow the earlier ones; . A chunk variable will also overshadow variables with the same names declared outside the `dist` statement. `<Body>` consists of a list of statements.

The `dist` statement works as follows. The expressions $< \texttt{expr}_i >$ are evaluated sequentially. Then we use the cursor routines in DIST{T} (`init_dist`, `is_done`, `curr_chunk`, `curr_c_index` and `next_dist`) to generate the chunks of each distributed structure sequentially. The `dist` statement executes its body in a separate *body invocation* for each $(n+1)$-tuple of chunks. The chunk variables are used to reference the chunks of the corresponding distributed structures during the execution of `<Body>`.[30] The only ordering between the evaluation of the chunks and the body invocations is the (obvious) one that the $(n+1)$-tuple of chunks must be generated before its corresponding body invocation. For example, it is valid to generate all the chunks of all $(n+1)$ distributed structures

[30] Thus if $< \texttt{expr}_1 >$ evaluates to DIST{T} object (i.e. it is a distributed structure whose chunk type is T), then the type of $< \texttt{chunk\_id}_i >$ is T.

before any body invocation starts executing. The parent thread executing the `dist` statement proceeds to the next statement only after all body invocations have terminated.

Each body invocation executes sequentially, but there is no ordering between separate body invocations.[31] The body invocations may be executed completely sequentially or in parallel, so programmers cannot assume any ordering of body invocations. For example, it is incorrect to write code that performs a synchronous message send from one body invocation to another because the implementation may execute the receiver before the sender.[32]

When a body invocation encounters a `break` statement:

```
dist ...do
   ⋮
   break;
   ⋮
end;
```

it terminates. This causes the parent thread to send a termination signal to other body invocations. Since the invocations are not ordered and latencies of termination signals are unknown, it is possible that other invocations all execute to completion, in which case the `break` statement only affects the control of one invocation. The parent thread must still wait until all invocations have terminated before it can proceed to the next statement. A similar semantics apply when an invocation encounters a `return` statement. The only difference is that the parent thread returns from its current routine after all invocations complete, instead of continuing to the next statement. (Section 2.7.3 explains what happens when an invocation encounters some other new Sather 1.0 statement constructs.)

Each body invocation is executed on the cluster of the chunk referenced by $<$ $\texttt{chunk\_id}_0$ $>$. The purpose is to bind parallel computation to the location of data, in order to exploit locality. When there are more than one distributed structures in the `dist`, they must have the same number of chunks and the chunks in an $(n+1)$-tuple must be on the same cluster.[33] A runtime error occurs if the distributed structures are not all aligned. This alignment requirement does not reduce the expressiveness of the language construct because it is possible to reference non-aligned objects from within the body of the `dist` statement.

The `dist` statement is also a scope for *body local* variables (i.e. local variables declared within a `dist` statement body). The body local variables and the chunk variables are only visible within the body of the `dist` statement. There is one instance of each body local variable per body invocation.

---

[31] Since only the separate body invocations *may be* executed in parallel, the body code determines the minimum granularity of parallelism in a `dist` statement.

[32] But it is safe for multiple body invocations to lock a gate to ensure exclusive access to a shared resource.

[33] We have fixed the meaning of "alignment" in the `dist` statement because it is simple, efficient and ensures co-located chunks (and hence data locality). An alternative approach is to let the meaning of "alignment" in the `dist` statement be redefinable by the user. In this approach, we would require that `x.is_aligned(y)` be true for every pair of `x` and `y`, where `x` and `y` are the results of $<$ $\texttt{expr}_i$ $>$ and $<$ $\texttt{expr}_j$ $>$, $i \neq j$ respectively. This would invoke the user-defined routine `is_aligned` in `x`'s class. With $(n+1)$ distributed structures, there would be $\sum_{i=1}^{n} 2i$ calls, while our current approach only requires n calls.

Next we consider a variable (say x) declared outside the scope of the **dist** statement. x may be any of the usual variables (e.g. local variable, attributes), or even a body local variable or a chunk variable. The same instance of x is accessible from all body invocations. As a simple example, if we have:

```
dist a as a_c do -- S1
  x:INT;
  dist b as b_c do -- S2
    x:=...; ...
```

each body invocation of S_1 has its own instance of x. But when one of S_1's body invocation executes S_2, only its instance of x is accessible by all body invocations of S_2.

PSather allows assignments to x within a **dist** statement's body (except when x is a constant feature). Since the body invocations may execute in parallel, when the programmer makes an assignment to x, he/she ought to follow the rules for atomicity and consistency of memory operations (Section 2.4.4), and apply the same programming techniques as those used when a variable is shared/updated by multiple threads. The rules for atomicity of memory operation are more complex in pSather 1.0; non-exclusive access may result in objects with an inconsistent state (Section 2.7.2).

Because multiple invocations may access a local variable/parameter, we have to examine any language construct which is designed with the assumption that local variables and parameters are accessible only within a single thread. One such construct is the **with-near** statement[34] (Section 2.5.1). In the design of **with-near** statement, we intentionally allow only local variables and parameters (including **self**, **res**) in the variable list because they cannot be updated by different threads and their near'ness property can be strictly enforced. But if we have:

```
x:$OB;
dist ...do
  ⋮
  with x near ...
  ⋮
end;
```

x becomes accessible by multiple invocations which, in the normal case, will execute on different clusters. Since x can only be near with respect to a single cluster, this situation is an error in most normal circumstances. Therefore a **with-near** statement, when it occurs in a **dist** statement body, cannot specify any variable that is declared outside its innermost enclosing **dist** statement.

Figure 2.21 shows how the **dist** statement is used (in the DIST_VEC{INT} class) to implement addition of distributed vectors. The DIST_VEC{INT} class is a descendent of DIST{VEC{INT}} with sequential vectors (of type VEC{INT}) as chunks. This also shows an important construction principle: many distributed classes are built on top of their sequential counterparts by using the sequential objects as chunks of a distributed data structure. In Figure 2.21, the **to_plus** routine adds

[34] Section 2.7.3 describes another construct, the typecase statement, found only in p/Sather 1.0.

```
1 :         to_plus(a:SELF_TYPE):SELF_TYPE is
2 :            -- Store the sum of "self" and "a" into "self".
3 :            assert (pre) is_aligned(a) end;
4 :            dist self as c, a as a_c do
5 :               c.to_plus(a_c);
6 :            end;
7 :         end;
```

Figure 2.21: Use of `dist` statement summing two distributed vectors.

`a` to `self` and stores the result in `self`. In order for addition to work on each pair of chunks, `self` and `a` must not only have the same number of chunks with corresponding chunks on the same cluster, their vector chunks must also have the same dimension pairwise. Since the `dist` statement does not enforce the same-dimension requirement, this precondition is enforced by the `assert` statement (line 3). The `dist` match up the chunks (line 4) and each body invocation (line 5) performs sequential addition on the pair of sequential vectors. This computation style is again typical in `DIST{T}` classes: the distributed operation is just a distributed application of the ordinary operation.

Implementation note: A naive implementation of the semantics of `dist` statement can cause memory bottlenecks when multiple body invocations access the same local variable or parameter that is declared outside the scope of the `dist` statement. In the description of `dist` statement's implementation (Section 3.4.5), we describe how the current implementation reduces this bottleneck by identifying *read-only* local variables/parameters.

If a read-only local variable/parameter refers to a reference object `O1`, then the pointer to `O1` is passed to every body invocation; if it is a value object, then every invocation gets its own copy of the object. We also note that the implicit read-only variable `self` is similarly passed to all body invocations (i.e. depending on whether it is a reference or value object). Thus if `self` is a reference object, the separate body invocations will be able to access attributes of a single object. But if `self` is a value object, and since value objects do not have named attributes and are immutable, each invocation can use its copy independently. This implementation strategy remains applicable in pSather 1.0 because there are no named attributes in user-defined value classes. Section 2.7.3 further describes how the introduction of bound objects/types and abstract types in pSather 1.0 may affect how local variables/parameters (declared outside `dist` statement) are implemented in the `dist` statement body.

The optimization strategy only replicate pointers for read-only variables holding reference objects. In order for each invocation to get its own local copy of the reference object (to improve data locality), the programmer may use the various constructs described in Section 2.5 (e.g. `SPREAD{T}` class and various copying/move operations).

2.6.3 Examples

In this section, we present several examples to illustrate how the `dist`-statement is used to distribute various forms of computation. Figure 2.22 shows some routines from a distributed vector class (`DIST_VEC{DOUBLE}`). Each chunk in a `DIST_VEC{DOUBLE}` object is a sequential vector implemented by an array.

The `scale_by` routine (Figure 2.22 (a)) is the simplest; it takes a double-precision floating point number `a` and scales the distributed vector by `a`. Since the scaling is a fully parallelizable operation, we use the `dist` statement to start a body invocation for each chunk (line 3). Each invocation sequentially updates the elements in its chunk (line 4). We note that even in a simple operation, it is often the case that a `dist` statement body has to access variables outside its scope (e.g. parameter `a` in line 4). It is therefore important that an implementation identifies read-only variables and pass them by value to all invocations.

The `scale_by` routine is an example of operations in which the chunks can be updated independently. But there are also many operations which combine the vector elements into a single result, e.g. computing dot-product of or distance between two vectors, and finding maximum/minimum value in a vector. Figure 2.22 (b) gives such an operation: a `length` routine which computes the Euclidean length of a distributed vector.

We parallelize such operations by using the `dist` statement to create separate invocations, each of which compute a *partial result* independently. The partial results are then reduced after all invocations terminate. In the `length` routine, each partial result is the sum of squares of elements in a sub-vector (a sequential vector) (lines 4–5). The final reduction step adds the partial sums and computes the square root (lines 7–9).

One question that arises is where the partial results are stored. One way is to build a `REDUCTOR{T}` class based on `SPREAD{T}`.[35] The invocations on cluster i accumulate their partial results into the ith element of a reductor (spread) object. The reduction step then combines the elements in the reductor object. Our approach is to use a simple array (as in the `length` routine, line 2). Each invocation stores its result into a unique element of the array based on the chunk's id[36] (line 6).

In both `scale_by` and `length` routines, all invocations execute to completion. There are also cases when a body invocation I may find the result before others finish. In this case, we can use the `break` statement to terminate I and let the parent thread send termination signals to other still-executing invocations. The `bounded_sq_distance_to` (Figure 2.22 (c)) routine is such an example. It is further complicated by the fact that an invocation may want to use the intermediate results of other invocations.

The routine computes the square of the Euclidean distance between `self` and the first parameter `a`. The return value is the square of the Euclidean distance iff it is less than or equal to a

[35] A dot product example in Section 7.3 of [181] shows how this is used.

[36] This obviously assumes that an n-chunk distributed structure numbers them $0, \ldots, n$.

```
scale_by(a:DOUBLE) is
   -- Scale "self" by "a".
   dist self as c do
      i:INT; until i>=c.size loop c[i]:=a*c[i] end;
   end;
end;
```

a. A distributed scaling routine in a distributed vector class DIST_VEC{DOUBLE}.

```
length:DOUBLE is
   arr:ARRAY{REAL} := ARRAY{REAL}::new(self.num_chunks);
   dist self as c do
      i:INT; partial:DOUBLE;
      until i>=c.size loop partial:=partial+(c[i]*c[i]) end;
      arr[c.chunk_id] := partial; end;
   k:INT; until (k >= arr.asize) loop
      res:=res+arr[i]; k:=k+1 end;
   res := res.sqrt;
end;
```

b. Computing the length of a distributed vector.

```
bounded_sq_distance_to(a:SELF_TYPE; sbnd:DOUBLE):DOUBLE is
   assert (pre) is_aligned(a) end;
   mutex:GATE0 := GATE0::new;
   dist self as c, a as a_c do
      lsum:DOUBLE:=0.0; ct,i:INT:=0; exceed:BOOL:=false;
      until (i >= c.size) loop
         lsum := lsum+(c[i]-a_c[i]).square; ct := ct+1;
         if ct >= 100 then
            lock mutex then res := res+lsum; end;
            ct:=0; lsum:=0.0;
            if res>sbnd then
               exceed:=true; break end; end; end;
      if exceed then break end;
      if (ct>0) then
         lock mutex then res := res+lsum; end;
      end; end;
   if res>sbnd then res:=-1.0 end;
end;
```

c. Compute the square of the Euclidean distance from "self" to "a" if it is less than or equal to "sbnd", '-1.0' if it is greater than the bound.

Figure 2.22: Various uses of `dist` statement in some distributed classes. (The codes become more compact/elegant when the iterator construct in 1.0 is used.)

```
1 :     transpose:DIST_MATRIX_BLK_ROW{T} is
2 :        res := DIST_MATRIX_BLK_COL{T}::create_dir(nrows, ncols);
3 :        dist self as c do
4 :           c_transpose := MATRIXT::create(c.ncols, c.nrows);
5 :           i:INT;
6 :           until (i >= c.nrows) loop
7 :              j:INT; until (j >= c.ncols) loop
8 :                 c_transpose[j,i] := c[i,j]; j := j+1;
9 :              end;
10:              i := i+1;
11:           end;
12:           res.set_chunk(c.chunk_id, c_transpose);
13:        end;
14:     end;
```

Figure 2.23: Transposing a distributed matrix using `dist` statement in `DIST_MATRIX_BLK_COL{T}` class.

given bound (parameter `sbnd`); otherwise, -1.0 is returned. Each invocation can decide independently if its local sum (variable `lsum`) exceeds the bound, but it is more effective if the local sum is periodically added to `res` (shared by all invocations). Thus for every 100 iterations, the value in `lsum` is added atomically to `res` (line 9) and we check if the partial sum already exceeds the bound (line 11). In the worst case, an invocation may execute 100 additional iterations before it discovers that the bound is exceeded. If the bound has already been exceeded, we execute `break` (line 13) so that the parent thread may send a termination signal to other invocations. Otherwise, the remaining `lsum` is added to `res` atomically (line 14). The parent thread again checks whether the bound is exceeded and if so, returns -1.0 (line 17).

We provide another example to show the usefulness of the `dist` statement in Figure 2.23. The routines work on distributed matrices (instances of `DIST_MATRIX_BLK_COL{T}` class) whose chunks are sequential matrices. Each chunk represents a block of consecutive columns. In the `transpose` routine (Figure 2.24), line 1 creates a directory for the result. Each chunk is then transposed independently (lines 4–11). The transposed chunks are stored in the result's directory (line 12). A simple change in line 8 from:

```
c_transpose[j,i] := c[i,j];
```

to:

```
c_transpose[j,c.nrows-i-1] := c[i,j];
```

produces a routine which rotates a distributed matrix by 90 degrees clockwise. Figure 2.24 illustrates how the transposition and rotation operations work on a distributed matrix with four chunks.

Although the results have a different partition from the initial input, this is not a major obstacle in coding. One reason is that the blocks-of-rows partition may be exactly what is needed

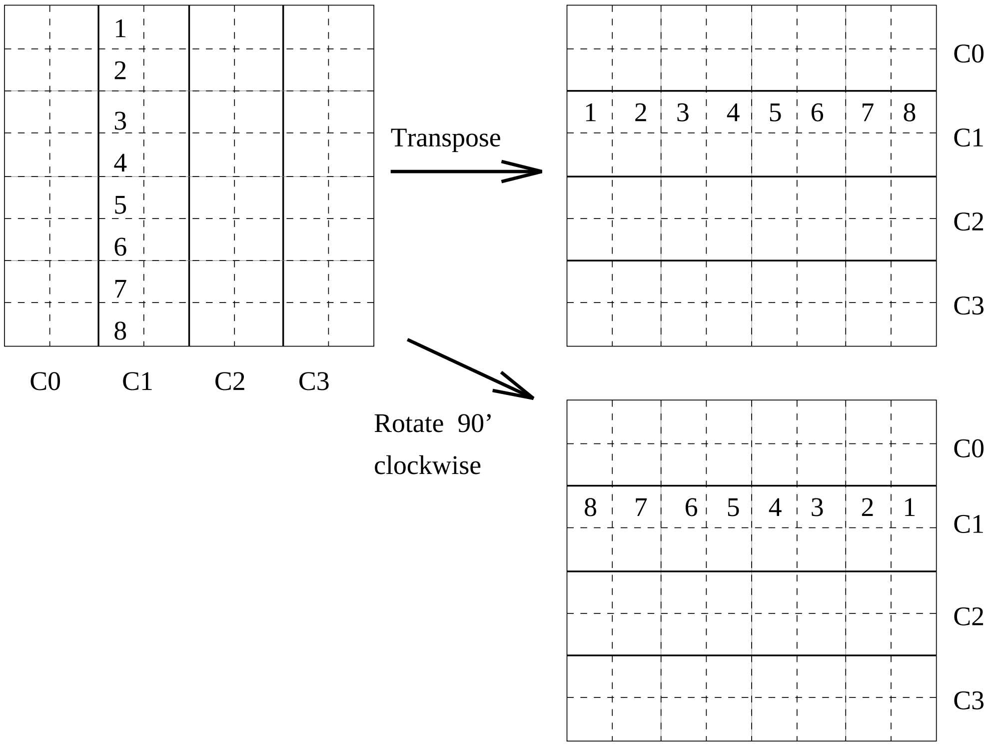


Figure 2.24: Pictorial illustration of DIST_MATRIX_BLK_COL{T} routines.

next in the program. Another reason is that the library is expected to provide routines that alter the partitioning of distributed matrices (because the same matrix may be partitioned differently in different parts of an algorithm). Hence if the block-of-rows partition is not appropriate, it is always possible to alter the result's partitioning.

2.7 PSather 1.0

Sections 2.3 – 2.6 describe how pSather extensions work in the context of Sather 0.1. This version is implemented in the current prototype (Chapter 3). In the following sections, we further clarify the semantics of the pSather extensions in the context of Sather 1.0[37] [189]. The overall semantics remain unchanged. There are, however, situations when new functionalities are introduced due to new language constructs (e.g. increased flexibility of thread creation (Section 2.7.1)); in other situations, we have to define the interaction of an existing pSather construct with new Sather 1.0 constructs (e.g. `dist` statement with exception handling (Section 2.7.3)). But the major advantage is the encapsulation power of new Sather 1.0 constructs, which help to build more general classes (e.g. parallel collections (Section 2.7.6) and actor-like classes (Section 2.7.5)), and simplify certain language requirements (e.g. interaction of `DIST{T}` class and `dist` statement (Section 2.7.4)).

2.7.1 Control-Parallel Constructs – New Functionalities

Two of the major revisions in Sather 1.0 are the separation of subtyping and implementation inheritance in the class hierarchy, and the unification of special case operations (e.g. attribute access) with general routine invocations. We look at how these revisions affect the synchronization and thread creation constructs in pSather 1.0.

Predefined `GATE` classes

In pSather 0.1, a class `A` can be used as the type of a variable in both dispatched ("`x:$A`") and non-dispatched ("`x:A`") forms. A class that inherits from `A` gets `A`'s implementation and is also a subtype of `A`. So we only need to specify the predefined `GATE` classes as reference classes. Unlike 0.1, pSather 1.0 has separate type and implementation inheritance (Section 1.2.1), so we have to define more clearly the role of the `GATE` classes.

PSather 1.0 has two abstract classes `$GATE{T}` and `$GATE0` whose interface consists of the gate routines. Objects whose type is a subtype of `$GATE{T}` or `$GATE0` can also be used in gate-related constructs (e.g. deferred assignment statements, locking statements). The abstract classes also contain the implementations of gate operations which cannot be redefined by the programmer. There are also two reference classes `GATE{T}` and `GATE0`. Each is a subtype of and inherits the code from its corresponding abstract class:

[37] The code examples will use pSather 1.0's syntax.

```
class GATE{T} < $GATE{T} is
  include $GATE{T} end;

class GATE0 < $GATE0 is
  include $GATE0 end;
```

By making GATE{T} and GATE0 reference classes, different threads using a gate for synchronization are able to reference a shared object, instead of each getting its own copy.

Forking Threads

In Section 2.3.3, when we describe the deferred assignment statement:

```
<gate-expr> :- <routine-call>;
```

we mentioned that the right-hand-side expression `<routine-call>` is restricted to a routine call (i.e. features which are defined as routines in classes).

Since the definition of "routine call" is broader in pSather 1.0, we now allow the right-hand-side of deferred assignment to include more types of expressions. For example, in pSather 1.0, an attribute definition:

```
class A is
  attr x:INT;
end;
```

automatically defines a read and a write routine. Both routine have the same name `x` and are distinguished by their type signatures. Therefore we may use these access routines on the right-hand-side of any deferred assignment statement. This provides a mechanism for reading and writing object attributes in a split phase fashion. Figure 2.25 shows how split-phase read and write operations can be implemented using gates. To do a split-phase write, the routine `x(INT)` is invoked asynchronously; `g.join` invokes a `join` routine (defined in Section 2.3.4) which suspends the current thread until the write completes. To do a split-phase read, the routine `x:INT` is invoked asynchronously and the result read is retrieved from the gate via `g.take`. The read and write routines of shared and constant features can also likewise be used in a deferred assignment statement.

(P)Sather 1.0 also includes many syntactic sugar expressions (Appendix B), including the array access expressions:

```
<expr>[<expr-list>]
```

which is syntactic sugar for `<expr>.aget(<expr-list>)` or `<expr>.aset(<expr-list>)`, depending on whether the expression is reading or writing an array element. Since `aget` and `aset` are routines, we can also implement split-phase read/writes on array elements using the deferred assignment statement.

With these sugars, the deferred assignment statement should also logically allow:

```
-- "g" is a gate.
g :- obj.x(...);
...
g.join;
-- "join" operation (defined in Section 2.3.4)
-- waits for completion of memory write.

g :- obj.x;
...
f(...,g.take,...);
```

Figure 2.25: Implementing split-phase soft write/read using the deferred assignment statement.

```
:- <expr1> + <expr2>; -- (1)
:- <expr1>.plus(<expr2>); -- (2)
```

But the syntax in (1) may mislead the human reader to think that the forked thread evaluates `<expr1>` and `<expr2>`, followed by the addition, when in fact, both `<expr1>` and `<expr2>` are evaluated before the thread is forked. We therefore disallow (1), but allow (2), because (2)'s syntax is consistent with forking threads for user-defined routines. Writing (2) also requires the programmer to think more carefully about the costs and benefits of creating additional parallelism.

Sather 1.0 also introduces the *bound routine* (Section 1.2.1); another new way of calling routines is to invoke bound routines. Thus, in pSather 1.0, the right-hand-side expression can also be a bound routine invocation (supplied with all the necessary arguments):

```
r:ROUT{T1,T2,...TN}:T; ...
g :- r.call(arg1, arg2, ...  argN); -- "g" is a GATE{T} object.
```

Bound routines and iterators are both constructs that reference executable code in pSather 1.0. Although bound routines are allowed on the right-hand-side of deferred assignment, iterators are not. The reason is that an iterator already embodies some form of coroutine-style control flow,[38] so combining iterators and thread creation would be confusing. The main confusing point is that there is a sort of goto-next state in any iterator call. If a thread is created to call the `elts` iterator,

```
loop
  g :- elts; -- "elts" is an iterator; illegal.
  ...
end;
```

it may not have completed when the deferred-assignment statement is next encountered, and the goto-next state is therefore unknown. One possible solution is to suspend the current thread until the goto-next state is defined, but this further complicates the semantics of the deferred-assignment statement without any benefit to expressiveness. (Although iterators do not work with deferred-assignment, they work well with *dist* statement to support data-parallel execution (Section 2.7.3).)

[38] An analogy is that an iterator is to structured coroutine, as a loop is to structured goto [156].

Another Detail

In pSather 0.1's deferred assignment statement, when the left-hand-side is a GATE0 object, the right-hand-side may or may not have any return value. This is in accordance with the semantics in Sather 0.1 which allows the result of any routine call to be discarded. But this leads to a common programming error when a routine call creates and returns a new updated data structure, and the user forgets to use the result and continues to operate on the old structure. So in Sather 1.0, it is stipulated that the results of all routine calls must be used or assigned to a variable. To be consistent with this new definition, in pSather 1.0, when the left-hand-side of a deferred assignment statement is a GATE0, the right-hand-side must not have any return type.

2.7.2 Atomicity of Memory Operations in 1.0 – A Discussion

Section 2.4.4 described the atomicity rule of pSather 0.1: every read/write of a variable of a built-in value type or of any reference type is atomic. The sum effect was that all variable reads/writes were atomic in 0.1. PSather 1.0 allows the programmer to define his/her own value types. It also allows a variable of an abstract type to hold both reference and value objects. We have to revise the atomicity rule to handle these changes in the type system. Any resulting rule has to be judged against several criteria: clean language specification, efficient realization and any effect on the language's support for encapsulation and reusable code.

There are two approaches. We can continue to ensure that read/write of variable of any type is atomic. Or we can allow read/write of variable of some type to be non-atomic, in which case, a clear and unambiguous rule is: every read or write of a variable is atomic, except for those variables whose type is BITS, a tuple, or a subtype of BITS or tuple.

The first rule has the advantage that it simplifies the reasoning of programs. For example, in a piece of code such as:

```
class SEARCH{T} is
  attr result:T;

  search(a:...) is
    ...
    result := ...
    ...
  end;

end;
```

the library designer is guaranteed that `result` is updated atomically, whatever actual type is given for T. We are able to avoid using additional synchronization mechanisms if the only requirement is that readers get a consistent value from `result`. If T is a user-defined value type (e.g. based on tuple), the obvious implementations may require `result` to be protected by a low-level lock. This is because most architectures only guarantee indivisible reads/writes for certain primitive data types. Therefore atomicity of more complex data types have to be guaranteed by the language implementation.

The second rule would eliminate such overheads most of the time, because on most machines, the abstract and reference types, and *most* of the built-in value types can be implemented directly using the primitive machine data types. There are still certain built-in value type such as `FLTE` (extended float) that may require multiple machine operations to read or write. The language implementation would still have to provide some low-level protection for such variables. The second rule does not guarantee atomicity for bits and tuple objects because unlike integers and floats, they may be viewed as objects with (unnamed) attributes.

Even if the first rule is adopted, the compiler may be able to analyse that access to certain variables are always atomic (e.g. local variables and parameters which are visible only to a single thread). Therefore the second rule should only be adopted if the language implementation suffers a significant performance penalty when atomicity of all types is guaranteed. Since no implementation of pSather 1.0 exists, we leave this issue open for further exploration.

2.7.3 `dist` Statement in pSather 1.0

Section 2.6.2 describes the semantics of the `dist` statement when one of the body invocations encounter a `break` or `return` statement. In general, we have to clearly define the actions of a `dist` statement when a body invocation encounters a statement which leads to a non-structured transfer of control. PSather has two additional non-structured control statements: exception and the `yield` statement (which only occurs in an iter definition (Section 1.2.1)).

- If an exception is passed on beyond the end[39] of a `dist` statement, all body invocations are eventually terminated and the exception is passed on to the next outer handler. One may look at the `dist` statement as an implicit exception handler, that handles all exceptions by properly synchronizing all body invocations and passing the exception on to the next outer handler.
- `yield` statements are not allowed in the body of a `dist`-statement (in an iterator definition). We also disallow any iter-calls in the body of a `dist`-statement (enclosed within a `loop` statement).

Section 2.6.2 also examines the interaction of `dist` statement with any construct which is designed with the assumption that local variables and parameters are accessible only within a single thread. One such example is the `with-near` statement; another is the `typecase` statement in 1.0. When a `typecase` statement occurs in the body of a `dist` statement, its discriminator variable must be a local variable, parameter or predefined variable (`self`, `res`), and there must not be any assignment to the variable in the `dist` statement body. For example:

[39] We say that an exception is passed on beyond the end of a statement if it was raised (explicitly by the `raise` statement, or implicitly by the runtime system or an inner exception handler that could not catch the exception) within the body of that statement and cannot be handled in the body.

```
a:$OB;
dist ...do
  typecase a
  when $A then ...
  when INT, FLT then a:=...
  else ...end;
  end;
```

the error is caught at compile-time, because there exists an assignment to `a` in `dist` statement body.

A local variable or parameter `v` may be declared outside the scope of a `dist` statement and accessed from within its body. We briefly discusses how the bound objects/types and abstract types affects any optimization strategy for read-only variables (Section 2.6.2).

- Bound objects are immutable. If we have:

  ```
  x:ROUT{INT}:INT; ...
  dist ...do
    f(x);
  end;
  ```

  there is no assignment to `x` in the `dist` statement body, so each body invocation can get its own copy of the bound routine. But the unique identity of the bound routine must be retained.

- Suppose we have a read-only variable `x`:

  ```
  x:$OB;
  dist ...do
    f(x);
  end;
  ```

  we do not know whether `x` holds a value or reference object until execution time. If it is a reference object $\mathsf{O}_{\mathsf{ref}}$, `x` contains the pointer to $\mathsf{O}_{\mathsf{ref}}$; if it is a value object $\mathsf{O}_{\mathsf{val}}$, `x` contains a pointer to a boxed version of $\mathsf{O}_{\mathsf{val}}$. Therefore we can only determine at execution time, whether each body invocation gets `x`'s pointer to $\mathsf{O}_{\mathsf{ref}}$, or a pointer to a local copy of boxed $\mathsf{O}_{\mathsf{val}}$.

2.7.4 DIST{T} and SPREAD{T} classes

Like the `GATE` classes, we introduce `$DIST{T}` and `$SPREAD{T}`, the abstract counterparts of the two predefined classes: `DIST{T}` and `SPREAD{T}` respectively.

```
class DIST{T} < $DIST{T} is
  include $DIST{T} end;

class SPREAD{T} < $SPREAD{T} is
  include $SPREAD{T} end;
```

Another change in 1.0 is that: the `dist` statement invokes an iterator (called `chunks`) in `$DIST{T}` to generate the chunks of a distributed structure. This is cleaner and simpler than the use of cursor routines in 0.1.

2.7.5 Dealing with Inheritance Anomaly

Section 2.2 describes how some parallel object-oriented languages provide synchronization mechanisms to separate the synchronization constraints from computation code so that the code can be more easily inherited. The general idea in the various approaches is to associate a condition with each routine, and execute the routine only when the condition holds.

We now show how the `$GATE` classes can be used to deal with inheritance anomaly explicitly in a relatively clear and succinct manner. We use a bounded queue example `QUEUE{T}` found in various papers (e.g. [147, 173]), and show how two different descendent classes can be introduced without altering the synchronization conditions in the ancestor class.

One clean approach to improving the inheritability of synchronization code is to separate the synchronization of a routine from the routine's computation. An example is given by use of *constraints* in HAL [140]. Each routine may have an associated constraint, such that the routine is executed only when the constraint is true. The actor model in HAL guarantees that at most one routine is executed in an object at any time.

Figure 2.26 shows how we can use `GATE0`, bound routines and iterators to model the constraints in HAL. The `QUEUE{T}` class has two routines – `get` and `put` (lines 2–11). Each routine `r` has an associated gate and a boolean function, which together act as a constraint (e.g. the gate in `get_cond` and the function `get_cond_fn` (lines 16–17) act as the constraint for `get`). Every routine `r` observes the following protocol. `r` calls enter(<cond$_r$>) before executing the routine proper; this waits until the synchronization constraint is satisfied. After `r` has executed the routine proper, just before control is returned to the caller, the routine `enable_routines` is called to allow other threads to enter.

Lines 12–40 show the code for the synchronization constraints. Line 13 defines an attribute that contains a gate to ensure exclusive access to the queue. Lines 14–15 define the gates that are part of the synchronization constraints for `get` and `put`. (`#GATE0` is the expression that creates a new gate object.) The gate is the part of a constraint that suspends/resumes a thread; the actual evaluation of a constraint is done by a boolean function. Lines 16–19 define the boolean constraint functions for `get` and `put`.

Next we define two iterators. The first yields all the boolean constraint functions (lines 20–22). The second yields (in the same order) the gates of the constraints (lines 23–25).

A routine is allowed to proceed only if it satisfies two conditions: no other thread is executing in the object and the routine's constraint is satisfied. When there is a thread executing in the object, the bind status of the `obj_lock` gate is set to *bound*. Similarly, when a constraint

```
class QUEUE{T} is
   get:T is
      enter(get_cond);
      <Get element from front.>
      enable_routines;
   end;

   put(T) is
      enter(put_cond);
      <Insert element at the end.>
      enable_routines;
   end;

   -- Code for synchronization constraints.
   private attr obj_lock:GATE0 := #GATE0;
   private attr get_cond:GATE0 := #GATE0;
   private attr put_cond:GATE0 := #GATE0;

   private get_cond_fn:BOOL is
      res := <Queue is not empty> end;
   private put_cond_fn:BOOL is
      res := <Queue is not full> end;

   private iter cond_fns:ROUT{SAME}:BOOL is
      res:=#ROUT(_:SAME.get_cond_fn); yield;
      res:=#ROUT(_:SAME.put_cond_fn); yield; end;

   private iter conds:GATE0 is
      res:=get_cond; yield;
      res:=put_cond; yield; end;

   private enter(c:GATE0) is
      lock obj_lock.is_unbound, c.is_bound then
         obj_lock.set;
      end;
   end;

   private enable_routines is
      -- Evaluate every condition and make the corresponding
      -- gate unbound ("true") or bound ("false").
      loop
         fn::=cond_fns.call!; cond::=conds.call!;
         if (fn.call(self)) then
            cond.set;
            elsif (cond.is_bound) then cond.take end;
         end;
         obj_lock.take;
      end; end;
```

Figure 2.26: QUEUE{T} class with constraints to simplify inheritance of synchronization.

```
class QUEUE2{T} is
  include QUEUE{T}
    cond_fns -> old_cond_fns,
    conds -> old_conds;

  get2:{T,T} is
    enter(get2_cond);
    <Get 2 elements from the front.>
    enable_routines;
  end;

  -- Additional synchronization constraint does
  -- not interfere with inherited code.
  private get2_cond:GATE0 := #GATE0;

  private get2_cond_fn:BOOL is
    res := <Queue has at least 2 elements>
  end;

  private iter cond_fns:ROUT{SAME}:BOOL is
    loop res:=old_cond_fns.call!; yield; end;
    res:=#ROUT(_:SAME.get2_cond_fn); yield;
  end;

  private iter conds:GATE0 is
    loop res:=old_conds.call!; yield; end;
    res:=get2_cond; yield;
  end;
  end;
```

Figure 2.27: QUEUE2{T} class illustrates how to add new synchronization constraints without interfering with inherited synchronization constraints.

```
class DQUEUE{T} is
  include QUEUE{T}
    cond_fns -> old_cond_fns,
    conds -> old_conds;

  get_back:T is
    enter(get_back_cond);
    <Get an element from the end of queue.>
    enable_routines;
  end;

  -- Code for additional synchronization constraints.
  private get_back_cond:GATE0 := #GATE0;

  private get_back_fn:BOOL is
    res := <Queue is not empty>
  end;

  private iter cond_fns:ROUT{SAME}:BOOL is
    loop res:=old_cond_fns.call!; yield; end;
    res:=#ROUT(_:SAME.get_back_cond_fn); yield;
  end;

  private iter conds:GATE0 is
    loop res:=old_conds.call!; yield; end;
    res:=get_back_cond; yield;
  end;
end;
```

Figure 2.28: DQUEUE{T} class illustrates how to add new synchronization constraints without interfering with inherited synchronization constraints.

```
class PARALLEL_DO{T} is
  do(elts:ITER:T; rout:ROUT{T}) is
    -- "elts" generate the objects from a collection
    -- (e.g. array, set).  A thread is created to execute
    -- "rout" for each element at its cluster.
    cobegin
      loop
        e::=elts.call!;
        :- rout.call(e) @ e.where;
      end; end; end;
      end;
```

Figure 2.29: A PARALLEL_DO{T} class supports a restricted form of data-parallel operations.

is satisfied, the constraint's gate is set to bound. So the `enter` routine uses the `lock` statement to suspend the current thread until the conditions are satisfied. Then it immediately sets `obj_lock` to prevent any further thread from entering the object.

When a routine has finished its computation, before it returns to the caller, it must recheck the constraints using `enable_routines` (lines 31–41). Each boolean constraint function is evaluated; if it is true, the gate is set to bound status (line 37), otherwise, a previously bound gate is reset to unbound (line 38). Finally we set `obj_lock` to unbound (line 40), to allow the next thread to come in.

The pSather implementation has the advantage that it shows explicitly the runtime costs incurred (as compared to the implicit costs in HAL). On the other hand, a HAL compiler might be able to do some analysis, and avoid re-evaluating some constraints at the end of a routine.

We can easily inherit the code in `QUEUE{T}` to implement specialized queue classes. The descendent class's synchronization does not lead to redefinitions of the parent's code (unlike the examples discussed in Section 2.2).

Figure 2.27 defines a class which allows the user to retrieve two elements from the queue atomically. The new routine is `get2`, and `get2_cond` and `get2_cond_fn` are its constraint. The slight inconvenience is that the programmer has to redefine the iterators (`cond_fns`, `conds`) that yield the constraints, so that `get2`'s constraint is now included. But this code is stylized and does not impose any burden.

Figure 2.28 defines a dqueue which allows the user to get an element from the back of the queue. We note that the change is similar to that Figure 2.27. In both cases, as long as the programmer follows a certain "protocol", a descendent class need not be concerned about any interference between the synchronization of the new and inherited routines.

The queue classes serve as examples of how the expressiveness and encapsulation capability of pSather is enhanced by the introduction of bound routines and iterators in 1.0.

2.7.6 Parallel Invocations on Objects.

Section 2.6 describes the data-parallel extensions in pSather, designed to work on dynamic distributed data structures. However, sometimes we want only to invoke the same operation on a collection of objects in parallel.

Figure 2.29 shows a `PARALLEL_DO{T}` class that supports parallel executions of a routine on different objects in a collection (e.g. array, set). It works as follows.

- A bound iterator that generates the elements of the collection is constructed and passed to the first parameter `elts` to the `do` routine.
- The second parameter `rout` is a bound routine which expects an object of type `T`.
- Successive objects in the collection are generated by invoking the `elts` iterator.

- For each object, a thread is created to invoke the desired routine using the object. Using the @ operator, we can improve locality by specifying that the thread executes on the cluster where the object is located.
- The cobegin-statement suspends the current thread until all the created threads (and whatever threads they create) terminate.

Some examples of routine calls that perform parallel invocations on different objects are:

```
PARALLEL_DOWINDOW::do(#ITER(window_array.elts),
          #ROUT(_:WINDOW.display)); -- (1)
PARALLEL_DOLEAF_REGION::do(#ITER(quad_tree.leaves),
          #ROUT(_:LEAF_REGION.compute_multipole_expansion); -- (2)
```

In (1), `#ITER(window_array.elts)` is a bound iterator that yields windows from an array. The bound routine `#ROUT(_:WINDOW.display` is invoked on each window. The threads that display different windows execute in parallel. (2) shows how an N-body application code (Section 4.8, Chapter 4) if it were written in 1.0. A bound iterator `#ITER(quad_tree.leaves)` generate the leaves in a quad-tree; threads are forked to compute multipole expansions in the leaf nodes in parallel. (This example gives a preview of the discussion in Section 4.9 on how the abstractions and applications in Chapter 4 can be improved with 1.0 language constructs.)

`PARALLEL_DO{T}` is an example of how abstractions can simplify the user's task. The user uses existing library classes to build parallel code and is insulated from the details on parallel programming.

2.8 Comparing pSather with Other Languages

We have earlier described pSather's design choices (Section 2.2.1) and given a table of summary for various parallel object-oriented languages (Table 2.1). Now that we have described pSather's parallel constructs (Sections 2.3 – 2.6), we are in a position to compare them with constructs in other languages. We focus on the high-level aspects of language features, rather than on low-level details (e.g. whether asynchronous or synchronous calls are default).

Synchronization is a key functionality in parallel languages; Section 2.8.1 compares `GATE` classes with other synchronization constructs. An important design goal of pSather is expressive language support for programming on distributed-memory machines. To achieve this, pSather introduces object placement in the language semantics; Section 2.8.2 compares our approach with others. Programming on distributed-memory machines also requires the construction of distributed data structures; Section 2.8.3 compares pSather's approach to building distributed structures with those of other languages. Data-parallelism is a well-studied technique for parallel algorithms, but not all parallel object-oriented languages provide suitable language constructs (if any) for writing data-parallel programs. Section 2.8.4 describes how data-parallelism is supported in pSather and other languages. Another distinct feature of pSather is its explicit support for a cluster model with distributed memory and shared address space. Section 2.8.5 discusses its implications for suitable target systems for pSather.

2.8.1 `GATE` Classes vs. Other Synchronization Primitives

The **`GATE`** classes are the main synchronization mechanisms in pSather. They provide functionalities which closely parallel other synchronization constructs in other languages, such as M-structures [24] in Id, and monitors in Mesa and Concurrent Pascal ([154, 120, 123, 136, 227, 50])

Monitors

PSather gates were previously called monitors [97] because of their similarity with the monitor concept in Mesa [154] and Concurrent Pascal [123].

Firstly a gate operation (monitor entry procedure) guarantees a thread (process) exclusive access to the object (Mesa module). But in pSather, the gate operations are predefined since **`GATE{T}`** and **`GATE0`** are predefined classes. In Mesa (for example), the entry procedures are user-defined because any Mesa module can be declared to be a monitor. This means that if a Mesa monitor operation is ever suspended, the user has to take care that it gets resumed correctly later and to prevent deadlocks.

In Mesa, condition variables can be declared in monitors such that a **`wait`** operation on a condition variable atomically suspends the process on a queue while a **`notify`** resumes one of the processes suspended on the condition variable. This functionality is achieved in pSather by the **`take`** and **`enqueue`** operations (corresponding to **`wait`** and **`notify`** respectively).

However, in terms of programming style, gates and monitors are used quite differently. We expect gates to be mostly used as components of objects to control synchronization among different routine calls acting on the same object. On the other hand, Mesa monitors are roughly "protected classes". "classes" because a monitor is an instance of a Mesa module (which encapsulates both data and procedures in a unit); "protected" because a monitor entry procedure is guaranteed exclusive access to the monitor. The **`wait`** and **`notify`** operations are also lower level than gate operations.

The deferred assignment unifies the functionalities of a gate with thread creation. As a result, a gate can be used as a future (section 2.3.3). This is not possible in Mesa because the monitor functionalities are independent of process creation.

M-structures

M-structures are designed to overcome the absence of state in Id, by allowing data structures to be updated. An M-structure has a state (*empty* or *full*) associated with it. This is like the bound state of gate objects and is used to synchronize access to M-structure. There are only two primitive atomic **`take`** and **`put`** operations that can be performed. When doing **`take`** on an empty M-structure, the thread is suspended, until another thread performs a `put`. The stored value is atomically retrieved by `take`. This is similar to the `take` and `set` operations in gates.

Unlike gates, M-structures do not store multiple values; thus there is no need to have any `enqueue` operation. The `examine` operation (similar to `GATE`'s `read`) is built on top of the `take` and `put`:

```
def examine c = { v = take c ;
                  _ = put c v;
              In v }
```

2.8.2 Object Placement / NUMA Programming Model

Emerald [146], like pSather, incorporates object placement in its language semantics. In fact, object mobility is a major design goal of Emerald and affects its design (e.g. parameter passing semantics). While Emerald's philosophy is to support transparent object mobility, pSather's philosophy is to let the programmer explicitly decide the distribution of objects.

Emerald's location-independent invocation vs. pSather's `@`-operator. Emerald supports location-independent invocation; a routine invoked on an object always executes at the object's processor, wherever the invoking thread might be. Therefore when an object moves, activation records of its routines have to be relocated as well. This means that the runtime system has to keep track of the activation records of every movable object; this might entail high runtime costs. Jul et al [146] describes how to reduce such costs and what to update when activation records are moved. In pSather, where a routine executes is independent of the object's location and is determined by the `@`-operator (executed in the caller). The `@`-operator simply specifies where a subthread executes. The orthogonality of the design allows the `@`-operator to be used with the copy/move operations for relocating objects. By using the `@`-operator, a thread's control can span multiple clusters even though subthreads (or activation records with locations) stay on a fixed cluster.

Emerald's transparent move vs. pSather's move/copy operations. The movement of Emerald objects is transparent to the user. Emerald distinguishes between objects which can be moved (*global objects*) and those which cannot be moved (*local objects*), and implements them differently. A global object is not referenced directly by any user variable. User variables refer directly only to local objects or local *object descriptors*. An object descriptor either holds a pointer to a resident global object or a forwarding address which gives the processor on which the global object is resident. When an object is moved, its object descriptors on the source and destination processors are updated accordingly.

PSather's move operations do not relocate objects in a transparent manner. The programmer has to update the references to the new object. But pSather does allow a programmer to implement objects which move in an almost transparent manner. (The `MOVEABLE{T}` class in

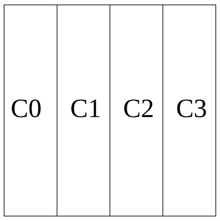


Figure 2.30: A `(*,BLOCK)` distribution for a 2-D matrix in Fortran D.

Section 2.5.2 corresponds to the object descriptor in Emerald.) There is also a variety of copy/move operations that allow a user to have complex rules to decide which parts of a data structure should be copied/moved.

2.8.3 Building Distributed Data Structures

Some parallel languages (e.g. Fortran D [99], High Performance Fortran [98], PC++ [103]) provide additional keywords/declarations so that the programmer can declare how a statically declared array can be partitioned on a distributed memory machine. For example, Fortran D has a `DISTRIBUTE` statement that assigns an *attribute* to each dimension of an array to describe how it is distributed. The possible attributes are `BLOCK`, `CYCLIC` and `BLOCK_CYCLIC`. The partition:

```
DISTRIBUTE A(*,BLOCK)
```

is illustrated by Figure 2.30, similar to the `DIST_MATRIX_BLK_COL{T}` class outlined in Section 2.6.3. The advantage of the Fortran approach is that the compiler may make use of its knowledge about the matrix partition to perform optimizations (e.g. collecting remote updates in a single message). The disadvantage is that the programmer only has a fixed number of choices about how the matrix ought to be partitioned.

PSather does not provide any predefined partitioning strategy. Instead it relies on the library designer to provide classes that build distributed structures. This library approach is more flexible because in addition to building Fortran D-like distributed matrices, the library designer can customize the partitioning (e.g. adding memory consistency protocols, Section 4.5). In addition to regular arrays, the facility to create dynamic objects also makes it easy to build dynamic, irregular distributed data structures e.g. the workbag (Section 4.1), the replicated hash table (Section 4.3), and quad-trees (Section 4.8). We rely on the library designer (rather than the compiler) to write code that handles synchronization and communication efficiently.

We note that although the construction of distributed structures is independent of the language's machine/programming model, the existence of a shared address space in pSather's cluster model (Section 2.4) and various NUMA language constructs (Section 2.5) simplify the task of designing and implementing classes for distributed structures (as compared to e.g. a message-passing programming model).

2.8.4 Support for Data-Parallelism

Section 2.6 describes pSather's `dist` statement which executes its body for each chunk in a distributed structure (whose type is a descendent of `DIST{T}`). There are five characteristics to note for pSather's style of data-parallelism.

- The so-called "data" which are operated upon in parallel are actually large-grained chunks (e.g. tree, array) consisting of finer-grained data (e.g. tree nodes, array elements).
- Although data-parallelism was previously associated with the execution mode of SIMD machines [6], the `dist` statement moves away from this association to an SPMD form of data-parallelism in which parallel threads execute the same piece of code on different chunks, but not necessarily in lock-step.
- The execution of body code is co-located with the cluster location of the corresponding chunk to ensure data locality. From the user's point of view, load balancing then consists of partitioning the distributed data structure such that each chunk requires approximately the same amount of computation.
- The setup can also be used for dynamic, irregular distributed data structures which can grow or shrink during program execution.
- A distributed object is just like any ordinary object and is not used exclusively only in `dist` statements. For example, it is possible to perform the usual object operations (e.g. routine calls, iterator calls) on distributed objects, just like ordinary objects.

In C* [126, 127], the user can declare a *domain* data type (just like a C struct) and then use this data type to declare domain arrays. A domain array is then used in a *domain select* statement such that the body of the domain select statement is executed in parallel on every element of the domain array. Comparing C* with pSather:

- The data operated upon in parallel in C* is fine-grained, compared to the large-grained chunks in pSather.
- The execution of the body is synchronous so that the model of computation is still SIMD.
- Like pSather, the execution of the body code is located on the same processor as the domain element. A difference from pSather is that in C*, the partitioning and partition locations of domain array are determined at compile-time, whereas in pSather, the chunk locations are determined during program execution.

- The size of a domain array in C* is fixed at declaration, and therefore there is no way to dynamically change its size, even when the number of needed domain elements changes. There is no such restriction in pSather.
- Because the idea of domain is closely associated with the data-parallel construct (domain select statement), it is not obvious that a domain object can exist independent of a domain array.

In PC++ [159], there is the concept of a homogeneous collection of objects which is analogous to the domain array in C*. The data-parallel construct is also similar to that in C*. One major difference is that a collection class in PC++ only needs to know the interface of the type of its elements. Thus any element type which satisfies the interface can be used in the collection. This allows the code in a collection to be reused; there is no similar facility for code reuse in C*.

There is also a research effort on dpSather [204] to add only "loosely synchronous data parallelism" to sequential Sather, without the notions of threads, synchronization etc. dpSather has bulk types; the elements in a bulk type is finer-grained than chunks. A variable can be declared to have a bulk type by writing "x:`par` <class>"; an invocation "`x.f`" then calls `f` in parallel on all elements of the bulk data. The "loosely synchronous data parallelism" breaks away from SIMD execution and is similar to our approach. dpSather's fine-grain approach is more suitable for vector machines. For distributed-memory multiprocessors with vector units (e.g. CM-5 [76], CNS-1 [18]), pSather will have to be extended further and/or borrow ideas from dpSather's fine-grain data-parallelism, so that the user can both distribute data and make use of vector operations.

2.8.5 Target Systems

In the design of parallel languages, there are inevitably some implicit assumptions on the characteristics of the ideal target architectures. Some languages such as Orca, Distributed Smalltalk are aimed at distributed systems (or distributed-memory systems with network latency of the same order). Other languages such as μC++ and parallel versions of Eiffel implicitly assume a uniform shared memory. PSather is suitable for both uniform shared memory multiprocessors,[40] and NUMA architectures whose network latencies are of the order of hundreds of cycles (or fewer). We feel that the cluster model (distributed memory, shared address space) is a justifiable choice because many parallel architectures are converging to this characterisation, and scalability and programmability considerations have led to many efforts at supporting distributed shared memory (e.g. [38, 13]).

[40] The first implementation of pSather was on a Sequent Symmetry.

Chapter 3

Implementation

This chapter describes the implementation of pSather. It is divided into the following sections. Section 3.1 gives an overview of the implementation strategies followed by a rough outline of the compilation process. Section 3.2 then describes the components of the runtime system. To make the description more specific, Section 3.3 describes the runtime implementation of pSather on the CM-5. We restrict our description to the CM-5 because it is a distributed-memory machine and does not have a shared address space. The more interesting implementation issues therefore arise in this context. After that, we devote Section 3.4 to the implementations of the parallel language constructs. The subsections 3.4.1, 3.4.2, 3.4.3, 3.4.4, 3.4.5 describe the implementation of the locking mechanism, `unlock` statement, `cobegin-end` statement, `with-near` statement, and `dist` statement respectively. Section 3.5 discusses some of the optimization strategies, including code size reduction, removing overhead in pointer dereference, inlining etc. Sections 3.6 describes the performance of the CM-5 implementation. We first measure the timings of important runtime operations and then use a direct $O(N^2)$ N-body program to demonstrate the performance improvements when some of the optimizations were incorporated. Finally Section 3.7 describes how the compiler can generate extra runtime checks to help the programmer identify the source(s) of errors during program development. These runtime checks are important in the current programming environment because we do not have a parallel debugger. The checks therefore serve as an important source of debugging information.

3.1 Overview of Compiler

The pSather compiler was adapted from the Sather 0.1 compiler. It implements the pSather extensions applied to Sather 0.1[1] and is itself written in Sather 0.1. The current pSather prototype supports the full set of 0.1 language features and pSather extensions, but does not support the newer Sather 1.0 language features.

[1] It also implements an `alias` construct which is not in Sather 0.1.

The compiler uses C as the intermediate language. This is similar to the strategy adopted by compilers by several other languages such as Standard ML [213], Scheme [26] and POOL2 [9]. The advantages of using C are obvious. Because of C's widespread availability, it is relatively easy to port the compiler. We can also take advantage of low-level optimizations (e.g. register allocation) performed by the C compiler. One disadvantage is that many parallel constructs have to be supported by runtime routines. The calls to these runtime routines may destroy information that the C compiler might have used for optimization. Another disadvantage is that the pSather compiler does not have fine-grained control of runtime resources (e.g. allocation of stack frames) which might improve system performance.

The compilation environment is similar to that in Sather 0.2i [144] (a public domain Sather implementation). Since our aim is to provide a relatively stable prototype to further experiment with writing parallel applications (in pSather), the compiler is implemented in a relatively straight-forward manner using standard techniques such as those found in [5, 218] and is itself not fully optimized. But because we want to demonstrate that it is possible to have a pSather implementation with reasonable performance, the later sections of this chapter describe some optimization strategies used in the compiler.

3.1.1 Compilation

The main data structure during compilation is the abstract syntax trees (AST's [218, Chapter 10]) for the classes being compiled. We make use of the Sather class hierarchy to represent different language constructs, so e.g. all AST nodes for lock-statement are Sather objects from the `LOCK_STMTOB_S` class. The classes are organized based on the category of the language constructs such as features, statements, expressions, variable declarations. Language constructs within a category share a common ancestor. For example all feature classes (routines, attributes, shareds, constants, inherited class) have a common ancestor. Multiple inheritance is used when a language construct belongs to more than one category (e.g. a shared feature is both a feature and a variable declaration).

The compiler has 17 phases. The delineation of the compilation process into phases is to simplify the description. It is often but not always the case that a phase corresponds to a complete traversal of the abstract syntax trees. The checks for syntax and semantic errors occur before phase 11. Subsequent phases, from phase 11 onwards, compute information that is used in various optimizations. The optimization-related phases are briefly outlined and described in more detail in Section 3.5. The main phases of the compiler are as follows.

1. **Read control files.** The compiler first finds and reads a user control file and a (default) system control file. The most important information provided by these files is the names of source files and additional runtime support to be linked in.

2. **Read pSather source files.** The next phase reads in the specified Sather source files. Each

file is passed through a handwritten scanner (written in C) and a YACC parser that builds up a code tree for each Sather class definition. There are two main data structures in this phase – a string table and a class definition table.

- **Class definition table.** A class definition table is simply a hash table. The abstract syntax tree of each class definition (a class definition may contain uninstantiated type parameters) is stored in the table.
- **String table.** A string table maps each string name to a unique positive integer. Using this strategy, string names are all stored as integers in the AST's and name equality is the same as integer equality. Even keywords such as `lock` are stored in the string table; this simplifies the identification of keywords by the scanner.

3. **Identify instantiated classes.** Class definitions may contain uninstantiated type parameters. The next phase in the compiler identifies classes with fully instantiated type parameters and constructs new abstract syntax trees for these instantiated classes. From this point onwards, all the semantic checks are performed on the abstract syntax trees for these instantiated classes (stored in a **class object table**).

4. **Expand class inheritance.** In this phase, the feature lists of inherited classes are recursively expanded into the feature lists of their descendents. A feature overwrites any earlier definition with the same name. This phase also eliminates features declared by `UNDEFINE`, and allows each class object to update a (possibly empty) list of direct-parent classes for later ancestor-descendent computation. It checks that there is no cycle in the class inheritance graph.

5. **Initial semantic phase.** After expanding the classes, we have all the instantiated classes to be used in the program and the abstract syntax trees of their features. This phase performs several semantic functions.

   It resolves any reference to `SELF_TYPE`, checks that the predefined classes `C`, `SYS`, `UNIX` and `UNDEFINE` are not used in any declaration and that `OB` is used only in a dispatched fashion (i.e. as `$OB`). It also counts the number of attributes in each class. The checks on the type declarations is done by each object recursively invoking its components. The type objects do the actual checking.

6. **Compute ancestor/descendent relationship.** In this phase, we determine all the ancestors and descendents of a given class. The result is that for a given class, there is a list of classes to which it conforms. Another side effect of this phase is to verify that classes obey the inheritance rules and identify the basic and array classes.

7. **Compute size.** After computing the ancestor/descendents, each class knows its C implementation type, and whether it is an array or not. In this phase, each class first rearranges the

ordering of the attribute features. Any attribute that occupies less than a word-size is moved to the beginning of the feature list, so that these attributes can be allocated in one contiguous block at the start of an object, reducing space requirement. Each class then sums up the sizes of its attributes and assigns a base size to itself. We also examine every attribute to find out whether a class is *atomic*. A class is atomic iff its objects do not store any pointer, i.e. either (1) the class is not an array and none of its attributes is a pointer, or (2) it is an array without pointer attribute and whose element type is not a pointer. This atomic information is used in the runtime object allocation routine, so that (atomic) objects not containing any pointers can be allocated differently to make garbage collection simpler.[2]

8. **Major semantic phase.** In this major semantic phase, a number of tasks are accomplished.
   - The identifiers are resolved to the correct parameter, local variable, attribute, shared, constant or routine.
   - Each `debug`- or `assert`-statement object is correctly marked to indicate whether it is to be turned on or not in the final program.
   - The type of each expression is computed. This is one of the main side-effects of invoking the semantic check operation on an expression.
   - Type-checking is performed for all expressions. Any conformance that is not absolutely safe and may need runtime checks (Section 3.7), is recorded in the expression object whose type has to be checked during runtime.
   - Record the list of local variables used by a routine. Local variables with the same name are distinguished by an attribute with positive integer suffix.
   - Mark features which might be used, so that during code generation, we can avoid generating code for those routines which are definitely not used (Section 3.5.1).
   - Verify that the LHS of assignment are assignable expressions; these include local variables, parameters, object attributes, shared features, array access expressions, but exclude constant features and routine calls.

9. **Check usage of @-operators.** The pSather grammar allows expressions of the form such as "`<identifier> @ <expr>`" to be accepted. But `<identifier>` may refer to variables such as parameters; such uses of @-operator are invalid. With the identifiers now resolved, during this phase, we recursively traverse the abstract syntax trees and make sure that the @-operator are used only with valid expressions (e.g. routine calls). The @-operator is valid only in the following contexts.
   - Predefined operations e.g. `new`, `copy`, `extend`

[2]This atomicity information is used only in the Sather compiler but not in the pSather compiler since there is no garbage collection in the pSather implementation.

- User-defined routine calls e.g. `insert(x)`, `list.insert(x)`, `LIST::insert(x)`
- Access to shared and constant features[3]

10. **Miscellaneous semantic checks.** This phase performs some miscellaneous checks on the use of pSather language constructs and collects relevant information for later code generation.

    - We check that `break` statements are allowed only within `loop`, `lock` statements and `then` branch of `try` statements. The `break` statement gets a goto tag associated with the end of its enclosed statement.
    - We also have to take care of `break` statements enclosed within (possibly more than one) `cobegin-end` statements.

      loop
      cobegin ...**break**; ...**end**;
      end;

      When the `break` statement is executed, it transfers control to the first statement after the loop but according to the semantics of the `cobegin-end` statement, execution should not proceed until after the all the threads created within the (dynamic) scope of `cobegin-end` have terminated. Therefore we record the number of `cobegin`'s with which a `break` statement has to synchronize when it transfers control. The counter for `break` statements counts the number of `cobegin`'s within the `break`'s enclosed statement (e.g. loop). During code generation, each `break` statement object generates code to synchronize with the correct number of `cobegin`'s.
    - A similar counter for the number of `cobegin`'s has to be kept for `return` statements. The counter is different because a `return` statement transfers control out of a routine. The counter for a `return` statement therefore counts the number of `cobegin`'s which syntactically enclose the specific `return` statement.
    - When a `break` statement transfers control out of a locking statement, the lock status of gates in that locking statement are restored to their previous values. Since a `break` statement knows its enclosing statement, it has information to generate the correct code.

      A `return` statement is not associated with any enclosing statement but it also has to deal with restoring the lock status of gates. The gates are those in all locking statements which enclose the `return` statement. During the top-down traversal of the abstract syntax trees, the `GATE` expressions of a locking statement are inserted (deleted) into a stack just before (after) the traversal enters into (exits from) the locking statement's body. When a `return` statement is encountered, it records the expressions found in the stack. From these expressions, a `return` statement is able to generate correct code to restore lock status of gates before exiting from a routine.

[3] Shared and constant features are allocated on a per-cluster basis. The @-operator allows the programmer to access shared and constant features on different clusters.

An `unlock` statement ("`unlock <gate-expr>;`") also records the `GATE` expressions in the stack. It uses this information to generate code which checks at execution time that `<gate-expr>` evaluates to a gate from any of the enclosing locking statements. (Sections 2.3.4 and 3.4.2 describe the semantics and implementation of `unlock` statement respectively.)

- A piece of information that is needed in a few optimization phases is whether a routine is *suspendable*. A routine `r` is *suspendable* if execution of the routine may cause it to be suspended (e.g. via an operation on gate). The point of suspension may be within `r` or in some other routines called directly or indirectly by `r`. A non-suspendable routine allows for certains optimizations because it guarantees that no state needs to be saved during its execution. Therefore in some cases, thread/subthread creation and its associated overhead can be eliminated.

  This phase decides whether a routine `r` is *directly suspendable* (i.e. there is a potential point of suspension within `r`). This information, together with a call graph built in a later phase, allows the compiler to determine if a routine is suspendable.

11. **Process program pragmas.** Although a compiler supporting full optimizations is outside the scope of this work, we do want to show that pSather code can demonstrate good performance. Furthermore some optimizations are obvious to the programmer but not to the compiler. Therefore one of our first-cut efforts in optimizing pSather programs is to allow pragmas to be given for routines in different classes. An example is to decide which routines can be inlined.

```
class_pragma MATH is
  log:INLINE is end;
end;
```

Since some of this pragma information is directly used in later optimizations, we process the program pragmas as the first of a sequence of optimization phases.

These pragmas are given in pragma files separate from the pSather source files. They were read in at the same stage as the pSather source code and a hash table of class pragma objects constructed. In this phase, for each class C, we look up the hash table for any class pragma object CP with the same name. If one exists, the class pragma object goes through each feature F_i in C and records any pragmas for F_i. There are two possible pragmas for routines: `INLINE` and/or `NO_POLL`. Inlining is described in Section 3.5.3; the meaning of `NO_POLL` is explained when we describe the CM-5 implementation (Section 3.3).

12. **Build call graph.** A call graph is needed to identify suspendable routines, prevent cycles in routine inlining and is useful if further optimizations in the compiler are attempted. The compiler therefore builds a call graph before any optimization is attempted. For each routine `r`, the compiler records the routines which are invoked from within `r`. To ensure correctness,

when a routine `r` is invoked on a dispatched variable (e.g. of `$POLYGON` type), the callees include not only the routine `POLYGON::r`, but also `<D>::r` where `D` is any descendent of `POLYGON`.

A way of viewing a call graph is as a pair (<initial routine>, <graph>) where <initial routine> is the first routine r_0 that starts executing and <graph> is a graph of routines reachable from r_0. In a sequential program r_0 is unique and so we usually identify a call graph with just <graph>. In a parallel language program it is useful to distinguish different initial routines for different threads. For example, if the compiler wants to relax the ordering of updates of object attributes and maintain sequential consistency, it would be useful to know the set of attributes accessed by different threads. Thus our "call graph" also includes a set of initial routines of forked threads.

13. **Perform inlining.** The inlining phase interacts with several other optimizations later. A more detailed description is given in Section 3.5.3.

14. **Recording usage of `near` variables.** We have described a `with-near` statement in Section 2.5.1. It allows a programmer to assert that a list of variables hold object references which are local with respect to the executing thread. With this knowledge, the compiler is able to optimize access to object attributes on a distributed memory machine. We delay description of this optimization and its interaction with inlining to Section 3.5.5.

15. **Perform in/out analysis on variables.** The intention of this analysis is to compute, in a conservative manner, whether the values of expressions exist before and/or after the execution of a routine. By combining this information with inlined procedures, we were able to identify temporary objects whose lifetimes are within a routine. The runtime controls allocation and deallocation of these temporary objects, and reduces memory leakage. Although this does not obviate the need for garbage collection, it allows us to postpone the study of distributed garbage collection so that it can be researched more thoroughly later. The analysis of object lifetime is described in more details in Section 3.5.4.

16. **Identify immutable attributes.** On a distributed machine, it is important that objects are local with respect to a thread for efficient access. But sometimes we do need to read or write attributes of remote objects. Caching is a standard technique in hardware/software to speed up reads. So one question that arise is: can the compiler identify variables whose values can be cached?

    Consider an attribute `attr` (in class `C1`) which is only updated when a `C1` object is created. Thereafter we only read the value of `attr` from any `C1` object. It would appear that the values of such *immutable* attributes are ideal candidates for caching.

    This phase identifies such immutable attributes and is described in Section 3.5.6. Although this information is not used in the generated code (reason in Section 3.5.6), we list this phase here to point out that identifying immutable attributes depends the results of earlier in/out analysis.

17. **Code generation.** The last phase performs code generation. In Section 3.3, we describe some of the issues that arise when generating code for a distributed memory machine, and our solutions.

3.2 General Components of Runtime Support

The compilation described above is largely independent of the machine architecture.[4] But this is not true of the runtime implementation. We therefore briefly describe the CM-5 here.

A CM-5 multiprocessor consists of many general-purpose processing nodes (each a Sparc processor) which are divided into partitions. Each partition has a control processor which serves as the partition manager. We currently use a 64-processor machine, which is divided into two partitions. Two front-ends to the CM-5 serves as partition managers. The machine is also usable as a 64-processor partition. Because CM-5 has one processor per cluster, we will use "processor" and "cluster" interchangeably.

This section describes the general components of the runtime system. The runtime system is written in C because it is a common programming language and supported on many platforms. It is hoped that using C will make it easier to port the runtime system to different multiprocessors. We would like to acknowledge that the pSather runtime relies on several sources. These include (a) Sather runtime system, (b) the FastThreads package [12], and in the CM-5 implementation, (c) the CMAM active message package [226].

We describe the functionalities of the major components in the runtime system and how they are inter-related.

Active Message Layer.

In the CM-5 implementation, one of the most fundamental layers on which most other components depend is the active message library built by von Eicken et al. [226] at University of California at Berkeley. This layer is not necessary in a shared memory implementation (e.g. Sequent). The active message layer is used to build remote operations, including remote reads/writes, remote routine calls etc.

Basic Thread/Synchronization Support.

This component (originally built by Jeff Bilmes) consists of assembly routines which perform functions that are machine architecture dependent. The functions include (a) the code that

[4] If the machine supports shared-memory, we do make use of that fact to generate simpler code.

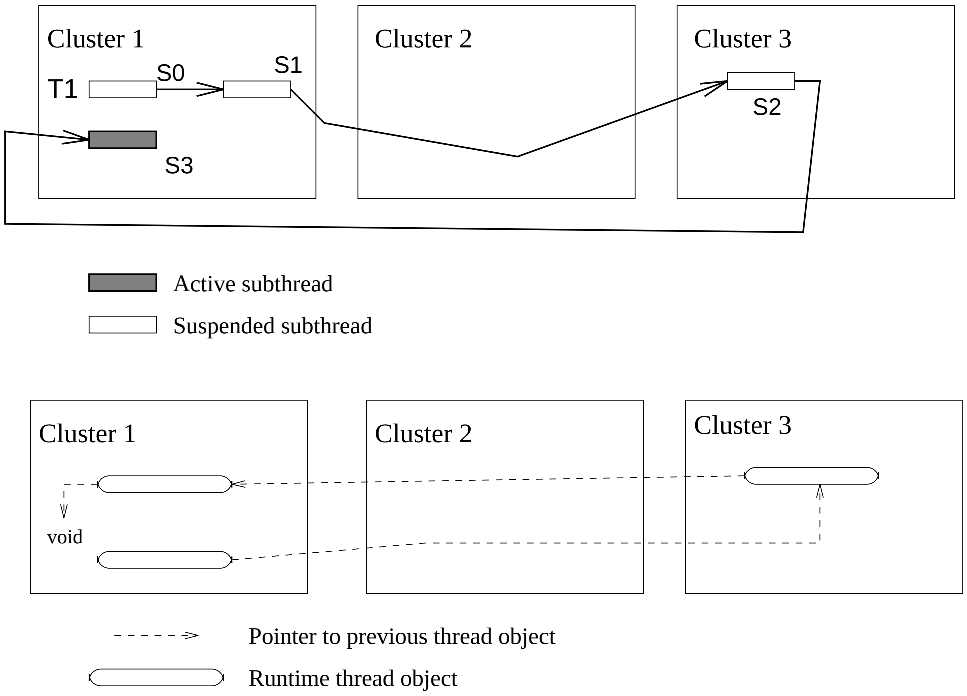


Figure 3.1: Linking of thread objects which make up a user-level thread.

performs context switching for threads and (b) routines that perform atomic locks and unlocks.

Thread Support.

This component manages the thread objects. We want to point out that thread objects are just an implementation mechanism and are different from the notions of subthreads and user-level threads (Section 2.4.3). They are related in the following ways. A thread object (plus stack) may be used to implement multiple subthreads and a user-level thread is made up of one or more thread objects depending on whether its execution spans one or more clusters. For example, Figure 3.1 shows a single thread T1 with four subthreads S0 – S3, implemented as three thread objects. This is however a simplified view because for efficiency reason, there might be times when it is safe to allow multiple user-level threads to share a thread object. (Section 3.3.2 describes such a situation when making a remote call on a non-suspendable routine.)

The thread support component contains routines to create new threads, to perform context-switching, to resume threads, to display the runtime trace stack and to clean up after a thread has completed. The cleanup after thread completion performs the following actions.

- If a thread is associated with a gate, the status of the gate is updated. The result (if any) of the thread is enqueued in the gate. If runtime checks (described in more details in Section 3.7) are enabled, the cleanup routine may have to check whether the thread result violates type safety.

- If a thread is created within the (dynamic) scope of a `cobegin-end` statement, when it terminates, the thread counter of the closest enclosing `cobegin-end` object has to be decremented. (Section 3.4.4 describes the implementation of the `cobegin-end` statement.)

In the CM-5 implementation most of a thread's state is local (e.g. runtime trace stack, thread stack). A thread however does know about objects which may be remote. The thread routines thus use CMAM to handle operations on such objects (if they are located on a remote cluster). A first example is the gate to which a forked thread is associated ("`m :- r @ cluster_id;`"). The routine `r` executes at `cluster_id`. When `r` terminates, and `m` is not located on `cluster_id`, a message is sent to `m`'s cluster to update `m`. A second example is related to the `cobegin-end` statement.

```
cobegin
  :- r @ cluster_id;
end;
```

When `r` terminates, a message to decrement the `cobegin-end`'s thread counter may have to be sent back to the cluster on which the `cobegin-end` statement is executed.

The thread support component also uses routines in a remote operations component (described next). For example the exit routine (that is executed by the main thread on every active cluster before program termination) includes code to print any statistics collected during program execution (e.g. number of reads/writes of attributes). On the CM-5, because the statistics are scattered on different processors, processor 0 relies on remote read operations to gather and reduce these statistics.

Remote Operations.

On a distributed memory multiprocessor (without shared virtual memory support), additional runtime operations are needed to emulate the shared address space in pSather programming semantics. CMAM is useful but not sufficient to provide all the language level operations. Thus an additional runtime component is built on top of CMAM to implement operations that conform to the pSather language semantics. For example when a remote procedure is called, it is not sufficient to send an active message. The current thread must remain suspended until the remote procedure finishes. This additional layer therefore includes (a) the creation of subthreads when a thread shifts its locus of control from one cluster to another, (b) a hash table to keep track of threads which are suspended when making remote calls, (c) broadcast, gather and scatter operations on shared attributes, and (d) reads and writes of remote locations.

This component calls thread support routines to create remote subthreads, to perform context switches, and to suspend (resume) local threads before (after) invoking a remote operation.

```
while (TRUE) {
  if (no more work) { return; }
  <Get thread from ready queue>
  <Execute thread>
}
```

Figure 3.2: Process scheduler loop.

Support for Predefined Operations.

This component consists of runtime routines used in the sequential language core. It supports various predefined object operations e.g. object creation (`new`), copying (`copy`), dispatch routines to locate features of objects or sizes of arrays. In a shared memory implementation, these routines rely only on C library functions (e.g. memory allocation). For a distributed memory machine (without distributed shared memory support), certain operations (e.g. copying to a remote cluster) have to make use of CMAM.

Parallel Execution Environment.

In the cluster model, there are two levels of initialization before a program starts executing. The first consists of creating a parent process on each cluster. This parent process initializes the resources shared by all processors within a cluster. An example would be the initialization required by a garbage collector.[5]

After initializing the common resources, the parent process then creates a process[6] for each processor within the cluster.

The parallel execution environment component contains the routines executed by the processes. Each process has its own queue of active threads, and processes within a cluster also share a common queue. A process executes a scheduling loop (Figure 3.2) that repeatedly checks the private and common queues, retrieves any active thread from the queues and starts running it. This is repeated until every cluster runs out of active threads; all the processes then terminate together.

Runtime Storage Management.

The runtime support has exact knowledge of the size of runtime objects (e.g. threads, stacks, queues for suspended threads in gates), and of how/when they are allocated and freed. Therefore, instead of relying on general memory allocation and deallocation routines, there is a component that maintains pools of various kinds of runtime objects. For each kind of object, there is a pool allocated

[5] Since a cluster has its own address space, the memory managed by a garbage collector would probably be handled on a per-cluster basis.

[6] Each such process is associated with a thread that can be scheduled/suspended just like any other user-level thread and that is referred to as a *processor thread*.

to each processor and a common pool allocated to each cluster. When a local pool becomes too large, some of its storage is returned to the common pool; this prevents a processor from hogging large amounts of unused memory. (Since CM-5 has only one processor per cluster, only a single local pool is needed.) The runtime pools help to reduce memory fragmentation and improve program performance.

Blocking Lock.

There is a support component to provide blocking locks. These are used solely as internal locks for gates. The reason that we cannot use spinlocks to protect gates is because there can be multiple threads on one processor. Using spinlocks may result in deadlock, because the threads are not preemptable.

Spinlock.

Spinlocks are used to protect any resources shared by processors within a cluster because the critical sections of shared resource are short. An example of a shared resource is a shared pool of thread objects.

In the current CM-5 implementation, however, spinlocks are not needed. This is because each cluster in CM-5 has only one processor and a thread that does not poll the network is not interruptible by the arrival of messages. Thus a sequence of instructions that neither polls nor performs context switching is guaranteed to be a protected section. In such a case, spinlocks can be avoided because no race conditions exist. Instead of eliminating the code for spinlock acquisition and release, such code is enclosed within conditional compilation statements for C preprocessors, because we want to retain essential code that may be needed for different architectures.

Synchronization Operations.

The `GATE` operations and other synchronization constructs (e.g. `lock`, `cobegin-end` statements) are translated into runtime routines. In some cases, to reduce runtime overhead, certain "runtime routines" are in fact implemented as C macros. For example, a header file contains the macros needed for executing `cobegin-end`'s. Although the number of `GATE{T}` operations is small, the runtime has to include specialized support for different possible instantiations of type parameter `T` (e.g. `GATE{CHAR}`, `GATE{$OB}`). The amount of C runtime code for `GATE` operations is therefore large. Using the observations that (a) for any one program, only a few of these operations are used and (b) many `GATE` operations are independent, the pSather compiler generates a makefile that includes only used `GATE` operations in the final executable. This selective runtime code inclusion reduces the executable's code size. (Other strategies to reduce code size are detailed in Section 3.5.1.)

3.3 CM-5 Implementation of pSather

In this section, we describe an implementation of pSather on CM-5. The implementation strategies can also be used for any distributed memory multiprocessor without distributed shared memory support. The issues discussed will also be relevant for languages that share similarities to pSather's model and semantics.[7]

On the CM-5, all pSather remote operations are implemented by using the CM-5 Active Message (CMAM) library that has been built by von Eicken et al. [226] at University of California at Berkeley. CMAM supports active messages each of which serves as a request to start a routine on a remote processor. In CMAM a remote processor must poll the network to recognize requests.

The following subsections describe how the compiler and runtime implement the various operations in pSather.

3.3.1 Accessing Object Attributes

Because CM-5 does not support a shared address space in hardware, the shared memory model in pSather has to be supported by other means. One possible approach is to alter the operating system, like Wisconsin Wind Tunnel [198] (WWT). WWT effectively simulates shared virtual memory by treating a CM-5 node's memory pages as caches for a virtual target machine. Instead of making each cache block equal to a page, WWT maps multiple smaller cache blocks to a page and uses CM-5's error correcting code (ECC) to mark invalid cache blocks. When the CM-5 SPARC cache loads an invalid block (marked with bad ECC value), a software trap handler then sends messages to other processors to retrieve the missed block.

We have however chosen to support shared memory in the compiler and runtime since the research environment does not permit periods of exclusive access to the machine. We represent far (or long) pointers as 64-bit entities. The first word denotes a cluster id and the second word is an address within the specified cluster. The implementation is a straight-forward one in which each access to a reference object is tested for remote access.[8] Although the language specifies a relaxed consistency model (Section 2.4.4), the prototype implementation waits for acknowledgement of all writes and presents a sequentially consistent model. Section 3.5.5 explains how the compiler can use the `with-near` statement to eliminate pointer tests.

Another implementation issue for attribute access is dynamic dispatching. The costs of reading/writing an attribute are doubled if dispatching is not performed locally (with respect to the object). Section 3.5.2 describes how the current implementation integrates dispatch and access at the object's cluster. Performance figures are given in Section 3.6.

The 64-bit representation for long pointers extend to the runtime which contains routines

[7] An example is the implementation of a dispatch mechanism, commonly found in object-oriented languages, on a distributed memory machine.

[8] Value objects such as `INT`, `CHAR` retain their shared memory representation and do not impose any overhead.

to create new threads, perform remote calls and perform thread scheduling, and execute the (predefined) `GATE` operations. The runtime operations become more complex in a distributed-memory machine, because the data structures for the gates or the threads may be on a remote cluster. Our general strategy here is to use long pointers for some runtime objects, in particular the threads and gates. We however retain short (32-bit) pointers for certain structures in threads/gates which are obviously local.

3.3.2 Remote Routine Calls

The implementation of routine calls is related to our subthread notion described in Section 2.4.3. A subthread (routine call) by default always executes on the same cluster as its caller subthread. It may be executed on a different cluster only when the @-operator is used (e.g. "x.compute @ <cluster_id>;").

Therefore at any instant, when we look at a user-level thread, it logically consists of a stack of subthreads (activation records) $< st_0, st_1, st_2, \ldots, st_{(n-1)} >$ which can be partitioned into contiguous subsequences such that subthreads within a subsequence are located on the same cluster. Because stack frames are a time-efficient mechanism for making routine calls and returns, we implement a subsequence of co-located subthreads as contiguous stack frames on one stack. This contiguity is implemented directly because we use C as the intermediate language.

But the disadvantage is that if a thread's stack is allocated once and non-extensible, this strategy is inefficient in terms of memory usage, because the machine may run out of stack memory for new remote subthreads while large portions of allocated stacks remain unused. There are more memory-efficient stack techniques used in parallel functional language e.g. cactus stack [111] (which maintains the activation records in a tree of linked lists) and meshed stack [138] (which keeps activation records of multiple threads on a processor on the same stack), but our choice of stack technique is restricted by C as an intermediate language. In spite of this shortcoming, the compiler reduces both time costs and the need to allocate a new stack for each remote sequence of subthreads by implementing suspendable and non-suspendable routines differently.

Our description of remote call implementation first outline the difference between suspendable and non-suspendable routines and introduce the notion of an *continuation state* object. Then we describe how operation states maintain a unique identity for each thread as it spans multiple clusters.

Suspendable vs. Non-suspendable Routines

The time costs of a remote subthread come from several sources. They are (a) network latency, (b) context switching costs in order to execute the remote subthread and (c) additional costs in allocating and deallocating thread stacks. The network latency problem is partially obviated with a fast message library like CMAM. So we turn to look at the other overhead costs. Both (b) and (c) can be avoided when we distinguish between invoking a suspendable vs. a non-suspendable routine at a remote cluster. Recall that a routine is said to be *suspendable* if, during execution, it might be suspended via a `GATE` operation or constructs with synchronization (e.g. a `cobegin-end` or a `dist` statement). This property of a routine is computed during phase 10 in the compiler (Section 3.1).

On top of CMAM, we have built support routines that enable a compiled pSather program to call suspendable and non-suspendable remote routines. Because our implementation allocates a stack for co-located subthreads, when a remote suspendable routine r_{susp} is invoked on processor p, a new runtime thread object and stack has to be allocated on p. The stack object is used to store states of subthreads when r_{susp} executes on p. The thread object is conceptually an extension of the caller thread on the originating processor. Its purpose is to ensure that if r_{susp} actually suspends, only the r_{susp}-related subthreads are saved and suspended. We can improve the performance of remote suspendable calls by carefully reusing the thread and stack.

When a message for r_{susp} arrives on p, we do not allocate a thread immediately. Instead p allocates an *continuation state* (or *cont-state*) object to store r_{susp} and the argument values. The cont-state object is stored in a `susp_op_queue` queue. When p is free to schedule other threads, it checks `susp_op_queue`; if there is a cont-state, p then allocates a new thread and stack that executes the `Susp_Op_Loop` routine:

```
while (TRUE) {
  while (susp_op_queue is not empty) {
    s ← Dequeue(susp_op_queue);
    Exec_Susp(s);
  }
  Poll network
  if (ready queue is not empty) { return; }
  /* This allows p to schedule other user-level threads.  */
}
```

The implementation ensures that cont-states from `susp_op_queue` are only dequeued and executed within this code, which in turn is only executed by a new thread. Thus activation records for remote suspendable calls are kept on a separate stack. The performance is improved because if there are multiple cont-states in `susp_op_queue`, a thread can start immediately on another remote suspendable call after it has completed one; there is no need to allocate a new thread and stack. If a remote call suspends (during `Exec_Susp(s)`) and `susp_op_queue` is not empty, this "demand" will cause p to allocate new threads and stacks.

When a non-suspendable routine call $\mathsf{r}_{non-susp}$ is invoked on processor p, p does not need to allocate a new thread and stack at all. There are two possible ways to schedule $\mathsf{r}_{non-susp}$. In the first way, p can start executing $\mathsf{r}_{non-susp}$ immediately. In this case, if p is executing a user-level thread T when the message to invoke $\mathsf{r}_{non-susp}$ arrives, the result will be as if the current thread T on p is preempted by $\mathsf{r}_{non-susp}$. This is consistent with the pSather semantics that allows the runtime to preempt or not preempt threads.

```
while (TRUE) {
  if (no more work) { return; }
  while (non_susp_op_queue is not empty) {
    s ← Dequeue(non_susp_op_queue);
  Exec_Non_Susp(s);
  }
  if (susp_op_queue is empty) {
    t ← new thread for "Susp_Op_Loop";
    Context_Switch(t);
  }
  <Get thread from ready queue.>
  <Execute thread.>
}
```

Figure 3.3: A more refined process scheduler loop.

In the second way, when a request to execute $\mathsf{r}_{non-susp}$ arrives, p allocates a cont-state object for $\mathsf{r}_{non-susp}$.[9] The cont-state object which stores the argument values of $\mathsf{r}_{non-susp}$ and address of routine $\mathsf{r}_{non-susp}$, is queued in a **non_susp_op_queue** (separate from cont-states for suspendable calls). Figure 3.3 refines the scheduler in Figure 3.2 by taking into account the management of cont-states. A processor p checks for queued cont-states when (a) the user-level thread which it is executing suspends or terminates, and when (b) p's active thread queue is empty but global program termination is not detected. If any queued cont-state is found, its routine $\mathsf{r}_{non-susp}$ can be invoked directly (and the effect is as if the processor thread is suspended and $\mathsf{r}_{non-susp}$ is activated). Because $\mathsf{r}_{non-susp}$ is guaranteed not to suspend and threads are non-preemptable, there is never any need to save any thread state. Thus $\mathsf{r}_{non-susp}$'s subthreads are allocated and deallocated directly on the stack of p's processor thread. The processor thread is always resumed if $\mathsf{r}_{non-susp}$ terminates. (If $\mathsf{r}_{non-susp}$ executes indefinitely without synchronization, it is correct for p's processor thread to remain descheduled indefinitely since pSather does not guarantee threads to be preemptable.)

Maintaining Unique Thread Identity

A request which arrives at a processor contains the routine to be invoked and the argument values. In order to retain the identity of the caller thread, it must also contain other information.

Unique thread id. Each thread is identified by a unique global identifier. When a new thread is forked, it gets a new identifier. When a thread T on a processor p calls a remote routine on another processor q, the identity of T has to be preserved on q while the remote call is executed. This is necessary for program correctness because gates are locked based on thread id independent of the thread's location. Therefore when executing a remote call, only the

[9] Both ways are implemented. The second is the default though Section 3.6 shows that it is more efficient for p to start executing $\mathsf{r}_{non-susp}$ immediately than to use a cont-state.

caller thread's unique id is visible. There are two other pieces of information which can reveal a thread's identity.

Long pointer to its previous subthreads. Figure 3.1 (a) (in Section 3.2) shows a possible stack of subthreads distributed on different clusters. Since contiguous subthreads on the same cluster share a thread object, the back pointer is actually from one thread object to the previous one (Figure 3.1 (b)). This pointer allows a thread to retrace its stack of subthreads which may span multiple clusters. This is useful for printing out a trace stack when a runtime error is encountered. A thread's unique id is independent of its subthreads' location and therefore insufficient for this trace back.

Long pointer to a cobegin-end record (if any). The need for this pointer is best demonstrated by an example. Suppose there is a deferred assignment statement in a `cobegin-end` statement.

```
cobegin
  :- f; end;
```

When the `cobegin` is executed, the current thread T gets a new cobegin-end record R.[10] When T1 (thread for `f`) is forked, R's counter must be incremented (to track T1 implicitly) because T cannot proceed beyond its cobegin-end statement until T1 has terminated. R must also be incremented by all other threads created directly or indirectly by the execution of `f`. Suppose the definitions of `f` and `g` are as follows.

```
f is g @ i end;
g is :- h end;
```

During `g`'s execution, when another new thread T2 is forked (for `h`), R's counter must again be incremented. The remote thread on cluster `i` (that serves as a remote extension of T1) executing `g` must therefore get a long pointer to R from its caller thread T1.

More generally, whenever a new thread is forked, we have to check for any cobegin-end with which this thread has to synchronize when it terminates. The pointer to the latest cobegin-end record (if any) is stored in the context of the parent thread and updated whenever the parent thread encounters any `cobegin` statement. This pointer to latest cobegin-end record must therefore be propagated as a thread spans across multiple clusters.

There are two other values which can reveal the identity of a thread. They are a *clear-level* counter and the gate (if any) associated with the thread when it is forked. These values can reveal a thread's identity because they are used to implement `clear` operation (Section 2.3.3) on gates.

The `clear` operation has a simple implementation. Every gate has a *clear-level* counter which is incremented each time `clear` is called. A thread also has a *clear-level* counter. When a thread is forked in a deferred assignment with a gate on the left-hand-side, its clear-level counter is

[10] Section 3.4.4 describes the implementation of `cobegin-end` statement.

```
/* "who" is the current thread.  */
<Keep the fields who->curr_cobegin, who->unique_id,
  and who->prev_thread in local variables.>

who->curr_cobegin = s->cobegin_end;
who->unique_id = s->unique_id;
who->prev_thread = s->prev_thread;
<Execute routine r_{non-susp} with argument values.>

<Restore the fields who->curr_cobegin, who->unique_id,
  and who->prev_thread from local variables.>
```

Figure 3.4: C code of remote non-suspendable call which interrupts executing thread.

initialized using the gate's clear-level counter and stays unchanged. The `CONFIG::clear_request` routine returns true when the thread's counter is smaller than the gate's counter.

There are two ways to get the correct counters independent of a thread's location. The first is to propagate the gate pointer and counter as a thread shifts its locus of execution to a different cluster. A second way is to locate the original thread based on its unique id, and get both gate and thread counters in the original thread. The current implementation of `CONFIG::clear_request` is able to locate the original thread from its location encoded in the thread id, so there is no need to send the counter and gate pointer during a remote call.

Executing Remote Call

After describing the execution information contained in a request (for remote routine call), we now examine how remote calls are executed. The first case is when a non-suspendable routine is allowed to preempt the executing thread. The compiler generates a remote version of each user-level routine $r_{non-susp}$. This remote routine remote_$r_{non-susp}$ (Figure 3.4) is activated immediately when the request arrives. It stores the thread-unique information in local variables, executes $r_{non-susp}$ and then restores the thread-unique information.

The other case is when a cont-state is allocated to store the execution information. This case applies to both suspendable routines, and non-suspendable routines which do not preempt the executing thread. The execution of a remote suspendable (non-suspendable) call is handled by the runtime routine `Exec_Susp` (`Exec_Non_Susp`).

When executing a non-suspendable routine (`Exec_Non_Susp`, Figure 3.5 (a)), the cont-state is pushed into a per-processor stack. The runtime maintains the invariance that if the stack is not empty, the top cont-state is exactly the one being executed. We record the cont-state so that the runtime can find the correct thread-unique fields. For example, to get the current thread id, if the stack is not empty, the id is retrieved from the top cont-state; otherwise, the id is retrieved from the thread. This works correctly because by definition, a non-suspendable routine never suspends,

```
Push_Op_State(s, exec_op_state_stack);
Call(<numargs>, <remote routine>, <argument list>);
Pop(exec_op_state_stack);
```

a. Definition of `Exec_Non_Susp`.

```
/* "who" is the current thread.  */
<Keep the fields who->curr_cobegin, who->unique_id,
  and who->prev_thread in local variables.>

who->curr_cobegin = s->cobegin_end;
who->unique_id = s->unique_id;
who->prev_thread = s->prev_thread;
Call(<numargs>, <remote routine>, <argument list>);

<Restore the fields who->curr_cobegin, who->unique_id,
  and who->prev_thread from local variables.>
```

b. Definition of `Exec_Susp`.

Figure 3.5: Executing remote call using cont-state `s`.

and as long as the remote call executes, it is possible to get all the essential information of the caller thread from the top cont-state. To the user, the thread's identity stays the same even though the runtime thread objects are different on different clusters.

The `call` routine invokes the remote routine. When the remote routine terminates, the cont-state is removed from the stack. The runtime does not need to send any reply because the compiler generates code to do this in the remote routine.

For suspendable routines (Figure 3.5(b)), the thread object's fields must be updated with the caller thread's information. If the cont-state-on-stack strategy were used, the stack must be updated everytime a thread suspends. This adds more overhead to thread suspension; the context-switch routine is less modular since it has to update global (per-processor) information.

Maintaining Routine Call Semantics

After discussing on the actions at the callee during a remote call, we now look at what happens at the caller. Whenever a routine is called on a remote cluster, according to the usual semantics of a routine call, the calling thread cannot proceed until a reply (possibly with result) is received from the remote processor. This delay is necessary to guarantee sequential semantics of routine calls.

While the calling thread waits, the "caller" processor need not be idle, because it can switch to a different thread that is ready for execution. It may also poll the network for a while before switching thread. When remote calls are short and context switch costs are high, polling may allow

```
pid ← <loc-expr>
if (pid is local) then
  <Call routine locally.>
else
  <Pack argument values into buffer.>
  <Send a message to pid.>
  <Suspend until result/ack is received.>
end;
```

Figure 3.6: Compiler-generated code to do remote call.

a processor to avoid switching threads.[11]

Compiler vs. Runtime Implementation

When a processor p sends an active message to invoke a procedure r on a remote processor q, q has to send an acknowledgement (possibly with result) back to p when r terminates. The sending and receipt of messages can be handled in two ways.

- The first way is to use runtime support. That is, the calling thread p uses a runtime routine that packs the arguments and sends the message. At the remote processor, a runtime support routine unpacks the message and invokes the routine r. When r finishes, the support routine returns the result and acknowledgement to p.

- A more efficient way is for the compiler to generate code that packs and unpacks the messages directly. A translation of the expression "<expr> @ <loc-expr>" at the caller is given by Figure 3.6. At the remote processor, a remote-handler of routine r unpacks the arguments and stores the values in an cont-state. Eventually the scheduler picks up the cont-state and executes the remote version of r which also handles the reply. We therefore need different versions of r to handle both local and remote calls. For each routine r, the compiler generates (a) a local version, (b) a remote-handler that unpacks the arguments and (c) a remote version of r.

The second strategy of generating a local and remote version for each routine r increases the code size. In Section 3.5.1, we describe how to avoid generating unused remote routines and reduce the code size of a remote handler. Section 3.6 shows that the performance of remote calls improves with compiler-generated handlers at both sender and receiver.

Dispatched Remote Calls

[11] In the current version of the runtime system, the default is to switch threads immediately. We have also implemented an alternative runtime that polls until a reply is received. The latter may result in deadlock, but allows us to measure performance gains with polling.

	100	10000	30000
Number of routine calls	16308	7795459	27438616
Number of dispatched remote calls	0	0	0

Table 3.1: Statistics on percentage of dispatched remote calls in a fast multipole N-body program on 100, 10000 and 30000 input points executing on 32 processors (CM-5).

```
1 :        tp ← TYPE_(x); /* Possible remote access.  */
2 :        if (tp != cached_type) {
3 :        cached_type ← tp;
4 :        cached_disp_routine ←
5 :           Get_Dispatch_(cached_type,<routine name>);
6 :        }
7 :        if (cid is local) {
8 :           <Call local version of "r" in cached_disp_routine>
9 :        }
10:        else {
11:           remote_rout ← TRANSLATE_REMOTE_(cached_disp_routine);
12:           <Call remote version of "r" in remote_rout>
13:        }
```

Figure 3.7: Current implementation of dispatched remote call (e.g. "`x.r @ cid;`").

The dynamic dispatching capability in object-oriented systems introduces a slight complication in remote calls. The situation is when we have "`x.r @ cid;`" where `x` is a dispatched variable. The current implementation translates it as in Figure 3.7.

- The object's type is first checked against a cached type (lines 1–2).
- If the values are not equal, the cache is updated with the new type and local routine (lines 2–6).
- If the placement expression evaluates to a local cluster id, then the local routine can be used (lines 7–8).
- Otherwise we have to translate to the remote version of the routine (line 11), and perform a remote call using it instead (line 12).

There are two performance penalties here. The first is that reading the type of an object may involve a remote access (line 1). The second is the translation to the address of the remote routine (line 11).

The first cannot simply be solved by integrating dispatch and routine call at the object's cluster. This is because unlike attribute access when it is always best to do dispatching and read/write at the object's location, the location of a remote call is independent of an object's location.

```
1 :      if (cid is local) {
2 :         tp ← TYPE_(x); /* Possible remote access.  */
3 :         if (tp != cached_type) {
4 :         cached_type ← tp;
5 :         cached_disp_routine ←
6 :            Get_Dispatch_(cached_type,<routine name>);
7 :         }
8 :         <Call local version of "r" in cached_disp_routine>
9 :      }
10:      else {
11:         if (x is local) {
12:            <Dispatch locally; call remote routine directly.>
13:         } else {
14:            <Call another remote routine that do dispatching,
15:               then execute the correct remote routine.>
16:         }}
```

Figure 3.8: Suggested alternative implementation of dispatched remote call (e.g. "`x.r @ cid;`").

The second cost is a result of our earlier decision to have a local and remote version of each routine. But this is not a concern at the current stage of our work because our applications (Chapter 4) have few dispatched calls and even fewer dispatched remote calls. In fact, in a fast multipole N-body program which has the most dispatched variables among our applications, there are no dispatched remote calls at all (Table 3.1).

Here we suggest an alternative strategy which may help to reduce both costs (Figure 3.8).

- We first check whether the call is local or remote (line 1).
- If the call is local, then the usual dispatch mechanism kicks into action (lines 2–8). Note that if the object is remote, a remote read (to get the object's type) must be performed because the language semantics dictate that the routine is executed locally.
- If the call is remote and the object is local, we can find out exactly which remote routine is to be called (line 12). Otherwise, we call a remote routine to do dispatching (line 14–15) at the remote cluster. In the latter case, if the object is not located on the remote cluster, a remote read (to get the object's type) is still necessary.

With the current implementation of objects and pointers, it appears impossible to completely integrate dispatch and routine call.

We have discussed implementation alternatives of remote calls (e.g. suspendable vs. non-suspendable remote calls, polling vs. context switching while waiting for routine completion etc). The performance results will be discussed in Section 3.6.

3.3.3 Thread Creation

The main difference between routine calls and thread creation is that in the former, the caller thread has to wait for a reply, while in the latter, the caller thread does not wait and can continue on after the active message is sent. Thus, the implementation of simple thread creation (i.e. "`:- r @ cluster_id`") resembles that of remote routine calls using cont-states[12] (Section 3.3.2) except for the following changes. (1) When a thread is created, we have to check whether it is within the dynamic scope of a `cobegin-end`. If it is, we increment the corresponding cobegin-end counter. We do not perform this cobegin-end check for remote routine calls. (2) The second difference is in the clean-up action performed when the thread/remote call finishes. In the former case, it decrements the cobegin-end record counter (if any). On the other hand, a remote call that finishes sends a reply to resume the caller thread. (3) In thread creation, the parent thread continues executing immediately after the message is sent; in remote calls, the caller thread is suspended until a reply is received.

Although the implementation strategy using cont-states can also be used for creating a thread which is associated with a gate (i.e. "`g :- r @ cluster_id`"), the prototype implementation uses a less time-efficient strategy because it evolved from the shared-memory implementation on Sequent Symmetry. In the current implementation, when a parent thread executes "`g :- r @ cluster_id`", it performs the following actions.

1. It checks whether the thread is within the dynamic scope of a `cobegin-end`; if it is, the cobegin-end counter is incremented.

2. It checks if the gate is locked; if it is, the parent thread is suspended until the gate is unlocked.

3. The parent thread adds the new thread to `g`'s set of associated threads (Section 2.3.1). (This actually only requires incrementing an internal counter in the gate `g`.)

4. It sends out a message containing the address of routine `r`, any argument values plus other execution information (e.g. a pointer to `g`, a pointer to any active cobegin-end record) to `cluster_id`. After the message is sent out, the parent thread continues execution.

We note that steps 1 and 2–3 may require the parent thread to act on remote cobegin-end record and remote gate respectively.

When the remote processor receives a thread-creation message, it allocates a new thread object, initializes it using information in the message and inserts it into the local scheduler queue. To conserve memory, a new thread object does not get any stack space immediately on creation. Instead we adopt the following strategy: before a thread is executed, we check whether it has a stack. If it does not, a stack is allocated from a per-processor runtime pool. This prevents the following

[12] This means that, like remote routine calls, we also have to distinguish between suspendable and non-suspendable routines for thread creation.

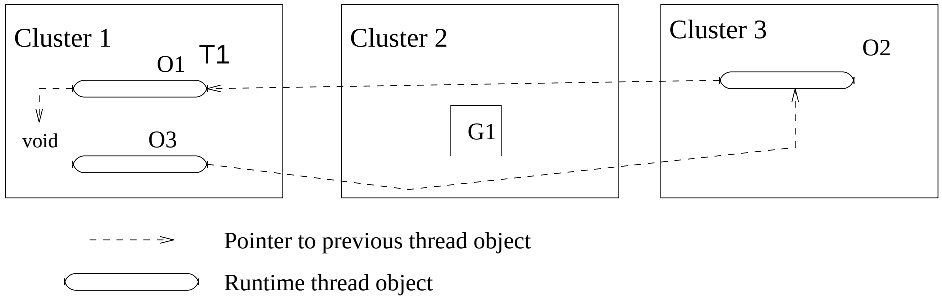


Figure 3.9: Distinguishing thread object id and caller thread id in gate operations.

situation from happening: Suppose we have 1 Mbytes of stack space, and each stack has a fixed size of 100 Kbytes. Furthermore suppose that 11 threads are created, and each thread completes its execution before the next starts executing. If a new thread gets a stack on creation, after creating 10 threads, we will run out of stack space for the 11th thread. But with the current strategy, the (terminated) ith thread's stack is re-allocated to the (i+1)th thread just before the latter starts executing. At any time, only 100 Kbytes are actually used.

When the child thread terminates, it performs the following cleanup actions before returning the stack to the runtime pool.[13]

1. It checks whether the gate is locked; if it is, the terminating child thread is suspended until the gate is unlocked. The result (if any) is enqueued in `g`'s queue of values.

2. If the thread was created within the dynamic scope of a `cobegin-end`, the cobegin-end counter is decremented.

3. The child thread is removed from `g`'s set of associated threads. (This actually only decrements an internal counter in gate `g`.)

We again note that the cleanup actions may involve accessing objects which are remote relative to the child thread. Performance of remote thread creation is discussed in Section 3.6.

3.3.4 `GATE` Operations

The `GATE` operations take a long address containing the cluster number and the address within the cluster. If the gate is local within the current cluster, the specified operation or predicate (e.g. `set`, `lock`, `unlock` etc) is performed as in a shared-memory implementation. If the gate is remote, we again distinguish between suspendable and non-suspendable calls. Non-suspendable calls (e.g. `is_bound` predicate) can be executed using active messages directly.

When executing a remote suspendable gate (e.g. `take`) operation, the caller thread sends an active message to the gate's cluster. The message specifies the remote routine, id of the thread

[13] Note that the stack is still needed as long as the thread is executing; hence the order of the actions is important.

```
while (gate_op_queue is not empty) {
  state ← Dequeue(gate_op_queue);
  Exec_Non_Susp(state);
}
```

Figure 3.10: Code that polls `gate_op_queue`, and executes cont-states for gate operations on behalf of remote processors.

object, id of the caller thread, a pointer to the thread object and a (short) pointer to the gate within the cluster. There is a distinction between the id of the thread object and the id of the caller thread. The former is used to identify which runtime thread object to resume when the gate operation completes. The latter is used when check the gate's lock status (i.e. whether the gate is unlocked or locked by the executing thread). For example, in Figure 3.9, a gate operation on G1 gets the ids of runtime thread object O3 and caller thread T1.

The remote (runtime) routine activated by the message allocates a cont-state at the remote processor. This cont-state is enqueued in `gate_op_queue`, separate from the cont-states of user routines. We explain why cont-states of gate operations must be treated separately to ensure that gate operations make progress. Suppose a user-level thread T1 on cluster $c1$ executes:

```
v := g.take;
```

where `g` references a gate on cluster $c2$. From the programmer's point of view, since the `@`-operator is not used, T1's locus of control has remained on $c1$ and so T1's execution does not depend on the scheduling of threads on $c2$. But since the cont-state is on $c2$ and we cannot use the execution cycles of $c1$'s processors, this means that we have to "steal" some execution cycles from $c2$'s processors to do `g.take` (on behalf of the processors in cluster $c1$), to ensure that the cont-state for `g.take` gets executed and makes progress.

To "steal" $c2$'s processors' cycles, the compiler inserts code to poll `gate_op_queue` in every basic block of the pSather routines.[14] Figure 3.10 shows the code that polls `gate_op_queue`. As long as `gate_op_queue` is not empty, a cont-state is removed and executed using the runtime routine `Exec_Non_Susp` (which is also used to execute cont-states of remote routine calls, Section 3.3.2). This scheduling and execution of gate operation cont-states is independent of the process scheduler loop in Figure 3.3.

Note that the runtime routine `Exec_Non_Susp` is used to execute a gate operation cont-state. We cannot use `Exec_Susp` because the execution of a gate operation cont-state temporarily preempts the executing thread (on $c2$) and "borrows" the use of the thread's stack. Therefore the cont-state must not have any operation that may cause the thread object to suspend.

We must reconcile this non-suspendability requirement with the fact that a gate operation

[14] The compiler also inserts network-polling statements into the generated code (Section 3.3.5).

```
1 :      Get_Internal_Lock_(g);
2 :      if (! Is_Bound_(g)) {
3 :         who->wait_condition_ = WAIT_TO_TAKE_;
4 :         <Suspend current thread and release internal lock.>
5 :      }
6 :      if ((! Unlocked_(g)) && (Lock_Id_(g) != caller_id)) {
7 :         who->wait_condition_ = WAIT_TO_TAKE_;
8 :         <Suspend current thread and release internal lock.>
9 :      }
10:      res ← Deque_Value_(g);
11:      <Either release internal lock, or keep internal lock and
12:         schedule another thread.>
13:      return (res);
```

Figure 3.11: Outline of gate `take` operation.

typically has several possible points where a thread can get suspended. We use the `take` operation as an example to show the possible suspension points in a local gate operation (Figure 3.11) and then explain how a single routine with multiple suspension points can be transformed into multiple non-suspendable routines.

A thread first get the gate's internal lock (line 1) to ensure that gate operations performed by multiple threads execute atomically. This internal lock may suspend the current thread. This is not a spinlock because we want the gate operations to be performed in a FIFO order. Suppose we have:

```
T1, T3:
  v := g.take;

T2:
  g.enqueue(w);
```

and by some other synchronization (not shown), we ensure that T1 executes `g.take` before T2 which in turn executes before T3. By the language definition, T1 should get the value enqueued by T2. When T1 first executes `g.take`, it is suspended because `g`'s queue of values is empty. After T2 executes `g.enqueue(w)`, T1 is resumed, holding `g`'s internal lock.[15] So to ensure FIFO ordering, the internal lock may be held by a waiting thread (in the scheduler queue), like T1. The latter condition in turn implies that if a thread T' cannot successfully get the internal lock, T' must be suspended; otherwise the program may not progress.

After getting the internal lock, if `g`'s queue of values is empty (line 2), the thread is suspended after we set a flag which indicates the reason for its suspension (lines 3–4). Another condition to check is whether the gate is locked by another thread (line 6). If it is, the thread is

[15] If T1 does not hold `g`'s internal lock when it is resumed, T3 may overtake T1, which again violates the language definition.

also suspended (lines 7–8). Finally the thread can remove the value from the front of the queue (line 11) and return it (line 13). Before the routine exits, we have to check whether any of the suspended threads in `g` can be resumed (line 12). (If no thread is resumed, the internal lock is released. Otherwise the internal lock remained locked and is now conceptually held by the resumed thread.)

To explain how a remote gate operation executes the `take` operation in Figure 3.11, the first thing to note is that the execution of any part of the `take` routine is in fact *piggybacked* on another thread. Let us again suppose that a user-level thread T1 executes on cluster *c*1:

```
v := g.take;
```

and `g` references a gate on cluster *c*2. When the runtime system executes `g.take`'s cont-state on *c*2, T1 is actually piggybacked on another thread T2 in *c*2, in the sense that T2 is preempted by T1 and T2's thread object and stack is temporarily borrowed and used by T1. So when T1 suspends (conceptually), we want to return T2's thread object and stack to T2 and resume executing T2. We use a cont-state to represent the continuation of T1 when it is resumed. To split `take` into multiple non-suspendable routines, at each suspension point (lines 1, 4, 8), we transform the code from:

```
L1
<Thread is suspended.>
L2
```

to separate routines:

```
L1
<Allocate cont-state with continuation routine r2 and store
  the value of any variable in L1 also used in L2.>

r2(....)  {
  L2
}
```

The result is shown in Figure 3.12. (We continue to use T1, T2, *c*1, *c*2 to refer to local/remote thread/cluster in the following description.)

The first step is to transform `Get_Internal_Lock_(g)` in line 1 of Figure 3.11 because it contains a suspension point. Lines 1–9 show the routine that is executed at the remote processor on *c*2 when the message for remote `take` operation is received. Since the most efficient CMAM message is limited to 4 words (5 if we include the address of the routine), the pointer to T1's thread object (`thread_obj_ptr`) is a short address. We reconstruct the long pointer from the thread object's id (`sender_id`) and `thread_obj_ptr` (line 1). Next we try to get the internal lock. Unlike the `Get_Internal_Lock_` operation (Figure 3.11), `Test_And_Get_Internal_Lock_` is non-suspending and returns a result on whether it succeeds in getting the internal lock. If it succeeds, then T1 does not suspend and proceeds to the next stage given by `take`$_1$ (line 4). If it does not succeed, the

```
remote_thread_obj_ptr ← Make_Ptr_(sender_id, thread_obj_ptr);
status ← Test_And_Get_Internal_Lock_(g, remote_thread_obj_ptr);
if (status == SUCCESS) {
   take1(sender_id, caller_id, thread_obj_ptr, g);
} else {
   state ← Alloc_And_Store_(take1,
         sender_id, caller_id, thread_obj_ptr, g);
   Insert_(gate_cont_table, remote_thread_obj_ptr, state);
}

/* Definition of ``take1 */
if (! Is_Bound_(g)) {
   remote_thread_obj_ptr ← Make_Ptr_(sender_id, thread_obj_ptr);
   state ← Alloc_And_Store_(take2,
         sender_id, caller_id, thread_obj_ptr, g);
   Insert_(gate_cont_table, remote_thread_obj_ptr, state);
   <Suspend "remote_thread_obj_ptr" in gate "g".>
   Release_Internal_Lock_(g);
} else {
   take2(sender_id, caller_id, thread_obj_ptr, g);
}

/* Definition of ``take2 */
if ((! Unlocked_(g)) && (Lock_Id_(g) != caller_id)) {
   remote_thread_obj_ptr ← Make_Ptr_(sender_id, thread_obj_ptr);
   state ← Alloc_And_Store_(take3,
         sender_id, caller_id, thread_obj_ptr, g);
   Insert_(gate_cont_table, remote_thread_obj_ptr, state);
   <Suspend "remote_thread_obj_ptr" in gate "g".>
   Release_Internal_Lock_(g);
} else {
   take3(sender_id, caller_id, thread_obj_ptr, g);
}

/* Definition of ``take3 */
res ← Deque_Value_(g);
<Send "res" back to caller thread object.>
<Either release internal lock, or keep internal lock and
   schedule another thread.>
```

Figure 3.12: Implementing remote `take` operation as separate non-suspendable routines.

remote thread object pointer (i.e. T1's thread object) is queued in the internal lock (just like the local threads), as part of the `Test_And_Get_Internal_Lock_` operation. So in lines 6–7, we create a cont-state that is to be activated when T1 is (conceptually) resumed. The association between the cont-state and remote thread object pointer is stored in a hash table (line 8).

This setup also requires a change in the runtime routine that resumes a thread waiting in the internal lock. Previously, we simply get a (local) thread object and insert it into the scheduler queue. Now we have to check whether the thread object is local or remote. If it is local, we proceed as before; otherwise, the associated cont-state is retrieved and inserted into **gate_op_queue** (the queue for cont-states of remote gate operations).

The same transformation to the rest of **take** produces the other routines $\texttt{take}_1$, $\texttt{take}_2$ and $\texttt{take}_3$ (Figure 3.12). In every routine, we maintain the invariance that the internal lock is held while the routine is executed. In $\texttt{take}_1$, if the gate is not bound, the same sequence of actions (lines 11–14) — allocating cont-state, associating cont-state with remote thread object pointer — are performed. T1 is now suspended in the gate (lines 15–16), instead of the internal lock.[16] If the gate is bound, we proceed to the next phase by calling $\texttt{take}_2$. The routine $\texttt{take}_2$ is similar to $\texttt{take}_1$ except that we check the lock counter of the gate. Finally $\texttt{take}_3$ removes the value from the front of gate's queue of values, and returns the value to the caller T1.

It is important to note there is no code in Figure 3.12 that may cause T2's thread object (on which T1 is piggybacked) to suspend. There is also no code that polls the network, so T2's thread object cannot be preempted by other remote calls. Each successive routine serves as a continuation for the previous one (e.g. $\texttt{take}_2$ is a continuation of $\texttt{take}_1$). Since the transformation is fairly mechanical, we feel that some variation of continuation-passing style may be useful in improving the compilation strategies. Although it is beyond the scope of this work, Section 5.2.4 (Chapter 5) discusses some issues pertaining to better compilation strategies.

3.3.5 Polling

As already mentioned, a request (from an active message) is only executed, if the receiving processor polls the network. It is therefore not sufficient that a processor p polls the network only if p itself has sent a message. This is because a processor that never sends any active message would not poll the network and a possible request is never served. The compiler therefore introduces additional polling points into the generated code.

In the current implementation, a polling point is simply inserted for every x (pSather) statements where x's default value is 3. x can be changed for different compilations for future experiments, but its value is fixed within a single compilation. The compiler also ensures that there is at least one polling point in each basic block. The frequency of polling points may be altered at compile-time for experimentation.

[16] When a thread is resumed from the gate, we again have to treat local and remote thread objects differently.

```
1 :     da := DIST_MATRIX_BLK_ROW{DOUBLE}::create(size, size);
2 :     b := MATRIX{DOUBLE}::create(size, size);
3 :     dist da as a do
4 :        lb:MATRIX{DOUBLE};
5 :        if b.is_far then lb:=b.copy else lb:=b end;
6 :        with a,lb near
7 :           a.to_multiply(lb);
8 :        end;
9 :     end;
```

Figure 3.13: Distributed matrix multiplication.

It is obvious that removing unneeded polling points will increase efficiency. Since it is sometimes possible for a user to identify routines in which message is neither sent nor received, the compiler allows the user to attach a pragma `NO_POLL` to any routine `r`. Polling statements are removed from routines marked `NO_POLL`.

But in general there is no simple strategy to remove polling points. We use a simple distributed matrix multiplication code (Figure 3.13) to illustrate this point. The algorithm uses a input matrix `da` which is partitioned into blocks of rows. The second input matrix `b` is not distributed but a local copy of `b` is made at every cluster. `MM1` is the code generated using the `with-near` statement to speed up pointer accesses. It would seem that after copies of `b` are made, no more communication is needed and polling points can be removed within the `to_multiply` routine (line 7). `MM2` adds the following code changes: (a) lines 6–8 are moved to a new routine marked as `NO_POLL`, (b) a barrier object is used to synchronize the threads between lines 5 and 6. As shown in the timings table (Table 3.2), `MM2` has a lower 1-processor time than `MM1`. But `MM2`'s 2-processor time did not improve over its 1-processor time. The reason is that removing polling point delays the resumption of threads suspended at the barrier. Suppose the barrier object is located on processor p_0. The definition of barrier does not guarantee that threads on processor p_0 proceed only after threads on other processors pass the barrier. Therefore, a possible scenario is that the thread `T0` on processor p_0 is resumed first after all other threads have reached the barrier. When `T0` starts executing, p_0 no longer polls and no remote operation on the barrier is allowed. The resumptions of threads on other processors are thus delayed.

The solution therefore is to make sure that after the copy phase all threads are resumed before they enter a no-polling state. `MM3` implements such a solution by reorganizing the code into two separate phases – a copy phase and a computation phase, each within its own `dist`-statement – and removing polling statements from the computation phase. `MM3` shows better performance than `MM2`.

The matrix multiplication example shows that removing all unneeded polling points is a tricky task even for the user, not to mention the compiler. Given that polling points must be

	1	2	4	8	16	32
MM1	2.81	1.42	0.73	0.43	0.33	0.38
MM2	1.30	1.31	0.68	0.39	0.37	0.40
MM3	1.30	0.66	0.35	0.23	0.22	0.33

Table 3.2: Time (in seconds) for a pSather program executing on CM-5 without using vector unit, on 100 x 100 matrices.

```
Sort (g1, ..., g<n>)
for i = 1,..., n
  if (lock(g<i>) != SUCCESS) {
    Suspend current thread on g<i>
  }
endfor
```

Figure 3.14: An implementation of `lock` statement.

generated, the next question to ask is whether there exists an "optimal" polling frequency. Increasing the polling frequency obviously adds more overhead but on the other hand, it makes sure that a message is handled quickly, thus allowing other threads to proceed as soon as possible. As an example, when measuring the performance of remote gate operations (Section 3.6), we find that increasing the polling frequency improves the measured timings.

3.4 Implementation of Parallel Constructs

In this section we give non-machine-specific descriptions of the implementation of various pSather extensions.

3.4.1 Lock Mechanism

This section describes the implementation for multiple-gate `lock` statement. We present several solutions and describe whether they implement the required semantics of the `lock` statement (as described in Section 2.3.4). The following discussion assumes that the `lock` statement has the form:

lock `g1, g2, ..., g<n>` **then** ...**end**;

Deadlock

The first criterion of a `lock` statement is that all the gates are acquired atomically. One solution to implement this is to sort the list of gates and acquire them according to the sorted order

```
  Sort (g1, ..., g<n>)
try_again:
  for i = 1, ..., n
    if (lock(g<i>) != SUCCESS) {
      for j = 1,...,i-1
        Release g<j>
      endfor
      Suspend current thread on g<i>
      Goto try_again
      /* Try again when the thread is woken up.  */
    }
  endfor
```

Figure 3.15: One possible implementation of `lock` statement.

(Figure 3.14). This, however, does not satisfy the second criteria for maximal concurrency: a thread that is waiting to lock one or more gates does not have priority over later threads trying to lock the same gate(s).

Suppose we have:

lock `g1` **then** ...**end**; `-- Thread 1`

lock `g2` **then** ...**end**; `-- Thread 2`

lock `g1, g2` **then** ...**end**; `-- Thread 3`

and the ordering is `g1` < `g2`, and thread 2 has acquired `g2`. When thread 3 tries to lock both `g1` and `g2`, it cannot succeed. Next thread 1 starts to execute. According to the semantics of the `lock` statement, since thread 3 did not succeed, `g1` should not have been locked and thread 1 will be able to lock `g1` and proceed. If we simply acquire the locks in sorted order, thread 3 would have locked `g1` and be blocked on trying to lock `g2`. Essentially, thread 3 has priority over thread 1 in locking `g1` and prevents thread 1 from locking `g1`, thus violating our "maximal concurrency" requirement.

Maximal Concurrency

The pseudo-code in Figure 3.14 is therefore elaborated to the form shown in Figure 3.15. By releasing the gates locked so far when not all the gates can be locked, we prevent the unsuccessful thread from gaining priority over other threads. We adopt the solution in Figure 3.15 for our implementation on CM-5, a distributed-memory machine. But the release of all previously locked gates may be inefficient; we discuss this issue next for shared-memory machines.

Efficiency

The locking operations must be done efficiently — on the first try if the thread can suc-

```
  Sort g1, ..., g<n>
try_again:
  for i = 1, ..., n
    if (lock(g<i>) != SUCCESS) {
      Suspend current thread on g<i>
      /* Check thread status when it is woken up.  */
      if (thread notified of preemption) {
        Goto try_again
      else
        Continue
        /* Continue with trying to lock the other
          gates.  */
      }
    }
  endfor
```

Figure 3.16: Another implementation of `lock` statement.

cessfully grab all the gates. Even if a thread cannot successfully grab all the gates, the releasing of all previously locked gates as shown in Figure 3.15 may not be needed. If we have:

lock `g3` **then** ...**end**; `-- Thread 1`

lock `g1, g2, g3` **then** ...**end**; `-- Thread 2`

and suppose thread 1 locks `g3` successfully. When thread 2 tries to lock the gates (assumed in the correct order), it will lock `g1` and `g2`, finds that `g3` is locked and releases the two gates. This release is not absolutely necessary, because in the time interval between thread 2 getting suspended and thread 2 being resumed (when thread 1 unlocks `g3`), no other thread is trying to acquire `g1` or `g2`. It would be more efficient to leave both `g1` and `g2` locked by thread 2.

We therefore further elaborate the code to the form in Figure 3.16.

In the locking code, when a thread (T1) finds that a gate is locked by another thread (T2) which has been suspended in a locking statement, it alters the status of T2, and surreptitiously "steals" the gate. In effect, T1 forces T2 to release the gate (and all others that T2 is currently locking). As a result, when T2 is finally woken up, it has to retry locking all the gates it needs.

On the other hand, if no other thread needs any gate locked by the currently suspend thread T2, none of T2's gates is released. When T2 is waked up, it can continue to try to grab the other gates. Figure 3.16 only shows the code for the thread whose locks might be stolen. Figure 3.17 describes the "lock-stealing" mechanism in more detail.[17] Here we give an argument outline to show that it achieves the desired semantics.

First, we give proofs of some properties of the mechanism.

[17] We only implemented this on Sequent Symmetry, a shared-memory machine.

```
/* "g" denotes the gate to be locked; "locking_thread" is the
  thread that currently locks "g".  We try to steal "g" from
  "locking_thread".  */

if (g is locked and g is not locked by current thread) {
  steal_lock ← 0
  /* "g" is successfully stolen only when the value of
    "steal_lock" is greater than 0.
  if (locking_thread is suspended) {
    /* "g1" is the gate on which the locking thread is suspended.  */
    Grab internal lock of g1.
    if (locking_thread is still suspended on g1) (*1*) {
      (locking_thread is waiting for a lock) (*2*) {

      /* The check at (*1*) is necessary because in the time
        interval between finding out about "g1", and getting
        "g1"'s internal lock, "locking_thread" may have been
        resumed.  At this point, we know that "locking_thread"
        is suspended at a lock statement; hence, current thread
        can proceed to steal "g".

      for each stealable gate g<i> locked by "locking_thread"
        if g<i> = g {
          steal_lock ← 1
          Notify "locking_thread" of preemption.
          Mark g<i> as non-stealable.
          Add 1 to counter.
          /* "counter" keeps track of number of times "g" has
            been locked in current lock statement.
        }
      endfor
      if (counter > 0) {
        if (number of times g has been locked != counter) {
          steal_lock ← 0
          Restore locking_thread preemption status.
          Re-mark g<i> as "stealable" for each g<i> = g.
        else
          Unlock all other gates currently held by "locking_thread",
          and mark them non-stealable.
    } } }
    Release internal lock of g1.
} }
/* Current thread either suspends or locks "g".  */
```

Figure 3.17: Lock preemption mechanism.

Claim 1 *If a thread t1 steals a lock* `g` *from* `locking_thread` *successfully, (in the semantics of* `lock` *statement),* `g` *should have been released by* `locking_thread` *before* `locking_thread` *suspends.*

Proof: Suppose $t1$ succeeds in stealing the lock `g`. From the implementation, we see that `g` must be one of the locks held by `locking_thread` in the current `lock` statement. In addition, `locking_thread` is suspended at a `lock` statement by the checks at (*1*) and (*2*) (in Figure 3.17). Hence `g` should have been released by `locking_thread` by the semantics of the `lock` statement. □

Claim 2 *If a thread t1 steals a lock* `g` *from* `locking_thread` *successfully,* `g` *has been locked by* `locking_thread` *only in the current* `lock` *statement (on which* `locking_thread` *is suspended).*

Proof: Suppose $t1$ succeeds in stealing the lock `g`. From Figure 3.17, we must have `counter` equal to the number of times `g` has been locked. But `counter` keeps track of the number of times `g` has been locked in current `lock` statement. Hence, `g` has been locked by `locking_thread` only in the current `lock` statement. □

Claim 3 *Once a thread t1 succeeds in stealing a lock* `g` *from* `locking_thread`*, all other locks held by* `locking_thread` *in the current suspended* `lock` *statement are restored to their status before the* `lock` *statement.*

Proof: This is obvious from the code in Figure 3.17. We also note that if a gate $g' \neq g$ is restored to the unlocked status, then another thread wanting `g'` can lock `g'` without executing the piece of code in Figure 3.17. If `g'` remains locked, then because it would have been marked as non-stealable, another thread will not be able to steal it.

Claim 4 *If two threads t1 and t2 try to steal the same lock* `g` *from* `locking_thread` *in parallel, exactly one of the threads will succeed, or none of them will succeed.*

Proof: Suppose both $t1$ and $t2$ succeeds in stealing the lock `g`. Then both $t1$ and $t2$ must have found g_{i_1}, g_{i_2} respectively, where $i_1 \neq i_2$ and $g_{i_1} = g_{i_2} = g$. Without loss of generality, suppose $t1$ grabs the internal lock of `g1` before $t2$. Since $g_{i_1} = g_{i_2} = g$, both g_{i_1} and g_{i_2} would have been marked non-stealable by $t1$, and it is not possible for $t2$ to steal `g` again. □

It does not really matter if neither $t1$ nor $t2$ succeeds in locking `g`. If `g` has been previously locked in a successful `lock` statement, then it is obvious that both $t1$ and $t2$ must fail to steal lock `g`. Otherwise, since the speed of execution is unknown, we can always assume that $t1$ and $t2$ are executing at such a speed that they both fail to steal `g` because `g` has not been released.

Suppose we have the following situation:

```
lock g1, g2 then ...end; -- Thread 1
lock g1 then ...end; -- Threads 2, 3
```

Thread 1 could have just locked `g1`. Just before it gets suspended on `g2`, threads 2 and 3 try to lock `g1`. According to the `lock` statement semantics, we can treat `g1` as being not yet released by thread 1 and as a result, both threads 2 and 3 will fail to lock `g1`.

From claims 1 and 2, we guarantee that a thread can only steal a lock `g` if `g` was not locked in a successful `lock` statement, and that `g` should have been released by `locking_thread` which is suspended on the `lock` statement. Furthermore, claim 4 guarantees that exactly one or no thread will succeed in stealing `m`. Claim 3 ensures that a successful steal attempt will make `locking_thread` release all its locks in the current `lock` statement (which is exactly what it should have done before it is suspended).

Intuitively, it may seem that not all locks held by `locking_thread` in the current `lock` statement need to be released. However, this would give rise to the following inconsistent situation:

```
lock g1, g2, g3 then ...end; -- Thread 1

lock g1 then          -- Thread 2
  try g2 then ...
  else
    -- Error, since 'g2' should not be locked
    -- when thread 1 is suspended, and thread 2
    -- can succeed in grabbing 'g1'.
  end;
  ...
end;
```

Suppose thread 1 locks `g1` and `g2` and is suspended on `g3`. Thread 2 then steals the lock `g1`. If `g2` is not released, then we have an inconsistent program state, because thread 1's `lock` statement is supposed to either grab all the locks or none.

Fairness

We can achieve a strong fairness property if the implementation follows what is outlined in Figure 3.14. That is, any thread trying to lock any set of gates will eventually succeed, provided that there is progress towards satisfying the locking condition.

Since we have traded off fairness in favor of maximal concurrency (Section 2.3.4), the shared-memory implementation (Figure 3.16) does not ensure strong fairness. Intuitively, a thread requiring fewer gates is more likely to succeed entering the critical section, than a thread requiring more gates (since the latter are more likely to be "preempted"). We discuss two properties of the current implementations.

- It does not ensure that `lock` statements are always executed in a FIFO order.

  Consider Figure 3.18. When a parallel thread for `f1` (line 8) tries to lock `g1` and then `g2` (lines 3), it fails to lock `g2` and becomes suspended on `g2`. The parent thread then steals `g1` from `f1`. When a second parallel thread for `f2` (line 11) tries to lock `g1` (line 4), it fails immediately

```
1 :     g1, g2:GATE0;
2 :     ...
3 :     f1 is lock g1, g2 then end; end;
4 :     f2 is lock g1, g2 then end; end;
5 :     ...
6 :     main is
7 :        lock g2 then
8 :           :- f1;
9 :           ...
10:           lock g1 then
11:              :- f2; ...
12:              unlock g2;
13:              ...
14:              unlock g1;
15:     end; end; end;
```

Figure 3.18: Non-FIFO locking

and becomes suspended on `g1`. When `g2` is unlocked (line 12), `f1` is resumed, and tries to re-lock `g1` (because it has already been stolen by the parent thread). It fails again and becomes suspended, but now it is enqueued waiting for `g1`, behind `f2` in the queue. Finally, when `g1` is unlocked (line 14), `f2` is resumed and executed, because `f2` is now the first thread in the waiting queue of `g1`.

We note that even though the `f1` executes its `lock` statement before `f2`, when finally both `g1` and `g2` becomes available (after line 14), it is `f2` which succeeds because `f1` has in some sense lost its FIFO position when it first waits at `g2`, then later swaps to the waiting queue of `g1`. This swapping from one waiting queue to another could be repeated several times, delaying `f1` arbitrarily.

- It allows a 'weak' fairness property to hold. If we have the following:

  lock `g3` **then** ...**end**; `-- Thread 1`

  lock `g1, g2, g3` **then** ...**end**; `-- Threads 2, 3`

  Thread 1 succeeds in grabbing `g3`. Thread 2 will be blocked on `g3`. When thread 3 starts executing, thread 2 will be forced to unlock `g1` and `g2`. Thread 3 then gets suspended on `g3` behind thread 2. When `g3` is unlocked, thread 2 will be resumed before thread 3; this time, thread 3 will be forced to unlock `g1` and `g2` to be grabbed by thread 2, which will succeed, provided that in the meanwhile, no other thread has grabbed `g3`.[18]

  This weak fairness property does not prevent starvation, but any thread trying to lock gates will eventually get a chance to execute again, provided that the locking condition is eventually

[18] Even if some other thread has grabbed `g3`, thread 2 has been given its chance.

```
tmp ← <gate-expr>;
unlock_(tmp);
if (tmp == V_{n-1,0}) { V_{n-1,0} ← void; }
else if (tmp == V_{n-1,1}) { V_{n-1,1} ← void; }
⋮
else if (tmp == V_{n-2,0}) { V_{n-2,0} ← void; }
else if (tmp == V_{n-2,1}) { V_{n-2,1} ← void; }
⋮
else if (tmp == V_{0,0}) { V_{0,0} ← void; }
else if (tmp == V_{0,1}) { V_{0,1} ← void; }
⋮
else if (tmp == V_{0,m_0}) { V_{0,m_0} ← void; }
else ERROR;
```

Figure 3.19: C code generated for "`unlock <gate-expr>`".

satisfied and it is not overtaken in the system scheduler queue.

3.4.2 `unlock` Statement

In this section, we describe the implementation of the `unlock` statement. We recall that the semantics of the `unlock` statement (Section 2.3.4) "`unlock <gate-expr>;`" require the gate object G (evaluated by `<gate-expr>`) to be one of the gates locked by a *syntactically enclosing locking statement*.[19] This condition can only be checked at execution. A further constraint is that the `unlock` statement performs early unlocking for only the *innermost* enclosing locking statement. We use the following strategy to check the unlocking.

The compiler checks that an `unlock` statement has a syntactically enclosing statement, and is not enclosed within any `loop` statement. The `unlock` statement therefore knows all the gate expressions used in its syntactically enclosing locking statements. Let the syntactically enclosing locking statements be denoted by $\mathsf{S}_0, \ldots, \mathsf{S}_{n-1}$. Each S_i has gate expressions $\mathtt{G}_{i,0}, \ldots, \mathtt{G}_{i,m_i}$. Each gate expression $\mathtt{G}_{i,j}$ ($0 \le i \le (n-1)$, $0 \le j \le m_i$) is evaluated once when the locking statement is executed, and the resulting reference to a gate object is stored in a temporary variable $\mathtt{V}_{i,j}$. Figure 3.19 shows the C code generated for "`unlock <gate-expr>`".

By comparing `tmp` against the results of all the enclosing gate expressions, we ensure that `unlock` is executed on a gate in one of the enclosing locking statements. When an equality is found, the corresponding temporary variable $\mathtt{V}_{i,j}$ is set to `void`. This ensures that at the end of locking statement S_i, a no-op is performed on $\mathtt{V}_{i,j}$, and there is no unlocking on the gate $G^{i,j}$. (We use $G^{i,j}$

[19] If an `unlock` statement S_{unlock} syntactically occurs within a `lock` statement S_{lock}, then S_{lock} is a syntactically enclosing locking statement of S_{unlock}. If S_{unlock} syntactically occurs within the `then` part of a `try` statement S_{try}, then S_{try} is a syntactically enclosing locking statement of S_{unlock}.

to refer to the gate evaluated by the gate expression $\texttt{G}_{i,j}$.)

The scheme ensures that within any routine, the number of locking and unlocking operations are equal. Intuitively, it works as follows. $\texttt{V}_{i,j}$ is used to reference exactly one gate during its lifetime. It is initialized to `void`. When a reference to a gate $G^{i,j}$ is assigned to it (i.e. $\texttt{V}_{i,j}$ becomes non-`void`), $G^{i,j}$'s lock counter is incremented by 1. When $G^{i,j}$'s lock counter is decremented by 1, $\texttt{V}_{i,j}$ is reset to `void`, preventing any further locking or unlocking operation on $G^{i,j}$ via $\texttt{V}_{i,j}$. Since the use of each $\texttt{V}_{i,j}$ corresponds to one increment and one decrement of $G^{i,j}$'s lock counter, the overall sum effect is that there are equal number of locking and unlocking operations within a routine.

But the above only implements one part of the semantics: the gate G evaluated by `<gate-expr>` is one of the gates locked by an enclosing locking statement. We also need the locking statement to be the *innermost* enclosing one. The comparison against $\texttt{V}_{i,j}$ $(0 \leq i \leq (n-1)$, $0 \leq j \leq m_i)$ are therefore done in the order from the innermost locking statement S_{n-1} to the outermost S_0.

For example, if we have a piece of code as follows.

```
lock g1, g2 then -- S0
  lock g2 then -- S1
    ⋮
    unlock g2;
  end;
  ⋮ -- L′
end;
```

`g1` and `g2` are variables pointing to distinct gates. The `unlock` statement performs early unlocking for statement S_1 and sets the temporary variable $\texttt{V}_{1,0}$ to `void`. S_1's unlocking operation gets a `void` pointer and performs a no-op. As a result, during the execution of L', the gate referenced by `g2` remains locked from S_0.

If the comparisons against temporary variables were done from outermost to innermost locking statement, then $\texttt{V}_{0,1}$ (instead of $\texttt{V}_{1,0}$) would be set to `void`. S_1's unlocking operation would again decrement the lock counter of `g2`'s gate. During L', the gate referenced by `g2` would no longer remain locked from S_0.

3.4.3 `with-near` Statement

The `with-near` statement has a simple translation to conditional statements which is done during parsing. The complicated aspect is how to make use of the information to eliminate overheads in pointer dereference and this optimization is described in Section 3.5.5. A `with-near` statement of the form:

```
with var0, ..., varn near
  S′
end;
```

is translated into a Sather conditional statement:

```
if var0._is_near then
  if var1._is_near then
    . .
      if varn._is_near then
        S'
      else Near_Exception("varn") end;
    ⋮
  else Near_Exception("var1") end;
else Near_Exception("var0") end;
```

The predicate `_is_near` tests whether a variable is void or has a local pointer. We make a distinct name to identify the near tests from `with-near` statements (as opposed to `is_near` tests in conditional statements in user code). `Near_Exception` is a call to a runtime routine which prints out the name of the variable which fails the near assertion, and the line number, file name and class in which the error occurs. When the `else` part of the `with-near` statement is present, `Near_Exception` is replaced by the statements in the `else` part.

3.4.4 `cobegin-end` Statement

The interesting aspects in the implementation of the `cobegin-end` statement arise because the `cobegin-end` statement (1) allows nested `cobegin-end`'s to exist during a thread's execution, and (2) requires all directly and indirectly created threads to terminate before resuming execution to the next statement after the `cobegin-end`.

The compilation for the `cobegin-end` statement is relatively simple: most processing is simply delegated to the syntax tree for the list of statements in the statement body. The functionalities of the `cobegin-end` statement are provided in the runtime system.

A thread can have two independent roles with respect to a `cobegin-end` statement. It can execute `cobegin-end` statements, or it can be a descendent thread within the dynamic scope of some `cobegin-end` statement. [20] The runtime support unifies the two roles by keeping a stack of pointers to cobegin-end records for each user-level thread. We next explain how a per-thread stack called $\mathsf{STACK}_{cobegin}$, that consists of cobegin-end record pointers, implements the semantics of `cobegin-end`.

A statement:

```
cobegin
  <statements>
end;
```

is simply translated into:

[20] Descendent threads and dynamic scope are describe in the subsection on `cobegin-end` in Section 2.3.3.

```
COBEGIN_;
  <C code for statements>
COEND_;
```

where COBEGIN_ and COEND_ are C macros that keep track of the execution of the cobegin-end statements within a thread. COBEGIN_ allocates a cobegin-end record whose pointer is pushed into $\mathsf{STACK}_{cobegin}$ of the executing thread. Intuitively, each record represents one execution instance of a cobegin-end statement.[21] This record keeps count of the number of descendent threads with which the executing thread has to synchronize. It is obtained from a per-processor pool (maintained by the runtime storage management component) if possible. Its counter is incremented by 1 (atomically) for every thread forked within the cobegin-end's dynamic scope. COEND_ checks the counter in the topmost record of $\mathsf{STACK}_{cobegin}$. If the counter is non-0, it suspends the current thread until the counter becomes 0.

The pointer to a cobegin-end record may be propagated to other threads in the following two situations. The first is if during execution of the cobegin-end body, the thread makes a remote call.

```
cobegin -- S1
  f @ cluster_id;
end;
```

When f executes at cluster_id, more threads might be created that require S_1's record to be incremented. Thus when remote calls are made, the pointer to S_1's record must be propagated to the remote cluster. (Section 3.3.2 describes the information that must be propagated when remote calls are made, and the cobegin-end record pointer (if any) is one of them.)

The second situation is when a new thread is created (whether at a local or remote cluster).

```
cobegin -- S2
  :- f;
end;
```

When a thread T is forked for f, S_2's record counter is incremented by 1. T keeps a pointer to S_2's record so that when it terminates, it knows which record counter (if any) to decrement. The pointer to S_2's record is the first to be pushed into T's $\mathsf{STACK}_{cobegin}$. This example also shows how the dynamic scope of a cobegin-end statement is propagated. Suppose we have:

```
cobegin -- S2
  :- f;
end;


f is :- g; end;
```

[21] Thus, there is no difference between the "dynamic scope of a cobegin-end statement" and the "dynamic scope of a cobegin-end record.

$$
\begin{array}{l}
\textbf{dist} \; < \texttt{expr}_0 > \; \textbf{as} \; < \texttt{chunk\_id}_0 >, \\
\quad < \texttt{expr}_1 > \; \textbf{as} \; < \texttt{chunk\_id}_1 >, \\
\quad \vdots \\
\quad < \texttt{expr}_n > \; \textbf{as} \; < \texttt{chunk\_id}_n > \; \textbf{do} \\
\;\; <\texttt{Body}> \\
\textbf{end};
\end{array}
$$

Figure 3.20: General syntactic structure of `dist` statement.

Then `g`'s thread is still within the dynamic scope of S_2. In our scheme, it so happens that pointer to S_2's record is topmost in T's $\mathsf{STACK}_{cobegin}$ when we fork off the thread for `g`. So we can correctly increment S_2's record counter.

Our scheme maintains the following invariant: suppose we have a thread T0 and T0's $\mathsf{STACK}_{cobegin}$ is non-empty. Then any thread forked by T0 is within the dynamic scope of the topmost cobegin-end record. The topmost record may be a result of T0 executing a `cobegin-end`, or T0's parent may have propagated it to T0.

The other property of $\mathsf{STACK}_{cobegin}$ is that if a terminating thread's $\mathsf{STACK}_{cobegin}$ is non-empty, there is exactly one record in the stack. Furthermore, that record represents a `cobegin-end` with which the terminating thread has to synchronize, so its counter is decremented by 1 (atomically). We get this property because in the generated code, every `COBEGIN_` is matched by a `COEND_`. So any cobegin-end record left over must have been propagated from the thread's parent, and the thread was in the cobegin-end record's dynamic scope when it was first created.[22]

3.4.5 `dist` Statement

The semantics of `dist` statement (Section 2.6.2) allows for several possible implementations. In this section, we first describe our implementation and then discuss a possible alternative implementation.

Given a `dist` statement (Figure 3.20), we put <`Body`> into a separate routine R_{body}. A new thread is forked to execute R_{body} for each n-tuple of chunks from the distributed data structures $DD_0, \ldots, DD_n$ (evaluated by expressions $< \texttt{expr}_0 >, \ldots, < \texttt{expr}_n >$).

During parsing, the syntactic representation of a `dist` statement keeps track of a list of pairs of ($< \texttt{expr}_i >$, $< \texttt{chunk\_id}_i >$), and the list of statements in <`Body`>. When the instantiated classes are created (phase 3, Section 3.1.1), this syntactic representation creates an abstract syntax tree node to represent the `dist` statement.

Figure 3.21 shows the execution of the `dist` statement node. The Sather portions (lines

[22] Actually, the implementation distinguishes between record pointers which are due to a thread's executing `cobegin-end` statements and that which is inherited from the parent thread. The distinction is unnecessary and the runtime code works as if a unified $\mathsf{STACK}_{cobegin}$ exists.

```
tmp0:$OB := < expr0 >;
⋮
tmpn:$OB := < exprn >;

index0:INT := tmp0.init_dist;
⋮
indexn:INT := tmpn.init_dist;
BEGIN(counter);
label1:
   tmp_is_done0:BOOL := tmp0.is_done(index0);
   ⋮
   tmp_is_donen:BOOL := tmpn.is_done(indexn);

   assert (align_)
      (tmp_is_done0 = tmp_is_done1) and ···
      (tmp_is_done0 = tmp_is_donen) end;

   if tmp_is_done0 { goto label2; }
   chunk_id0:$OB := tmp0.curr_chunk(index0);
   ⋮
   chunk_idn:$OB := tmpn.curr_chunk(indexn);

   assert (align_)
      (chunk_id0.where = chunk_id1.where) and ···
      (chunk_id0.where = chunk_idn.where) end;

   chunk_index0:INT := tmp0.curr_c_index(index0);
   ⋮
   chunk_indexn:INT := tmpn.curr_c_index(indexn);

   assert (align_)
      (chunk_index0 = chunk_index1) and ···
      (chunk_index0 = chunk_indexn) end;
   counter++;
   <Pack arguments for Rbody and create a new>
      thread for Rbody>

   index0 := tmp0.next_dist(index0);
   ⋮
   indexn := tmpn.next_dist(indexn);
   goto label1;
label2:
END(counter);
```

Figure 3.21: Outline of generated C code for `dist` statement.

1–6, 9–14, 16–27, 31–33) are stored as subtrees in the node. The compiler generates C code for these Sather portions, just like any other Sather statements from the user program. The Sather portions also contain local variable declarations (lines 1–6, 9–11, 16–18, 22–24), some of whose types are not known exactly until phase 8 of the compiler. When the exact type is unknown (lines 1–3, 16–18), `$OB` is used. During phase 8, the `$OB` types are replaced by the exact types (inferred from the RHS of the assignment statements). The non-Sather portions (lines 7–8, 15, 28–30, 34–36) show the C code generated explicitly by the `dist` statement node.

The implementation works as follows. The expressions $<$ $\texttt{expr}_i$ $>$ are evaluated, and the pointers to the distributed data structures stored in temporary variables $\texttt{tmp}_i$ (lines 1–3). Each distributed data structure must be from the `DIST{T}` class or its descendent (Section 2.6.1), and has routines — `init_dist`, `is_done`, `curr_chunk`, `curr_c_index`, `next_dist` — which are used to generate its chunks (Figure 2.20, Section 2.6.1). Lines 4–6 therefore calls `init_dist` on the distributed data structures to initiate the generation of chunks.

To keep track of the number of created threads, the compiler generates a temporary thread counter which is initialized to 0 (line 7). It is incremented by 1 each time a new thread is created for an n-tuple of chunks, and decremented by 1 whenever such a thread completes.

Lines 9–11 check the status of the chunk generations via the `is_done` routine. When the distributed data structures are aligned, their chunk generations will end together (lines 12–14). If there is no more chunk to be generated, we quit the loop (line 15). Otherwise, we retrieve the chunks and assign them to the variables $\texttt{chunk\_id}_i$ $(0 \leq i \leq n)$ in lines 16–18. (Note that the names of these variables are the same as those in the original `dist` statement.) In lines 19–21, we check that all the chunks are located on the same cluster. In lines 22–27, the indices of the chunks are computed and checked to be all equal. The temporary counter is incremented by 1 each time we create a new thread (lines 28–30). Then we proceed on to the next index (lines 31–33), and repeat the loop (line 34).

When all the chunks have been generated, the current thread is suspended until the counter becomes 0, which implies that all the threads created within the `dist` statement have completed.

The previous description was deliberately vague on how the R_{body} routine is created from the <`Body`>, and how threads are created for R_{body} (e.g. argument values to be passed to R_{body}). These are described next.

We first describe the compiler processing that is necessary to put <`Body`> into a separate routine R_{body}. During phase 8 of the compiler, after processing the subtrees of a `dist` statement node is encountered, we extract from the symbol table a list of all the visible local variables and parameters. For each of these declarations, we create a parameter declaration with the same name; the new parameters and <`Body`> are used to create a routine R_{body} which is then processed (like any user-defined routine). R_{body} is inserted into the list of features of the class in which the `dist` statement appears.

The parameters of R_{body} are divided into three categories: unused, read-only and sharable.

For example, in the following (useless) `dist` statement, `x` is unused, `y` is read-only and `z` is sharable by the threads (because there is an assignment to `z`).

```
x,y,z:INT;
dist a as chunk_a do
  z := y+1;
end;
```

The C routine of R_{body} excludes unused parameters in its declarations, so that values of unused variables are not passed. Read-only variables are passed by value to the R_{body}; this prevents memory bottleneck caused multiple invocations of the `dist` statement's body accessing the same location. Sharable variables must be passed by reference, so that all threads see the update of any one thread. Because the variables which contain the references to the chunks are also passed as parameters to R_{body}, in the above example, the value of `chunk_a` is not sent to R_{body}, since `chunk_a` is unused. Excluding unused variables thus cuts down the communication costs (when creating a remote thread).

In addition to user variables, the R_{body} routine also needs the following parameters:

- pointer to thread counter,
- pointer to location storing caller thread, if the caller is suspended, and
- cluster id of caller thread

When R_{body} finishes execution, it sends an active message containing the values of these parameters, back to the cluster of the caller thread. The message invokes a routine which decrements the thread counter, and check if the counter has become 0. If the counter is 0, we resume any suspended caller thread.

A source of optimization is to again distinguish between suspendable and non-suspendable R_{body}'s (like for remote calls and threads). When R_{body} is non-suspendable (suspendable), an cont-state is inserted into the `non_susp_op_queue` (`susp_op_queue`) (Section 3.3.2), to be scheduled like any remote routine calls. After exiting the loop which generates threads for the n-tuple of chunks (lines 8–35, Figure 3.21), before checking whether the thread counter is 0, the caller thread looks into `non_susp_op_queue` for any non-suspendable cont-states to execute. Thus, if R_{body} is non-suspendable, threads created for local chunks are executed without switching the system threads.

Possible Alternative Implementation – `dist` Statement

In the current implementation, a new thread is forked for each chunk of a `DIST{T}` object. If there is more than one chunk per cluster, separate messages have to be sent for each chunk, and each message will result in a different thread. An alternative implementation might collect all the chunks for a cluster, send one message and activate a single thread that repeatedly executes R_{body} for each chunk.

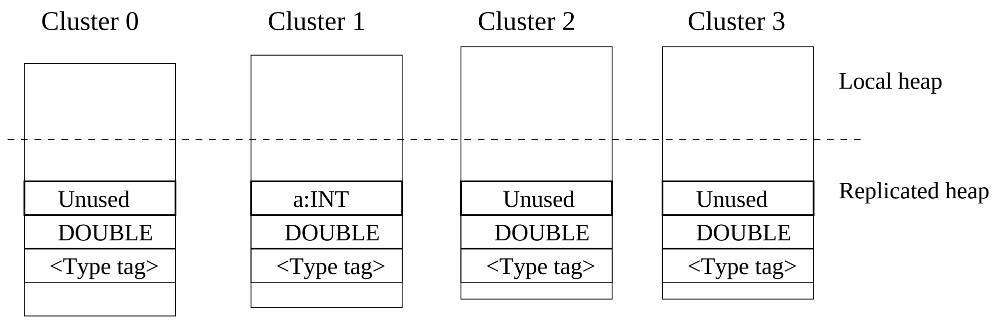


Figure 3.22: How a spread object of type `S` is allocated in current implementation.

3.4.6 `SPREAD{T}` Class

Section 2.5.3 briefly described how we can use a replicated heap to allocate spread objects. But the description omitted a detail for the sake of simplicity: a spread object actually consists of a replicated part and a non-replicated part. Consider a spread class:

```
class S is
  SPREAD{DOUBLE};
  a:INT;
  ...
```

an `S` object needs space for a `DOUBLE` on every cluster (replicated part) and for attribute `a` (non-replicated part). Figure 3.22 shows how our implementation allocates space for `S` (when the non-replicated part is on cluster 1). We include the space for the non-replicated part when allocating space from the replicated heap. This wastes some amount of memory, but has some advantages. The first is that the compiler can generate the same address for accessing attributes in either a spread or a non-spread object. The second is that the unused space may be used to cache immutable attributes. For example, the type tag of an attribute never changes, and this value can be replicated on all the clusters.

Possible Alternative Implementations

One possible implementation is allocate two blocks of memory — one from replicated heap and one from the ordinary heap (Figure 3.23). The replicated part allocates space for a `DOUBLE` and a pointer to the non-replicated part. A pointer to an `S` object consists of a long address: the cluster id on which the non-replicated part is situated, and a short pointer to the replicated part. We do not adopt this implementation because (a) attribute access requires an extra indirection and (b) the compiler must generate different code for accessing attributes in spread and non-spread objects. Case (b) is further complicated when we access an attribute via a dispatched variable:

```
x:$TABLE; x.size := ...
```

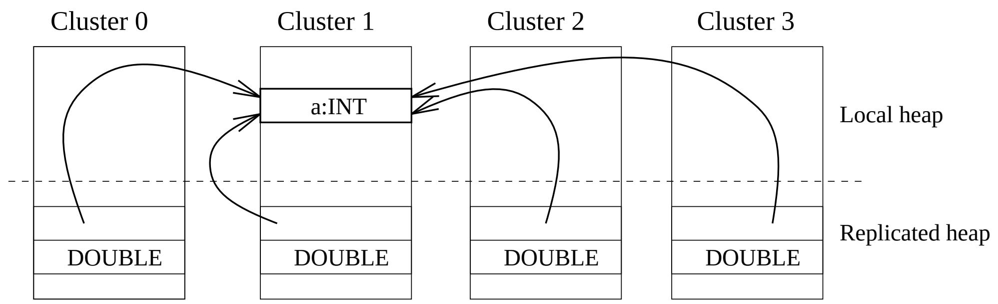


Figure 3.23: How a spread object of type `S` would be allocated by a possible alternative implementation.

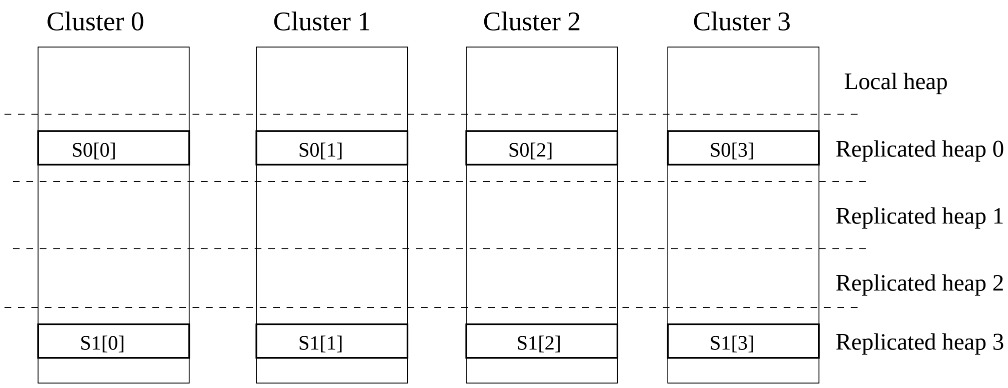


Figure 3.24: Distributed allocation of replicated heap.

and the dispatched variable `x` can point to either a spread or a non-spread object.

The address space for the replicated heap may be managed in a centralized or distributed manner. The current CM-5 implementation uses a centralized allocator. If a multiprocessor has virtual memory, a more time-efficient may be a totally distributed one (Figure 3.24). Each cluster is responsible for a part of the replicated heap's address space. In Figure 3.24, replicated heap 0 is managed by an allocator on cluster 0 etc. The figure shows a possible configuration after two threads — on clusters 0 and 3 — each allocates a spread object.

3.5 Optimization Strategies

Although the objectives of the research do not focus on a highly optimized compiler for pSather, we do examine several ways to improve the generated code. The improvements include code reduction, dispatching on remote objects, inlining (or procedure integration), analysis of object lifetime, lazy forking, eliminating overhead in pointer dereference. We also discuss a strategy to identify object attributes which are immutable, but the current implementation does not use this information.

3.5.1 Code Reduction

Because the pSather compiler treats a class (e.g. `LIST{T}`) parametrized by different type parameters separately, this can result in a large amount of code generated. Furthermore, a large number of functionalities are supported in the runtime libraries.

There are several heuristics used by the compiler to reduce the size of the final executable.

Eliminate Unused Classes

The first is to not generate code for classes and routines which are definitely not used in the program. When the instantiated classes are created, we make up a list containing classes for which we must generate code for the final executable. This list includes the predefined classes, user root class (containing the `main` program) and any externally-used classes (i.e. pSather classes used in any included C code). This list L is further expanded to include other classes during the initial semantic phase. In this phase, initially we only traverse abstract syntax trees of classes in L. During traversal, any type object that is reachable (e.g. `COMPLEX` in `COMPLEX::create`) includes the class in L. If new classes are included in L, the abstract syntax trees of the new classes are in turn traversed. This is repeated until no new classes are added to L. All classes in L are marked used.

A object can only be created within a Sather program if it class is given somewhere in a reachable portion of the program. Therefore this marking is correct. A typical example of an unused class identified by this strategy is one which serves only as the parent of another class. For example:

```
class POLYNOMIAL is
  ARRAY{DOUBLE};
  ...
end;
```

the class object for `ARRAY{DOUBLE}` is not marked as used if `ARRAY{DOUBLE}` is not used in any type declaration or class access.

This marking however is a conservative estimate. A class may be called only within a non-reachable function and still be marked as "used". It is not possible to determine the reachability of functions at this point because we have not resolved the various identifiers.[23]

Eliminate Unused Routines

The previous heuristic describes how to identify classes which are not used in the program. A further refinement is to identify routines which are not used in the program. This is

[23] In a more elaborate scheme, we would re-compute the used classes after the reachable functions have been determined, re-determine the reachable functions from the (possibly) reduced set of used classes, and keep iterating until a fixed point is reached.

done during the major semantic phase: when a routine is retrieved from the symbol table to resolve an identifier, it is also marked as "used". When a routine is used by a dispatch variable (e.g. "`x.compute; -- x:$REGION;`"), its counterparts in the descendent classes are also marked "used". Since this marking is also conservative, a routine may be called from a non-reachable function and still be marked as "used".

Exclude Unnecessary Runtime Support

The pSather runtime support includes a large number of routines to support predefined operations (e.g. operations on gates). Typically only a few of these routines are actually needed in a program. Thus it is natural that only runtime routines which are used are included in the final executable.

One possible approach is to keep runtime routines as library archives. We did not adopt this approach because during the development of the compiler and runtime, changes were many and frequent. As a result the runtime C code is not precompiled, but always compiled together with the user program. The advantage is that it ensures pSather users get the most updated runtime even while changes are being made. A major disadvantage is the additional compilation time.

The compiler has two ways of informing the runtime what routines are needed. The first is used for routines (e.g. `GATE` operations) which are independent from other runtime functionalities. Each such routine `r` and any helper routines it needs are kept in a file named "`r.c`". The compiler gets the names of these routines at startup from a system command file. If the compiler outputs any of these routine names, the name is stored in an internal hash table and finally its corresponding `.c` filename is output to the makefile. This is the cleaner approach. However there are runtime routines which are either inter-dependent or used in other parts of the runtime support. In this case, the compiler gets a mapping from routine name to a flag. If a routine name is ever used, this flag is output to the makefile and used in the C compilation. The runtime files can build further dependencies based on the supplied flags.

Eliminate Unused Remote Routines

Section 3.3.2 described why the compiler generates a local and a remote version of each routine to handle both local and remote calls. If this code generation is done without regard to whether a remote routine is ever used, code size will be doubled. But we know that in the semantics of pSather, a remote call is explicitly made only when the @-operator is used. Therefore during phase 8, when the compiler encounters a "`<expr> @ <loc-expr>`" expression, if `<expr>` is a routine call, the routine object is marked "need_remote_handler". The final code generation phase uses this flag to determine whether it needs to generate a remote handler code for this routine (after determining whether the routine is or used).

We have discussed how to reduce the size of compiled code. A possibly more effective

Program	V1	V2	V3	V4
Primes-finding	3232	2862	2719	2475
Gröbner basis	3592	3192	3058	2674
Fast Multipole N-body	5913	4655	4522	3398

(a) Includes host code

Program	V1	V2	V3	V4
Primes-finding	1580	1210	1067	823
Gröbner basis	1940	1540	1406	1022
Fast Multipole N-body	4261	3003	2870	1746

(b) Excludes host code

Table 3.3: Size (rounded to nearest Kbytes) of executable.

approach is to use dynamic link editing (e.g. Dld [135]) to add/remove object code based on program needs during execution, but this research (dynamic linking on distributed-memory multiprocessors) is beyond the scope of this work.

We measure the code sizes of several programs (which will be described in Chapter 4) by adding each additional heuristic to the compilation process. V1 includes code only for potentially used classes. V2 eliminates unused routines. V3 further excludes unused runtime routines, while V4 also avoids generating remote version of a routine if it is never used with the `@`-operator. Figure 3.3 (a) shows the code sizes for various compiling strategies; the overall reductions (from V1 to V4) range from $\sim 25\%$ (in primes and Gröbner basis programs) to $\sim 43\%$ in a fast multipole N-body program. The absolute sizes are still large because a CM-5 executable consists of code for both the host and nodes.[24] Figure 3.3 (b) shows a more accurate view of code reduction after subtracting the host code size (about 1652 Kbytes). The overall reductions are now $\sim 47\%$ for primes and Gröbner basis programs to $\sim 59\%$ for fast multipole N-body. The reason for the larger decrease in the N-body program size is that class inheritance is used more frequently, resulting in many classes which are only inherited from and never used. The code for these classes are eliminated in V2.

3.5.2 Integrating Dispatch and Access

One interesting aspect of Sather/pSather implementation was a compiler-generated software cache for dispatching [164]. Given a dispatched call "`<obj>.<feature>`" where `<obj>` has an abstract type, the Sather compiler generates the code sequence in Figure 3.25.

We note that there is a "critical section" (as indicated by the arrows in Figure 3.25): the cache must not be altered during its update/use, from the point the type of the object is read (`TYPE_(tmp_obj)`) till the time it is used. On a shared- or distributed-memory machine, multiple

[24] The host code is always needed to start up the program even though pSather's programming model only uses the nodes.

```
tmp_obj ← <obj>
                            <----
if (TYPE_(tmp_obj) != cached_type) {
  cached_type ← TYPE_(tmp_obj);
  cached_dispatch_value ← Get_Dispatch_(cached_type,<feature>);
}
<Use "cached_dispatch_value">
                            <----
```

Figure 3.25: Compiler-generated software cache for dispatch calls.

threads can be executing the same code in parallel. To prevent the threads from interfering with one another, one obvious solution is to allocate these caches on a per-thread basis for pSather. This is, however, not a desirable solution, because:

- it adds additional overhead to thread management. Caches might get allocated but never used.
- threads cannot benefit from earlier dispatches performed by other threads at the same code location.

It turns out that the software caches can be allocated on a per-processor basis[25] if the following invariant holds: the current routine is never re-entered on the local processor when any of its software caches is used. This invariant can be maintained by the following conditions:

- The current thread does not re-enter the routine during the execution of the critical section. For example, the following must not happen for a dispatched routine call: before the call is made (i.e. before the dispatch value is used), the evaluation of the argument values causes the current thread to re-enter the routine and rewrite the caches.
- The current thread must not suspend or be preempted by other threads when the software caches are used. This ensures that no other thread can execute on the local processor and cause the routine to be re-entered.

To guarantee the first condition, we need to look only at dispatched routine calls because a thread can never re-enter a routine when using a dispatched variable to access an attribute, shared or constant feature. For dispatched routine calls, we use the simple strategy of pre-evaluating the argument values and storing them in temporary variables. When the call is made (i.e. dispatch value is used), the argument values are retrieved from temporary variables. Since there is no routine call at all in the critical section, the current routine is never re-entered.

The second condition is helped by the language definition because the definition of pSather says that threads may or may not be preempted. So the implementation is at liberty to ensure

[25] Since we have implemented software caches as C static variables, these are allocated per-processor on both the Sequent Symmetry and CM-5. There is, therefore no need to alter the Sather software cache implementation on a parallel machine.

```
tmp_obj ← <obj>
if (NEAR_(tmp_obj)) {
                         <----
  if (TYPE_(tmp_obj) != cached_type) {
    cached_type ← TYPE_(tmp_obj);
    cached_dispatch_value ← Get_Dispatch_(cached_type,<feature>);
  }
  <Use "cached_dispatch_value" locally>
                         <----
}
else {
  <Perform dispatching and operation at remote processor.>
}
```

Figure 3.26: Using software cache for dispatch calls on a distributed memory machine.

threads are not preempted or suspended during the critical section when the cache is used. Since threads are suspended only at synchronization points (e.g. `GATE` operations), it is not possible for a thread to be suspended when using the software caches.

A remote call from another processor may interrupt and preempt the current thread (Section 3.3.2). But such interruptions can only happen if an active message for the remote call is received, which in turn can only happen if the network is polled.

On the CM-5, the compiler inserts polling points in the generated code (Section 3.3). A polling point can result in a remote request being handled locally. This handler logically represents an extension of another thread and thus might interfere with the caches of the current thread. We carefully avoid this interference by generating polling points only between pSather statements. Since a code fragment such as Figure 3.25 can only be generated as part of a pSather statement, no polling point ever gets generated within the "critical section" of a cache. For example, if it is dispatched routine call, the next polling point will occur in the called routine, after the dispatch value is used.

There is however an efficiency issue in the use of such software caches on the CM-5. Referring again to Figure 3.25, a dynamic dispatch might involve two remote accesses – first to get an object's actual type, then the actual (potentially remote) operation on an object. (Note that when reading a remote object's type, the current thread might be preempted by a remote call. This does not break the atomicity of the critical section which starts only *after* the type value has been received.)

Ideally, on the CM-5, we would like the dispatch call to proceed as in Figure 3.26. For dispatched routine calls, this strategy may or may not be effective because an operation need not be executed on the object's cluster location.

```
obj:$POLYGON;
obj.compute_area @ c2;
```

```
tmp_obj ← <obj>
if (CLUSTER_(tmp_obj) == local_cluster_id) {
   if (TYPE_NEAR_(tmp_obj) != cached_type) {
      cached_type ← TYPE_NEAR_(tmp_obj);
      cached_offset ← Get_Dispatch_(cached_type,<attrib>);
   }
}
((CLUSTER_(tmp_obj) == local_cluster_id) ?
   <Read from address:"LOCAL_ADDR_(tmp_obj)+cached_offset".>:

read_attr_(CLUSTER_(tmp_obj),get_<attrib>,LOCAL_ADDR_(tmp_obj)));
```

Figure 3.27: Generated code for a dispatched attribute read on CM-5.

```
static int type = 0;
static int dispval = 0;

if (TYPE_NEAR_(obj) == X) {
   result ← <Read from address:  "LOCAL_ADDR_(obj)+Y".>
}
else {
   if (TYPE_NEAR_(obj) != type) {
      type ← TYPE_NEAR_(obj);
      dispval ← Get_Dispatch_(type,<attrib>);
   }
   result ← <Read from address:  "LOCAL_ADDR_(obj)+dispval".>
}
<Send result back in reply.>
```

Figure 3.28: Compiler generated routine that combines remote read and dispatching.

Suppose the code is executed on $c0$ and the object is located on cluster $c1$. Whether the dispatching mechanism is executed locally or at $c2$, there is always a remote access to get `obj`'s type from $c0$. There will be three messages, one to make the remote call and two to read `obj`'s type. In optimized code, we can reduce the cost to two messages, one from $c0$ to $c1$ to read `obj`'s type and do the dispatching, and one from $c1$ to $c2$ to make the remote routine call. This optimization may be possible if the compiler generates code in a continuation passing style. Section 5.2.4 discusses the potential advantages and disadvantages of using continuation passing in the compiler.

Although the ideal code (Figure 3.26) only improves dispatched routine calls for some cases, it will always improve dispatched accesses for array elements, attributes, shared and constant features. The mechanism is essentially the same for these cases, so we use attribute read as an illustration.

Figure 3.27 shows the code for a dispatched attribute read ("`<obj>.<attrib>`"). The `<obj>`

expression is first evaluated and stored in a temporary variable (line 1). Then we check the cluster id of the pointer (line 2). If the object is local, then we invoke the dispatching mechanism locally (lines 3–6). Note that in line 3, we use `TYPE_NEAR_` instead of `TYPE_`, in order to eliminate the extra check for near vs. remote pointer. If the object is local (i.e. the test on line 8 is true), the correct offset is stored in `cached_offset`, so we can use it to read the attribute value locally (line 9). Otherwise, we invoke a runtime routine `read_attr_` with the parameters: object's cluster (`CLUSTER_(tmp_obj)`), a remote dispatch/read routine (`get_<attrib>`), and the address of the object within the cluster (`LOCAL_ADDR_(tmp_obj)`).

The routine `get_<attrib>` is generated by the compiler and tailored specifically to the particular attribute being read. Figure 3.28 shows an example of `get_<attrib>` in a class whose index is X and for an attribute whose offset is Y. The runtime routine `read_attr_` invokes this routine at the object's cluster using an active message. First we check whether the type of the near object is X (line 3). If it is, then we can avoid using the dispatching mechanism altogether because the compiler knows that the offset of the attribute is Y (line 4). If the object's type is not X and if it is not equal to the cached type (line 7), the dispatch mechanism is invoked (lines 8–9). Otherwise, the cache contains the correct offset. At the end, the result is returned in a reply message (line 13). Note that lines 3–12 are executed atomically (e.g. there are no polling statements and/or suspension points), so the cache values are consistent.

3.5.3 Inlining

Procedure calls add extra overhead to a program because of additional costs in passing arguments and returning a result. Inlining (also called procedure integration) allows this overhead to be eliminated by replacing a call to a procedure with the body of that procedure. In addition to reducing procedure call overheads, inlining also allows the body of an inlined procedure to be analyzed and optimized specific to its called context [200]. One disadvantage of inlining is that it increases the size of the executable although a study in [83] concludes that this size increase is not a problem in practice. [84] is another reference that discusses various factors that influence the speedup of inlined code.

Inlining is a useful optimization technique in an object-oriented language because there often exist short but often-called routines in classes. We therefore added a phase in the compiler to perform inlining. This phase executes after all the usual type-checking and semantic checks have been done, and before any further optimization-relevant analysis is done on code tree. This allows us to reap some of the benefits of doing analysis across procedures without using sophisticated interprocedural analysis techniques. The benefits of inlining in the analysis of object lifetimes (Section 3.5.4), in the search for immutable attributes (Section 3.5.6), and in the reduction of execution costs in dereferencing near variables on a distributed-memory machine like the CM-5, will be described later. Section 3.6 shows performance improvement of an $O(N^2)$ N-body program using inlining.

We first show how inlining is implemented in pSather. The compiler performs inline expansion only if the user specifies a compiler option. A second option decides whether the routines to be inlined are decided by the compiler or the user (default). In the former case, the compiler simply selects non-recursive routines with n or fewer pSather statements. In the latter case, the programmer selects the routines to be inlined, in a pragma file that has a format similar to a pSather source file. (Section 3.1 of this chapter briefly describes a compiler phase during which program pragmas are processed.) As an example, in the following file "`complex.pragma`" (whose relevance will be described in Chapter 4, Section 4.8 when we described a fast multipole N-body application), the user has specified that the routines `create`, `length` in `COMPLEX` class should be inlined in their callers when possible. Baker [20] notes that an inline pragma for routine `r` can apply all or selected occurrences of `r` and argues that the latter case gives the programmer more flexibility. We however choose to apply an inline pragma to all occurrences of `r` because we want to keep the pragma specifications strictly separate from the program texts.

```
class_pragma COMPLEX is
  create:INLINE is end;
  length:INLINE is end;
  ...
end;
```

A call graph is constructed before inlining. The call graph construction is slightly complicated by dispatched variables. So if we have a call "`x:$POLYGON; x.draw;`" within a routine `MAIN::main`, the call graph has to record the caller-callee relationships for (`MAIN::main`, `<X>::draw`) where `<X>` is `POLYGON` or any of its descendent classes.

The reason for having a call graph is that we need to check if there exists a cycle in the *inline routines* (i.e. those marked for inlining expansion). The existence of a cycle implies that mutually recursive routines are considered for inlining and would result in infinite amount of code. We use a relatively simple strategy to do this check while inlining the routines in a bottom-up order in the call graph. (The bottom-up order is also a clean way to make sure that nested routine calls are inlined correctly.[26])

(1) We first create an initial set S consisting of all inline routines (i.e. all routines marked for inline expansion).

(2) We then consider each routine r in S and look at r's callees. If every callee r' satisfies one of the two critiera: (a) r' is not an inline routine, or (b) r' is an inline routine but not in S (i.e. r' has been inlined in its callers), r itself can be inlined in its callers. After inlining routine r in its callers, it is removed from S.

(3) After we have considered every routine in S, we next check whether S is empty. If it is empty, inline expansion is finished. If it is non-empty and the previous loop has not caused at least

[26] Baker [20], in pointing out the subtle "semantics" of inlining, describes several straight-forward strategies that might not inline nested routines correctly.

one routine to be inlined, then there is a cycle among the inline routines. Otherwise, we repeat the previous step again with a now smaller set S.

Claim 5 *There is a call cycle among the inline routines if and only if there exists a loop in which S is non-empty and none of the S's routines is inlined.*

Proof: Consider the call graph G consisting of routines in S at the beginning of step (2) each time it is executed. The criteria on selecting a routine ensures only leaf nodes in G are inlined. So effectively step (2) trims off the leaf nodes in G. Suppose there is a call cycle among the inline routines. Eventually, G consists of a cycle without any leaf nodes and a loop satisfying the condition is encountered.

On the other hand, suppose that we encounter a loop in which S is non-empty and none of the S's routines is inlined. This means that for every r in S, there exists another r' in S such that $< r, r' >$ is the call graph. Since S is finite, there must be a cycle among the inline routines. □

When a routine r is given to its callers for inline expansion, each caller c passes r to its statement and expression nodes. Each expression checks r against its resolved identifiers. If an expression's identifier is the same as r, it creates a compiler object `INLINE_TEMPLATE` which takes (a) the routine r, (b) an expression which represents the value of `self` when r's code is inlined in c and (c) a list of expressions for r's parameters. A new local variable l is created to hold the r's `self` object. New local variables are also created to replace r's parameters. The inline expansion makes a copy of r's code body, possibly translating certain constructs in the process.

During the copying process, any reference to `self` in r's body has to be replaced by the new local variable `<local-var>`. Thus the result of copying an expression like "`call_foo;`" is "`<local-var>.call_foo;`". The copy of an expression for local variable `lvar` returns a new expression that refers to the copy of `lvar`'s declaration. The copy of r's body is a code block with a label at the end to which any `return` statement in r (translated into a `goto` statement) can transfer control.

The current implementation of inline expansion only supports the minimal functionalities that are necessary to achieve our main aim, i.e. to simplify any analysis and optimizations done by the prototype compiler. We have, therefore, avoided elaborate techniques such as inline expansion for dispatched variables which is a technique adopted in the SELF compiler [139]. This tradeoff does not have a major impact on the applications because situations in which the optimizations are applicable do not normally use dispatched variables. Further issues on inline expansion will undoubtedly arise in the implementation of Sather 1.0 features such as polymorphic routines and bound routines, iterators but they are beyond the scope of this work.

3.5.4 Analysis of Object Lifetime

Garbage collection is a complex research problem, particularly on distributed memory machines. The reason for garbage collection is that explicit memory deallocation makes it difficult to write well-encapsulated classes and complicates coding. Suppose we have an object O from class A. Since O is used by some other class, A does not know when it becomes unreachable and so is unable to reclaim it. It may also be unsafe for O's user to deallocate it because references to O may be kept in other classes and data structures. The problem of keeping track of object references is further complicated in a parallel environment because multiple threads may hold references to the same object.

One way to reduce the amount of garbage is to reduce the number of objects created, e.g. by modifying and reusing the `self` object, instead of returning a new object. However, there are situations when it is conceptually cleaner to return a new object (e.g. operations on complex numbers).

```
x := y.plus(z);
```

It is easier to think of `y.plus(z)` returning a new complex number instead modifying `y`. If `y` is used again later, it can be quite confusing to remember that the value of `y` is actually $y + z$.

Another way to reduce garbage is to allocate objects on the stack, so that when a routine returns, the object is automatically deallocated. In pSather, this strategy is applicable only for value objects (which are immutable and passed by value). Although 1.0 permits user-defined value classes, the 0.1 definition has only a fixed set of *built-in* value classes (e.g. `INT`, `CHAR`). So the object-on-stack strategy is not generally applicable in 0.1. One obvious example is the complex number class: `COMPLEX`. Since `COMPLEX` is not a predefined value class in 0.1, it is implemented as a reference class with operations such as `plus` which allocates and returns new `COMPLEX` objects.

The new `COMPLEX` objects, however, are often only intermediate values in a long chain of complex number computations. Without garbage collection, such intermediate `COMPLEX` objects remain unreclaimed and memory requirements of the program are high.

For the `COMPLEX` class, one possible solution is to redesign the user interface such that new values are stored into an old object. For example `x.to_sum(y)` will store the sum in `x` instead of returning a new object. This however imposes additional burden on the user when using `COMPLEX` objects. When calculating the difference vector of two particle's position, in order not to overwrite the values in `pt1` and `pt2`, instead of writing simply `pt1.minus(pt2)` (or even more simply `pt1-pt2` in version 1.0 which supports syntactic sugar), the user has to make sure that a new complex number is allocated and initialized.

```
x:=COMPLEX::create(pt1.x, pt1.y);
x.to_subtract(pt2);
```

Since our main goal is the design of clean class libraries, we decided to adopt the approach of creating and returning new `COMPLEX` objects. We rely on the compiler (with a little user help) to detect intermediate short-lived objects whose memory allocation is then managed runtime user help) via a simple data-flow analysis on variables. This analysis to detect intermediate objects is also useful for other kinds of objects such as `CURSOR` [187]. With the language extensions in 1.0, `COMPLEX` can be implemented as a value class, and the functionalities are cursors are handled by iterators [182]. These extensions would probably reduce the usefulness of this analysis. This analysis however is relevant in application program written in pSather 0.1 (e.g. fast multipole N-body algorithm) and also demonstrates how inlining impacts interprocedural analysis.

We now describe the technique used. The aim is to check whether an object O allocated via a `new` call (e.g. `COMPLEX::new`) in a routine (say r) is still reachable after r has ended. The check is a conservative one; it is possible that O is not reachable after r terminates, but still considered to be reachable. The check maintains program correctness and seems to detect most of the desired cases.

A reference object O is reachable iff a pointer to O is stored somewhere. And since pointers are stored in variables, intuitively we track how the values in variables are passed around.

Let us consider the creation of an object O in a routine `r`. If the pointer is stored in an attribute or shared or constant feature, obviously the pointer to O can continue to exist even after `r` terminates. But if the pointer is stored in a local variable or parameter (`var`), then O is no longer reachable after `r` terminates, as long as the `var`'s value is not stored somewhere which has a longer lifetime than the current invocation of `r`. Suppose `var` is passed as an argument to a routine call to `r1`, a pointer to this object might be stored in a global data structure during `r1`'s execution. In this case, we say that `var` is marked "out".

More formally, a variable in routine `r` is marked "in" ("out")[27] iff at least one of the values it can hold can exist before (after) `r` is called. The only kinds of variables in pSather are local variables, parameters, attributes, shared features and constant features. Out of these variables, we only maintain the in/out status of local variables and parameters because:

- the in/out status of an attribute depends on that of the object expression in which it is used (e.g. `self.attr` has both in/out status marked because `self` exists both before and after the routine call), and
- the in/out status of shared and constant features are always marked because they always exist before and after the routine call

So we have to define what is meant by the in/out status of expressions and we need to compute the in/out status of local variables and parameters.

An expression in a routine `r` is marked "in" iff its value can exist before `r` is called. (For an expression that returns a reference object, its value is the *pointer* to that object.) It is marked "out" iff its value can exist after `r` has terminated. For example, a function call `fn` is marked "in" because its return value may exist before `r` is called (e.g. the return value may be a pointer to an object stored in some global data structure).

[27] This has nothing to do with in/out sets of basic blocks in compiler optimization [5]. In our analysis, we are computing a property of a variable.

The in/out status of variables and expressions interact as follows. Local variables get their *initial* in/out status from their initializing expressions. When variables are used, their in/out status are further updated. An expression computes its in/out status based on its component expressions (e.g. an expression consisting of only a local variable gets its in/out status from that of the local variable) and its usage (e.g. an expression on the right-hand-side of an assignment statement incorporates the in/out status of the left-hand-side expression).

Consider a local variable declaration with initializing expression.

```
l:COMPLEX := <init-expr>;
```

If <init-expr> is marked "in", i.e. its value can exist before `r` (current routine) is called, then `l` must be marked "in" as well. Also if <init-expr> is marked "out", then it is possible for `l` to hold a value which exists after the routine terminates, e.g. "`l:COMPLEX := res;`". This sets up a local variable's *initial* in/out status, which is later combined with the in/out status computed from its usage (e.g. a routine call `f(l)` cause `l` to be marked "out" whatever its initial status was).

The algorithm repeatedly computes the in/out status of expressions in a routine until none of the expression's in/out status is changed. We have to do a fixed-point computation because a variable or an expression's in/out status may affect other variables or expressions, and then gets updated later.

```
x:COMPLEX;
y:COMPLEX := x;
f(x);
```

Initially `x` is not marked; neither is `y`. `f(x)` causes `x` to be marked "out". So we have to propagate this new status to `y`.

How do we use the in/out status of variables to detect short-lived intermediate objects?

Step 1. Given an expression that creates a new object (e.g. `COMPLEX::new`), we check whether it is marked "out". If so, a reference to the object may exist after the routine terminates. If not, then we can safely deallocate the object when the routine terminates. But we cannot allocate the object on the stack, because the expression may be executed multiple times:

```
loop
  ...COMPLEX::new ...
end;
```

and we do not know how much stack space to pre-allocate. Therefore we use a runtime strategy. Instead of allocating from the system heap, we call a routine that allocates from a per-thread per-invoked-routine heap space. When the routine terminates, that heap space is automatically deallocated.

Step 2. We note that the above strategy is useful only if the object creation expressions are not marked "out". But most of the time, we have something like:

```
plus(y:COMPLEX):COMPLEX is
  res := COMPLEX::new; L'
end;
```

In this case, the variable `res` is marked "out" (since it holds the result), and so `COMPLEX::new` is marked "out" with respect to the `plus` routine. By inlining `plus` in its callers, we make the analysis of `COMPLEX::new` specific to certain contexts. Inlining:

```
x:COMPLEX := y.plus(z);
```

produces:

```
tmp_res := COMPLEX::new;
L'
x:COMPLEX := tmp_res;
```

If `x` is not marked "out", neither will `tmp_res` and `COMPLEX::new`. So this specific instance of `COMPLEX::new` can allocate memory from a runtime-managed heap.

3.5.5 Eliminating Overheads in Pointer Dereference

The pSather implementation on CM-5 uses a 2-word address as pointers (Section 3.3.1). Whenever an object attribute is read/written, the cluster id of the address is checked for local vs. remote access. To reduce this source of overhead, a general optimization strategy would analyze the program to identify when certain variables would hold only near pointers (i.e. pointers which are on the same cluster as the executing thread).

However, to make the optimization more manageable, we only make use of the `with-near` statement to eliminate the extra pointer checks. Even in this restricted domain, there are some interesting strategies to propagate information about the near'ness of variables.

The description is divided into the two parts. We first describe how the compiler uses the `with-near` statement to mark near variables and to record expressions which use near variables. Then we look at how to propagate information about the near'ness of variables.

During parsing, a `with-near` statement is transformed into one or more conditional statements using `_is_near` predicate. When such a conditional statement is found during a top-down traversal of the abstract syntax tree, the near variable is pushed onto a stack STACK_{near}. After traversing the subtrees, the near variable is popped from STACK_{near}. (If the near'ness is propagated to other variables, these will also be popped from STACK_{near}.)

During the traversal of the expressions in the subtrees, we want to identify expressions which use near variables, and which involve memory access. In particular the expressions we look for are: (1) reads/writes of an object attribute (via a near variable), and (2) reads/writes of an array element (when the array pointer is stored in a near variable):

```
with l near
  l.attrib ...
  l[i] ...
```

Therefore, when we encounter a simple single-identifier expression (which is an attribute), we check whether `self` is found in STACK_{near}. If it is, then we mark the expression. During execution, the attribute of `self` can be accessed without checking if `self` has a near or remote pointer. Similarly, when we encounter an array element access "`<arr-expr>[<indices>]`", we check whether `<arr-expr>` is found in STACK_{near}, and when we encounter a dotted expression "`<obj-expr>.attrib`", we check for `<obj-expr>` in STACK_{near}. If `<arr-expr>` (`<obj-expr>`) has been asserted as near, we mark the expression "`<arr-expr>[<indices>]`" ("`<obj-expr>.attrib`"). Using this information, the compiler generates more efficient code to read/write array elements or object attributes because it does not need to check whether the pointer is near or remote.

But the above strategy is limited in its usefulness, because the near'ness information is used only for attribute/array element accesses within the body of the `with-near` statement. For example in the code:

```
with l near
  l.rout;
```

if `rout` accesses the attribute of `self` (i.e. `l`), we cannot make use of the near'ness information to speed up such accesses. One strategy is to generate a version of `rout` in which accesses to `self` are treated as near. However this strategy, when applied to routines with many arguments, might result in many different versions of the same routine:

```
with l near
  l.rout(a0,a1,a2); ...

with l,a0 near
  l.rout(a0,a1,a2); ...

with l,a0,a1 near
  l.rout(a0,a1,a2); ...

with l,a0,a1,a2 near
  l.rout(a0,a1,a2); ...
```

The number of possibilities is combinatorial (though limited by the number of ways in which the routine is called).

Instead of propagating the near'ness information to the routines, we use the fact that the code of inlined routines can be customized to their contexts. Therefore, we perform this optimization after the inlining phase (Section 3.5.3).

Another situation in which the near'ness information is propagated is as follows:

```
with l near
  m := l;
  <Use "m">
```

Since `m` gets the value of `l`, `m` can also be treated as a near variable (within the `with-near` statement) as long as it continues to hold the value that originates from a near variable. There are two ways for such a *compiler-induced* near variable `m` to stop holding the value from a near variable. The first is when `m` is assigned a new value from somewhere else:

```
with l near
  m := l;
  <Read "m">
  m := fn(...);
  -- "m" is no longer a near variable.
```

The second is when the program flow exits the scope which contains the assignment "`m := l`" (e.g. either branch of a conditional statement):

```
with l near
  if test then
    m := l;
    -- "m" is a near variable in this scope.
  end;
  -- "m" is no longer a near variable.
```

In this example, `test` may be false and we cannot be sure that after the conditional statement, `m` holds a value from a near variable. Other syntactic structures which limit the scopes of compiler-induced near variables are the branches of `try` statements, bodies of `loop` statements and branches of a `switch` statement.

The current prototype detects compiler-induced near variables only for those variables which are initialized and never used in any other assignment statement:

```
with l near
  m:$OB := l;
  -- "m" is never used in an assignment statement elsewhere.
```

This eliminates the case when a compiler-induced near variable's value is replaced by a possible remote pointer. Furthermore, during the top-down traversal of the syntax trees, the compiler keeps track of the scopes of compiler-induced near variables and removes them STACK_{near} when it exits from their scopes.

By using inlining and compiler-induced variables, we can propagate the near'ness information to non-dispatched routines within the body of `with-near` statements.

```
with l,a0 near
  l.rout(a0,a1,a2); ...
```

If `rout` is inlined,[28]

```
with l,a0 near
  tmp_l:  <TYPE of l> := l;
  tmp_a0:  <TYPE of a0> := a0;
  tmp_a1:  <TYPE of a1> := a1;
  tmp_a2:  <TYPE of a1> := a2;
  < Body of "rout" >
```

and if `tmp_a0` is assigned once only,[29] then `tmp_l` and `tmp_a0` are compiler-induced near variables in the body of `rout`.

In general, the optimizations need not be restricted to `with-near` statements, but can be extended to conditional statements using `is_near` tests. This is beyond the scope of the prototype implementation.

How Design of `with-near` Affects Implementation

If the identifier list of a `with-near` statement were to include the attributes, shared or constant features (of the class in which `with-near` statement appears), the implementation becomes more complicated. Consider the statement:

```
with foo near
  -- "foo" is an attribute/shared/constant.
  self.foo.attrib := ...
```

According to program semantics, `foo` is the same as `self.foo`, and so the optimization should apply to the assignment statement. Therefore we cannot simply store the attribute `foo` in STACK_{near}.

Consider another situation when inlining is applied:

```
with l near
  ⋮
  -- This statement is from the body of an inlined
  -- routine "l.rout".
  with l.attrib near
    L'
  end;
```

The body of the inlined routine contains a `with-near` statement that lists `attrib` as a near variable. After inlining, `attrib` (or `self.attrib`) becomes `l.attrib`. Any attribute access via `l.attrib` can be optimized.

A solution is to generalize the definition of near variables. A near variable may now either be a simple variable (local variable/parameter), or a list consisting of a simple variable followed by one or more attributes, shared or constant features. In the example, STACK_{near} contains both `l` and [`l`, `attrib`] as near variables when the compiler traverses the syntax trees of statements `L`'.

[28] The inlining of `rout` may depend on program pragmas.

[29] `self` is not assignable and hence `tmp_l` can never be used in any assignment statement.

3.5.6 Immutable Attributes

Section 3.5.4 describes an analysis to compute the in/out status of variables and expressions. We can also use this information to make an approximate guess of which attributes are immutable.

A named attribute (in class `C1`) is immutable if it is updated only when a `C1` object is created. (The creation and update must occur syntactically in the same routine.) Attributes can only be updated in assignment statements:

```
<obj-expr>.<attr> := ...;
```

(Suppose the statement is in routine `r`.) Intuitively, if `<obj-expr>` is not marked "in", its value does not exist before `r` is called. Therefore, we cannot be updating the attribute of a pre-existing object. The object for `<obj-expr>` must have come into existence during `r`'s execution. If `<obj-expr>` is marked "in", then we may be updating the attribute of a pre-existing object, so the attribute may be mutable.

The steps to compute immutable attributes are as follows. First we mark the attributes in all classes as immutable. Then we traverse the abstract syntax trees and check all assignment statements which update object attributes. If an `<obj-expr>` (of type `T`) is marked "in", then we mark `attr` in class `T` as mutable. If an `T` is a dispatched (abstract) type, then we must mark `attr` as mutable in all descendents of `T`.

Since the analysis does not include control flow information, we cannot simply use the information about immutable attributes to do caching. Suppose a class `C1` has a `create` routine:

```
res := new @ remote_cid;
res.attr := ...;
f(res.attr);         -- (1)
res.attr := ...;     -- (2)
g(res.attr);         -- (3)
```

If `attr` is not updated elsewhere, the compiler will mark `attr` as immutable. At (1), when we read `res.attr`, the value should not be cached because `res.attr` is again updated later at (2). (If `res.attr` is cached as a result of (1), the cached value should be invalidated at (2).) It is incorrect to use the cached value from (1), in statement (3).

One way to overcome this problem is as follows. Given an attribute read in routine `r`:

```
...<obj-expr>.attr ...
```

the compiler should generate code to cache the value[30] only if in all control paths (within `r`) that follow `<obj-expr>.attr`, there is no assignment statement that updates `attr`. So there can be no further change to O's `attr` within `r` (where O is the result of evaluating `<obj-expr>`). After `r` returns, we can only retrieve the pointer to O via expressions or variables which are marked "in".

[30] At execution time, actual caching takes place if `<obj-expr>` references a remote object.

Since the compiler analysis says that `attr` is not updated in any `<obj-expr>` marked "in",[31] O's `attr` attribute cannot be changed by any other routine.

Although immutable attributes suggest a way to implement runtime caching, the generated code does not make use of this information. The reason is that our strategy is to focus on compiler optimizations that yield significant performance improvements, and from statistics gathered about read accesses of immutable attributes, it is not clear that the overhead of runtime caching will be offset by significant improvement in reading immutable attributes.

	100	10000	30000
(1) Total reads/writes of attributes	80127	3640560	14179104
(2) Total reads of immutable attributes	66976	2981397	11818122
(3) = (2) as fraction of (1)	0.84	0.82	0.83
(4) Local reads of immutable attributes	65430	2861331	11479239
(5) = (4) as fraction of (2)	0.977	0.960	0.971
(6) Remote immutable attribute reads (fraction)	0.023	0.040	0.287

Table 3.4: Statistics on usage of immutable attributes in a fast multipole N-body program on 100, 10000 and 30000 input points executing on 32 processors (CM-5).

We take the measurements from a fast multipole N-body algorithm which is a good example which would favor runtime caching, because the program is relatively static in nature. The compiler determines that the fraction of immutable attributes is 0.55 which is high among the application programs we examine. (An attribute in a parametrized class is counted for each time the class is instantiated with a different type parameter.) We ran the program on inputs with 100, 10000 and 30000 points on 32 processors. From Table 3.4, we see that reads of immutable attributes (item 2) make up a significant fraction (0.82–0.84) (item 3) of the total number of reads and writes of all attributes (item 1).[32]

To estimate the effectiveness of a runtime cache, we must however examine the fraction of immutable attribute reads which are remote. From item 6, the fraction of remote attribute reads out of the total number of immutable attribute reads is small (0.023–0.040). Since this is the upper bound on the fraction of times a runtime cache is accessed, performance improvements from such a cache appears limited.

From these figures, we conclude that a program written with memory access locality in mind has little use of runtime caches. Since a user often has better knowledge of what is cacheable (e.g. it might be correct to cache slightly out-of-date objects), we decided not to implement caching of immutable attributes.

[31] That is why `attr` is recognized as immutable in the first place.

[32] We exclude array element access, since the compiler analysis does not detect array elements which are unchanged.

3.6 Performance

After the earlier descriptions of the CM-5 implementation and certain optimization strategies, in this section, we examine the performance of important runtime operations and improvements obtained from the optimization strategies. Since the optimization strategies used are not exhaustive, the performance figures will also help locate other overheads which can be reduced in future implementations.

3.6.1 Accessing Attributes of Objects.

Object attributes are the principal means by which values are stored and retrieved in an object-oriented language. In Section 3.3, we have described how the pSather compiler+runtime provide a shared memory abstraction by building reads and writes of remote object attributes on top of active messages. The reads/writes of remote attributes in an object-oriented language such as pSather is further complicated by the fact that the exact type of the object may not be known at compile time. We have also described in Section 3.3 how the dispatching mechanism was altered to improve the performance when accessing remote object attributes by dispatch.

The times in Table 3.5 (a) are obtained from a loop. The generated C code is compiled with -O option by `cc` for the Sparc implementations and `gcc` for the CM-5 implementation.

```
until (i >= num_iters) loop
  <x>.attr := i; -- <x> may be dispatched or not.
  i:=i+1 end;
```

The number of iterations were varied from 100000 to 500000 to verify the accuracy of the results. Table 3.5 (a) first shows the times to read and write an attribute using Sather 0.1 on a SPARCstation 2. The implementation of pSather on a Sparc workstation has comparable performance to sequential Sather (Table 3.5 (b)). In both cases, accessing attributes via dispatching more than double the times. We may therefore use the pSather Sparc timings as a basis to compare the timings on a CM-5.

Table 3.5 (c) shows the times on a CM-5 when accessing attributes on a local and remote object respectively. This version does not include any optimization. In the remote case, the object is allocated on cluster 2 and its attribute is read/written from cluster 0. Since the number of iterations vary (100000, 300000 and 500000), each time is shown as a range of values if necessary.

Table 3.5 (d) shows the times for the same test program when the compiler integrates dispatching and access for remote objects (Section 3.5.2). This integration saves one roundtrip message time and thus reduces the remote access times by about half. In this case, the remote times are still about twice as long as pure roundtrip active message because of runtime overheads – the object pointer has to be checked for remoteness and the remote processor has to check for queued active threads besides polling the network.

	Time/read (μs)	Time/write (μs)
Non-dispatched	0.2	0.2 – 0.27
Dispatched	0.48 – 0.5	0.53 – 0.6

a. Sather on Sparc workstation.

	Time/read (μs)	Time/write (μs)
Non-dispatched	0.2	0.2 – 0.37
Dispatched	0.48 – 0.5	0.5 – 0.53

b. PSather on Sparc workstation.

	Local		Remote	
	Time/read (μs)	Time/write (μs)	Time/read (μs)	Time/write (μs)
Non-dispatched	1.5 – 1.6	2.0	44.6 – 44.9	31.6
Dispatched	3.9	2.4	63.1	63.1

c. PSather on CM-5 (no optimization).

	Local		Remote	
	Time/read (μs)	Time/write (μs)	Time/read (μs)	Time/write (μs)
Non-dispatched	1.4	1.6	29.6	29.6
Dispatched	2.1	2.4	30.1	30.1

d. PSather on CM-5 when remote dispatch and access are integrated.

	Local	
	Time/read (μs)	Time/write (μs)
Non-dispatched	1.1	1.3
Dispatched	1.6	2.0

e. PSather on CM-5 when `with-near` statement is used.

	Local	
	Time/read (μs)	Time/write (μs)
Non-dispatched	0.09	0.61
Dispatched	0.67	0.95

f. PSather on CM-5 when `with-near` statement is used, and polling statements are eliminated.

Table 3.5: Times to read and write object attributes.

Even though the times for remote access improves after integrating dispatch and access, the times for local access remain high compared to the implementation on a Sparc workstation. There are two major sources of overheads.

The first is the extra check for remote object. This can be eliminated when the programmer uses the `with-near` statement and the compiler uses this statement as a hint to optimize pointer dereferences (Section 3.5.5).

```
with <x> near
  <Repatedly read or write local object attribute>
end;
```

Table 3.5 (e) shows the time improvements over Table 3.5 (d) when the `with-near` statement is used.

```
L151:
        cmp %l1,%o0
        bl L151
        add %l1,1,%l1
        b L152
```

Figure 3.29: Generated assembly code for attribute reads after applying optimizations.

The second source of overhead is the extra polling statements that the compiler has to insert into the generated C code for correct program behaviour. The compiler removes the polling statements from a routine tagged with a program pragma `NO_POLL`. Table 3.5 (f) shows the local access times after removing this second source of overhead. These times are still a little slower than the times in 3.5 (a) because of two reasons. The first is that Sparc 2 is a 40 MHz processor, while each CM-5 node is a 33 MHz processor. The second reason is that on the CM-5, each pointer is represented as two words (cluster id and address); this representation prevents the C compiler from storing the address in a register. As a result, there is a load each time the second (address) word is read whereas on the Sparc, the address is retrieved from a register.

From the local access times in Table 3.5 (f) and remote access times in Table 3.5 (d), we see that remote reads/writes are about 30–45 times slower than local ones. There is an odd case when reading an attribute from a local object in a non-dispatched manner takes only 0.09 μsec (thus making remote read about 330 times slower), but this result is not accurate since the use of `with-near` statement and elimination of polling statements has made it possible for the `gcc` compiler to optimize and remove the reads from the final code. The code that remains only performs a sequence of additions on a register (Figure 3.29). (The `cc` compiler however did not perform this optimization for the measurements in Tables 3.5 (a) and (b).)

Therefore with careful use of pragmas and pSather constructs, we are able to take advantage of the C compiler optimizations and achieve performance comparable to C. We expect the library designers to take advantage of such optimizations in commonly used library routines; an example is a distributed matrix multiplication routine described in [163].

3.6.2 Invoking Routines.

Another set of important operations is the invocation of routines. Section 3.3 describes how the compiler distinguishes between suspendable and non-suspendable routines in order to generate more efficient code for non-suspendable remote calls. In our current implementation, a calling thread waits for the completion of a remote call before proceeding on. While waiting, the processor of the calling thread normally switches to another thread. However, if the remote routine finishes and replies in a short time, it can be worthwhile for the processor to poll the network for a while before

	Time/$f_{non-susp}$ call (μs)	Time/f_{susp} call (μs)
Sather (Sparc)	0.22 – 0.23	N/A
PSather (Sparc)	0.23 – 0.24	2.80 – 2.83
PSather (CM-5)	2.22	7.31
PSather (CM-5, no polling)	0.33	4.13

Table 3.6: Timings of local routines calls.

	Time/$f_{non-susp}$ call (μs)	Time/f_{susp} call (μs)
V1	129 – 137	243 – 255
V2	123 – 132	154 – 161
V3	114 – 119	130 – 135
V4	69	94 – 99
V5	49 – 55	N/A
V(local)	3.2	8.3 – 8.5

Table 3.7: Timings of remote/local routines calls on CM-5.

switching to another thread. This avoids any unnecessary thread switching overhead. Another criterion that affects the costs of remote calls is whether the packing and unpacking of messages is done by compiler-generated code or runtime support routines.

We discuss the performance of routine calls under these criteria (i.e. suspendable vs. non-suspendable, switching thread immediately vs. poll a short time, message handling via runtime-support vs. compiler-generated code).

We first look at the times of a simple call. The sample program consists of a simple loop as shown.

```
until (i >= num_iters) loop
  f; -- Routine call, f_non-susp or f_susp.
  i:=i+1 end;
```

We measure the times of both non-suspendable and suspendable routines, given by $f_{non-susp}$ and f_{susp} respectively. ($f_{non-susp}$ has an empty body while f_{susp} contains an empty cobegin-end statement.) The measurements were made using 1000000 and 2000000 iterations, each repeated twice.

The CM-5 timings is slower because of the polling overhead in both the loop and routine body. Once this overhead is removed, the CM-5 timings is comparable to those on the Sparc. (It is still a little slower because `f`'s implicit argument — an object pointer — is 2 words on CM-5.)

Now we look at the performance of a remote call (both suspendable and non-suspendable) on CM-5 (Table 3.7). Each entry is obtained by running the test program on 2 and 32 processors (i.e. the remote processor is 1 and 31 respectively). The range indicates the extent of timing errors and does not correspond to the distance of the remote processor.[33]

[33] On a 32-processor CM-5, the distance is an insignificant factor in message latency.

We will list the characteristics of each version and explain how it improves over the previous version.

- V1 depends on runtime routines to pack argument values into a message and send it off. At the remote processor, if the remote routine is suspendable, a thread is allocated immediately when the message arrives. If the routine R is non-suspendable, a remote-handler of R (generated by the compiler) unpacks the message and stores the argument values into an cont-state. In both cases, the compiler generates a remote version of R which returns an acknowledgement back to the sender when R ends. The caller processor switches thread immediately every time a remote call is made.

- V2 incorporates the optimization described in Section 3.3.2 for remote calls made on suspendable routines. This reuses the thread and stack for a remote suspendable call whenever possible. (In fact, in the experiment, the thread and stack is reused for every remote suspendable call.) This optimization affects only suspendable routines; hence we see an improvement in the times of f_{susp}, but not $\mathsf{f}_{non-susp}$.

- V3 reduces the remote call time further by having the compiler generate the code that packs the argument values and sends an active message directly. This eliminates one procedure call at the caller thread.

- In V4, when a remote call is made, the caller processor polls the network until a reply is received instead of switching to a new thread immediately. (A correct implementation will impose a limit on the amount of time spent polling before performing a context switch; otherwise, a deadlock may result. This implementation therefore demonstrates the best possible times if the caller processor adopts the poll-before-switch strategy.) Because the use of register windows slows down thread switching on CM-5's Sparc processors, we see a constant time improvement over V3 for both $\mathsf{f}_{non-susp}$ and f_{susp}.

- In Section 3.3.2, we have described how the implementation retains the non-preemption semantics by allocating cont-states for remote calls. If threads were preemptible, an active message for a non-suspendable routine r could invoke r directly. V5 shows the timings when remote calls to non-suspendable routines are handled immediately on message receipt.

- Lastly V(local) gives the timings of local calls. The timings are larger than those in Table 3.6, third row because the @-operator is used (i.e. the call is "`f @ cid`" instead of just "`f`").

The implementation provides two possibilities: V3 (default) or V5 with thread switching (i.e. a processor switches to a new thread immediately each time a remote call is made, and threads may be preempted by remote non-suspendable calls). We do not implement V4 because it is not clear how long a processor should poll before switching to a different thread. With more information on the time distribution of remote calls, a future implementation may be able to choose a polling

	Time/$f_{non-susp}$ call (μs)	Time/f_{susp} call (μs)
V3, no polling	113 – 124	122 – 131
V(local), no polling	1.3	5.1

Table 3.8: Timings of local/remote routines calls on CM-5 without polling.

```
i:INT;
cobegin
  until (i >= N) loop
    :- f @ <cid>;
    i := i+1;
  end; end;


f is end; -- Empty routine
```

Figure 3.30: Program used to measure timings of local/remote thread creation.

interval that reduces the times of remote calls on average. Table 3.8 give the times for V3 and V(local) after polling points are removed. Polling overhead becomes insignificant in remote calls.

3.6.3 Costs of Forking Threads.

Section 3.3.3 describes how cont-states can be used to implement thread creation, just like the implementation of remote routine calls. In this section, we look at the costs of thread creation using two implementation strategies. The first allocates a thread object and stack for every user-level thread. The second uses cont-states[34] (Section 3.3.2) and relies on the runtime scheduler to reuse thread objects and stacks whenever possible.

Figure 3.30 shows the test program used to obtain the timings. The program simply forks off N threads which execute an empty routine. We take measurements for N = 1000 and N = 10000. `<cid>` is set to 0 (1) for local (remote) thread creation. The measured timing is divided by N to give an average thread creation time.

[34] This is only implemented for the case when a thread is not associated with a gate, i.e. "`:- f @ cid`".

	Thread object with stack	Cont-state
Local (N = 1000)	152 – 153	22
Local (N = 10000)	150	21
Remote (N = 1000)	183	58
Remote (N = 10000)	175 – 176	51

Table 3.9: Average time (in μs) of a local/remote thread creation for a non-suspendable routine.

	Thread object with stack	Cont-state
Local (N = 1000)	156	28
Local (N = 10000)	156	25
Remote (N = 1000)	189	52
Remote (N = 10000)	186	52

Table 3.10: Average time (in μs) of a local/remote thread creation for a suspendable routine.

The results are shown in Table 3.9;[35] each entry is repeated five times. The cont-state implementation significantly reduces the local/remote thread creation times. This is because the scheduler simply reuses the same runtime thread object and stack, thus eliminating any context-switch for runtime thread objects. The 21 – 22 μs for local thread creation using cont-state consists of time spent in allocating/deallocating cont-states, invoking the empty routine, updating the cobegin-end record and other overheads (e.g. polling the network). In all cases, remote thread creation consistently adds an overhead of about 30 μs; this is consistent with the message latencies[36] we have seen in other measurements (e.g. accessing object attributes).

We note that the `f` routine in Figure 3.30 is a non-suspendable routine. So we take measurements for a suspendable routine:

```
f is g.set end;
-- g is a near pointer to a gate object.
```

Table 3.10 shows the results. The timings increase marginally due to the additional local gate operation. The thread creation times have stayed constant for the thread-object-with-stack implementation because it is oblivious to whether a routine is suspendable or not. On the other hand, although the cont-state implementation is different for suspendable and non-suspendable routines (Sections 3.3.2 and 3.3.3), we find that in the best case the overheads in creating a remote local/thread for a suspendable routine is the same as that for a non-suspendable routine. This best case occurs when the scheduler is able to reuse a runtime thread object and its stack.[37]

3.6.4 Costs of `GATE` Operations.

Gates are the central user-level synchronization construct in pSather. It is therefore important that they are reasonably efficient. A straight-forward way of implementing gates would require a new thread and stack to be allocated for every remote gate operation (because gate operations are suspendable). But this is undesirable because of the high context switch costs on the CM-5. Section 3.3.4 describes how we implement the gate operations (in the runtime) in a continuation-passing

[35] The CM-5 system memory allocator is time-expensive, especially whenever the program heap is increased. To prevent this overhead costs from affecting the measurements, all test programs preallocate a large amount of heap space before the user program starts executing.

[36] The message latency is larger than the ideal case, because of additional system overheads.

[37] This scheduling is explained in Figure 3.3, together with the description of the runtime `Susp_Op_Loop` routine.

```
i:INT;
cobegin
  until (i >= CONFIG::current_num_clusters) loop
    :- read_from_gate(g) @ i;
    i := i+1;
  end; end;


read_from_gate(g:GATE{T}) is
  j, val:INT;
  until (j >= <No.  of iterations>) loop
    val := g.read;
    j := j+1;
  end;
end;
```

Figure 3.31: Program used to measure timings of gate `read` operation.

style. This strategy avoids allocating a new thread and stack for each remote gate operation.

We obtain timings for both thread-based and continuation-based implementations of the gate `read` operation. The thread-based implementation allocates a new thread object and stack for every remote gate `read` and was our initial implementation. Our current prototype uses the continuation-based implementation.

The times are first measured using the program shown in Figure 3.31 with varying number of processors and iterations. Then we measure the overheads by removing the `g.read` operation in `read_from_gate`. (The overheads however become insignificant for large number of processors.) The difference is then divided by the *total number* of gate `read`'s executed by all the threads, giving an "average time". This "average time" gives an indication of the performance when there are local and remote gate operations (although the proportion of local gate operations decrease as with increasing number of processors).

Table 3.11 (a) shows the average time for a gate `read` operation when polling points are inserted between every 3 pSather statements (Section 3.3.5). For each implementation, we measure timings for 1000 and 10000 iterations with different number of processors. Each table entry of the continuation-based implementation is based on five repetitions and shows a range when there is significant variation. [38] This variation may be due to the non-deterministic way in which remote gate operations are scheduled. It may also be due to processor 0 (where the gate is located) not receiving requests fast enough. Table 3.11 (b) shows that the average time and its variation both become smaller when polling becomes more frequent. But there is no improvement when a polling point is inserted between every pSather statement (Table 3.11 (c)). Any effort in trying to reduce

[38] For overhead timings, we also measure five repetitions for each (number-of-processors, number-of-iterations) combination, and the variation is insignificant. We use the smallest overhead timing from each set of five repetitions to calculate the corresponding average time.

No. of Processors	Thread-based (i) / (ii)	Continuation-based (i) / (ii)
1	26 / 10	7 / 5 – 7
2	282 / 278	41 / 44 – 51
4	368 / 415	28 – 58 / 31 – 35
8	681 / 634	28 / 32 – 38
16	709 / 764	31 – 61 / 48 – 72
32	644 / 811	46 – 138 / 79 – 130
64	832 / 830	77 – 113 / 78 – 133

a. Polling points inserted between every 3 pSather statements — default.

No. of Processors	Continuation-based (i)	Continuation-based (ii)
1	6 – 8	8
2	40	40
4	24	24
8	25	24 – 26
16	26 – 57	44 – 69
32	41 – 72	63 – 74
64	49 – 94	82 – 91

b. Polling points inserted between every 2 pSather statements.

No. of Processors	Continuation-based (i)	Continuation-based (ii)
1	6 – 12	6
2	52 – 131	48 – 51
4	30	29 – 33
8	27	24
16	48 – 60	41 – 56
32	44 – 75	57 – 70
64	67 – 90	74 – 113

c. Polling points inserted between every 1 pSather statement.

Table 3.11: Average time (in μs) of a gate `read` operation. ((i) = 1000 iterations, (ii) = 10000 iterations)

the average time is limited by the overheads of a remote call (thread context-switch and message latency), because all local and remote operations on a gate `G` are really executed on `G`'s cluster (Section 3.3.4). In all three tables, as the number of processors increase, there is more contention and hence the timings generally increase.

Since the outlines of gate operations are similar, the timings for `read` should be representative of other gate operations. We check this by measuring the timings of the `lock` statement. The statement in Figure 3.31:

```
val := g.read;
```

by:

lock g **then** **end**;

Table 3.12 show the measurements taken for 1000 iterations. We note that a `lock` statement consists

No. of Processors	Polling freq. = 3	Polling freq. = 2
1	9 – 18	12
2	94 – 127	81
4	140 – 142	139 – 154
8	141 – 203	144 – 145
16	174 – 175	176 – 184
32	142 – 157	164 – 169
64	146 – 148	156 – 157

Table 3.12: Average time (in μs) of a `lock` statement (measured using 1000 iterations).

of two `gate` operations: a lock and an unlock, and so we expect the timings to be twice those for the `read` operation. This is indeed the case when large number of processors are used. When the number of processors is small, the timings are more than twice as large. Overall, the timing of a `lock` statement remain relatively independent of the number of processors.

Our intuition for this phenomenon is that: the `lock` statement creates more "contention" in that when there are two or more threads, there is a higher chance for a lock operation to be suspended (than for a `read` operation on a gate with a non-empty queue of values). This means that more suspension points are executed and the scheduler has to do more work to allocate/deallocate, queue/dequeue the cont-states. The timings $\sim$ 70 – 100 μs (found in Figures 3.11 and 3.12) are probably close to the worst possible for a remote `gate` operation. To the extent that the increase in average time of a gate operation is kept in check as the number of processors increase exponentially, we feel that the gate implementation has performed satisfactorily.

3.7 Runtime Checks

In Chapter 2 when we discussed the semantics of several language constructs, we mentioned that certain runtime errors can be caught at execution time if the program has been compiled with a *runtime-check* option. This option asks the compiler to generate extra code that tests for various possible runtime errors. When an error actually occurs, a message shows the type of error and the line and file name where the error occurs, together with a trace of the current execution stack. In an environment without parallel debugger, this information helps the user to track down common program mistakes, and saves a lot of debugging effort. These runtime checks are therefore a useful and relevant aspect of the programming environment. We spend some time describing the kinds of runtime errors and how they are checked.

We use two general strategies to incorporate extra code that does runtime error checking. The first is for the compiler to generate code that performs specific checks. The second strategy is to define sections of code that do runtime checks within the C preprocessor `#ifdef` command [124] in the runtime system. When the runtime-check option is on, the compiler generates a `-DRT_CODE_CHECK_` flag that is fed to the C compiler and causes the `#ifdef`'d (or conditionally compiled) code to be included. When the runtime-check option is off (default), none of this extra code is included in the final executable.

In addition to generating the flag, the compiler also generates extra code at the entry and exit points of a routine to update a per-thread trace stack. At routine entry, the names of the file, class and routine are pushed onto the trace stack. Just before a routine exits, the top set of <file, class, routine> are popped off the trace stack. If an error is caught, the trace stack is printed in LIFO order. On a distributed-memory machine, a thread may be (logically) split across multiple clusters; therefore a thread keeps track of its earlier subthreads located on a different cluster with a back pointer (Section 3.3.2).

The advantage of this implementation is that it needs minimal change in the compiler. The disadvantage is that the trace stack only records the called routines but not the locations where the routines are called. Despite this shortcoming, the implementation has proven adequate so far.

Void Object.

An error occurs when a void pointer is dereferenced. There are two situations when dereferencing a void pointer is an error. The first is when reading/writing the attribute of a reference object or elements of an array. The second is when dispatching is needed, e.g. invoking a routine or accessing an attribute of a dispatched type:

```
x:$POLYGON;
x.draw;
```

The dispatching mechanism deferences object pointers to read the type tag of an object and then uses the type tag to get the correct value of a feature (e.g. offset of an attribute or address of a routine).

A call such as "`x:SQUARE; x.draw;`" does not result in any runtime error even if `x` is void. An error occurs only when an attribute of `self` is accessed within the `draw` routine in the `SQUARE` class. This slightly peculiar case arises because pSather permits the `self` variable in a routine to hold a `void` reference.

The pSather compiler identifies the above situations and generates extra code to check for a void pointer before an attribute/array element access or before invoking the dispatching mechanism:

```
  if (<temporary variable> == void) {
    err_quit_(VOID_OBJ_ERR_, <line number>);
  }
  < Read or write object attribute >
OR
  < Get value from dispatch table >
```

The pointer value to be tested is pre-evaluated and stored in a temporary variable in case of side effects. If it is void, an error routine is called to print out relevant error messages. The `<line number>` gives the location of the error within the pSather routine.

Void `GATE` Object.

A void gate leads to an runtime error when it occurs:

- on the LHS of a deferred assignment statement,
- in the list of gate expressions of a `lock` or `try` statement, and
- during the invocation of a predefined `GATE` operation or predicate.

The generated test code is similar to that for testing ordinary void objects. For example, to test for "`g :- fn;`", the compiler generates code:

```
if (temp == void) {
  gate_err_quit_(VOID_GATE_ERR_,<line number>);}
```

to test for a void gate pointer before calling the runtime thread creation routine. The gate pointer is evaluated and stored in a temporary variable `temp`, and `<line number>` is the location where the deferred assignment takes place.

Out-of-Bound Array/Spread Access.

Another potential source of error is when an index into an array is out of bound. For example, given an array expression such as `a[i,j]`, the compiler generates test code as follows:

```
if ((tmp_i < 0) || (tmp_i >= tmp1)) {
  arr_out_of_bound_(0,tmp_i,tmp1,<line number>);
}
if ((tmp_j < 0) || (tmp_j >= tmp2)) {
  arr_out_of_bound_(1,tmp_j,tmp2,<line number>);
}
```

before reading/writing the array element. The first argument of `arr_out_of_bound_` gives the position of the index. `tmp_i`,`tmp_j` are temporary variables holding values of indices `i` and `j` respectively. tt tmp1 and `tmp2` are temporary variables containing the array sizes along dimensions 1 and 2 respectively. `<line number>` is the line where the offending expression occurs.

These out-of-bound checks are executed only after the array object is checked to be non-void. The same kinds of checks apply when accessing elements of a spread object. In the case of a spread object, the bounds are given by 0 and the number of clusters.

Invalid Object Type.

Although Sather 1.0 is completely type-safe, this is not the case with Sather 0.1 from which the current implementation of pSather is derived. There are two kinds of assignments which violate the type safety of a pSather program:

```
x:$POLYGON; y:SQUARE; ...
y := x; -- Case (1)
-- SQUARE is a descendent of POLYGON.

z:$QUADRI;
z := x; -- Case (2)
-- QUADRI is a descendent of POLYGON.
```

The language specification also permits such reverse assignments when argument values are passed to routines.

In case (1) if `x` is not a pointer to a `SQUARE` object at execution time, the assignment is invalid. This is because after the assignment, any operation on `y` would incorrectly treat a non-`SQUARE` object as a `SQUARE`.

In case (2) if the object referenced by `x` at execution time is neither a `QUADRI` nor a descendent of `QUADRI`, the assignment is again invalid. But in this case because the dispatching mechanism is independent of the class hierarchy, any operation on `z` will execute correctly. Although the code would work correctly, this is still a semantic error that is not detectable by the compiler and is subjected to runtime check.

To maintain type safety, the compiler marks source expressions which are assigned to variables with stricter types. (`$POLYGON` is considered less strict than `$QUADRI` and `SQUARE` because it permits a variable to refer to more types of objects.) The source expression to be tested has to be pre-evaluated and stored in a temporary variable in case of side effects. The C code that tests for the two different cases is as follows:

```
-- Case (1)
if (tmp_x == void) { /* OK */ }
else {
  type = TYPE_(tmp_x);
  if (type != SQUARE) {
    type_mismatch_(type,SQUARE,<line number>);
  }
}
```

```
-- Case (2)
if (tmp_x == void) { /* OK */ }
else {
  type = TYPE_(tmp_x);
  if ((type != QUADRI) &&
    (IS_NOT_A_DESCENDENT_OF(type,QUADRI))) {
    type_mismatch_(type,QUADRI,<line number>);
  }
}
```

In both cases, a temporary variable `tmp_x` contains the pointer to the object to be tested. In case (1), the type of `tmp_x` must be a `SQUARE`; otherwise, an error message is printed and the program terminates. In (2), the type of `tmp_x` should either be a `QUADRI` or a descendent of `tmp_x`. If not, an error message is printed.

Invalid Object Type of Thread Result.

Another situation when type safety can be violated is when the result of a thread is enqueued in a `GATE{T}` object:

```
g:GATE{SQUARE}; ...
g :- fn;
```

Suppose the routine `fn`'s return type is `$POLYGON`. If during execution, the result of `fn` is not a `SQUARE`, an object of wrong type gets enqueued in a `GATE{SQUARE}` object that expects to hold only references to `SQUARE` objects.

This kind of runtime error is detected in the runtime system after the thread terminates and before the result is enqueued in the gate object. The main reason for adopting this strategy (instead of generating extra code) is that all the epilogue code after thread termination resides in the runtime system and this seems the cleanest approach in building the system. But the compiler still has to generate extra arguments to tell the runtime the expected type of the thread's result, and the file name and line number of the offending statement, in order to print a more informative error message.

Invalid Cluster Id.

The language has various constructs that allow the user to specify in which cluster an object is to be allocated, or a thread or subthread is to be executed (Chapter 2.4). The use of cluster id's in the user program present another potential source of runtime errors because the user might have inadvertently specify an invalid id. Therefore all runtime routines that accept cluster id from user code includes conditionally compiled code to check that the id is valid.

Consider the example: "`x:=SQUARE::new @ i`". This is translated to a call to a runtime routine `new_at_` whose parameters include the cluster id given by `i`. The `new_at_` routine has the following conditionally compiled code.

```
#ifdef RT_CODE_CHECK_
  if (!  (valid_cluster_id_(id))) {
    <Announce invalid id and print trace stack>
  }
#endif
```

Similar conditionally compiled code is included in runtime routines that perform predefined operations (e.g. `copy`, `extend`), make remote routine calls, or fork new threads on user-specified clusters.

Stack Overflow.

Each thread has a fixed contiguous block of memory which serves to hold the stack frames during program execution. This block of memory is not extensible during program execution. This restriction is a result of using C as the intermediate code and relying on the runtime for thread manipulation. In practice this restriction does not pose any problem except when deep recursive calls are made. (Deep recursive calls are probably not a good idea anyway when good performance is desired.) To warn the user of a stack overflow, the compiler inserts the following code at the entry point of every routine.

```
tmp ← new stack pointer;
if (tmp <= address-of-memory-block) {
  <Print stack overflow message.>
  <Display current call stack.>
}
```

The new stack pointer is obtained from a runtime routine. Since the stack grows in the direction of decreasing address, if the stack pointer is less than the address at the end of the memory block, there is an overflow.

Near Variables with Remote Pointers.

As described in Section 2.5.1, the `with-near` statement allows the user to assert that the specific variable(s) will reference only local objects within the scope of the statement. It is a runtime error to update a near variable[39] to reference a remote object.

If we want to maintain the near property strictly, we must check for every update of near variables when the statements enclosed within `with-near` are executed. Because only local variables/parameters are allowed to be near variables, and local variables/parameters can only be updated in assignment statements, we only need to generate extra checking code for assignment statements which update near variables (e.g. "`l := <RHS-expr>`").

[39] Note that if attributes, shared and constant features can be listed in a `with-near` statement, the more general definition of "near variable" (Section 3.5.5) is used.

```
with attrib near
  g(attrib[...]);
  attrib := ...;
```

a. `attrib` (an attribute) is checked for near'ness in both statements.

```
with attrib near
  ...x := attrib; ...
```

b. `attrib` (an attribute) may be updated by another thread to a remote pointer.

```
with attrib near
  ...f; ...
```

c. `f` is a routine which updates attribute `attrib`.

```
with shared_feature near
  ...broadcast_shared_feature(...); ...
```

d. This error is caught by the compiler directly.

Figure 3.32: Examples of runtime errors using near variable.

```
tmp ← <RHS-expr>;
if ((tmp != void) && (cluster_of_(tmp) != local_cluster)) {
  err_quit_(NEAR_VAR_ERR_, <line number>);
}
```

Suppose attributes, shared and constant features are allowed in `with-near` statements. The above simple strategy does not prevent bugs that result from race conditions because the update may come from another thread. Suppose thread T0 executes this code:

```
with attrib near
  ...end;
```

When `attrib` is an attribute, it is possible for another thread T1 to have concurrent access to the same `self` object and hence T1 can write a remote pointer (relative to T0) into `attrib`.

In this alternative design of `with-near` statement, we need to perform a relaxed check on near variables. We would check that a near variable refers to a local object when it is updated (via an assignment statement) and when it is used to access object attributes (Figure 3.32 (a)) (although more strictly, we should also check the near attributes/shareds/constants during any routine call in the `with-near` statement). In this less strict implementation of near checks, the compiler generates code to test near variables in the following cases.

- A near variable is dereferenced to read/write an object attribute

- A near variable (referencing an array) is used to read/write array elements.
- A near variable is used on the LHS of a deferred assignment statement.
- A near variable holds a gate to be locked by a `lock` or `try` statement.
- A near variable is updated in an assignment statement.

Figures 3.32 (b)–(c) show examples of runtime errors for near variables that would not be caught by this implementation (although in Figure 3.32 (c), if `f` were inlined, any assignment to `attrib` would be checked for near'ness.) A case which could be caught during compilation is a broadcast operation on a near shared feature (Figure 3.32 (d)).

Broadcast Statement.

The last source of runtime errors that we identify comes from the broadcast operations on shared features. As discussed in Section 2.5.4, every shared feature (e.g. `f`) has a corresponding predefined broadcast operation (e.g. `broadcast_f`). The broadcast operation takes an argument whose type is the same as `f` and writes this value to `f` on every cluster. It is an error to have two or more simultaneous broadcasts on the same shared feature.

The current prototype does not check for this runtime error because broadcast operations on shared features rarely occur and are mostly used to initialize some program parameters. If such a runtime check is desired, the following simple strategy suffices.
(1) When a thread performs a broadcast operation, it records the address of the shared feature in a per-cluster table. This address is deleted when the broadcast operation completes.
(2) When a processor receives a request to update a shared feature `f` locally, it checks whether `f`'s address is in the per-cluster table. If it is, then a local thread is performing a simultaneous broadcast operation for `f` and an error is signalled.

3.8 Summary

This section summarizes what we have learned from the CM-5 implementation. We look at how the machine/execution model affects our implementation and discusses architectural features which have helped or impeded the runtime implementation.

Implications of the Machine/Execution Model for Implementation

So far this chapter has described our implementation on a CM-5, a one processor per cluster machine. Since pSather adopts a general cluster model, we briefly discuss how our current implementation may be generalized when there is more than one processor per cluster.

Load Balancing

The `@`-operator in pSather can change a thread's locus of control to a different cluster, but does not specify on which processor within a cluster a thread is executed. The result is that a programmer is able to use the `@`-operator explicitly to manage load balancing among different clusters. Since there is little understanding of how to do fine-grain automatic load balancing among clusters, our current approach – programmer-managed inter-cluster load balancing – is practical and efficient, but does not offer a completely location-transparent model.

On the other hand, for intra-cluster load balancing, there are various runtime systems (e.g. Uniform [215], Presto [94], and any state-of-the-art operating system for symmetric multiprocessors e.g. Solaris, Mach) which perform well for shared-memory machines. In a machine with > 1 processor per cluster, such systems can be used to provide automatic load balancing within a cluster.

Our model restricts the location of a subthread to within a cluster (Section 2.4.3). The language therefore does not allow a (user-level) thread to be migrated to a different cluster in the midst of execution, and precludes any runtime inter-cluster load balancing mechanism. This restriction is necessary to in order to maintain consistent semantics in the language. For example, if we have:

```
insert(e:T) is
  pre (self.where = CONFIG::current_cluster)
                                      <----
  if (self.where = CONFIG::current_cluster) then
    <Insert e>
  end; end;
```

and if the runtime performs inter-cluster thread migration, even though the pre-condition holds, the thread may be preempted and migrated just before the `if` statement is executed. As a result, `e` may not be inserted, contrary to what the user expects.[40]

Although the model precludes runtime inter-cluster load-balancing, it leaves enough room for the system to incorporate well-understood techniques to do load-balancing within a cluster.

Thread Implementation

The distinction between subthreads and threads in our execution model gives the implementation flexibility in deciding how subthreads/threads are mapped to actual runtime threads. For example, a subthread on a remote processor can be implemented by allocating a new runtime thread and stack, or by sharing an existing runtime thread with other subthreads (Section 3.3.2). On the other hand, if a routine call is local, the subthread simply allocates a new activation record on a local stack.

[40] Intuitively, since location information (e.g. objects' cluster location) is accessible at the user level, the runtime is not allowed to do things "behind the programmer's back", which would alter such information.

Architectural Features for Efficient Runtime

The pSather runtime implementation on the CM-5 relies a lot on two basic libraries — message and threads.

Message Library

We have found that an active message library such as that by von Eicken et al. [226] is a suitable substrate on which to build the pSather runtime. The ability to access the network interface directly is important for reducing remote latencies, and making the shared address space model feasible (even though CM-5 is a message-passing machine).

We have described earlier how the runtime relies on an active message layer to perform remote procedure calls and remote reads/writes. A message library can handle receipt of messages in two ways – it can either rely on polling or be interrupt-driven. The CM-5 implementation of pSather uses CMAM which relies on polling to receive messages.[41] Our experience shows that polling has the following disadvantages (−) and advantages (+).

- (−) The compiler has to be modified to generate polling points in user code, making it dependent on the architecture (Section 3.3.5).
- (−) The additional polling points degrade the performance of the system (Section 3.6). The degradation may be up to a factor of 2 (as in the performance of N-queens program, Section 4.2.1).
- (+) On the other hand, the reliance on polling means that an executing thread is never preempted unless it chooses to poll. Thus, on a 1-processor cluster, critical sections are protected "for free" by simply ensuring that there is no polling within the critical sections. For example, the absence of polling during dispatch guarantees that the critical section is protected when the dispatch cache is computed and used (Section 3.5.2). As a result, the implementation of dispatch caches on CM-5 remains relatively unchanged from that on the Sequent.[42] But we have to point out that our dispatch cache is one way (but not necessarily the most efficient way) of implementing dispatched variables. The Sather compiler implemented at Karlsruhe demonstrates an alternative dispatch implementation [156] that is both efficient and theoretically will not rely on the protection given by non-polling during dispatching.
- (+) The runtime can avoid locking runtime data structures (e.g. ready queue) because the absence of polling guarantees that the data structure is protected during update (Section 3.2). But if we use an interrupt-driven message library, the runtime overhead from having to lock

[41] The latest version of Thinking Machines' message library (CMMD 3.x) provides active message library routines which are interrupt-driven, but it came too late to affect our design decisions.

[42] The only change is that different dispatch mechanisms are used for near variables (Section 3.5.5).

runtime data structures ought to be more than compensated by the overhead reduction in the user code.

From the list of tradeoffs between polling and interrupts, whenever there is a choice, an interrupt-driven library ought to be preferred for an efficient implementation. The implication for multiprocessor architecture is that although it is important to reduce message latencies, this hardware performance gain can only benefit the higher-level software (e.g. language implementation) if the architecture supports a suitable message model (e.g. by providing the mechanisms to receive or turn off interrupts from the network interface).

Threads

A major source of overhead in the implementation of lightweight threads on CM-5 Sparc processors is due to the large processor state (a consequence of the register windows in the architecture). Since the software has a choice of whether to use the register windows or not, one possible approach would be to generate code that does not use the windows. But the pSather compiler generates C code and hence relies on the C compiler's code generation.[43] So we adopted another approach to reduce thread costs, i.e. using cont-states to implement thread forking and allocate runtime threads based on demand (Sections 3.3.2 and 3.3.3). This reduces the cost of thread forking (Section 3.6.3) but it does not reduce the cost of thread context-switch. Therefore, although using standard off-the-shelf chips is a way for multiprocessor manufacturers to leverage off the progress in sequential processor design, it is important that the processor supports fast thread context-switching (e.g. by having a small processor state). We think that if a machine like CM-5 were to use MIPS R4000 [178] instead of a Sparc, the overheads of our implementation would be much less. Otherwise, a processor design targeted for multiprocessor also holds better promise. One such example is Sparcle [93] which supports fast context-switching by partitioning the set of register windows among multiple threads.

[43] Altering a C compiler was beyond the scope of our work and we could not find any reliable C compiler that does not use the Sparc register windows.

Chapter 4

Abstractions and Applications

This chapter discusses the design of some library classes and medium-sized applications. We show how the language constructs described in Chapter 2 help in the design of library classes, and how the library classes are used in actual application programs. As a summary, Table 4.1 shows how two library classes have been used in several different application programs.

	Distributed Workbag	Replicated Hash Table[1]
N queens	X	
Primes sieve	X	X
Gröbner basis	X	X
Fast-multipole		X

Table 4.1: Use of abstractions in various applications.

Section 4.1 shows a distributed workbag that is used in various programs (N-queens (Section 4.2), prime finding (Section 4.4), Gröbner basis (Section 4.7)). Section 4.3 describes a replicated hash table that is used in both the prime finding and Gröbner basis programs. Section 4.8 describes a fast multipole N-body program and the data structures used (including a software cache implemented by the replicated hash table). Last but not least, since the current prototype is based on an old version of Sather, we also evaluate how the new 1.0 design improves the capability of the language to build well-encapsulated parallel library classes (Section 4.9).

4.1 Workbag

Our first abstraction generalizes a sequential queue into a parallel, distributed workbag. This class is commonly used in master-worker style computations. In later sections, we will show how

[1] We use the replicated hash table as a map in primes sieve and Gröbner basis, and as a set in fast-multipole.

```
<Initialize queue with tasks.>
while not queue.empty do
  w ← queue.dequeue
  <Compute with w, and enqueue
  new work generated into the
  queue.>
end;
```

Figure 4.1: Code skeleton using sequential queue.

```
<Initialize bag with tasks.>
while not (bag.is_empty and bag.is_quiescent) do
  w ← bag.get
  if (w exists) then
    <Compute with w, and insert
    new work generated into the bag.>
  else
    <No work>
  end;
end;
```

Figure 4.2: Code skeleton of a worker thread using parallel workbag.

the distributed workbag is used in the N-queens parallel search problem (Section 4.2), a parallel prime sieve program (Section 4.4) and a parallel variation of Buchberger's algorithm [48] for computing Gröbner basis (Section 4.7).

Queues are common data structures in sequential programming. A sequential program using a queue generally has a structure as in Figure 4.1. In many cases, the ordering of elements in the queue is irrelevant to the algorithms' correctness, e.g. algorithms such as graph problems (Goldberg-Tarjan's max-flow algorithm [110]), combinatorial search problems (traveling salesman problem using "2-opting", searching all solutions for an n-queens problem). These algorithms are especially amenable to parallelization, because several threads can then dequeue elements in parallel, and work independently (Figure 4.2). The queue effectively functions as a *workbag* [2] into which tasks are inserted and from which tasks are served out to different threads.

4.1.1 Overview

We implement a parallel workbag for a distributed-memory machine. The workbag is distributed by having one sub-bag at each cluster, and it provides an interface for the worker threads to insert and retrieve tasks from the workbag. Tasks are always inserted in the local sub-bag.

[2] We will not use the terms *workqueue*, *enqueue*, *dequeue* in a parallel context because the strict FIFO semantics is not preserved. The corresponding terms *workbag*, *insert*, *get* are more appropriate.

During retrieval, a thread looks for task in the local sub-bag before looking in the remote sub-bags. (Section 4.1.5 will describe the insertion and retrieval routines in more details.) Our setup pays more attention to data locality than load balancing, because with non-uniform memory accesses, it is generally important to reduce remote communication between processors in different clusters [170].

A complication with a parallel workbag is termination detection. We want to terminate the program when the bag/queue becomes empty *permanently*. In the sequential case (Figure 4.1), because there is only one thread of control, when the queue becomes empty, it stays empty. The execution can therefore terminate when the queue is empty. In a parallel program with multiple threads, when a bag becomes empty, it need not stay empty. For example, one of the threads may be in the midst of executing and generating new tasks while another thread finds the bag empty. When we have multiple threads, a bag becomes empty permanently only when the bag is empty and no thread is producing new tasks. Only then can the threads terminate. We call a thread that is not producing new tasks a *quiescent* thread. The workbag has a routine for the worker threads to signal their quiescence to the workbag.

4.1.2 Data Structure

Figure 4.3 shows the setup of the workbag. The workbag consists of a sub-bag at each cluster. In order to locate the sub-bags, the pointers to sub-bags are kept in a *directory* (implemented as an array in this case). Therefore, we implement `DISTRIB_BAG{T}` as an `ARRAY{BAG{T}}` (Figure 4.6) and is an array of pointers to sequential sub-bags. One of the attributes of a workbag is an array of gates "`locks:ARRAY{GATE0}`"; each gate is used as a lock to protect the corresponding sub-bag.

We note that although the sub-bags are located on different clusters, if only a single directory were shared by all the threads, this implementation would then shift the bottleneck from the bag to the array of pointers. Data locality would be reduced because each thread then has to access a (possibly remote) array to get at the pointer of a local or remote sub-bag.

We prevent this inefficiency[3] by requiring each thread to make a local copy of the directory. Each local directory can be thought of as a thread's local port [229] to the workbag abstraction, and is represented by a dashed box in Figure 4.3.

Each directory has additional attributes. The `local_id` attribute gives the cluster id in which the directory is located and is used to find the local sub-bag. Each local directory keeps a pointer (`master` attribute) to the original master directory; as will be seen later, it is used to help detect termination.

In the master directory, we keep a counter for the number of quiescent threads. We choose to use a centralized counter because it is simple. This centralized counter is kept in the `num_quiescent` attribute of the master directory. The counter is protected by a shared gate object during updates. The `lock_quiescent` attribute of the master and local directories contains the pointer to the shared

[3] Other ways to improve the efficiency of reading the pointers will be described in Section 4.1.7.

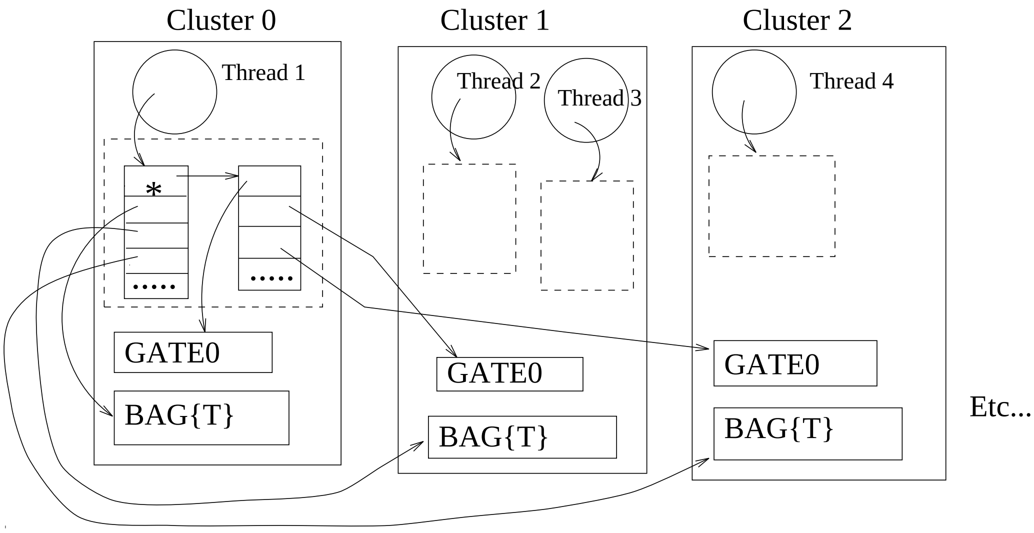


Figure 4.3: Setup of workbag with each thread having a local directory.

gate object.[4]

We keep a `has_signaled_quiescent` flag in each local directory. Before a quiescent thread updates the `num_quiescent` counter, it first check the directory's `has_signaled_quiescent` flag. If the flag is true, the master directory's `num_quiescent` counter already includes this thread. If the flag is false, we lock the shared gate object, atomically increment `master.num_quiescent` counter by 1, and set the flag to true. The flag ensures that each quiescent thread updates the centralized counter exactly once.

4.1.3 Interface

Figures 4.6, 4.7 and 4.8 give the definition of the workbag class. It has the following interface.

- `create:DISTRIB_BAG{T}.` This allocates a distributed workbag with a directory which keeps the pointers to sub-bags on different clusters.
- `insert(e:T).` A task `e` is inserted into the local sub-bag.
- `get:T.` This routine tries to retrieve a task from the local sub-bag. If none exists, then it tries to retrieve a task from some other remote sub-bags. Otherwise, it returns `void`.
- `is_empty:BOOL.` The result is `true` if we cannot find any task in all the sub-bags; it is `false` otherwise. The workbag, however, may not remain empty.

[4] Since we only use the lock functionality of gate objects here, we will use "locks" and "gates" interchangeably.

```
shared dist_bag:DISTRIB_BAG{NQUEEN_SOLN};
-- Declaring a per-cluster pointer to a common workbag object.

-- Master thread distributes the pointer to all clusters.
...  broadcast_dist_bag(DISTRIB_BAG{NQUEEN_SOLN}::create);

-- Worker gets its local directory.
bag := dist_bag.dir_copy
```

Figure 4.4: How a client might use a workbag.

```
loop
  w := bag.get
  if (w /= void) then
    <Compute with w, and insert
    new work generated into the bag.>
  else
    bag.signal_quiescent;
    if (bag.num_quiescent_threads = N) then
      <Break out of loop>
    end;
  end;
end;
```

Figure 4.5: Code skeleton of a worker thread using parallel workbag's routines.

- `dir_copy:DISTRIB_BAG{T}`. This returns a copy of the workbag directory. In addition to improving data locality, the directory also contains information for termination, so every thread *must* have its own copy of the directory.
- `num_quiescent_threads:INT`. This routine returns the number of quiescent threads.
- `signal_quiescent`. A thread calls `signal_quiescent` on a workbag when it is not producing any new tasks. This enables a workbag to keep track of the number of quiescent threads so that termination can be properly detected.

4.1.4 Initialization and Use

Together with the broadcast operation on shared variables (Section 2.5.4), a user can set up a workbag as in Figure 4.4. Figure 4.5 shows how the interface routines may be used in a thread's code. In our implementation, if a workbag does not have any task, a thread will continually execute the loop and call `bag.get`. Although this approach has the disadvantage that a thread may be idling, it is simple and the performance of the applications (N-queens, primes sieve, Gröbner basis) also shows that this is an acceptable choice.

```
class DISTRIB_BAG{T} is
  ARRAY{BAG{T}};
  locks:ARRAY{GATE0};
  local_id:INT;

  lock_quiescent:GATE0;
  num_quiescent:INT;
  has_signalled_quiescent:BOOL;
  -- Related to the quiescent property of all the threads.

  master:SELF_TYPE;
  shared num_clusters:INT := CONFIG::current_num_clusters;


  create:SELF_TYPE is
    -- Create an array of bags, one on each cluster.
    res := new(num_clusters);
    res.locks := ARRAY{GATE0}::new(num_clusters);
    i:INT; until (i >= num_clusters) loop
      res[i] := BAG{T}::create @ i;
      res.locks[i] := GATE0::new @ i;
      i := i + 1;
    end; -- loop
    res.local_id := CONFIG::current_cluster_id;
    res.lock_quiescent := GATE0::new;
    res.num_quiescent := 0;
    res.has_signalled_quiescent := false;
    res.master := res;
  end; -- create


  dir_copy:SELF_TYPE is
    -- We copy both the pointers to sub-bags and
    -- pointers to locks.
    res := copy;
    res.locks := res.locks.copy;
    res.local_id := CONFIG::current_cluster_id;
    res.has_signalled_quiescent := false;
  end; -- dir_copy


  num_quiescent_threads:INT is
    res := master.num_quiescent;
  end; -- num_quiescent_threads
```

Figure 4.6: Part 1 of definition of DISTRIB_BAG{T} implemented in current prototype.

```
signal_quiescent is
  with self near
    if not (has_signalled_quiescent) then
      has_signalled_quiescent := true;
      lock lock_quiescent then
        master.num_quiescent := master.num_quiescent+1;
      end; end; end;
end; -- signal_quiescent


is_empty:BOOL is
  with self near
    local_subbag:BAG{T} := [local_id];
    with local_subbag near
      lock locks[local_id] then
        res := local_subbag.is_empty;
      end; end;
    if (res) then
      i:INT; until (i >= num_clusters) loop
        if (i /= local_id) then
          lock locks[i] then
            res := [i].is_empty @ [i].where;
          end; end;
        if not (res) then return end;
      end; end; end;
end; -- is_empty


insert(e:T) is
  local_subbag:BAG{T};
  with self, local_subbag near
    local_subbag := [local_id];
    l:GATE0 := locks[local_id];
    -- Protect the insertion of local sub-bag using gate.
    lock l then local_subbag.insert_back(e) end;
  end; -- near
end; -- insert
```

Figure 4.7: Part 2 of definition of DISTRIB_BAG{T}.

```
get:T is
  -- LIFO for local sub-bag, FIFO for remote sub-bag
  with self near
    -- First look for any task in local sub-bag.
    l:GATE0 := locks[local_id];
    local_subbag:BAG{T} := [local_id];

    -- Protect the retrieval from local sub-bag using gates.
    lock l then
      with local_subbag near
        res := local_subbag.get_back;
      end; -- near
      if (res /= void) then
        if (has_signalled_quiescent) then
          has_signalled_quiescent := false;
          lock lock_quiescent then
            master.num_quiescent:=master.num_quiescent-1;
          end; end; end; end;

    if (res = void) then
      i:INT; until (i >= num_clusters) loop
        if (i /= local_id) then
          ith_subbag:BAG{T}:=[i]; l:=locks[i];
          lock l then
            res:=ith_subbag.get_front @ ith_subbag.where;
            if (res /= void) then
              if (has_signalled_quiescent) then
                has_signalled_quiescent := false;
                lock lock_quiescent then
                  master.num_quiescent :=
                    master.num_quiescent-1;
                end; end;
              return;
            end; end; end;
        i := i + 1;
      end; end; end;
end; -- get


end; -- class DISTRIB_BAG{T}
```

Figure 4.8: Part 3 of definition of DISTRIB_BAG{T}.

Suppose there are two threads T1, T2 sharing a workbag. If the counter `master.num_quiescent` is not updated in the `get` routine while the sub-bag's gate is locked, the following race condition can occur. (Time line goes down.)

T1	T2
bag.get = void	<Computing>
bag.signal_quiescent	
	bag.insert(...)
bag.get /= void	
	bag.get = void
	bag.signal_quiescent
	– At this point, T2 may incorrectly detect termination and exit.

Figure 4.9: Potential race condition in incorrect workbag.

4.1.5 Implementation Details

The pSather code in Figures 4.6– 4.8 shows how the language constructs (Chapter 2) are used in the workbag implementation.

In the `create` routine (Figure 4.6), the `@`-operator is used to create the different sub-bags and the corresponding locks on different clusters. The `@`-operator is also used in `get` and `is_empty` when we perform operations on remote sub-bags. The `dir_copy` routine (in Figure 4.6) uses the `copy` operations (Sections 2.5.2) to make a copy of the arrays of pointers to sub-bags and gates, and returns a local directory to the distributed workbag. The shared address space allows us to do various reads and writes (e.g. the shared counter for the number of quiescent threads) and remote operations easily.

Next we explain how the `insert` and `get` routines work (Figure 4.7 and 4.8). The `insert` routine simply puts a new task in the local (locked) sub-bag. It does not try to redistribute new tasks to remote sub-bags. A task gets redistributed only when the `get` routine cannot find any task from a local sub-bag. A thread first tries to look for a task from its local sub-bag because the order of task retrieval is irrelevant. To retain load balancing, if a thread does not find any task locally, it then searches the remote sub-bags. Since more than one thread can be inserting or getting a task from a sub-bag, the access to each sub-bag is protected by a lock. When a thread finds a task, it becomes non-quiescent, so `master.num_quiescent` has to be updated. Because of race conditions (Figure 4.9), if a thread finds a task and its `has_signalled_quiescent` flag is true, then `master.num_quiescent` must be updated while the corresponding sub-bag remains locked. That is, the update of the counter and retrieval of a task must happen atomically to prevent any race conditions. We will later describe how this enables the threads to detect termination correctly.

An interesting note about `insert` and `get` is that there are two independent implementation choices when inserting or retrieving work from a workbag.

The first is that for a particular local/remote sub-bag, tasks may be retrieved in either a FIFO, LIFO or even random fashion. We use a deque as a sub-bag. A deque has four operations: `insert_back`, `insert_front`, `get_back` and `get_front`. In Figure 4.7, new tasks are always inserted at the back of the deque. In Figure 4.8, tasks are removed from the back of local deque and the front of remote deques. The result is that the tasks are retrieved from local and remote sub-bags in a LIFO and FIFO order, respectively.

The second aspect is the order in which remote bags are searched (if no task is found locally). The ways in which remote bags may be searched include:

- a cyclic search starting from sub-bag 0, 1, ..., $C-1$,
- a cyclic search starting from a random sub-bag with a stride s, where s is coprime with C, or
- a tree-like search starting with nearby sub-bags.

Section 4.2 describes the performance of these strategies.

We can use code inheritance mechanism to implement these different strategies as subtypes of the `DISTRIB_BAG{T}` abstraction. However, in the current prototype, this is not easily achieved, because there is no way to abstract the `insert_X`, `get_Y` operations without rewriting the class definition. In Section 4.9.1, we see how this can be overcome using 1.0 language features.

The routine `is_empty` (Figure 4.7) first checks if the local sub-bag is empty. If it is, the routine then checks each of the remote sub-bags until it finds a non-empty sub-bag or when all sub-bags have been examined to be empty. Thus `is_empty` does not help to detect termination because `bag.is_empty` being true does not imply that the workbag has become empty permanently. It is, however, included as part of the interface of a workbag for completeness.

4.1.6 Distributed Termination

Now we give an argument why this implementation correctly detects when a workbag becomes empty permanently (i.e. $empty(bag)$ becomes stable), if every thread follows the protocol: If a thread finds that it has no task to do, it must tell the workbag that it is quiescent (the worker code structure in Figure 4.5).

We note that:

$$\texttt{bag.get} = void \Leftrightarrow \texttt{bag.is\_empty} = true$$

Thus Figure 4.5 uses the test $\texttt{bag.get} = void$ to check whether the workbag is empty. The predicate `bag.is_quiescent` is implemented by calling "`bag.num_quiescent_threads` $= N$" where N is the number of workers. The thread also obeys the workbag requirement that it calls `signal_quiescent` whenever it is not producing any task.

We outline an argument that if there are N threads and if the threads use the workbag as in Figure 4.5, then:

$$\texttt{bag.num\_quiescent\_threads} = N \quad \Leftrightarrow \quad empty(bag) \ \text{ is stable}$$

Suppose $empty(bag)$ is a stable property. From the definition of `get`, `bag.get` $= void$ holds for all N threads. From the protocol, all N threads will inform the workbag that it is quiescent, and so eventually `master.num_quiescent` $= N$ (i.e. `bag.num_quiescent_threads` $= N$ is true).

Now suppose `master.num_quiescent` $= N$. Let $op_i(b_j)$ denote op executed by thread t_i on sub-bag b_j. If we consider the $get_i(b_j)$ for all threads i and sub-bags j, all these operations are linearizable, because whenever a sub-bag is accessed, it is protected by a corresponding lock. Since the sub-bag remains locked while the counter `master.num_quiescent` gets updated (in `get` routine), we guarantee that if there is any thread i, sub-bag j such that $get_i(b_j) \neq void$, then `master.num_quiescent` $< N$. That is:

$$get_i(b_j) \neq void \Rightarrow \texttt{master.num\_quiescent} < N$$

Therefore when `master.num_quiescent` becomes N, we have:

$$\forall t_i, b_j \quad get_i(b_j) = void$$

where $get_i(b_j)$ is the latest `get` operation for thread t_i on sub-bag b_j. Since every thread's latest `get` operation on all sub-bags returns $void$, this means that no thread is working on any task and thus no new task can be generated. This also implies that the whole bag is empty and must stay empty.

4.1.7 Alternative Implementations

We discuss the important implementation choices in the workbag and the reasons for not adopting the alternative implementations.

Non-Suspending vs. Suspending get

In the `get` routine, if no task is found, it returns `void` immediately instead of suspending the thread. In our implementation, if a workbag does not have any task, a thread will continually execute the loop and call `bag.get`. The disadvantage is that a thread may be doing useless work.

In an alternative implementation, a worker thread may be suspended if the workbag does not have any task. This avoids useless work. A more complete workbag implementation may extend the current interface to include a routine which may suspend the worker thread when it does not find any task. This gives the worker thread a choice to call a suspending or non-suspending `get` routine. For example, even if a thread cannot find any task, it may still have other computation and may not want to be suspended; in this case, it will call the non-suspending version of the `get` routine.

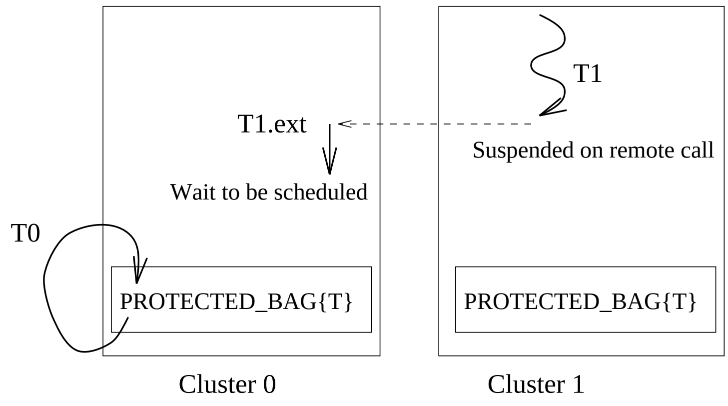


Figure 4.10: Unfair access to a sub-bag using `PROTECTED_BAG{T}`.

We chose the simpler, non-suspending implementation; the performance of the applications (N-queens, primes sieve, Gröbner basis) shows that this is an acceptable choice.

Non-Protected vs. Protected Sub-bag

Our implementation uses the sequential, non-protected `BAG{T}` class to implement the local sub-bags. This increases the amount of communication. For example, to create the ith sub-bag and its lock, two separate remote calls are made, resulting in two messages and two replies. It is more efficient if one message is sent to the remote cluster for both a sub-bag and a lock to be created. The addresses of the created sub-bag and lock are returned in a single reply. This increase in communication also appears in the `get` routine when looking up a remote sub-bag.

A way to reduce communication is to use a `PROTECTED_BAG{T}` class which works like `BAG{T}` but with its operations protected by an internal gate. In the workbag's `create` routine, instead of two remote calls, there would be only one remote call "`PROTECTED_BAG{T}::create @ i`" that creates both sub-bag and gate. Similarly in the `get` routine (Figure 4.8), there would be only one remote call that locks the sub-bag, performs a `BAG{T}::get_back` operation and unlocks the sub-bag.

Although the alternative implementation reduces the number of remote messages, we did not implement workbag using `PROTECTED_BAG{T}` because the current prototype does not guarantee the threads fair access to all the sub-bags. Consider Figure 4.10. When thread T1 does not find any task in its local sub-bag, it makes a remote call to the sub-bag on cluster 0, resulting in a new subthread (T1.ext) which waits to be scheduled. Since the pSather semantics guarantees neither preemption nor non-preemption, T1.ext might never be scheduled if T0 is never suspended. This

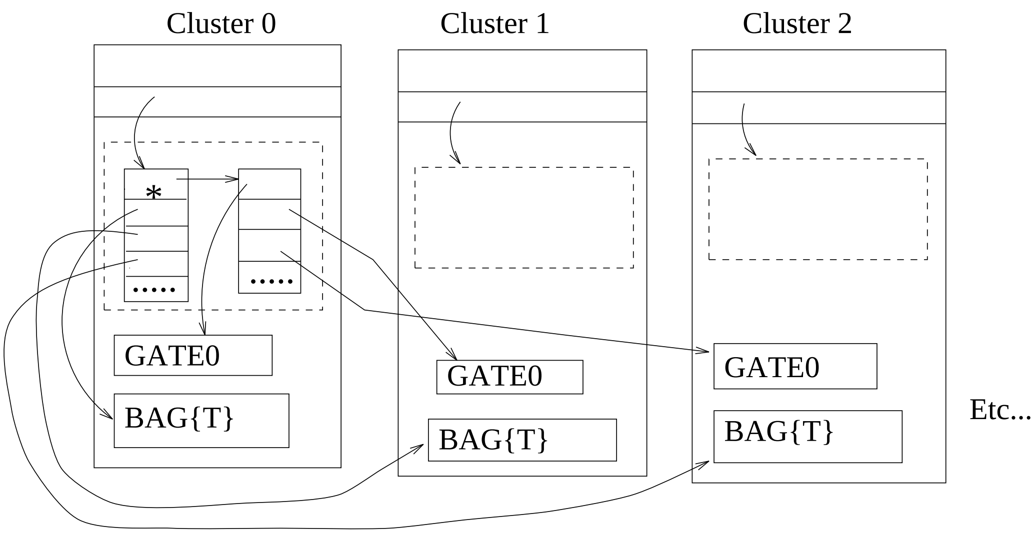


Figure 4.11: Alternative implementation of `DISTRIB_BAG{T}` based on `SPREAD{T}`.

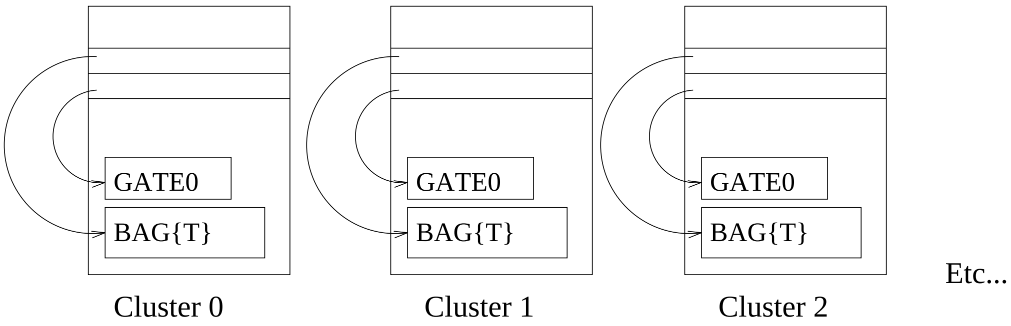


Figure 4.12: Another alternative implementation of `DISTRIB_BAG{T}` in pSather.

does not happen in the current workbag (Section 4.1) because when T1 locks the gate of a remote sub-bag, its locus of control remains on cluster 1. Fairness of access to gate objects guarantees that T1's lock operation will eventually succeed, allowing T1 to look for any task in cluster 0's sub-bag.

Directory Per Thread vs. Per Cluster

Our workbag (in Section 4.1) duplicates the directory of pointers for every thread. If there are multiple threads per cluster, one possible implementation is for threads on the same cluster to share a directory (Figure 4.11). In this case, however, we can no longer allocate the per-thread termination detection flag in the local directory; we would need a more sophisticated way to detect termination (e.g. using a bit array to implement multiple flags per directory).

Non-Spread vs. Spread Directory

We use an array as a directory, so the directory is located on a single cluster. An alternative is to spread out the directory of pointers, so that each cluster gets its sub-bag and lock pointers (Figure 4.12). This cannot be implemented in the current prototype because there is no way to group two pointers together by making `DISTRIB_BAG{T}` inherit from `SPREAD{{GATE0,BAG{T}}}`.[5] There is also the additional inefficiency that to perform an operation on a remote sub-bag, a thread has to first read the pointers from remote clusters.

4.1.8 Summary

The workbag class illustrates the following points.

- By encapsulating the functionalities of a workbag in a class the user is protected from dealing with certain detailed synchronization issues (e.g. protection of sub-bags).
- Certain aspects of parallelism are, however, still visible to the user, particularly the creation of parallel threads and efficiency of accessing objects. This deficiency can be overcome with the iter construct in 1.0 when we provide a master-worker abstraction to the user (Section 4.9.1) instead of a workbag. In 1.0, it also becomes simpler to use code inheritance to implement workbags with different internal search strategies (Section 4.9.1).
- The encapsulation mechanisms also provide a way for library programmers to implement parallel libraries on top of existing sequential library classes. This simplifies the job of library programmers and demonstrates that the object-oriented paradigm not only helps the general users but also the library programmers.

[5] `{GATE0,BAG{T}}` is the tuple type in 1.0 (Section 1.2.1), and `SPREAD{T}` is described in Section 2.5.3.

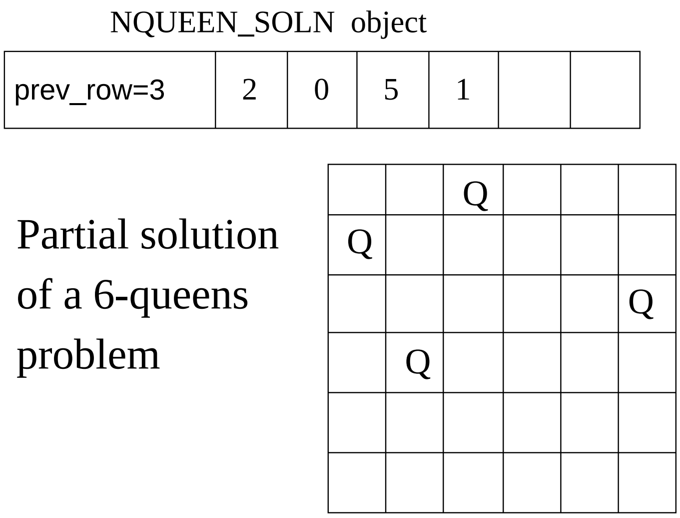


Figure 4.13: An `NQUEEN_SOLN` object and the chess board it represents.

4.2 Application of Workbag I — N-Queens

We show how the distributed workbag can be used in an N-queens search problem. The goal is to count all possible ways to place N queens on an N by N chess board, such that every queen is safe from all other queens. (A queen at position (i,j) is not safe if there is another queen which is on the same row, column or either of the two diagonals crossing (i,j).)

A "task" in this program is an `NQUEEN_SOLN` object; it is an N-element array with an attribute `prev_row`. The ith element ($0 \leq i \leq$ `prev_row`) contains the column position of a queen in row i. The positions of the queens in the array are all safe. An `NQUEEN_SOLN` object therefore represents a partial solution and is a node in the search tree. Figure 4.13 shows the configuration of a chess board represented by a corresponding `NQUEEN_SOLN` object.

Figure 4.14 shows the pSather code executed by each worker thread. Before forking off the threads, the workbag is initialized with N tasks; the ith task contains a queen on row 0, column i. Each worker first makes a local copy of the directory (line 3). It also finds out which cluster it is in (line 4), and the number of processors used (line 5). The thread uses `local_cid` when moving objects to the local cluster. Since one thread is created for each processor, `numprocs` gives the number of workers. The loop (lines 6–43) is where the main work is done.

A thread first tries to retrieve a partial solution from the workbag (line 7). If none is found, the thread signals to the workbag that it has become quiescent (line 8) and checks whether the computation has terminated (lines 10–12).

If a partial solution is found, it is first moved to the local cluster (lines 14–15). We use a `with-near` statement (line 16) to assert the fact that `near_poss_soln` references a local solution

```
search is
   -- Make local copies of directory to prevent bottleneck.
   workbag := workbag.dir_copy;

   local_cid:INT := CONFIG::current_cluster_id;
   numprocs:INT := CONFIG::current_numcpus;
   loop
      poss_soln:NQUEEN_SOLN := workbag.get;
      if (poss_soln = void) then
         workbag.signal_quiescent;
         if (workbag.num_quiescent_threads = numprocs) then
            break;
         end; -- if
      else
         near_poss_soln:NQUEEN_SOLN
            := SYS::move_to(poss_soln, local_cid);

            with near_poss_soln near
               i:INT := near_poss_soln.prev_row;
               -- 'i' is the last row updated.

               curr_row:INT := i + 1;
               if (curr_row >= nqueens) then
                  local_num_solns:=local_num_solns+1;
               else
                  -- Search for possible positions to put
                  -- queen in current row.
                  prev_col:INT := near_poss_soln.prev_column;
                  j:INT;
                  until (j >= nqueens) loop
                     if (j /= prev_col) and (j /= (prev_col-1))
                        and (j /= prev_col+1) then
                     if (near_poss_soln.is_valid(curr_row, j))
                     then
                        new_poss_soln:NQUEEN_SOLN
                           := near_poss_soln.copy;
                        with new_poss_soln near
                           new_poss_soln.update_row(curr_row, j);
                           new_poss_soln.update_prev_row(curr_row);
                           workbag.insert(new_poss_soln);
                        end;
                     end; end;
                  j := j + 1;
               end; end;
         end; end; end;
   end; -- search
```

Figure 4.14: Actual code for worker thread in N-queens problem using `DISTRIB_BAG{T}`.

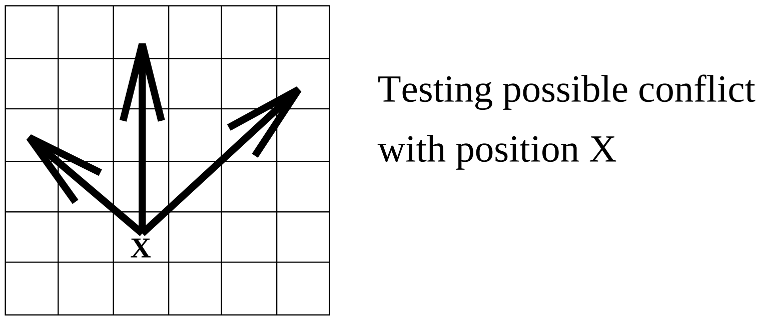


Figure 4.15: Testing positions which may conflict with **X**.

object, so that the compiler might optimize the generated code. Then we look at the last row processed in the partial solution (line 17) to find out which row we should work on next (line 19). If we have exceeded N, the size of the board, that means the solution is in fact complete, so we increment the number of solutions by 1 (lines 20–21).[6] If the solution is not complete, we try to put a new queen into the new row (given by `curr_row`). To do so, first we find out the column position of the queen in the previous row; this value is stored in `prev_col`. Then for each of the possible column positions (lines 26–40), except those directly in conflict with the previous positions (lines 28–29), we make a call:

```
near_poss_soln.is_valid(curr_row, j)
```

to test whether the position given (`curr_row`)th row and `j`th column is safe. If it is, we make a local copy of the partial solution (lines 32–33), and fill in the new row and column positions (lines 35–36). The new partial solution is inserted into the workbag.

Figure 4.15 shows how the `is_valid` routine tests the safety of a given position. When there is a queen occupying a position that is directly above, or on the top-left or top-right diagonal, the position is unsafe.

4.2.1 Performance of N-Queens Program

In Section 4.1.5, we pointed out there are various strategies to retrieve tasks from a workbag. We tried out some of the strategies. Table 4.2 illustrates the timings on a CM-5.[7] The strategies used are:

[6] The thread has a local counter `local_num_solns` to keep track how many complete solutions it has found. To find the total number of complete solutions, we sum up the local counters.

[7] The 1-processor times ought to be similar; they differ because of different memory requirements and timing disturbance from the system memory allocator whenever the system heap is increased. When a large heap is preallocated, the 1-processor times for breadth-first, depth-first, mixed and mixed-tree are 38.05, 37.67, 37.34 and 37.22 seconds respectively.

No. of processors	Breadth-first	Depth-first	Mixed	Mixed-Tree
1	41.59	39.17	37.63	38.02
2	33.21	21.23	19.22	19.98
4	19.05	10.25	9.52	9.92
8	9.67	5.10	4.76	5.02
16	3.91	3.12	2.68	2.72
32	4.11	1.78	1.49	2.21
64	4.48	1.12	0.94	1.32

Table 4.2: Execution times (in seconds) for 11-queens program.

Version	N = 10	N = 11	N = 12	N = 13
C	2.70	13.57	74.44	434.50
pSather/V1	3.15	16.20	87.31	511.20
pSather/V2	2.94	15.25	83.01	490.61
pSather/V3	4.13	20.42	106.64	608.14
Mixed	7.18	37.63	209.90	1260.88

Table 4.3: Execution times (in seconds) for different sequential N-queens program.

Breadth-first Both local and remote sub-bags keep tasks in a FIFO fashion. Remote sub-bags are searched in a cyclic fashion starting from sub-bag 0, 1, ... $C-1$. (C = number of clusters)

Depth-first Both local and remote sub-bags keep tasks in a LIFO fashion. Remote sub-bags are searched in a cyclic fashion starting from sub-bag 0, 1, ... $C-1$. (C = number of clusters)

Mixed Local sub-bag is LIFO while remote sub-bags are FIFO. Remote sub-bags are searched in a cyclic manner starting from sub-bag 0, 1, ... $C-1$. (C = number of clusters)

Mixed-Tree Local sub-bag is LIFO while remote sub-bags are FIFO. If remote sub-bags are searched, it is done in a tree-like fashion. This may alleviate contention to the 0th sub-bag, if there are tasks in the nearby sub-bags.

When using a workbag for the N-queens problem, we find that a mixed strategy generally gives better performance than either breadth-first or depth-first. In a mixed strategy, when a thread runs out of tasks, by getting tasks from the front of remote sub-bags, it tends to get tasks which are higher up in the search tree and generate more local tasks, reducing further remote accesses. A mixed-tree strategy reduces the contention to a particular remote sub-bag (e.g. sub-bag at cluster 0) if the tasks are more evenly distributed. Otherwise, it can result in more remote accesses, which offset the gains from reducing contention.

We obtained several sequential measurements (Table 4.3) to get a better perspective of the performance. All the sequential programs use a depth-first strategy. (All pSather programs are compiled with the same compiler options as the parallel program. The C compiler used in all cases is `gcc` version 2.4.5 with `-O` optimization.)

C. We implemented the algorithm in a sequential C program and ran it on a single CM-5 node.

pSather/V1. We change the parallel pSather version to a sequential one in the following ways. (1) Instead of forking multiple threads for `search` (Figure 4.14), a routine call is made. (2) We use a sequential class abstraction `BAG{T}` instead of a `DISTRIB_BAG{T}`. (3) The bag is not protected by locks (unlike the sub-bags in `DISTRIB_BAG{T}`. (4) There is no need to detect termination. The `BAG{T}` routines are not inlined. Since this is a sequential program, the compiler does not generate polling points because there is no communication between processors.

pSather/V2. This is the same as pSather/V1, except that the `insert` and `get` routines on the `BAG{T}` are inlined.

pSather/V3. We compile the parallel Mixed strategy which uses `DISTRIB_BAG{T}` but the compiler does not generate polling points. Like the parallel programs (and pSather/V2), the `BAG{T}` routines are inlined.

Before we discuss the speedups of the parallel programs, we make some observations about the program's overhead costs in a parallel environment. A sequential Sather/pSather program on CM-5 (pSather/V1) is about 15% to 20% slower than a corresponding C program. This is consistent with the 10% penalty reported in [164]. The additional penalty is incurred by having long pointers on CM-5. Even though the compiler optimizes pointer dereferencing using the information provided by `with-near` statement (Section 3.5.5), the C compiler is able to optimize less with the address part of a long pointer than with a C pointer (Section 3.6).

With inlining, the performance penalty of a sequential pSather program (pSather/V2) relative to C dropped to 8 – 13%. Comparing pSather/V2 to pSather/V1, inlining improves the timings by 4 – 7%.

pSather/V3 shows the overhead incurred by a parallel program executing sequentially. The sources of overheads are: (1) the `lock` statements used to protect the sub-bags in `DISTRIB_BAG{T}`, (2) an extra level of indirection from the `DISTRIB_BAG{T}`. Although the `BAG{T}` routines are inlined, the `DISTRIB_BAG{T}` are not; thus the number of routine calls is more comparable with pSather/V1. The overheads make pSather/V3 about 19 – 31% slower than pSather/V1. We also compare pSather/V3 with C because it is the best circumstance under which we can take a parallel program and run it on a sequential processor without modification. The timings show that pSather/V3 is 40 – 53% slower than a sequential C program.

pSather/V3 shows the timings one would ideally get on a distributed-memory machine that supports interrupt-driven active messages. However, polling is used in the current implementation on CM-5, and the Mixed strategy shows that the polling overhead slows down the timings of pSather/V3 by 74 – 107%. Thus polling overheads, long pointers and the synchronization costs in `DISTRIB_BAG{T}` make pSather/V3 160 – 190% slower than a corresponding C program.

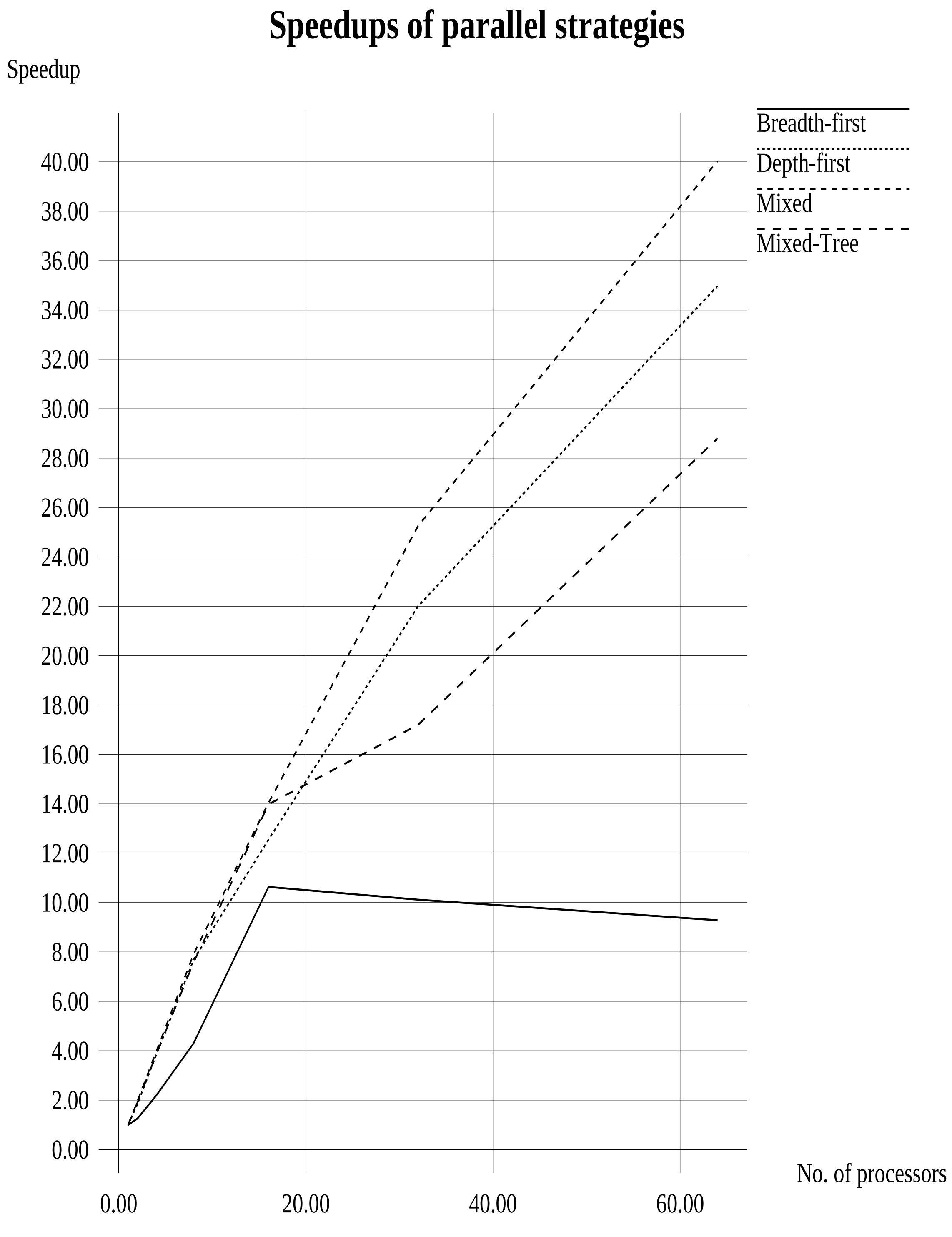


Figure 4.16: Speedups w.r.t. the same parallel program on 1 node.

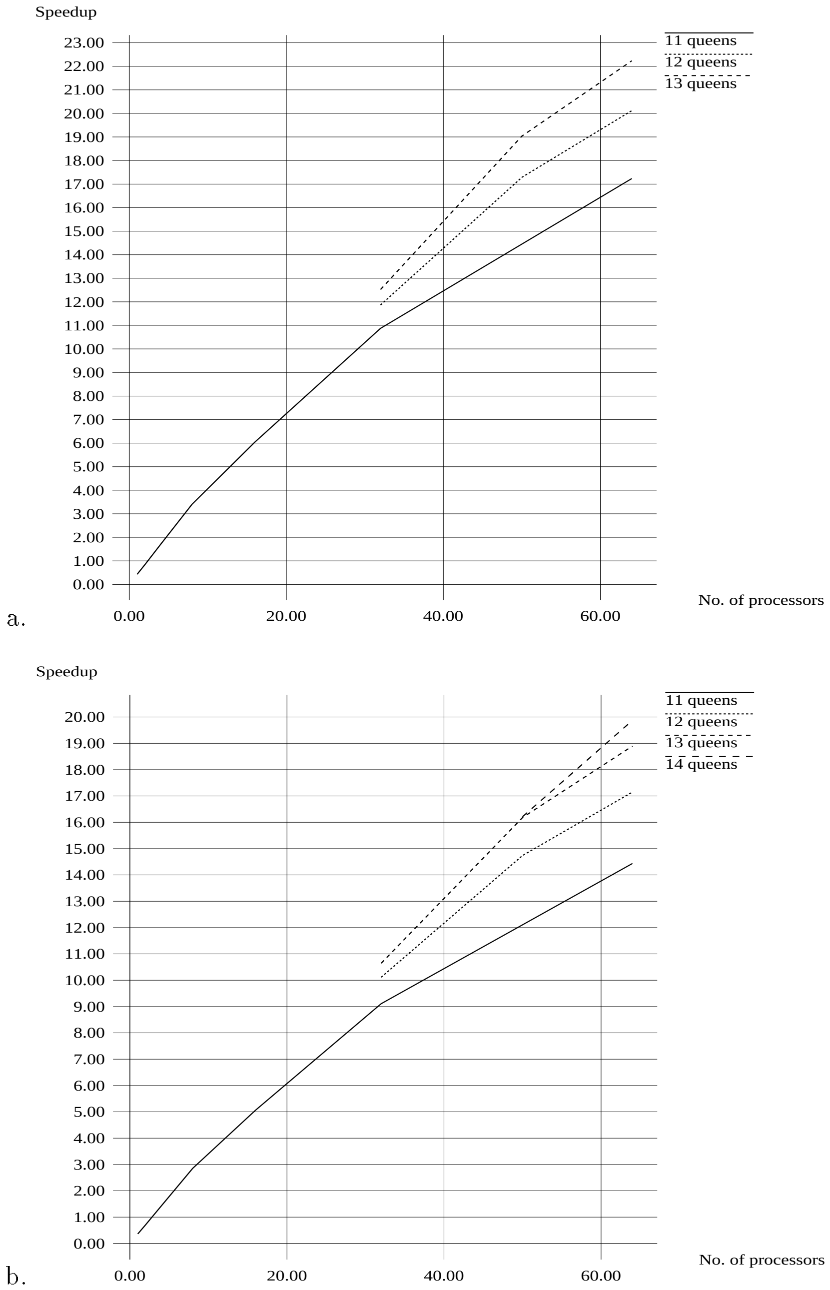


Figure 4.17: Speedups of Mixed w.r.t. (a) pSather/V1 program on 1 node CM-5 and (b) sequential C program on 1 node CM-5. The time of C program for 14 queens is 2716.84 s.

No. of processors	N = 12	N = 13	N = 14
32	7.36	40.82	261.05
50	5.05	26.85	167.56
64	4.34	22.99	136.71

Table 4.4: Execution times (in seconds) for different sequential N-queens program.

Figure 4.16 shows the speedups of different strategies with respect to the times of the same parallel program on 1 node. Since the Mixed version has the best parallel performance, we use this to get the times for larger inputs N = 12, 13 and 14 (Table 4.4). We then compare the parallel times against the C and pSather/V1 versions to get more realistic speedups.

Figure 4.17(a) shows the speedups of Mixed for N = 11, 12 and 13 when compared to a sequential Sather program.[8] This gives an idea of the parallel overheads and performance improvements when one is working exclusively in a Sather environment. Figure 4.17(b) shows the speedups of Mixed for N = 11, 12, 13 and 14 w.r.t. to a sequential C program on a single node. This tells us the possible performance gains when one changes platform from e.g. a SPARCstation (with performance comparable to a single CM-5 node) to programming a CM-5 using the current pSather prototype.

4.3 Replicated Hash Table

In this section, we look at another library class that is used in two parallel programs: finding primes (Section 4.4) and a parallel variation of Buchberger's algorithm to compute Gröbner basis [48] (Section 4.7).

The class is a replicated hash table (Figures 4.18, 4.19, 4.20). It combines the functionalities of a hash table and a software cache. It is also an example of how the programmer can flexibly build a memory consistency protocol based on the needs of the program and not rely on the system to provide a set of predefined protocols (e.g. Munin [60, 32]).

The hash table can be used as either a map or set abstraction and is useful in the following general situation: The computation has a monotonically increasing global state which consists of the results of different threads. These results are associated with unique integer id's. When a result is inserted into the hash table, it is never deleted. A thread's execution uses part or all of the global state and thus it may need results of threads from other clusters. Since the global state increases monotonically, results needed by local threads can be cached to improve data-locality. This caching is done within the hash table and is transparent to the user, who simply uses the hash table interface to retrieve the results.

[8] To avoid excessively long execution times, for N = 12 and 13, we did not run the program with the smaller number of processors.

```
class REPL_INT_HASH_MAP{T} is
  -- This class associates entities of type 'T' to keys of type
  -- 'INT' in a distributed memory machine.  'T' has routine
  -- 'make_copy_to(cid:INT):T' in its interface.
  -- The hash table is replicated on every cluster.

  SPREAD{PROTECTED_INT_HASH_MAP_HEADER{T}};

  constant local_cid:INT := CONFIG::current_cluster_id;

  private local_set:PROTECTED_INT_HASH_MAP{T} is
    -- Return the local cache table.
    res := [local_cid].local_set;
  end; -- local_set

  private shared_set:INT_HASH_MAP{T} is
    -- Return the shared table.
    res := [local_cid].shared_set;
  end; -- shared_set

  private shared_set_gate:GATE0 is
    -- Return the gate for shared table.
    res := [local_cid].gate;
  end; -- shared_set_gate

  private waiting:INT_HASH_MAP{GATE0} is
    -- Return the hash table for waiting threads.
    res := [local_cid].wait_gates;
  end; -- waiting

  ------------------------------
  create:SELF_TYPE is
    -- Allocate an empty replicated hash table.
    res := new;
    num_clusters:INT := CONFIG::current_num_clusters;
    s:INT_HASH_MAP{T} := INT_HASH_MAP{T}::create;
    gs:INT_HASH_MAP{GATE0} :=
      INT_HASH_MAP{GATE0}::create;
    g:GATE0 := GATE0::new;
    i:INT;
    until (i >= num_clusters) loop
      -- Allocate a chunk header with the corresponding gate
      -- object and local table.
      res[i] := PROTECTED_INT_HASH_MAP_HEADER{T}::create(s,g,gs) @ i;
      i := i+1;
    end; -- loop
  end; -- create
```

Figure 4.18: Part 1 of definition of REPL_INT_HASH_MAP{T} implemented in current prototype.

```
get_wait(k:INT):T is
  -- Get the entry associated with key 'k'; current thread suspends
  -- if entry is absent.

  s:PROTECTED_INT_HASH_MAP{T} := local_set;
  with s near
    res := s.get(k);
    if (res = void) then
      g:GATE0 := shared_set_gate;
      -- Then wait on any insertion into the shared table.
      lock g then
        res := shared_set.get(k) @ shared_set.where;
        if (res /= void) then
          -- If we find something, then the value is cached.
          unlock g;
          if (res.is_far) then
            res := res.make_copy_to(local_cid);
            s.insert(k, res);
          end; -- if
        else
          wait_gate:GATE0 :=
            waiting.get(k) @ waiting.where;
          if (wait_gate = void) then
            wait_gate := GATE0::new;
            waiting.insert(k, wait_gate) @ waiting.where;
          end; -- if
          unlock g;
          wait_gate.read;
          -- Wait until it is set.
          lock g then
            res := shared_set.get(k) @ shared_set.where;
            if (res.is_far) then
              res := res.make_copy_to(local_cid);
              s.insert(k, res);
            end; end; end;
      end; end; end;
end; -- get_wait
```

Figure 4.19: Part 2 of definition of REPL_INT_HASH_MAP{T} implemented in current prototype.

```
get_cached(k:INT): T is
  -- The entry associated with key 'k' or 'void' if absent.
  s:PROTECTED_INT_HASH_MAP{T} := local_set;
  with s near
    res := s.get(k);
    if (res = void) then
      -- Then check the shared (up-to-date) table.
      lock shared_set_gate then
        res := shared_set.get(k) @ shared_set.where;
      end; -- lock
      if (res /= void) and (res.is_far) then
        -- If we find something, then the value is cached.
        res := res.make_copy_to(local_cid);
        s.insert(k, res);
      end; end; end;
end; -- get_cached

get(k:INT): T is
  -- The entry associated with key 'k' or 'void' if absent.
  -- Unlike "get_cached", this does not have the side effect of
  -- copying objects.
  lock shared_set_gate then
    res := shared_set.get(k) @ shared_set.where;
  end; -- lock
end; -- get

insert(k:INT; e:T) is
  -- Insert key 'k' with entity 'e'.  Ignore if present.
  s:PROTECTED_INT_HASH_MAP{T} := local_set;
  with s near
    s.insert(k, e);
    lock shared_set_gate then
      wait_gate:GATE0 := insert_help(k, e) @ shared_set.where;
      if (wait_gate /= void) then
        wait_gate.set end;
    end; end;
end; -- insert

private insert_help(k:INT; e:T):GATE0 is
  shared_set.insert(k, e);
  -- "shared_set" is located on the same cluster as the hash table
  -- of waiting-gates.
  res := waiting.get(k);
end; -- insert_help

end; -- class REPL_INT_HASH_MAP{T}
```

Figure 4.20: Part 3 of definition of REPL_INT_HASH_MAP{T} implemented in current prototype.

```
class PROTECTED_INT_HASH_MAP_HEADER{T} is
  shared_set:INT_HASH_MAP{T};
  -- Shared by all other replicated local sets.

  local_set:PROTECTED_INT_HASH_MAP{T};

  wait_gates:INT_HASH_MAP{GATE0};
  -- Store the gates on which the threads are waiting.

  gate:GATE0; -- Gate for shared set.

  create(s:INT_HASH_MAP{T}; g:GATE0;
        gs:INT_HASH_MAP{GATE0}):SELF_TYPE is
    res := new;
    res.shared_set := s;
    res.gate := g;
    res.wait_gates := gs;
    res.local_set := PROTECTED_INT_HASH_MAP{T}::create;
  end; -- create

end; -- class PROTECTED_INT_HASH_MAP_HEADER{T}
```

Figure 4.21: Definition of a header class used in `REPL_INT_HASH_MAP{T}`.

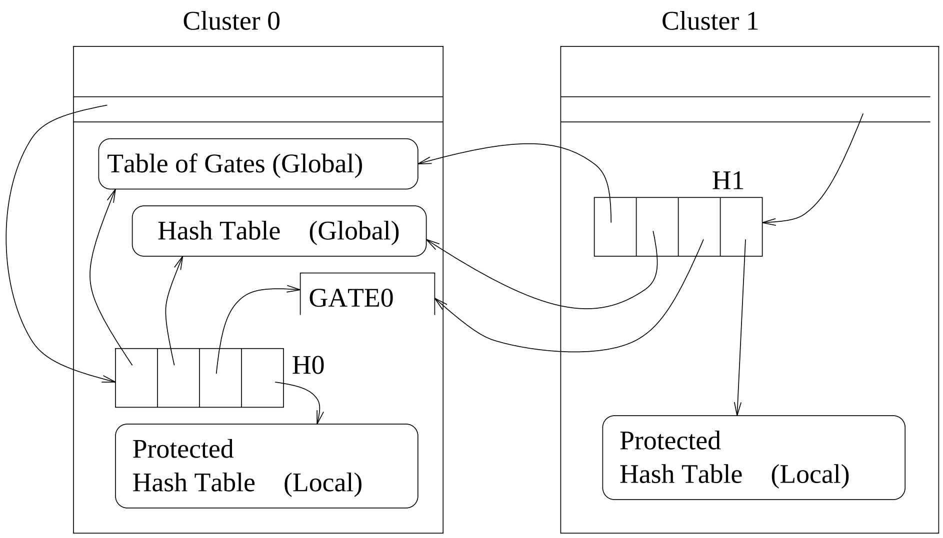


Figure 4.22: A replicated hash table for two clusters.

4.3.1 Data Structure

The replicated hash table is a spread object (Figure 4.22). On each cluster, we have a header object with the following attributes.

- `shared_set`: This is a pointer to a centralized hash table which stores all the key-object pairs.
- `local_set`: Each header also has a pointer to a local hash table which serves as the local "cache". Each local hash table is protected by its own internal lock, so that its insertion and retrieval operations work atomically.
- `wait_gates`: When a thread performs a lookup on a key, the associated object may not be available yet. The hash table uses a gate to suspend such a thread. Every header has a pointer to a centralized key-gates hash table which stores the gates.
- `gate`: Both centralized hash tables are protected by a shared gate object.

The centralized tables are implemented by a sequential, non-protected `INT_HASH_MAP{T}` class from the existing Sather library. Each local hash table is an object from the `PROTECTED_INT_HASH_MAP{T}` class (Appendix D) with an internal lock which ensures that its insertion and retrieval operations work atomically. We use centralized tables for the key-object pairs and the gates, although distributed tables might be preferable. However, since our focus is on improving data locality by caching, the centralized tables work relatively well. Figure 4.21 gives the class definition of the header objects.

4.3.2 Interface

The interface to the replicated hash table has the following routines:

- `create`. This allocates an empty replicated hash table.
- `get_wait(k:INT):T`. This returns the object (of type `T`) associated with key `k`. If the object is missing, the executing thread is suspended on an unbound gate object. When some other thread inserts the key-object (`k`, val) into the hash table, the gate is set. The thread is resumed and gets val (or a locally-cached copy of val) as the returned value. The library designer, by providing the `get_wait` routine, gives a clean interface to the user who does not need to do caching explicitly.
- `get_cached(k:INT):T`. This returns the object (of type `T`) associated with key `k`. If the object is missing, the value `void` is returned. (Unlike `get_wait`, the executing thread is not suspended.) If the object is present but not in the local cluster, then a copy of it is inserted into the local hash table and the new copy is returned.

- `get(k:INT):T`. This works like `get_cached`, except that there is no attempt to make a local cached copy of the non-void object.
- `insert(k:INT; e:T)`. This routine inserts the pair (`k`,`e`) into the shared hash table, and wakes up any suspended threads waiting for the object associated with `k`.

4.3.3 Implementation Details

Each replicated hash table (Figure 4.18) has a constant feature. The constant feature is replicated (Section 2.5.4) and is initialized such that the instance on cluster i contains the id of cluster i. There are four private access routines, which retrieve the attributes of the header object. A thread accesses the nearest header object by indexing using the local cluster id. For example, a thread gets the pointer to the shared hash table via:

```
res := [local_cid].shared_set;
```

The `create` routine (Figure 4.18) first allocates a spread object. It then creates a sequential (unprotected) hash table which serves as the centralized table. It also creates a sequential (unprotected) hash table of gate objects which are used to suspend and resume threads. Both these shared tables are protected by a gate object. The routine then allocates a header for each cluster; each header has a pointer to a local hash table (which acts as the cache). All headers contain the same pointers to the shared tables and gate. The setup for a 2-cluster machine is shown in Figure 4.22.

Next we look at the `get_wait` routine (Figure 4.19). Given a key `k`, we first search the local cache. If an object is found, we return it as the result. Otherwise, we lock the shared gate and search the shared hash table. If an object is found from the shared table, we immediately unlock the gate and check whether the object is near or far (with respect to the executing thread). If the object is far, we make a local copy of it and store this copy in the local cache.[9] If no object is found from the shared table, the gate remains locked and we search the `INT_HASH_MAP{GATE0}` table for any gate G associated with `k`. If no gate is found, a gate G is created and the pair (`k`, G) is inserted into the `INT_HASH_MAP{GATE0}` table. After getting a gate G associated with `k`, we unlock the shared gate and suspend the executing thread:

```
unlock g;
wait_gate.read;
```

Even though the unlocking and thread suspension does not execute atomically (unlike the wait operation on condition variable in Mesa's monitor [154]), the code works correctly because we maintain the following invariant: G is bound iff there exists an object O1 such that (`k`, O1) is in the shared hash table.

[9] Two or more threads may simultaneously try to insert copies for the same key into the same local table. Since the local table's insert operation is atomic and later insertions overwrite earlier ones, the result is still correct.

When the thread is resumed, it is guaranteed that the shared table contains an entry (**k**, O1). So the resumed thread simply looks up the shared table for O1, caches a copy of it (if O1 is far) and returns the result.

The next routine `get_cached` (Figure 4.20) is simpler than `get_wait`. Given `k`, it first checks whether the local cache contains an object associated with `k` and returns the object if found. If not, it checks the shared table. If it also fails to find any entry in the shared table, the thread simply returns `void`. Otherwise, if the found object is far (in another cluster), we make a local copy of it and store the copy in the local cache. The routine always returns a pointer to a near object or `void`.

The `get` routine (Figure 4.20) simply locks the shared gate, looks up `k` in the shared table and returns any result that is found. It is similar to the sequential hash table's `get` routine, except that we use the `lock` statement to protect access to the shared (sequential) table.

The `insert` routine (Figure 4.20) first inserts the pair (`k`, O1) into the local cache. This avoids any future need to look in the shared table, for the object associated with `k`. Then we insert (`k`, O1) into the shared table and look for any gates associated with `k` in the table of gates (referenced by `waiting`). If there is, the gate is set to signal that there is now an object O1 associated with `k`. The `insert` routine uses an `insert_help` helper routine to reduce overhead costs for remote calls. Since both tables referenced by `shared_set` and `waiting` are located on the same cluster and we have to invoke a routine for each table, it is more efficient to do one remote call that performs two local routine calls than to do two remote calls (each performing one local call). In the next subsection, we will explain why `insert_help` is not written in some other ways.

4.3.4 Overcoming Lack of Fairness Guarantee

There are several implementation details that are due to pSather's not guaranteeing fair scheduling for threads and remote calls.

The first is that in the `get_wait` routine, the executing thread is suspended when no object is found associated with a key `k`. We do not use a loop to check the shared table repeatedly:

```
loop
  lock shared_table_gate then
    x:POLYNOMIAL := shared_table.get(k) @ shared_table.where;
  end;
  if (x /= void) then
    if (x.is_far) then
      <Store a cached copy of "x".>
      break;
    end; end;
end;
```

This is because without fair scheduling for threads, the thread that would eventually insert (`k`, O1) may never be scheduled if the executing thread does not suspend and relinquish processor control.

Another implementation detail is that although the shared table and its gate are allocated on the same cluster (say $c1$), we do not perform the lock/unlock operations and invoke `shared_set.get(k)` (or `shared_set.insert(k,e)`) on $c1$. The lock/unlock operations are always executed locally. For example, the code from `get_cached` is:

(A)

```
lock shared_set_gate then
  res := shared_set.get(k) @ shared_set.where; end;
```

Since remote gate operations are more expensive than local gate operations (Section 3.6.4), why do we not perform the lock/unlock operations and `shared_set.get(k)` altogether on $c1$? The reason is that if we invoke the combined lock/get/unlock operation on $c1$, the lack of fairness guarantee might delay the combined operation indefinitely. By writing the code as in (A), the local thread can compete for the gate "fairly" with other remote threads (cf. implementation of gate operations in Section 3.3.4). After the gate is locked, we know that `shared_set.get(k)` will not be delayed at $c1$ because the `get` routine in the unprotected hash table is a non-suspendable routine and $c1$ services remote calls to non-suspendable routines immediately[10] (cf. implementation of remote calls in Section 3.3.2).

A third detail is that we did not use a protected table for the centralized hash table. If `shared_set` had been implemented as a protected hash table (`PROTECTED_INT_HASH_MAP{T}`), we would have written:

```
shared_set.insert(k, e) @ shared_set.where; -- (*)
```

in lieu of the following:

(B)

```
lock shared_set_gate then
  wait_gate:GATE0 := insert_help(k, e) @ shared_set.where;
  if (wait_gate /= void) then
    wait_gate.set end; end; end;
```

in the `insert` routine (Figure 4.20). In (*), the lack of fairness means that the remote call might never be scheduled. To prevent indefinite delay, we use the knowledge that the prototype does not delay non-suspendable remote operations (Section 3.3.2). Since the `insert` routine for the unprotected hash table (`INT_HASH_MAP{T}`) does not have synchronization and is non-suspendable, `insert_help` (Figure 4.20) is also a non-suspendable routine. This non-suspendability property guarantees that in (B):

```
insert_help(k, e) @ shared_set.where;
```

will definitely execute and complete.

[10] Since we want to prevent any threads from being starved (i.e. unable to access the shared table), it is reasonable to use some knowledge about the system to get the desired effect.

4.4 Application II — Finding Primes

Our second application finds the primes between 2 and N (inclusive), where N is the input parameter. This algorithm uses both the workbag (Section 4.1) and replicated hash table (Section 4.3), and shows how the library classes can be reused in different applications.

One algorithm that computes primes is the sieve of Eratosthenes. Bokhari [43] described a parallel implementation of the sieve on a Flex/32 multiprocessor. It used a master-worker model, where each time the master program finds a prime p, it asks an idle (worker) process to mark out multiples of p. It was, however, unable to obtain speedup beyond six processors. Lansdowne et al. [157] describes a better parallel implementation with better speedup (speedup of about 18.4 on a 20-processor CSI-150 system). Their approach was to find all primes from 2 and $\sqrt{N}$ sequentially, and divide the range $\sqrt{N}+1$ to N among the processors. Each processor gets all the primes, and uses them to cancel out the multiples in its range.

Our implementation follows the algorithm used in Carriero et al. (Chapter 5 of [59]) and uses a master-worker model.[11] Each worker thread repeatedly retrieves an odd number v from a distributed workbag and works on the numbers (v, v+2, v+4, ..., v+range). Instead of using known primes to filter out non-primes, the worker checks, for each odd number i in the range, whether it is divisible by all primes from 2 to $\sqrt{i}+1$. When looking up the primes $\leq \sqrt{i}+1$, some of the primes may still be in the process of being computed by other threads. This synchronization is provided by the replicated hash table which suspends a thread until the set of primes it is looking for has been computed. The computed primes constitute a global state which grows monotonically and can be cached. The replicated hash table also transparently provides the caching functionality to the user.

Figure 4.23 is the pSather 0.1 routine which is executed by every worker thread. As required by the workbag interface (Section 4.1), each thread first makes a local directory of the distributed workbag (line 1). The master creates the same number of threads as there are processors. In order for the threads to coordinate their termination using the workbag's `num_quiescent_threads` routine, each thread must know the number of workers (line 2). Then the thread enters into a loop (lines 5–46) in which it repeatedly retrieves a task from the workbag (line 6), inserts a new task into the workbag (lines 11–12), and work on the task (lines 13–45).

Each task is an integer v (stored in variable `next_task`, line 6). The worker works on each odd number i in the range from v to $\min(v + \text{range} - 1,\, N)$ (lines 17–18), where range is given by the variable `grain`. In line 14, we create a new list to store any primes found in the range. This list of primes is associated with v and stored in the replicated integer hash table (referenced by variable `primes`).

The variable `jth_list` (line 15) holds successive lists of primes which may be factors of i. The variable `i` (line 19) varies over the range of odd numbers.[12] If i is not a prime, then its prime

[11] The master just starts the worker threads and waits for them to complete. It does not interact with the workers.
[12] `i` is incremented by 2 in each iteration, line 42.

```
find_primes is
   -- Make local copies of directory to prevent bottleneck.
   workbag := workbag.dir_copy;
   np:INT := CONFIG::current_numcpus;
   loop
      next_task:INT := workbag.get;
      if (next_task = 0) then
         workbag.signal_quiescent;
         if (workbag.num_quiescent_threads = np) then break end;
      else
         new_task:INT := next_task+np*grain;
         if (new_task <= max) then workbag.insert(new_task) end;
         -- Find primes between 'next_task' and 'next_task+grain'.
         new_primes:PRIME_LIST := PRIME_LIST::create;
         jth_list:PRIME_LIST;
         with new_primes, jth_list near
            limit:INT := next_task+grain;
            if (limit > max) then limit := max end;
            i:INT:=next_task; until (i >= limit) loop
               max_poss_factor:INT := i.sqrt+1;
               maybe_prime:BOOL:=true; task_id:INT:=first_task_id;
               end_search:BOOL;
               loop
                  if (task_id=next_task) then jth_list:=new_primes;
                  else jth_list:=primes.get_wait(task_id) end;
                  k:INT; list_size:INT := jth_list.size;
                  until (k >= list_size) loop
                     kth_prime:INT := jth_list[k];
                     if (i.u_mod(kth_prime) = 0) then
                        maybe_prime:=false; end_search:=true; break;
                     end; -- if
                     if (kth_prime > max_poss_factor) then
                        end_search:= true; break; end;
                     k := k+1;
                  end; -- loop
                  if (end_search) then break end;
                  if (task_id = first_task_id) then
                     task_id:=start_task_id;
                  else task_id:=task_id+grain end; end; -- loop
               if (maybe_prime) then
                  new_primes:=new_primes.push(i) end;
               i := i+2;
            end; end;
         -- Insert the new list of primes.
         primes.insert(next_task, new_primes);
      end; end; -- loop
end; -- find_primes
```

Figure 4.23: Actual code for worker thread in prime-finding program using DISTRIB_BAG{T} and REPL_INT_HASH_MAP{T}.

No. of processors	$N = 100000$	$N = 800000$	$N = 2000000$
1	9.57	114.44	361.89
2	6.04	75.37	240.66
4	3.20	40.50	129.77
8	1.95	21.40	68.21
16	1.18	11.93	36.86
32	1.01	7.95	23.00
64	0.90	6.01	16.47

Table 4.5: Execution times (in seconds) of primes program when `grain` = 100, and N = 100000, 800000, 2000000.

factors must lie between 2 and $\sqrt{i}+1$ (line 20). To locate the earlier primes (between 2 and $\sqrt{i}+1$), we use the id's of the earlier tasks. We know that the first task is given by `first_task_id` and that each task v has a list of primes associated with it. Therefore we start with (line 21):

```
task_id:INT:=first_task_id;
```

Before looking into the hash table, we first check whether `task_id` is equal to the current task. If so, we look for prime factors from the list currently computed (line 24):

```
jth_list:=new_primes;
```

If not, we call (line 25):

```
jth_list:=primes.get_wait(task_id)
```

The hash table will transparently suspend the thread if another thread has not finished computing the list of primes. It also automatically caches the list of primes. We then check whether any prime in `jth_list` is a factor of i (lines 26–35). If we fail to find any prime factor, we go on to the next list of primes using next task id (lines 37–39):

$$\texttt{first\_task\_id},\ \ \texttt{start\_task\_id},\ \ \texttt{start\_task\_id} + \texttt{grain},\ \ \texttt{first\_task\_id} + 2 \times \texttt{grain}, \ldots$$

At line 26, we have obtained a list of prime factors to be checked against i. We iterate over this list (lines 27–35). If there exists a prime p which divides i, then i is not a prime and we quit the loop (lines 29–31). If p is greater than the maximum possible prime factor of i, we also stop searching (lines 32–33). At the end, if i is a prime, it is inserted into the new list (lines 40–41).

4.4.1 Performance of Primes-Sieve Program

Table 4.5 shows the execution times of the primes program on a 64-processor CM-5 when the grain size is 100 and N ranges from 100000 to 2000000. For N = 8000000 and N = 10000000, the program ran out of memory because of the large number of gate objects which were allocated for waiting threads (in the replicated hash table), but never reclaimed. When we increased the grain

No. of processors	$N = 800000$	$N = 2000000$	$N = 8000000$	$N = 10000000$
1	90.11	273.24	N/A	N/A
2	49.85	155.21	N/A	N/A
4	26.80	83.08	462.40	N/A
8	16.75	47.99	260.75	355.10
16	7.86	23.20	127.11	174.59
32	3.59	10.72	62.31	83.32
64	2.29	6.50	34.27	44.17

(a) `grain` = 1000

No. of processors	$N = 800000$	$N = 2000000$	$N = 8000000$	$N = 10000000$
1	89.76	267.58	N/A	N/A
2	47.99	145.92	N/A	N/A
4	25.02	77.90	436.40	N/A
8	14.29	44.88	251.63	349.28
16	7.34	21.48	126.14	178.03
32	3.60	10.03	56.96	75.87
64	2.17	5.88	32.37	43.53

(b) `grain` = 2000

Table 4.6: Execution times (in seconds) of primes program with N = 800000, 2000000, 8000000, 10000000. (N/A = Not Available, because we want to avoid excessively long running jobs)

```
<Insert 2 into the list of primes>
for (i=3; i<max; i+=2) {
  is_prime = TRUE;
  for (block=first; block!=void; block=block->next) {
    for (j=0; j<block->last_insert; j++) {
      if ((i % block->primes[j]) == 0)
        { is_prime=FALSE; goto end_check; }
      if (i < block->primes_square[j]) { goto end_check; }
    } }
end_check:
  if (is_prime) <Insert i into list of primes>
}
```

Figure 4.24: Code for sequential C primes program.

Program	$N = 100000$	$N = 800000$	$N = 2000000$	$N = 8000000$	$N = 10000000$
C	2.60	38.62	129.53	841.33	1142.03
pSather/V3	5.31	59.25	179.33	1035.29	1382.63

Table 4.7: Execution times (in seconds) of sequential primes programs for N = 100000, 800000, 2000000, 8000000, 10000000. The grain size of pSather/V3 is 2000.

size to 1000, fewer gates were allocated and the insufficient-memory problem disappears. Table 4.6 shows that the execution times improve with a larger grain size.

To calculate the speedups, we measure the timings on a single CM-5 node for two different programs. (We use the same naming convention as for the N-queens program, Section 4.2.1.) All pSather programs are compiled with the same options, and use the `gcc` compiler version 2.4.5 with `-O` optimization.

C. We implemented a simplified version of the algorithm (Figure 4.24) in a sequential C program.

The algorithm keeps the primes and their squared values in a linked list of blocks (20 primes per block). For each number `i`, we check whether it is divisible by any of the primes. If it is, then we go on to the next number. We stop the divisibility checks when `i` is smaller than the square of a prime $\mathtt{p}^2$. This is because if `i` is non-prime, then there must exist a prime $\mathtt{p}_1 < \mathtt{p}$ such that $\mathtt{p}_1$ divides `i`.

pSather/V3. We compile the parallel pSather program without generating polling points.

The sequential times are shown in Table 4.7. It is worth noting the tradeoffs between the pSather and C programs. The pSather program computes the square root function of every number `i`, but the C program does not. The C program calls the square function for every prime found, but the pSather does not. But since the squared values are stored in a linked list of blocks, there is an extra indirection to read each squared value, unlike the pSather program which simply stores the maximum possible factor of `i` in a local variable (line 32, Figure 4.23). PSather/V3 is about 21% – 104% slower than the C program because it has additional synchronization overheads and uses long pointers. PSather/V3 is 104% slower than C when $N = 100000$, but only 21% slower when the problem size is large ($N = 10000000$).

Figure 4.25 shows the speedups of the parallel program using the sequential C times as base, while Figure 4.26 shows the speedups relative to pSather/V3. Although this is highly parallelizable algorithm, the relatively clean structure of the program (Figure 4.23) with synchronization hidden in a replicated hash table class, and the reasonable speedups both serve as another piece of evidence to support our claim that pSather is a clean and efficient parallel object-oriented language.

4.5 Distributed Matrix Classes

Section 2.6.3 briefly describes a possible way to build a distributed matrix using sequential matrices as chunks.[13] It also shows how to write a `transpose` routine on a distributed matrix using the `dist` statement. Another example is a distributed matrix multiplication program (Section 3.3.5). One important characteristic of these examples is that the operations on each chunk can execute

[13] Besides using each chunk to represent either a block of consecutive rows or columns, other partitions are possible. It is up to the library designer to provide various classes.

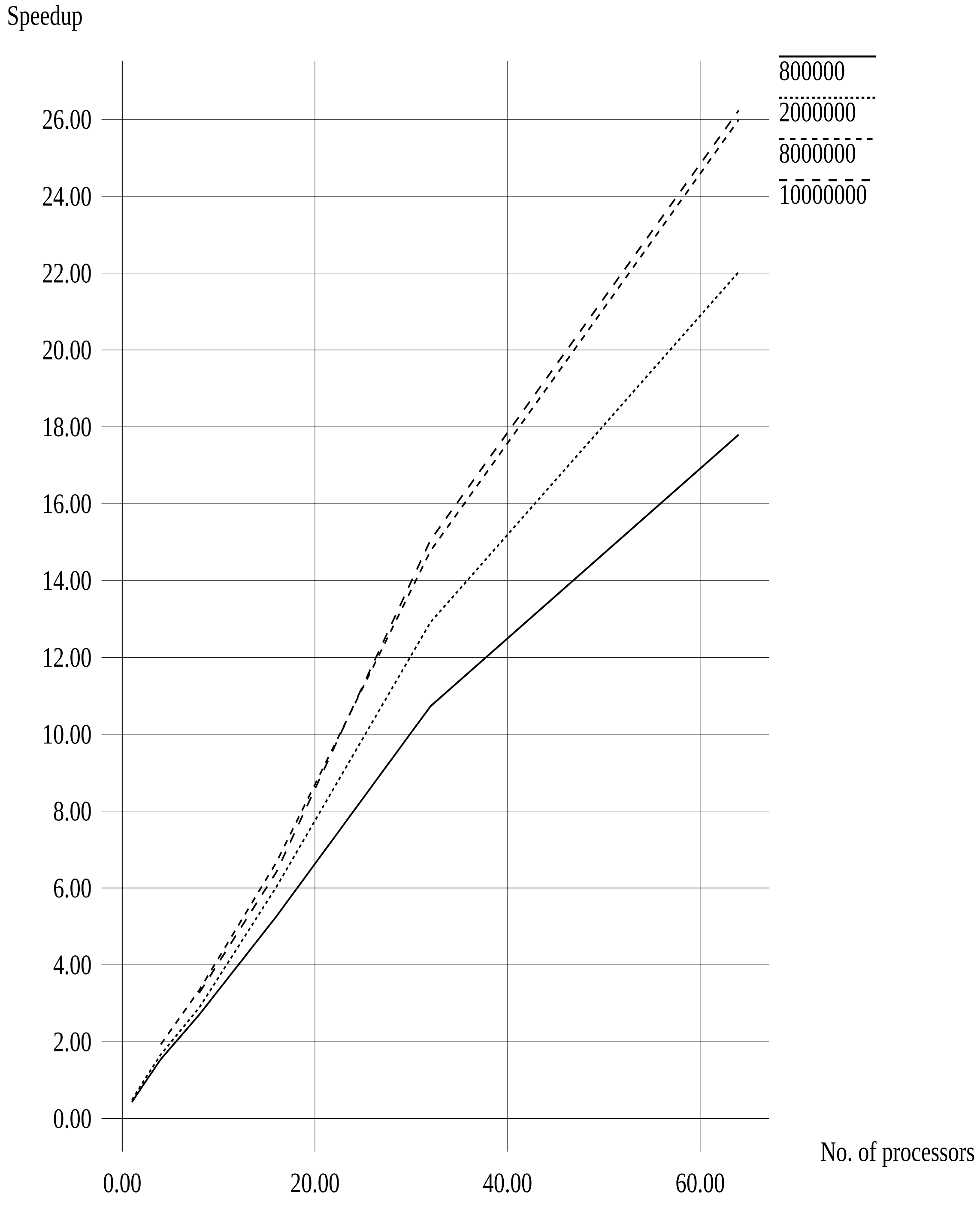


Figure 4.25: Speedups of primes program w.r.t. sequential C program on 1 node CM-5.

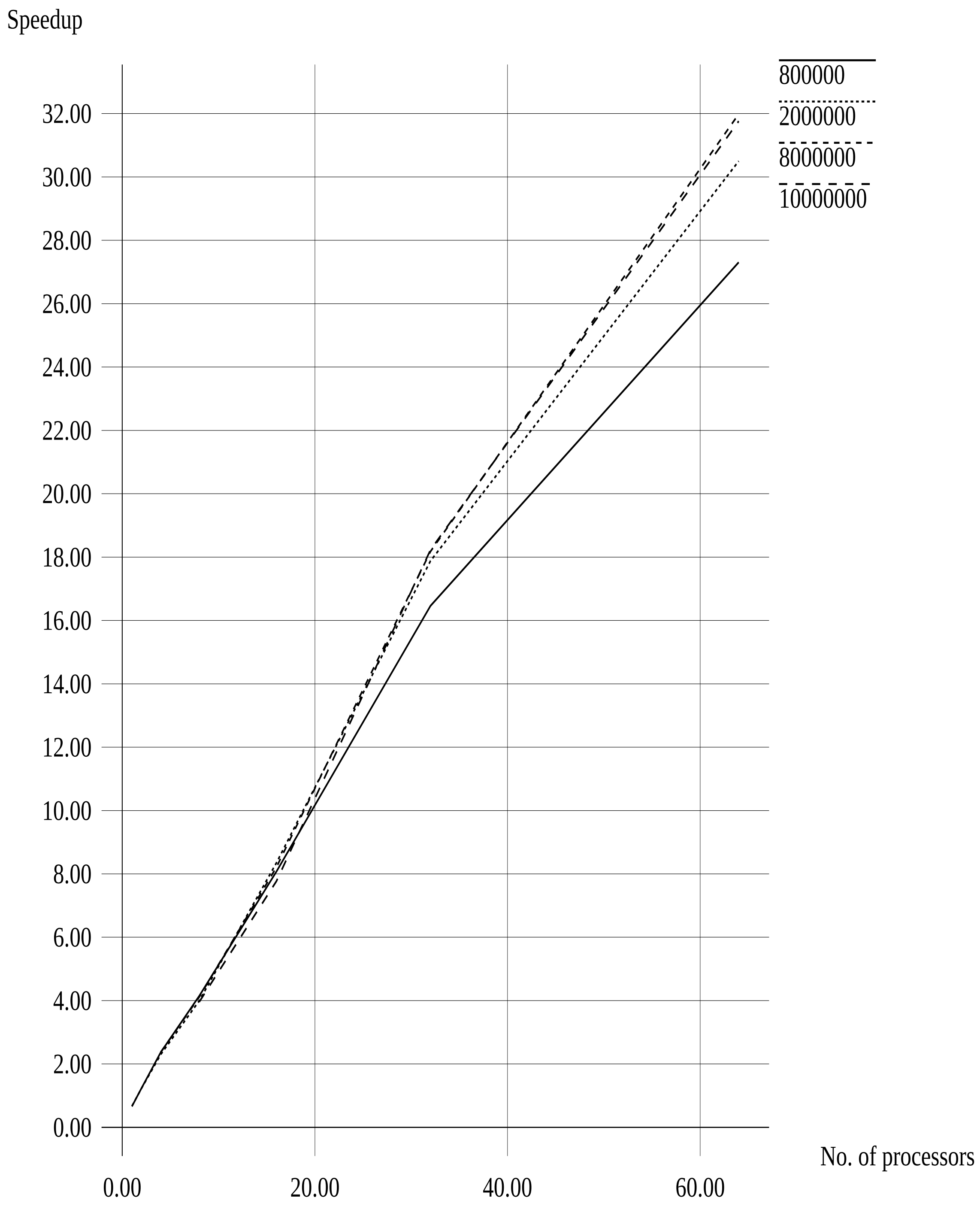


Figure 4.26: Speedups or primes program w.r.t. pSather/V3 on 1 node CM-5.

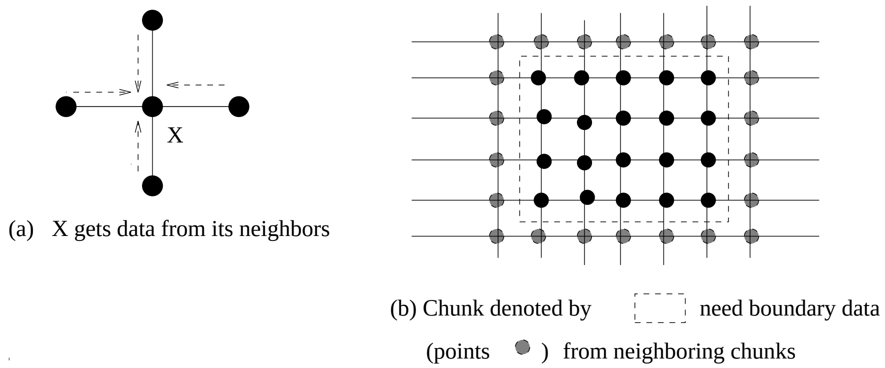


Figure 4.27: An example of when a chunk needs data from its neighbors.

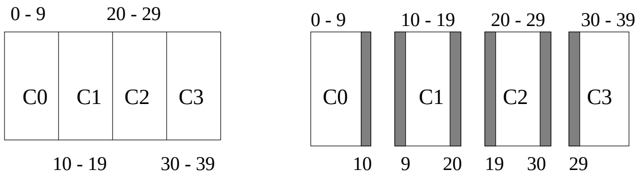


Figure 4.28: A possible configuration of a DIST_MATRIX_BLK_COL_OVERLAP{T} object representing a matrix partitioned into four chunks.

independently (without using data from other remote chunks). But not all matrix programs share this characteristic.

There are examples when a chunk operation needs to access data elements in other remote chunks. For example, in a direct O(N^2) N-body algorithm, each particle needs to know about the current position of all other particles. Another example is successive overrelaxation, in which a grid element uses its neighbors' data to update itself. In the second example, if a chunk consists of a set of neighboring grid elements (Figure 4.27), each chunk operation needs to access elements in its neighbors' boundary. To handle this, we implement a distributed matrix class in which the chunks overlap. The library writer implements an update protocol (for the boundary elements) that allows the user to determine when a chunk's boundary elements should be updated.

Figure 4.29 shows a DIST_MATRIX_BLK_COL_OVERLAP{T} class which inherits most of the code and interface from DIST_MATRIX_BLK_COL{T}. (We do not show the routines that are redefined because of the new partitioning.) Each chunk (sequential matrix) still represents a block of consecutive columns. The difference is that the left and right boundary columns of a chunk serve as caches for the chunk's left and right neighbor columns respectively.[14] Figure 4.28 shows the implementation of a matrix partitioned into four chunks. (The grey columns are the caches; and the numbers give

[14] The first (last) chunk has no left (right) neighbor column, so it caches only the right (left) neighbor column.

```
class DIST_MATRIX_BLK_COL_OVERLAP{T} is
  -- Code inheritance.
  DIST_MATRIX_BLK_COL{T};

  -- Redefinition of some routines (e.g. aget, aset) not shown.

  flush_updates is
    if (n_chunks = 1) then return end;
    with self near
      n_rows:INT := nrows;
      dist self as self_chunk do
        id:INT := self_chunk.chunk_id;
        if (id > 0) then
          receive_right_col(chunk(id-1),
            copy_ith_col(self_chunk, n_rows, 1), n_rows);
        end;
        if (id < n_chunks-1) then
          receive_left_col(chunk(id+1), copy_ith_col(self_chunk,
            n_rows, self_chunk.asize2-2), n_rows);
        end; end;
    else flush_updates @ self.where; end;
  end; -- flush_updates

  private copy_ith_col(sender:MATRIX_CHUNK{T}; nr, j:INT):ARRAY{T} is
    res := ARRAY{T}::new(nr); i:INT;
    until (i>=nr) loop res[i]:=sender[i,j]; i:=i+1; end;
  end; -- copy_ith_col

  private receive_right_col(recv:MATRIX_CHUNK{T};
            values:ARRAY{T}; nr:INT) is
    with recv near
      values := SYS::move_to(values, CONFIG::current_cluster_id);
      i:INT; j:INT := recv.asize2-1;
      until (i>=nr) loop recv[i,j]:=values[i]; i:=i+1; end;
      values.invalidate; values:=void;
    else receive_right_col(recv, values, nr) @ recv.where; end;
  end; -- receive_right_col

  private receive_left_col(recv:MATRIX_CHUNK{T};
            values:ARRAY{T}; nr:INT) is
    with recv near
      values := SYS::move_to(values, CONFIG::current_cluster_id);
      i:INT; j:INT := 0;
      until (i >= nr) loop recv[i,j]:=values[i]; i:=i+1; end;
      values.invalidate; values:=void;
    else receive_left_col(recv, values, nr) @ recv.where; end;
  end; -- receive_left_col
```

Figure 4.29: Library code for a distributed matrix with overlap (used in SOR).

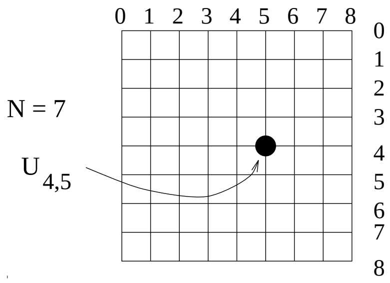


Figure 4.30: Discretized problem domain for solving Laplace's equation.

the column indices in the overall matrix.)

The class has a `flush_updates` routine (Figure 4.29) which updates the cached columns. It works as follows. If there is only one chunk, there is no need to update any cache. Otherwise, for each chunk C which has a left neighbor ("`id > 0`"), we pack C's second column into an array:

```
copy_ith_col(self_chunk, n_rows, 1)
```

C's second column is cached in C's left neighbor's (call it C_{left}) right boundary column. C_{left} get the updates via the `receive_right_col` routine. In this routine, we first check if C_{left} the chunk (`recv`) is near. If it is not, we shift the locus of execution to the chunk's cluster using the `@`-operator. The update array is moved to the local cluster and its values are stored into C_{left}'s rightmost column.

Similarly for each chunk C which has a right neighbor ("`id < n_chunks-1`"), we pack C's second last column into an array:

```
copy_ith_col(self_chunk, n_rows, self_chunk.asize2-2)
```

C's second last column is cached in C's right neighbor's (call it C_{right}) left boundary column. The `receive_left_col` routine is analogous to `receive_right_col` and updates C_{right}'s first (leftmost) column.

4.6 Application III — Successive Overrelaxation (SOR)

This section describes how a user program makes use of the distributed matrix class described earlier. This example also shows that pSather is useful not only for manipulating symbolic data structures, but also numerical computations. The problem is to compute Laplace's equation, which is a partial differential equation:

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0$$

on a set $(x, y) \in \Omega$ with given boundary condition $u(x, y)$ for (x, y) on the boundary of Ω. The domain Ω is discretized so that we consider only values at points (x_i, y_j), i.e. solutions for $u_{i,j}$. In the simple but important case when Ω is a unit square, we have $u_{i,j} = u(ih, jh)$ where $h = 1/(N+1), 0 \leq i, j \leq (N+1)$ and N is the level of discretization. The discretized problem can be cast into a system of N^2 linear equations $Ax = b$ where

$$x = \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ \vdots \\ u_{1,N} \\ u_{2,1} \\ \vdots \\ u_{2,N} \\ \vdots \\ u_{N,1} \\ \vdots \\ u_{N,N} \end{pmatrix} \quad A = \begin{pmatrix} T_N & -I_N & & \\ -I_N & T_N & -I_N & \\ & \ddots & \ddots & \\ & & -I_N & T_N \end{pmatrix} \quad T_N = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & -1 & \\ & \ddots & \ddots & \\ & & -1 & 4 \end{pmatrix}$$

We can in turn use various iterative algorithms [209] to find solutions for $u_{i,j}$.

Our program implements an iterative successive overrelaxation (SOR) algorithm for this problem. The general idea is that for a linear system $Ax = b$, during each iteration, the new value of x_i is given by:

$$x_i{}^{new} = x_i{}^{old} + \frac{\omega}{a_{i,i}}(b_i - \sum_{j<i} a_{i,j} x_j{}^{old} - - \sum_{j>i} a_{i,j} x_j{}^{new} - a_{i,i} x_i{}^{old})$$

where ω is a relaxation parameter. When applied to the system of equations with variables $u_{i,j}$, the update is given by:

$$u_{i,j}{}^{new} = (1-\omega) u_{i,j}{}^{curr} + \frac{\omega}{4}(u_{i-1,j}{}^{curr} + u_{i+1,j}{}^{curr} + u_{i,j-1}{}^{curr} + u_{i,j+1}{}^{curr})$$

The algorithm can be parallelized by dividing the variables into two sets, reds and blacks, so that $u_{i,j}$ is red (black) when $i - j$ is even (odd). In each iteration, the reds can be updated in parallel, followed by the blacks. The reds use the black values from previous iteration, and the blacks use the newly updated red values in the current iteration.

We implement this algorithm using the distributed matrix described in Section 4.5. Each chunk in the matrix contains consecutive columns of $u_{i,j}$'s. Figure 4.31 shows the part of the program when the red variables are updated during an iteration of a parallel SOR program. Figure 4.32 shows how the portions of each chunk are used, and helps to explain the program. The i-index increases

```
dist arr as arr_chunk do
   with arr_chunk near
      i:INT := 1;
      end_i:INT := arr_chunk.asize1-1;
      end_j:INT := arr_chunk.asize2-1;
      I:INT := arr_chunk.first_i+1;
      J0:INT := arr_chunk.first_j;
      if (arr_chunk.chunk_id = 0) then J0:=J0+1 end;
      until (i >= end_i) loop
         j:INT := 1;
         J:INT := J0;
         until (j >= end_j) loop
            if ((I-J).u_mod(2) = 0) then
                 arr_chunk[i,j] := (omega1 * arr_chunk[i,j])
                  + omega * (arr_chunk[i,j-1] + arr_chunk[i,j+1]
                  + arr_chunk[i-1,j] + arr_chunk[i+1,j]);
               end;
            j := j+1; J := J+1;
         end;
         i := i+1; I := I+1;
      end; end; end;
arr.flush_updates;
```

Figure 4.31: Updating red variables in different chunks during an iteration of a parallel SOR program.

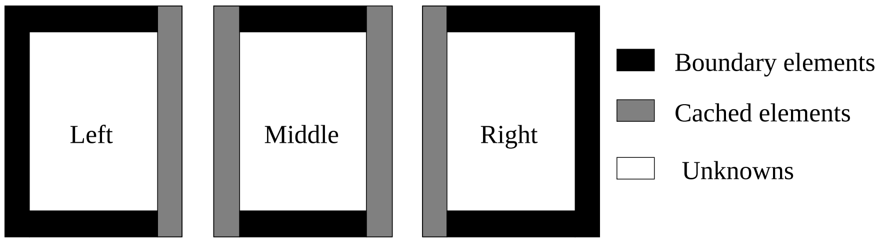


Figure 4.32: What each portion of a chunk is used for in an SOR program.

from top to bottom while the j-index increases from left to right. In a (generic) middle chunk, only the elements in the black and white portion are considered as part of the overall matrix; the grey portion (left and right) columns are only caches. Since the black portion consists of the boundary, updates are performed only on the white portion. In the left chunk, even though the left column is part of the overall matrix, it is now a boundary and therefore we again do not need to consider it in the computation. Similar reasoning applies to the right chunk. (In the extreme case, when there is only one chunk, we get a totally black boundary.)

In the following description, we distinguish between a chunk and the *used* (black and white) portion of a chunk.

In line 1, we use the `dist` statement to initiate parallel operations on the chunks. Since each body invocation executes on the chunk's cluster location, we can use the `with-near` statement to assert that the chunk is near with respect to its body invocation (line 2). We have seen that the white portions in all chunks turn out to have the same structure, i.e. starting with (1,1) (in a chunk) and ending with (*nrows*-2,*ncols*-2) (in the same chunk). This explains the initializations in lines 3–5. In order to find out if an element is red or black, we have to know its index within the overall matrix (not within the chunk). So lines 6–8 get the overall index of the (1,1) chunk element. Each chunk contains two attributes: `first_i`, `first_j`. These give the overall i- and j-indices of the top-left element in the *used* portion of the chunk. Therefore by adding 1:

```
I:INT := arr_chunk.first_i+1;
```

we get the overall i-index of top-left element in the *white used* portion. For the overall j-index, we only need to compensate only for the left chunk ("`arr_chunk.chunk_id = 0`"). In lines 9–21, we update the red unknowns. (The variables `omega` and `omega1` hold values $\omega/4$ and $1-\omega$ respectively.) At the end of the `dist` statement (line 22), we invoke `arr.flush_updates;` to update the caches in all the chunks. The code that computes the blacks is similar except that in line 12, instead of checking `I-J` is even, we check if `I-J` is odd.

It is interesting to note that the operations on each chunk (lines 2–20) are almost the same as when we implement this algorithm serially. This provides another example of the common paradigm that arises repeatedly: the distributed operation is a distributed application of the ordinary operation. In the SOR example, the main differences between the distributed and sequential versions are:

- The library writer has to distinguish between the indices of an element within a chunk and within the overall matrix.
- The user has to recognize that the first element of the *used* portion of a chunk starts at (0,0) for the first chunk and (0,1) for all other chunks.
- At the end of the updates, the routine `flush_updates` is called to make the caches consistent. This routine is provided in the library class and the user decides the frequency of the updates;

No. of processors	$n = 128$	$n = 256$	$n = 512$	$n = 768$	$n = 1024$
1	8.47	34.22	137.61	311.59	554.94
2	4.62	18.05	70.63	160.97	288.71
4	3.06	10.33	37.98	84.16	149.16
8	2.03	5.97	20.93	45.09	78.78
16	1.61	3.83	12.20	24.82	43.06
32	1.52	2.94	7.77	14.89	24.88

Table 4.8: Execution times (in seconds) of parallel SOR program (pSather) using various matrix sizes ($n \times n$) for 100 iterations.

Program	$n = 128$	$n = 256$	$n = 512$	$n = 768$	$n = 1024$
C (1-node CM-5, `gcc`)	6.62	27.04	114.69	263.49	473.68
C (1-node CM-5, `cc`)	3.44	13.92	56.04	126.39	224.88
pSather/V1	2.98	12.08	48.67	167.43	982.68[15]
pSather/V3	3.82	15.43	62.66	142.74	253.56

Table 4.9: Execution times (in seconds) of various sequential SOR programs using various matrix sizes ($n \times n$) for 100 iterations.

more frequent updates give faster convergence, but also requires more frequent communication.

4.6.1 Performance of Successive Overrelaxation

This section presents the performance results of the SOR program. Table 4.8 shows the timings for a 32-node CM-5 for[16] various sizes of $n = N + 2$. In each $n \times n$ matrix, the (i,j)th elements where $i, j = 0$ or $n - 1$ are the boundary elements. The pSather compiler invokes the `gcc` compiler version 2.4.5 with `-O` optimization.

To provide some bases for speedups comparison, we measure the timings of the following sequential programs:[17]

C / 1-node CM-5. We implemented the sequential algorithm in C and ran it on a single CM-5 node. The program is given in Figure C.1 (Appendix C). The arrays are dynamically allocated, just like those in pSather programs. The array elements are updated sequentially (in row-major form), instead of using a red-black access pattern. We compile this program using both the Sun native C compiler (`cc`) and `gcc` (version 2.4.5), both with `-O` optimization.

pSather/V1. We change the parallel pSather version to a sequential one by removing the `dist`

[15] One possible reason for these timings being worse than pSather/V3 is that the structure of 2-D array in pSather causes bad caching behavior.

[16] We only give timings up to 32 processors, because each node in our 32-node partition has more memory per node, than the 64-node partition.

[17] Unless other specified, the sequential pSather programs are compiled with the same options as the parallel program, and we use `gcc` version 2.4.5 with `-O` optimization.

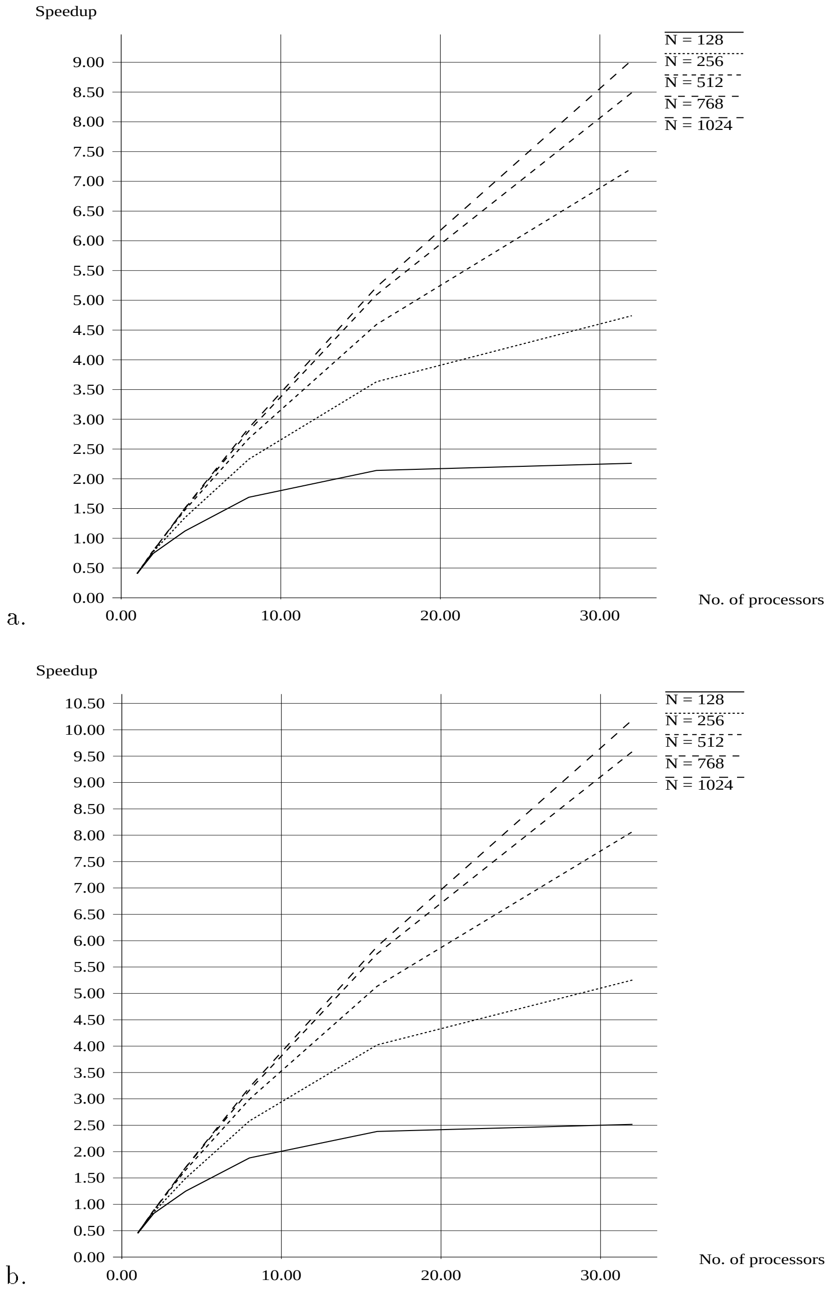


Figure 4.33: Speedups of SOR program w.r.t. (a) C on 1-node CM-5 (compiled with `cc -O`) and (b) pSather/V1, using 100 iterations in all cases.

No. of processors	$n = 128$	$n = 256$	$n = 512$	$n = 768$	$n = 1024$
1	3.85	15.52	63.34	142.99	254.08
2	2.25	8.55	32.95	72.85	128.55
4	1.70	5.39	18.52	39.35	67.87
8	1.31	3.36	10.89	22.33	37.79
16	1.14	2.37	6.81	13.05	21.41
32	1.20	2.10	6.95	8.53	13.21

Table 4.10: Better execution times (in seconds) of parallel SOR program (pSather) using various matrix sizes ($n \times n$) for 100 iterations, after polling is selectively removed.

statement and using an `ARRAY2{DOUBLE}` for the matrix (instead of a distributed matrix). The code is given in Figure C.2, Appendix C. Like the C program, the array elements are updated sequentially in row-major form, instead of following a red-black update pattern. The compiler is given the `-nopoll` option and does not generate any polling points.

pSather/V3. We compile the parallel program with the `-nopoll` option so that the compiler does not generate any polling points.

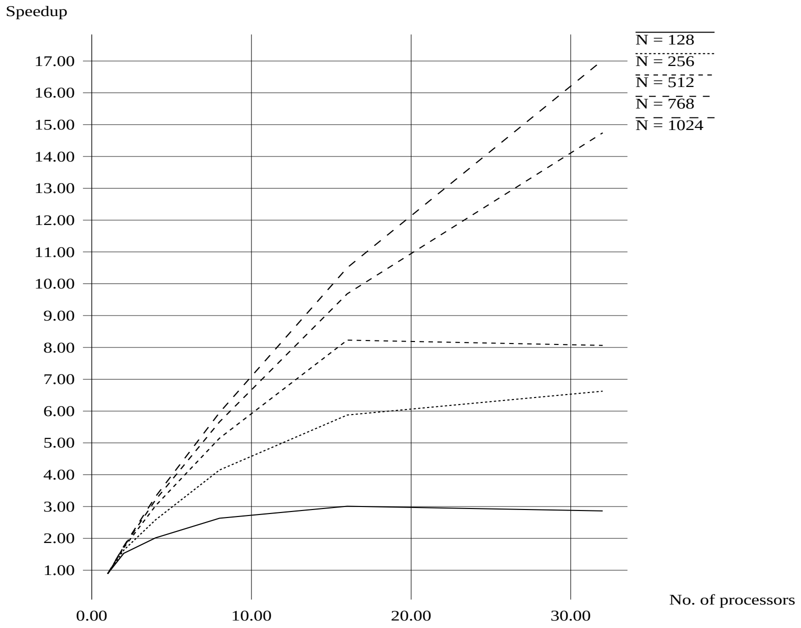
The sequential timings are shown in Table 4.9. The `cc` compiler performs better optimization than `gcc` with `-O`, so when we compute the speedups of the parallel SOR program with respect to C (Figure 4.33 (a)), we use the timings given by the second row. We find that the pSather/V3 red/black access pattern scales better than the sequential patterns given by pSather/V3. Thus when we compute the speedups of the parallel SOR program with respect to a sequential pSather program, we use pSather/V3 (Figure 4.33 (b)). The differences between pSather/V3 timings and the 1-node timings in Table 4.8 reflect the overheads due to polling.

To see how the parallel performance can be improved if polling is removed, we instruct the compiler not to insert any polling points during the update of array elements.[18] The resulting timings are shown in Table 4.10. The speedups are computed with respect to the same sequential programs as earlier and are shown in Figure 4.34. Like earlier performance measurements, the speedup improvements show the significant overheads imposed by polling, and the importance of using an interrupt-based message library in the pSather runtime.

The benefits of a multiprocessor are not only in improved execution times, but also in being able to execute problems of larger input sizes. So we execute the parallel SOR for various values of $n > 1024$ using 32 CM-5 nodes, for 100 iterations (Table 4.11). Most of these problem sizes will not fit on a single CM-5 node. To get an idea of how the speedup improves (i.e. program scalability), we compute the factor increase in problem size (n^2) and timings with respect to $n = 1536$. The timings increase slower than the problem size, indicating that the speedup curves continue to improve with larger n. Since the amount of computation is proportional to n^2 and the amount of communication

[18] This is done using the `NO_POLL` pragma (Section 3.3.5).

a.



b.

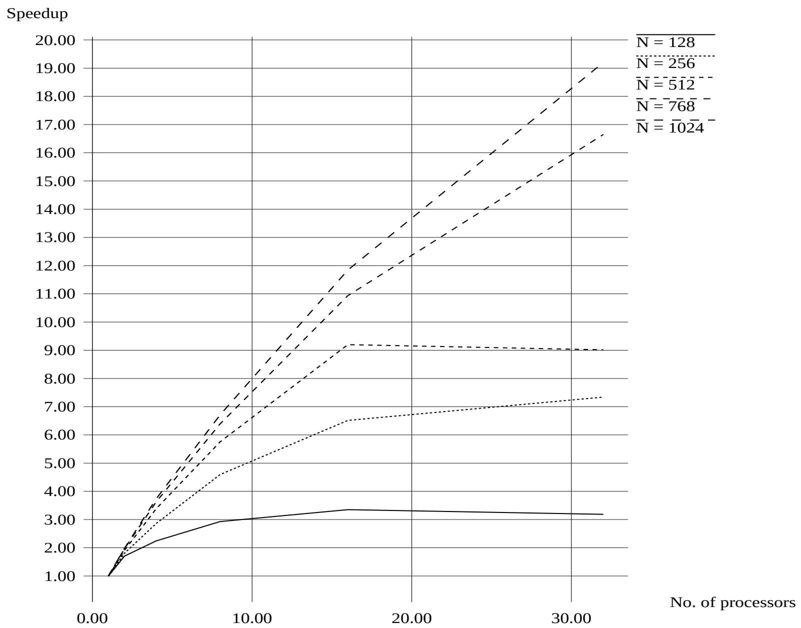


Figure 4.34: Improved speedups of SOR program (after polling points are selectively removed) w.r.t. (a) C on 1-node CM-5 (compiled with `cc -O`) and (b) pSather/V1, using 100 iterations in all cases.

	$n = 1536$	$n = 2048$	$n = 3072$	$n = 4096$	$n = 5120$	$n = 6144$
Parallel SOR	25.88	43.34	89.13	151.69	230.80	326.15
"Normalized" n^2	1.00	1.78	4.00	7.11	11.11	16.00
"Normalized" time	1.00	1.67	3.44	5.86	8.92	12.60

Table 4.11: Execution times (in seconds) of parallel SOR program (pSather) for problem sizes > 1024 using 32 processors, for 100 iterations. (Polling has been selectively removed.)

is proportional to n, as n increases, the communication overheads become a smaller proportion of total time and the processors can execute better in parallel.

4.7 Application IV – Gröbner basis

This section looks at a medium-sized Gröbner basis program that uses both the workbag and replicated hash table. Given a field K and set of variables $\{x_1, \ldots, x_n\}$, we denote $R = K[x_1, \ldots, x_n]$ as the ring of polynomials in n variables over K. Now suppose we have a set of polynomials $B \subset R$. The *ideal* of B (ideal_B) is the closure of B under standard polynomial addition and multiplication by elements in R.

Let ">" be a total ordering defined on the monomials in R. For example in lexicographic ordering, ${x_1}^{\alpha_1}{x_2}^{\alpha_2}\ldots{x_n}^{\alpha_n} > {x_1}^{\beta_1}{x_2}^{\beta_2}\ldots{x_n}^{\beta_n}$ if there is a k, $1 \leq k \leq n$ such that $\alpha_k > \beta_k$ and $\forall i < k$, $\alpha_i = \beta_i$. So any polynomial $\in R$ can be put in a canonical representation such that its monomial terms are in decreasing order with respect to the specified ">" ordering. For two polynomials, p, r, if the head term of r divides the head term of p, r can reduce p to p', by finding a monomial γ such that head term of $r\gamma$ = head term of p, and $p' = p - r\gamma$.

A Gröbner basis of a set of polynomials B is a set $G \subset R$ such that $\text{ideal}_B = \text{ideal}_G$ and every $p \in \text{ideal}_B$ is reduced by polynomials in G to 0. A reduced Gröbner basis G is a Gröbner basis such that every $p \in G$ is not reducible by any other polynomial in G.

Buchberger [49] gives an algorithm which computes a reduced Gröbner basis. The computation has two phases: the first produces a Gröbner basis G and the second produces the reduced basis from G. There have been various attempts at parallelizing Buchberger's algorithm [194, 61, 225], but our aim is slightly different. We want to produce a reasonably efficient parallel program using existing software components. Since our focus is not on algorithm design, we adopt the same approaches as others, in parallelizing only the first phase which produces the Gröbner basis.

4.7.1 Outline of Algorithm

To understand Buchberger's algorithm, we have to define what an *S-polynomial* is. Let hterm_p be the leading monomial of polynomial p. The S-polynomial of two polynomials p and q is

```
(1)
  G ← B
  PP ←  (p, q) | p, q ∈ B, p ≠ q
  while (PP is not empty)
     (x, y) ← a pair from PP
     PP ← PP -  (x, y)
     s ← S-poly(x, y)
     s' ← Reduce(s, G)
     if (s' /= 0) then
        PP ← PP +  (p, s') | p ∈ G
        G <- G +  s'
     endif
  endwhile

(2)
  G' ← ∅
  while (G /= 0)
     p ← a polynomial from G
     G ← G -  p
     p' ← Reduce(p, G')
     if (p' /= 0) then
        G' ← G' +  p'
     endif
  endwhile
```

Figure 4.35: Pseudocode for phases of Buchberger's algorithm: (1) Production of S-polynomials and (2) Final inter-reduction phase.

given by $ap - bq$ where a and b are chosen so that $\text{hterm}_{ap} = \text{hterm}_{bq} = h$ and h is the least common multiple of the hterm_p and hterm_q (denoted as $\text{lcm}(\text{hterm}_p, \text{hterm}_q)$).

Figure 4.35 shows the two phases of Buchberger's algorithm. The input is a set of polynomials B. The Gröbner basis G is initially set to B (line 1). Then we form the set of pairs of polynomials, PP (line 2). We repeatedly pick out a pair (lines 4 – 5), produce an S-polynomial using the pair (line 6) and reduce the S-polynomial using the current basis (line 7). If the result s' is not zero, then we form new pairs for s' and $p \in G$ (line 9), and add s' into the current basis (line 10).

At the beginning of the second phase, the reduced basis G' is empty. We reduce each polynomial in G (lines 15 – 16) with respect to the reduced basis that has been built up so far (line 17). If the reduction result is non-zero, it is added to the reduced basis (line 19).

There are two improvements to phase 1 which are essential to the efficiency of the computation. The idea is that not all pairs need to be added into the set PP (line 9). The algebraic arguments are summarized in [225] (which gives further references). We describe the algorithm refinements.

In the first refinement,[19] for any pair (p, q_1), if there exists a polynomial q_2 such that $p, q_1, q_2 \in G$ and one of the conditions hold:

- $\text{lcm}(\text{hterm}_p, \text{hterm}_{q_1}, \text{hterm}_{q_2}) = \text{lcm}(\text{hterm}_p, \text{hterm}_{q_1})$
- $\text{lcm}(\text{hterm}_p, \text{hterm}_{q_1}, \text{hterm}_{q_2}) = \text{lcm}(\text{hterm}_p, \text{hterm}_{q_2})$
- $\text{lcm}(\text{hterm}_p, \text{hterm}_{q_1}, \text{hterm}_{q_2}) = \text{lcm}(\text{hterm}_{q_1}, \text{hterm}_{q_2})$

then the S-polynomial of p and q_1 can be expressed in terms of the members in G. This means that the S-polynomial will be reduced to 0 and (p, q_1) can be eliminated. So when we form the pairs (line 9), if we find (p, q_1), (p, q_2) such that $\text{lcm}(\text{hterm}_p, \text{hterm}_{q_1}) = \text{lcm}(\text{hterm}_p, \text{hterm}_{q_2})$,[20] then we need to add only one of the pairs to PP.

If we find (p, q_1), (p, q_2) such that $\text{lcm}(\text{hterm}_p, \text{hterm}_{q_2})$ divides $\text{lcm}(\text{hterm}_p, \text{hterm}_{q_1})$, then (p, q_1) can be eliminated and only (p, q_2) needs to be added to PP.

The second refinement is called Buchberger's criterion 2 [49]. For any pair (p, q), if $p, q \in G$ and $\text{lcm}(\text{hterm}_p, \text{hterm}_q) = \text{hterm}_p \times \text{hterm}_q$, then the S-polynomial of p and q can be expressed in terms of the members of G. So (p, q) need not be added to PP.

[225] demonstrates how these two refinements greatly reduce the number of S-polynomials. We therefore use these two refinements in our parallel implementation which is described next.

---

[19] This is referred to as Buchberger's criterion 1 [104, 225].

[20] This implies that $\text{lcm}(\text{hterm}_p, \text{hterm}_{q_1}, \text{hterm}_{q_2}) = \text{lcm}(\text{hterm}_p, \text{hterm}_{q_1})$.

4.7.2 Parallel Implementation in pSather

Our implementation of Buchberger algorithm computes and reduces the S-polynomials in parallel (i.e. phase 1 of Figure 4.35).

The first abstraction that comes in useful is the workbag (Section 4.1), which is used to implement the set of pairs PP. We fork off one worker thread per processor, which repeatedly gets a pair from the bag, computes an S-polynomial s from the pair and reduces s locally.

The parallelization uses a key idea in Buchberger's sequential algorithm which is that we can time-stamp each S-polynomial that is produced. An S-polynomial is reduced with respect to polynomials with earlier time-stamps. The result may or may not be a zero polynomial. If the result is non-zero, it is added to the basis for reducing S-polynomials with later time-stamps.

We need to use the time-stamps to maintain correctness. This prevents the following incorrect situation from following: two processors produce S-polynomials p, q in parallel such that p reduces q to 0 and vice-versa, and none gets added to the basis.

The set of basis polynomials G grows monotonically during program execution, so we find that G is naturally represented by the replicated hash table abstraction (Section 4.3. We use the polynomials' integer time-stamp as the key. To make it easy to identify the keys, we use consecutive integers for the time-stamps, starting from 1. Since data locality is crucial for good performance, during an S-polynomial's reduction, we want the basis polynomials to be cached locally. This caching facility is transparently provided by the replicated hash table.

Before looking at the pSather implementation, we first explain some preliminaries.

- The unique time-stamps are generated by a global `GATE{INT}` object (in a shared variable `global_mpol_counter`).

```
get_unique_mpol_id:INT is
  -- Return a globally unique id for (multivariate) polynomial.
  res := global_mpol_counter.take;
  global_mpol_counter.set(res+1);
end;
```

- We implement the `MPOL` (multivariate polynomial) and `EXPO` (exponent) classes directly on top of C packages which we extracted from a shared-memory Gröbner basis program developed at Carnegie-Mellon University.[21] An example routine in `EXPO` looks like:

```
expoequal(exp0:SELF_TYPE):BOOL is
  res := C::expoequal(self, exp0);
end; -- expoequal
```

[21] I would like to acknowledge Steve Schwab who provided various packages and a shared-memory Gröbner basis program developed at CMU. PSather's ability to interface with the C language has made it possible for our program to use the existing packages for polynomial arithmetic and infinite precision integers.

```
workbag := w.dir_copy;
loop
   pair:PAIR_INT := workbag.get;
   if (pair /= void) then
      produce_spoly(pair);
   else
      workbag.signal_quiescent;
      if (workbag.num_quiescent_threads = num_workers)
      then break end;
   end;
end;
```

Figure 4.36: Worker thread for Buchberger's algorithm.

- The Gröbner basis G is implemented by `REPL_INT_HASH_MAP{MPOL}`. So `MPOL` needs to define a `make_copy_to` routine. To reduce the amount of copying, the `MPOL` class allocates a special zero polynomial (in shared feature `zero_mpol`) per cluster. Every cluster keeps an array `zero_mpol_array` such that `zero_mpol_array[i]` points to the special zero polynomial on the ith cluster. Whenever we need to copy a polynomial to cluster i, we first check if it is a zero polynomial and if so, we return the zero polynomial on i, instead of making a new copy.

```
make_copy_to(cid:INT):SELF_TYPE is
  -- Make a new copy of the MPOL object.
  if (zero.is_zero) then
    res := zero_mpol_array[cid]; return;
  end;
  if (cid = CONFIG::current_cluster_id) then
    res := C::local_copy_mpol(self);
  else
    res := make_copy_to(cid) @ cid;
  end;
end; -- make_copy_to
```

Note that the copying is done by a C routine[22] since `MPOL` are implemented as C objects.

Figure 4.36 shows the top-level code which is executed by the worker threads. It has a similar structure to the codes for N-queens (Figure 4.14) and primes sieve (Figure 4.23). A worker obtains a pair of polynomial id's from the bag (which, transparent to the user, may be from its local sub-bag or some other remote sub-bags).

Figure 4.37 shows the more detailed code which produces the S-polynomials. Lines 1–3 shows how we use the `get_wait` routine from `REPL_INT_HASH_MAP{T}` to transparently retrieve cached polynomials. If the S-polynomial is zero, then we simply return (line 4). Otherwise, we reduces the S-polynomial.

[22] This routine was modified from one implemented by Soumen Chakrabarti.

```
mpol1:MPOL := global_mpol_set.get_wait(pair.first);
mpol2:MPOL := global_mpol_set.get_wait(pair.second);
spoly:MPOL := MPOL_PAIRSET_OP::spoly_from(mpol1, mpol2);
if (spoly.is_zero) then return end;

-- Perform reduction with respect to the local cached set of
-- polynomials.
spoly := reduce(spoly, first_mpol_id, last_cached_id);

if not (spoly.is_zero) then
   -- When S-poly is non-zero, reduce again.
   next_pos:INT := get_unique_mpol_id;
   spoly := reduce(spoly, last_cached_id+1, next_pos-1);
   last_cached_id := next_pos;

   if not (spoly.is_zero) then
      global_mpol_set.insert(next_pos, spoly);
      MPOL_PAIRSET_OP::form_pairs(spoly, next_pos, workbag);
   else
      global_mpol_set.insert(next_pos, MPOL::zero_mpol);
   end;
end;
```

Figure 4.37: S-polynomial production and reduction.

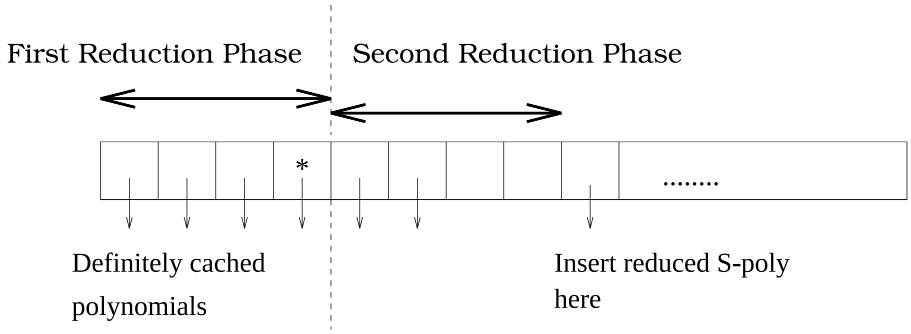


Figure 4.38: Relaxed reduction by already-computed polynomials.

Our implementation has two separate reduction phases because of the following reasons.

- We note that in our algorithm, once we have obtained a time-stamp t for an S-polynomial s, after s is reduced to s', s' must be inserted into the basis set even if $s' = 0$. The reason is that other workers will be trying to retrieve polynomials with keys $1, \ldots, t$. A deadlock results if s' is not inserted. So we want to reduce s as much as possible before getting a time-stamp. If s become 0 "early enough", we might avoid inserting a polynomial.
- The time-stamps are generated by a global gate, so the process of obtaining a time-stamp might potentially be a bottleneck, and we want to reduce the number of time-stamps.

In the first phase (line 7), an S-polynomial is first reduced by all polynomials that have already been cached. Each worker keeps a counter `last_cached_id` such that if `last_cached_id` $= i$, then we are guaranteed that polynomials $1, \ldots, i$ have been cached. This variable is initialized to $|B|$ (where B is the input set of polynomials). This is correct because if we were to get a time-stamp t, t must be greater than i, we have to reduce the S-polynomial against all others whose time-stamp is $\leq i < t$ anyway.

We obtain a time-stamp for the S-polynomial s only if it remains non-zero after the first phase (line 10). During the second phase (line 11), s is then further reduced by polynomials p with earlier time-stamps (all polynomials beyond (*) but before s in Figure 4.38). In the second phase, the replicated hash table automatically makes sure that p is ready before reducing s by p.

After that we know that all polynomials time-stamped $1, \ldots, t$ have been cached (line 12). We insert the reduced S-polynomial (lines 14, 17) and form pairs if it is non-zero (line 15).[23] We use the refinements in Section 4.7.1 to reduce the number of pairs added to the workbag.

4.7.3 Performance

This section presents the performance of our Gröbner basis program. The program uses a workbag with breadth-first strategy. (This performs better than workbags with other strategies because the "better" pairs are inserted earlier.[24])

Because this is a dynamic problem, we took 5 measurements for each set of parameters for the parallel program. For sequential timings, we compile the parallel program with `-nopoll` option so that the compiler does not generate any polling points. This is the version that has been called pSather/V3 throughout.

The main difficulty in measurements is that the program runs to completion too quickly. Table 4.12 shows the pSather/V3 timings for the typical inputs [225]. Even when we tried out the large problems used by [61] (which is generated by replicating an input with renamed variables), the

[23] The class `MPOL_PAIRSET_OP` (used in lines 3, 15) encapsulates the operations that produces S-polynomial and pairs.

[24] A pair (p_1, q_1) is better than (p_2, q_2) if $\mathrm{lcm}(\mathrm{hterm}_{p_1}, \mathrm{hterm}_{q_1}) < \mathrm{lcm}(\mathrm{hterm}_{p_2}, \mathrm{hterm}_{q_2})$.

Name	Ordering	Time
Arnborg4	L	0.05
Arnborg5	TR	5.73
Katsura4	TL	9.56
Lazard	TR	2.02
Morgenstern	TR	18.96
Pavelle4	TR	0.90
Robbiano	TL	0.49
Robbiano	L	7.44
Rose	TR	2.49
Trinks1	TL	2.05
Trinks2	L	0.87

Table 4.12: Sequential computation time (in seconds) of Gröbner basis program (pSather/V3) on typical inputs. The orderings are: L = lexicographic, R = reverse lexicograhic, TL = total refined by lexicographic, TR = total refined by reverse lexicographic.

Input	`Lazard` × `5`	`Rose` × `2`
Ordering	TR	TR
pSather/V3	22.60	5.40
P = 1	27.85 – 28.14	6.13 – 6.22
P = 2	15.24 – 15.83	3.76 – 3.88
P = 3	11.69 – 12.42	2.67 – 2.86
P = 4	7.63 – 10.30	3.01 – 4.42
P = 5	6.81 – 7.26	2.12 – 3.18
P = 6	5.66 – 7.62	1.81 – 2.35
P = 8	5.88 – 8.02	1.71 – 3.17
P = 10	5.85 – 8.21	1.92 – 4.38

Table 4.13: Computation time (in seconds) of Gröbner basis program on replicated inputs. P = Number of processors. `<Name>` × `<n>` = `<n>` copies of `<Name>`, with renamed variables.

Input	$\lvert G\rvert$	$\lvert S\rvert$	Total No. of reductions
`Lazard × 5`	140 – 148	280 – 301	4030 – 4956
`Rose × 2`	42 – 54	92 – 137	1958 – 3723

Table 4.14: General statistics on the amount of work in the parallel Gröbner basis program. The range is given for executions using 1 – 10 processors. $|S|$ = Number of S-polynomials produced (whether reducible to 0 or not).

1-processor timings remain small[25] (Table 4.13). If we consider the minimum times, it is encouraging that we still get some amounts of speedups for `Rose × 2`.

There are several reasons why the speedups are limited. First we note that the polynomial replication and production/reduction of S-polynomials are the main costs. As a first approximation, the amount of replication is proportional to $P\,|G|$, where G is the final (non-reduced) basis. If the refinements in Section 4.7.1 are not used, the S-polynomial production cost is $O(|G|^2)$. But when the refinements are incoporated, the S-polynomial production cost appears to be more of the order $|G|$. From our measurements, the S-polynomial reduction cost is also less than $O(|G|^2)$. These statistics are measured in Table 4.14. We observe that the statistics for the sequential program (pSather/V3) are always at the minimum of the range. Because of this, plus the fact that from earlier measurements (e.g. N-queens, primes, SOR), the speedup graphs based on pSather/V3 have always been similar to those calculated relative to C, we think that it is justifiable to give our speedups based on pSather/V3.

We measure timings for even larger problems (Table 4.15). Figure 4.39 shows the best and worst speedup curves. Even though the curves are not totally smooth due to the dynamic nature of the algorithm, we can discern a constant trend of increasing speedups as the number of processors are increased. When the problem is large enough (`Robbiano` $\times 10$), we manage to achieve a speedup of 15 on 26 processors in the best situation. (In other cases, the improvements taper off at about 20 processors.)

The absolute performance of the program, and the fact that the program was implemented using existing abstractions in a relatively short time are encouraging evidence of pSather's suitability for implementing parallel applications.

4.8 Application V – Fast Multipole N-body

We describe the parallelization of an N-body algorithm which computes potential fields for each of N particles in a region. The field of a particle is the result of interaction between it and each of the other $N-1$ particles. A brute-force computation takes time $O(N^2)$. The Greengard-Rohklin's

[25] The reason that our 1-processor times do not take hours (unlike [61]) is that the refinements (described in Section 4.7.1) reduce the amount of work generated; the difference may also be due to the different orderings used.

Input	`Lazard` × `15`	`Robbiano` × `10`
Ordering	TR	TR
pSather/V3	422.47	170.27
P = 1	505.86 – 507.36	202.05 – 203.05
P = 2	286.06 – 302.46	97.48 – 132.44
P = 3	215.21 – 234.05	64.82 – 79.03
P = 4	165.79 – 185.29	56.85 – 77.68
P = 5	144.72 – 154.96	48.09 – 67.53
P = 8	88.45 – 101.44	39.90 – 99.76
P = 10	68.26 – 77.83	26.14 – 40.02
P = 12	58.02 – 63.12	28.96 – 34.87
P = 14	51.37 – 55.10	19.36 – 26.96
P = 16	45.72 – 58.78	16.25 – 21.57
P = 18	44.63 – 52.24	16.52 – 20.53
P = 20	39.29 – 49.24	14.77 – 20.84
P = 22	41.54 – 49.50	13.43 – 17.54
P = 24	43.21 – 52.43	13.02 – 17.63
P = 26	41.60 – 47.18	11.26 – 16.83
P = 28	45.10 – 49.07	13.52 – 18.56
P = 30	47.89 – 53.08	12.89 – 15.83
P = 32	39.32 – 49.90	12.37 – 16.31

Table 4.15: Computation time (in seconds) of Gröbner basis program on replicated inputs. P = Number of processors. `<Name>` × `<n>` = `<n>` copies of `<Name>`, with renamed variables.

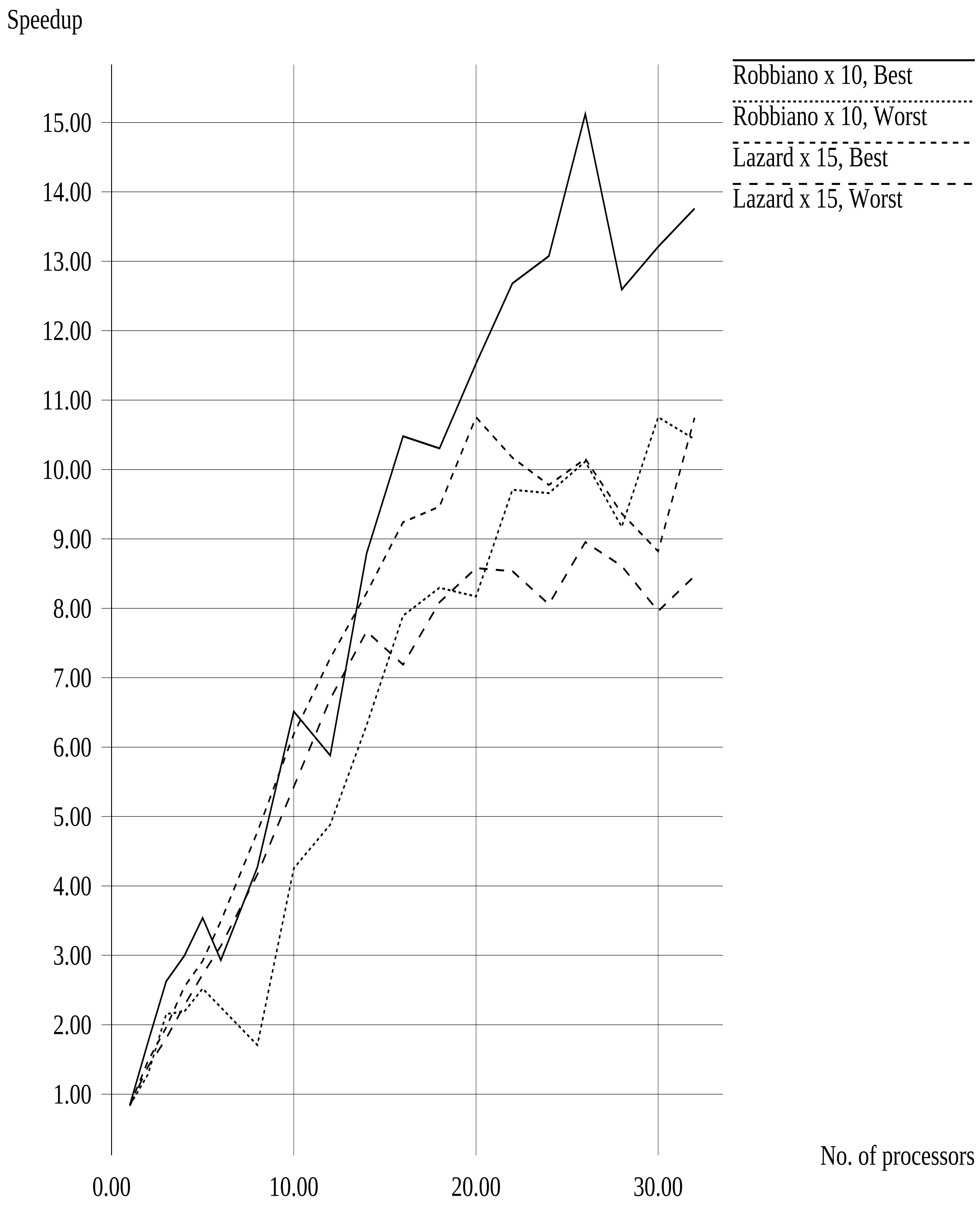


Figure 4.39: Speedups of parallel Gröbner basis program w.r.t. pSather/V3.

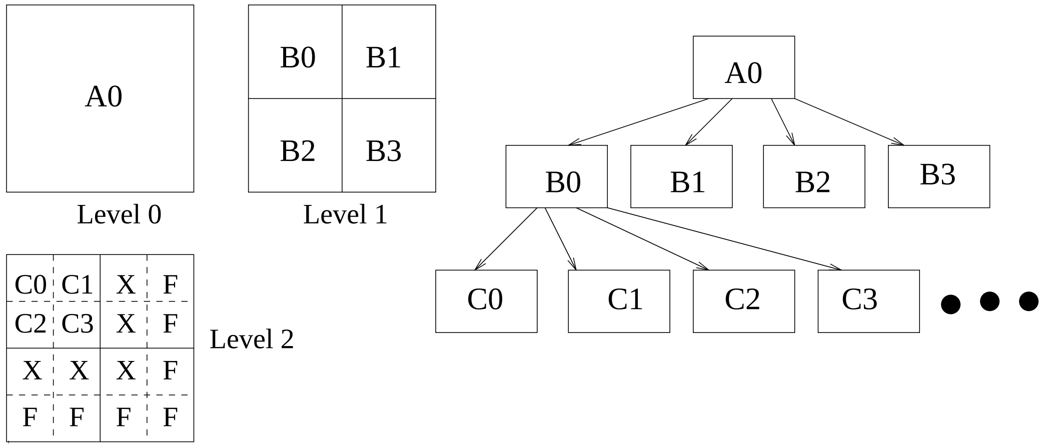


Figure 4.40: A region and two levels of refinement, and a corresponding quadtree representation.

algorithm [114] takes time $O(N)$.

We implemented two versions of the algorithm, for when the particles are evenly or unevenly distributed throughout the region. A *non-adaptive* algorithm is used when the particles are evenly distributed and an *adaptive* algorithm when they are unevenly distributed. Section 4.8.1 describes the general outline of the algorithm, but does not go into the detailed mathematics described in [114]. Section 4.8.2 describes our parallel implementation and data structures used. Section 4.8.3 gives our timing measurements.

4.8.1 Outline of Greengard-Rohklin's N-Body Algorithm

The crucial idea of the algorithm is that the potential on a particle p due to other $N-1$ particles can be divided into two parts – the near and far fields. The near field is computed by pairwise evaluation of particles which are near to one another. The value of far field is similar for all particles sufficiently close to p, and is represented by a complex polynomial. Once the far field polynomial is computed, the far field value on a particle can be computed by evaluating the polynomial at the particle position. This computation takes linear time in the order of the number of terms in the polynomial, and does not depend on the number of particles which are in the far field.

The main data structure is a quadtree in which each node represents a region.[26] A parent node has four children nodes, representing its four equal square sub-regions. The leaf nodes represent the finest division of the problem region. Each leaf node (sub-region) contains particles whose far fields are identical and represented by the same complex polynomial. When particles are evenly distributed, the height of the quadtree is determined by $\log(N/n)/\log(4)$ which approximately allo-

[26] We will use the terms "node" and "region" interchangeably.

cates n particles in each of the finest regions. Figure 4.40 shows an initial region and two levels of refinement. C0, C1, C2 and the X regions are all *neighbors* of C3 and are said to be *adjacent* to C3. Two regions, each of side s, are *well-separated* if they are separated by a distance s. For example, all the F regions are well-separated from C3.

An essential part of the algorithm is to compute the far field polynomial (called *local expansion*) for each of the leaf regions. There is an upward and a downward pass over the quadtree to calculate the local expansions.

The computation of particle potential fields by a non-adaptive fast multipole algorithm can be summarized as follows.

1. Execute the upward and downward passes over the quadtree; the goal is to get the local expansions of leaf regions.

2. For each particle in a region, its far field is calculated by evaluating the local expansion at its position.

3. Compute near field of a particle by pairwise computations with particles in *neighboring* regions.

4. The potential field of a particle is sum of its near and far fields.

We first describe the computation of local expansion for uniformly distributed particles.

During the upward pass, each leaf region computes the potential due to particles contained in itself. The potential is represented by a complex polynomial called *multipole expansion*. A multipole expansion of a (sub-)region is computed relative to the center of that region. Therefore, in order for a parent region to compute the potential of its particles, the multipole expansions of its children have to be first translated to the parent's center. The sum of these translated expansions gives the parent region's multipole expansion. Let center_{R} be the center of region R; in Figure 4.40, we get:

$$\begin{aligned}\text{multipole(A0)} = {} & \text{translate(multipole(B0),center}_{\text{A0}}) + \\ & \text{translate(multipole(B1),center}_{\text{A0}}) + \\ & \text{translate(multipole(B2),center}_{\text{A0}}) + \\ & \text{translate(multipole(B3),center}_{\text{A0}})\end{aligned}$$

We use Figure 4.41 to outline how the local expansion is computed in the downward pass. We first note that if we consider a region R and its parent, there are regions which are part of R's far field, but not part of R's parent's far field. These regions (marked F) are said to be in R's *interaction list*. More specifically, R's interaction list consists of R's parent's neighbors' children which are well-separated from R.[27]

Therefore, the far field of a region R has two components – a field *F1* (from region VF in Figure 4.41) and a field *F2* from regions in the interaction list. The *F1* component of region R

[27] We note that the regions in the interaction list are of the same size as R so that they are well-separated from R. Even if a region R1's children are all in the interaction list, we cannot simply replace the four children by R1 because R1 is not well-separated from R.

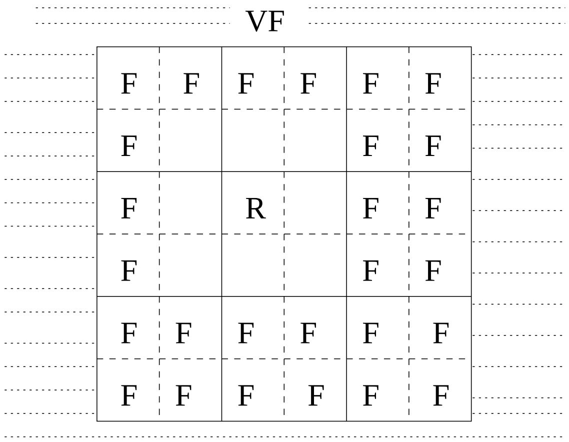


Figure 4.41: Computation of far field of region R

is exactly the far field of R's parent and can be computed by translating R's parent's far field to R's center. The *F2* component is computed by transforming each of the interaction list region's multipole expansion (with respect to R's center), and summing them. If R has any children, its far field in turn becomes the *F1* field of its children. If we refer to Figure 4.40 again, we get:

$$\text{local\_expansion(C3)} = \text{translate(local\_expansion(B0),center}_{C3}) + \sum_{F} \text{transform(multipole(F),center}_{C3})$$

The non-adaptive case is simple because all the leaf regions have the same size and there are only three possible relationships between a region R and any other region S which are at the same depth in the tree.

- S is R's neighbor, in which case we need to use S's particles when computing the near field of R's particles.
- S is in R's interaction list. We use S's multipole expansion to compute R's local expansion.
- S is neither of the above. S's contribution to R's far field is obtained from the local expansion of R's parent.

The situation becomes more complex in the adaptive case. At each node of the quadtree, the region is further divided into four sub-regions only when it contains more than s particles (where $s > 0$ is fixed). This avoids creating unnecessary empty regions. The unequal refinement results in leaf nodes of unequal sizes.

Each region R maintains four lists of regions, simply called u-, v-, w- and x-lists. The u-, v-, w- and x-lists of a region R are denoted by U_{R}, V_{R}, W_{R} and X_{R} respectively. We use Figure 4.42 (from [114]) to explain their meaning.

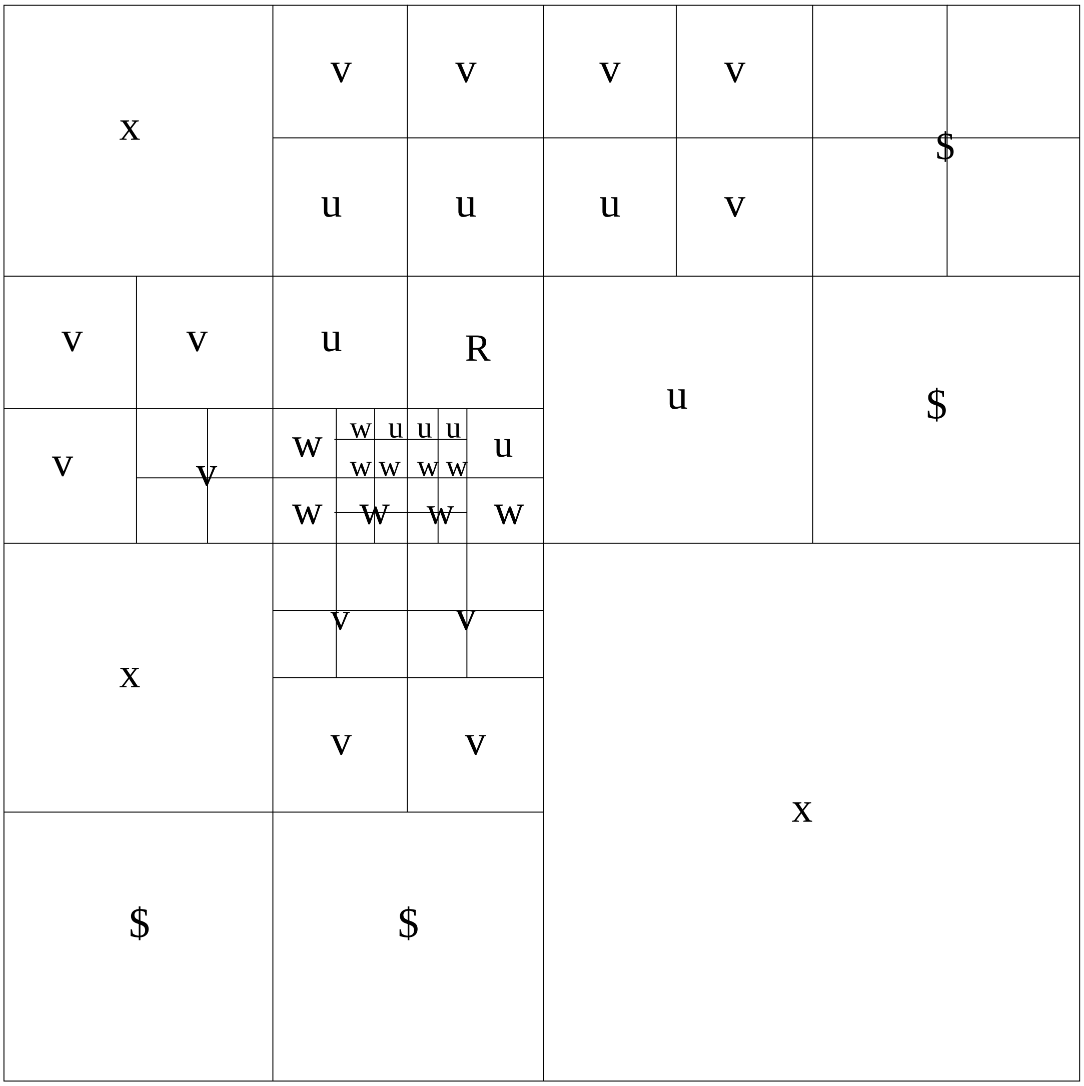


Figure 4.42: Region R and its associated lists.

- If R is a parent (i.e. internal) node, U_{R} is empty. If R is a leaf node, U_{R} consists of R and all leaf nodes adjacent to R. U_{R} consists of nodes whose particles are in the near field of R. Each region $b \in U_{\mathrm{R}}$ contains particles which contribute to the near field of particles in R.

- V_{R} contains the children of R's parent's neighbors which are well-separated from R. In fact, this is a subset of the interaction list (in the non-adaptive case).

- If R is a parent (i.e. internal) node, W_{R} is empty. If R is a leaf node, $b \in W_{\mathrm{R}}$, if b is a descendent of R's neighbors, b's parent is adjacent to R, but b is not adjacent to R. Like U_{R}, W_{R} is in the near field of R. The difference is that in the adaptive case, each $b \in W_{\mathrm{R}}$ is smaller than R and hence is separated from R by a distance greater than or equal to the length of the side of b. This separation allows the field due to particles in each $b \in W_b$ to be summarized by b's multipole expansion. This fact is used in the algorithm later. We also note that if a region $b \in W_{\mathrm{R}}$, then b's descendents are not in W_{R}.

- We consider two cases for X_{R}: when R is a leaf node and when R is an internal node. When R is a leaf node (as in Figure 4.42), for any b such that $\mathrm{R} \in W_b$, we put b in X_{R}. When R is an internal node and b is such that $\mathrm{R} \in W_b$, we recursively put b in R's leaf descendents.[28] Intuitively, X_{R} covers the rest of the interaction-list which is not covered by V_{R}. Unlike V_{R}, a region in $b \in X_{\mathrm{R}}$ is larger than R and although b is separated from R by a distance greater than or equal to the length of the side of R, b is *not* well-separated from R. We describe later how this affects the algorithm.

The regions marked with $-sign are in R's parent's far field, and their contributions are inherited from R's parent during the calculation. Therefore R does not need to know about them.

To compute the local expansion, the adaptive algorithm consists of an upward and downward pass, like the non-adaptive case. During the upward pass, each leaf region computes a multipole expansion due to its particles. A parent region translates the multipole expansions of its four children to its center and computes the sum of these translated expansions.

We use the following notation to describe the downward pass: let Γ_{R}, Δ_{R} be the local expansions (about $\mathrm{center}_{\mathrm{R}}$) of the field due to particles in the regions in V_{R}, X_{R} respectively.

Γ_{R} is obtained by transforming the multipole expansions of the regions in V_{R}, and summing them:

$$\Gamma_{\mathrm{R}} = \sum_{\mathrm{F} \in V_{\mathrm{R}}} \text{transform}(\text{multipole}(\mathrm{F}),\text{center}_{\mathrm{R}})$$

Δ_{R} is the local expansion due to the regions in X_{R}. It is calculated differently from Γ_{R} because, unlike V_{R}, a region in X_{R} is larger than R and hence is not well-separated from R.

$$\Delta_{\mathrm{R}} = \sum_{\mathrm{p} \in \mathrm{F}, \mathrm{F} \in X_{\mathrm{R}}} \text{field}(\mathrm{p}, \ \text{center}_{\mathrm{R}})$$

[28] If we look at an internal node W in Figure 4.42, R is in the $X_{r'}$ where r' is a leaf descendent of W.

We get the local expansion due to X_{R} by summing the local expansions due to each particle $p \in X_{\mathrm{R}}$.[29]

If R is an internal node, its local expansion is given by:

$$\text{local\_expansion(R)} = \text{translate(local\_expansion(R's parent),center}_{\mathrm{R}}) + \Gamma_{\mathrm{R}}$$

The local expansion of an internal node R does not include the contributions from X_{R}, because X_{R} is already included in R's leaf descendent's x-list. So when we get to a leaf node R, we simply make up for the rest of the contributions from X_{R}.

$$\text{local\_expansion(R)} = \text{translate(local\_expansion(R's parent),center}_{\mathrm{R}}) + \Gamma_{\mathrm{R}} + \Delta_{\mathrm{R}}$$

Now that we have seen the calculation of the local expansion (far field), let us look at R's near field. R's near field is given by U_{R} and W_{R}. The contributions from U_{R} are obtained by calculating pairwise interactions of particles p_0, p_1 where $p_0 \in$ R and $p_1 \in b \in U_{\mathrm{R}}$. We avoid this pairwise computation for $b \in W_{\mathrm{R}}$ because each such b is separated from R; hence b's effect on R is given by b's multipole expansion. Let Θ_{R} denote the multipole expansion due to W_{R}.

$$\Theta_{\mathrm{R}} = \sum_{\mathrm{F} \in W_{\mathrm{R}}} \text{translate(multipole(F),center}_{\mathrm{R}})$$

We summarize the adaptive fast multipole algorithm.

1. Execute the upward and downward passes over the quadtree to calculate local expansions of leaf regions. The local expansion calculation uses the v- and x-lists (instead of the interaction lists).

2. For each particle in a region, its far field is calculated by evaluating the local expansion at its position.

3. The near field of a particle p (in region R) is obtained by (i) pairwise interactions with particles in U_{R} regions, and (ii) evaluating Θ_{R} at p's position.

4. The potential field of a particle is sum of its near and far fields.

4.8.2 Parallel Implementation using pSather

In this section, we describe the parallel implementation of the fast multipole N-body algorithms. We wrote the programs in two stages. In the first stage, we focused on building a modular, correct program based on class libraries with clean interfaces. The main problem in this construction is that since pSather 0.1 does not support user-defined value class, we had to implement `COMPLEX` (which is an important class in our computation) as a reference class. As a result, a new `COMPLEX` object is allocated for each complex operation. Although this overhead is partially alleviated by having the compiler analyze object lifetimes (Section 3.5.4), it is still a major source of inefficiency, and causes memory leaks (because our prototype does not a garbage collector). In the second stage,

[29] This is valid because each particle $p \in X_{\mathrm{R}}$ is well-separated from R.

```
1 :    main is
2 :       -- Implements the main body of fast multipole algorithm.
3 :       read_input;
4 :       setup_quadtree_skel;
5 :       setup_quadtree;
6 :       distribute_pts;
7 :       upward_pass;
8 :       downward_pass;
9 :       expand_and_sum_at_particles;
10:    end;
```

Figure 4.43: Main phases of fast multipole program.

to improve program efficiency, we re-implemented some of the library classes with complex number operations in a way that pretends `COMPLEX` is a value class (as it would be in pSather 1.0). Even though this causes some change in the routine interfaces (e.g. replacing each `COMPLEX` parameter by two `DOUBLE`'s), the functionalities of the routines do not change. The tuning was easily accomplished because of the modular, object-oriented approach adopted in the first stage. In our description of the implementation, we will use the programs developed in the first stage. (Section 4.9.3 describes how 1.0 language constructs can improve the fast multipole programs in general, including implementing `COMPLEX` as a value class.)

Figure 4.43 shows the main phases of the program which are the same for both adaptive and non-adaptive versions.[30] The particle positions and charges are first read in sequentially (line 3). Then we determine the structure of the quadtree (line 4). In the non-adaptive case, we simply calculate the depth of uniform quadtree. In the adaptive case, we determine the quadtree's outline using the distribution of the particles. This pre-evaluation of the quadtree structure allows us to decide how the quadtree nodes should be distributed among the processors. In line 5, a distributed quadtree is constructed based on the pre-evaluated structure. Lines 4 – 5 build the distributed data structure sequentially. After that, the particles are distributed amongst the leaf nodes (line 6). The distribution is done in parallel for the non-adaptive algorithm, and sequentially for the adaptive algorithm. Next we compute the multipole expansions in the upward pass (line 7), the local expansions in the downward pass (line 8) and the final summation of near and far fields (line 9).

Even though the construction of the quadtree and particle distribution phases (lines 5 – 6) are done sequentially, they are included in the timings in Section 4.8.3 to provide a fair picture of the overall performance improvement.

In the following subsections, we first describe the classes shared by both programs. Then we describe the classes related to the construction and distribution of the quadtree. Finally we

[30] Since the program outlines are the same, we combine both programs in our implementation, but in the following discussion, we will treat them as two separate programs.

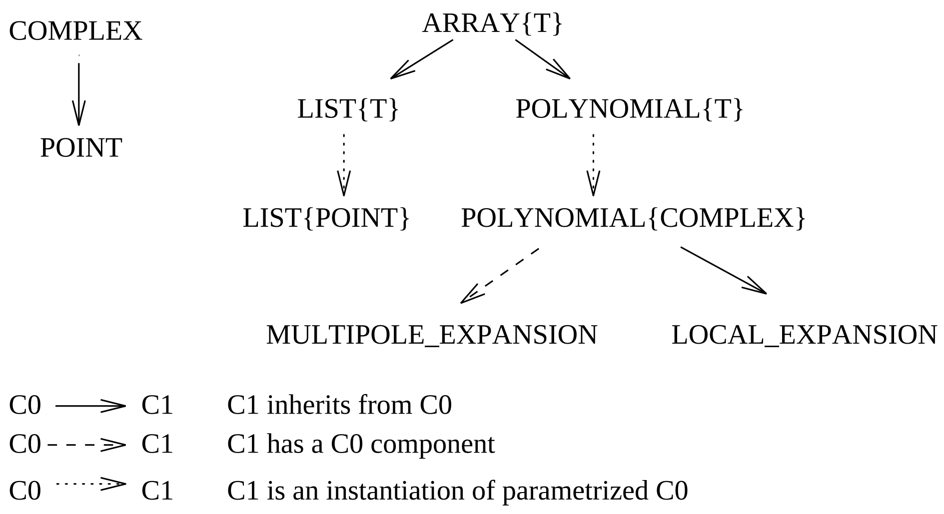


Figure 4.44: Classes shared by both adaptive and non-adaptive fast multipole programs.

describe how the routines in quadtree classes are parallelized.

Shared Classes

Figure 4.44 shows the classes shared by both adaptive and non-adaptive programs. The most fundamental classes are `COMPLEX` and `ARRAY{T}`. `COMPLEX` has an `x` and a `y` component (both of type `DOUBLE`) and provides various complex operations, e.g. `length_sq`, `log` etc. It is inherited by `POINT` (Figure 4.45) because a point, in addition to having an (x,y) position, also has a charge value (type `DOUBLE`), a far field effect (type `COMPLEX`), a near field effect (type `DOUBLE`) and a potential field (type `DOUBLE`). A `POINT` has operations to compute its potential field due to a list of points or another point.

The other fundamental class is `ARRAY{T}`. It is inherited by a `LIST{T}` class whose instantiation `LIST{POINT}` is used to hold the list of points in a leaf region. `ARRAY{T}` is also inherited by `POLYNOMIAL{T}` whose interface is shown in Figure 4.46. A `POLYNOMIAL{T}` represents a single variable polynomial; its type parameter `T` gives the type of the polynomial coefficients. The polynomial class is turn used to construct the `MULTIPOLE_EXPANSION` and `LOCAL_EXPANSION` classes (Figure 4.47). The major language flaw in 0.1 that is exposed by `POLYNOMIAL{T}` is that operations (e.g. addition) on the coefficients (of type `T`) have to be written as:

```
res[i] := [i].plus(p[i]);
```

in order for `T` to be parametrizable by `COMPLEX`. `POLYNOMIAL{T}` cannot use the more natural "`res[i] := [i] + p[i];`" because in 0.1, `+` is predefined, and we cannot define a `+` routine in `COMPLEX`. In Section 4.9.3, we describe how this problem will be solved in 1.0 using syntactic sugar.

A multipole expansion with p terms is given by:

```
class POINT is
  COMPLEX;

  charge:DOUBLE;
  far_field_effect:COMPLEX;
  near_field_effect:DOUBLE;
  potential:DOUBLE;

  to_complex:COMPLEX is ...end;
  -- Return a complex number.

  compute_near_field_wrt(pt:POINT):DOUBLE is ...end;
  -- Compute direct effect on "self" due to "pt".

  compute_near_field(pts:LIST{POINT}):DOUBLE is ...end;
  -- Compute sum of effects on "self" from each point in "pts".

end;
```

Figure 4.45: Interface of `POINT` class.

$$\phi(z) = Q\log(z) + \sum_{k=1}^{p} a_k/z^k$$

`MULTIPOLE_EXPANSION` does not inherit from `POLYNOMIAL{COMPLEX}`, because the latter is intended to represent polynomials whose terms have positive power:

$$f(z) = \sum_{k=0}^{p} c_k z^k$$

But since `POLYNOMIAL{COMPLEX}` keeps an array of coefficients, we can reuse it to represent the coefficients a_k in $\phi(z)$. Other attributes of a multipole expansion include the sum of particle charges Q, and a point about which the expansion is centered. The class provides routines to create a new multipole expansion from a list of points and a center (`create`), to translate an expansion to a new center (`translate`), to add two expansions (`plus`), to compute a local expansion from a multipole expansion (`local_expansion`) and to copy an expansion to a remote cluster (`make_copy_to`).

A local expansion with p terms is given by:

$$\varphi(z) = \sum_{k=0}^{p} b_k z^k$$

```
class POLYNOMIAL{T} is
  -- Parametrized by the type of the polynomial coefficients.
  ARRAY{T};

  create(n:INT):SELF_TYPE is ...end;
  -- Create a zero-initialized polynomial with "n" terms.

  degree_of:INT is ...end;
  -- Degree of polynomial.

  nth_coefficient(n:INT):T is ...end;
  -- Read coefficient of "n"th degree term.

  assign_nth_coeff(n:INT; v:T) is ...end;
  -- Update coefficient of "n"th degree term.

  plus(p:$POLYNOMIAL{T}):$POLYNOMIAL{T} is ...end;
  -- A new polynomial which is sum of the input polynomials.
  -- Does not destroy original polynomial objects.

  negate:SELF_TYPE is ...end;
  -- Negate the polynomial; destructive operation.

  multiply(p:$POLYNOMIAL{T}):$POLYNOMIAL{T} is ...end;
  -- A new polynomial which is product of the input polynomials.

  value_at(v:T):T is ...end;
  -- Return the value of the polynomial at "v", i.e. f(v).

end;
```

Figure 4.46: Interface of POLYNOMIAL{T}.

```
class MULTIPOLE_EXPANSION is
  sum_of_charges:DOUBLE; -- Coefficient Q.
  finite_series:POLYNOMIAL{COMPLEX};
  -- Complex coefficients a_k in expansion.

  center:POINT;

  create(pts:LIST{POINT}; origin:POINT):
      MULTIPOLE_EXPANSION is ...end;
  -- "origin" is the center of new multipole expansion.

  translate(v:POINT):MULTIPOLE_EXPANSION is ...end;
  -- "v" is the center of the new expansion.

  plus(mp:MULTIPOLE_EXPANSION):MULTIPOLE_EXPANSION is ...end;
  -- Return sum of two multipole expansions.

  local_expansion(v:POINT):LOCAL_EXPANSION is ...end;
  -- Convert a multipole expansion into a local expansion.
  -- "v" is the center of the local expansion.

  make_copy_to(cid:INT):SELF_TYPE is ...end;
  -- Copy multipole expansion to destination cluster "cid".
end;


class LOCAL_EXPANSION is
  POLYNOMIAL{COMPLEX}; -- Inherits "plus" and "value_at".
  center:POINT; -- Origin coordinate of this expansion.

  create(n:INT; v:POINT):SELF_TYPE is ...end;
  -- Create a zero-initialized polynomial with 'n' terms.

  init_with(c:COMPLEX) is ...end;
  -- Initialize local expansion with this value 'c' for all
  -- coefficients.

  translate(v:POINT):LOCAL_EXPANSION is ...end;
  -- Compute a new local expansion for new center "v".

  create_with_pts(pts:LIST{POINT}; origin:POINT):SELF_TYPE is ...end;
  -- The new local expansion is centered w.r.t.  "origin".
  -- "pts" and "origin" are given in the same co-ordinate system.

  make_copy_to(cid:INT):SELF_TYPE is ...end;
  -- Copy local expansion to the destination cluster "cid".
end;
```

Figure 4.47: Interfaces of `MULTIPOLE_EXPANSION` and `LOCAL_EXPANSION` classes.

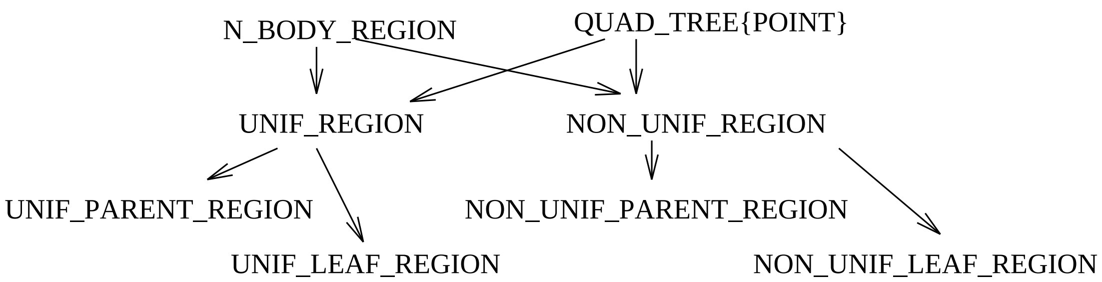


Figure 4.48: Classes to build distributed quadtree in adaptive and non-adaptive fast multipole programs.

Therefore `LOCAL_EXPANSION` (Figure 4.47) inherits from `POLYNOMIAL{COMPLEX}` and reuses the code for routines such as `plus` and `value_at`. In addition, a local expansion has routines to translate to a new center (`translate`), to create a local expansion from a list of points (`create_with_pts`) and to copy an expansion to a remote cluster (`take_copy_to`).

Computing Quadtree Distribution

Next we describe the two set of classes used in the quadtree constructions. The first is used to determine the structure of the quadtree and distribution of the nodes. The second set (Figure 4.48) consists of classes which are used to create the actual nodes in the trees, and which contain routines that compute the multipole and local expansions.

To determine the skeleton and distribution of the quadtree, we use the following classes: `UNIF_PARTITION`, `QUADTREE_SKEL{T}` and `QUAD_TREE{T}`. The interesting aspect of these classes is their encapsulation of a quadtree's distribution strategy.

Given a uniform distribution, we can directly compute the number of leaf nodes and on which processor (cluster) each leaf node should be located. The `UNIF_PARTITION` class (Figures 4.49 and 4.50) encapsulates this information. When a `UNIF_PARTITION` is created, it is given the total number of points N in the region and the average number of points n per leaf region (line 13). This allows us to calculate the height of the quadtree which is given by $\log(N/n)/\log(4)$ (lines 16 – 18). By default, we assume that the quadtree will be distributed among all the processors (line 19). Given `nr` leaf regions, we calculate the average number of leaf regions per processor (lines 26 – 28). For example if we have 16 leaf regions and 3 processors, `avg_num_nodes` is 3, `extra` is 1 and `id_divide` is 6, so the leaves 0 – 5 will be on processor 0, 6 – 10 on processor 1 and 11 – 15 on processor 2. This means that we need an ordering among the leaf nodes. The ordering should be such that spatially nearby leaf nodes are located together as much as possible, because each node needs to know about its neighbors when computing the near field. For example, the distribution in Figure 4.51 (b) requires less communication than Figure 4.51 (a). One solution is to order the leaf nodes depth-first (Figure 4.51 (c)).

```
class UNIF_PARTITION is

   num_clusters:INT;
   height:INT;
   avg_num_nodes:INT;
   extra:INT;
   id_divide:INT;

   -- The actual integer values are crucial to the calculation of
   -- the depth-first ordering of the leaves.
   constant top_left:INT := 0;
   constant top_right:INT := 1;
   constant bottom_left:INT := 2;
   constant bottom_right:INT := 3;


   create(avg_numpts, numpts:INT):SELF_TYPE is
      -- Return a partitioner object.
      res := new;
      height:INT := (MATH::log(numpts.to_d/avg_numpts.to_d)/
         MATH::log(4.0)).ceiling;
      res.height := height;
      res.num_clusters := CONFIG::current_num_clusters;
      nr:INT := 1; -- Number of leaf regions.
      i:INT;
      until (i >= height) loop
         nr := nr * 4;
         i := i+1;
      end;
      res.avg_num_nodes := nr/res.num_clusters;
      res.extra := nr.u_mod(res.num_clusters);
      res.id_divide := res.extra*(res.avg_num_nodes+1);
   end; -- create


   private position_in_parent(i,j:INT):INT is
      -- Given the (i,j) position of a child region, determine
      -- its position within the parent region.
      diffi:INT := i.bit_and(0x1);
      diffj:INT := j.bit_and(0x1);
      if (diffi = 0) then
         if (diffj = 0) then res:=top_left
         else res:=bottom_left end;
      else if (diffj = 0) then res:=top_right
         else res:=bottom_right end; end;
   end; -- position_in_parent
```

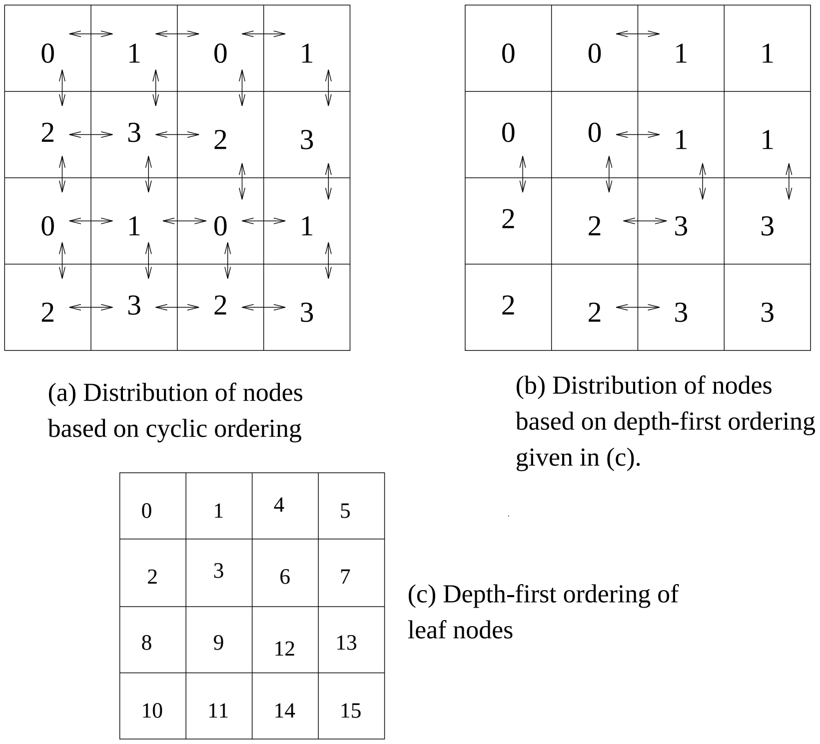
Figure 4.49: Part 1 of implementation of UNIF_PARTITION class which determines the location of a uniform quadtree node.

```
   order_of_leaf_node_at(i, j:INT):INT is
      -- For a uniform quad-tree, it is possible to compute the
      -- depth-first ordering of the leaf region.
      if (height = 0) then return end;
      exp:INT := 1; -- 2^0, 2^1 ...  2^(height-1)
      order:INT;
      lev:INT := height;
      until (lev <= 0) loop
         pos:INT := position_in_parent(i, j);
         order := order+pos*exp*exp;
         i := i.rshift(1); j := j.rshift(1);
         exp := exp*2;
         lev := lev-1;
      end; -- loop
      res := order;
   end; -- order_of_leaf_node_at


   cluster_id_of_leaf_node_at(i, j:INT):INT is
      -- For a uniform quad-tree, it is possible to compute the
      -- cluster on which a particular node is to be located.
      if (height = 0) then return end;
      -- First compute the depth-first ordering of the leaf region.
      order:INT := order_of_leaf_node_at(i, j);
      if (order < id_divide) then
         res := order/(avg_num_nodes+1);
      else
         res := (order-id_divide)/avg_num_nodes+extra;
      end;
   end; -- cluster_id_of_leaf_node_at

end;
```

Figure 4.50: Part 2 of implementation of UNIF_PARTITION class which determines the location of a uniform quadtree node.



(a) Distribution of nodes based on cyclic ordering

(b) Distribution of nodes based on depth-first ordering given in (c).

(c) Depth-first ordering of leaf nodes

Figure 4.51: Node distributions based on different orderings of leaf nodes. The number in each box gives its cluster location.

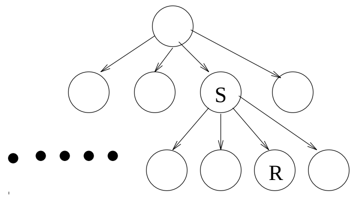


Figure 4.52: Computing depth-order of a leaf node.

To compute this ordering, we assign an (i,j) position to each node, where i increases from left to right and j increases from top to bottom. If a parent node has position $(i,\ j)$, its top-left, top-right, bottom-left and bottom-right children have position $(2i,\ 2j)$, $(2i+1,\ 2j)$, $(2i,\ 2j+1)$ and $(2i+1,\ 2j+1)$ respectively. Given a child node's (i,j) position, it is simple to find out its position within the parent node by looking at whether i, j are odd or even (lines 30 – 40).

An (i,j) leaf node's depth-first order is obtained by adding the number of leaf nodes that precede it and each of its ancestor. For example in Figure 4.52, there are 2 nodes that precede R and share the same parent as R, 2×4 nodes that precede S and share the same parent as S. So R's number is 10. This is the computation performed in the routine `order_of_leaf_node_at` (lines 41 – 56, Figure 4.50). We start from the leaf node at the finest depth (lines 45 – 47). For each depth, we get the current node's position within the parent (line 49). The local variable `exp` keeps track of the number of leaves on each side of a subtree of the current node. The value "`pos*exp*exp`" gives the number of *additional* preceding nodes and is added to a cumulative sum (line 50). Then we move up one level to the node's parent (lines 51 – 53). The final sum gives the depth-first order of a leaf node (line 55).

To find out the cluster location of a leaf node, we first get its depth-first order (line 62). Given r leaf nodes and p clusters, we divide them p groups, such that for $(r \bmod p)$ groups, each has $\lfloor r/p \rfloor + 1$ nodes, and the rest of the groups each has $\lfloor r/p \rfloor$ nodes. Using this information and a leaf node's order number, we can compute its location (lines 63 – 67).

For a non-uniform distribution, we cannot a priori calculate the distribution of quadtree nodes. So we sequentially construct a quadtree skeleton using `QUADTREE_SKEL{T}` (whose interface is given in Figure 4.53). The routines `create_uniform` and `create_non_uniform` are used to create skeletons for uniform and non-uniform quadtree respectively. After a non-uniform skeleton is created, we use `incr_distrib_pt` to distribute the input points, and incrementally determine the quadtree structure. Then we call `partition` to decide the location of both internal and leaf nodes.[31]

Given p clusters, and $r = kp$ leaf nodes, the `partition` uses a cursor that generates r

[31] The `partition` routine works for both uniform and non-uniform quadtrees.

```
class QUADTREE_SKEL{T} is
  create_uniform(avg_numpts:INT; numpts:INT;
      min_x, max_x, min_y, max_y:DOUBLE):SELF_TYPE is ...end;
  -- Create a skeleton for uniform quadtree.

  create_non_uniform(avg_numpts, numpts:INT;
      min_x, max_x, min_y, max_y:DOUBLE):SELF_TYPE is ...end;
  -- Return a skeleton structure that reflects the distribution
  -- of points.

  incr_distrib_pt(pt:T) is ...end;
  -- Incrementally distribute a point; skeleton may change.

  partition is ...end;
  -- Perform a depth-first traversal, and assign the nodes to
  -- clusters.

end;
```

Figure 4.53: Public interface of QUADTREE_SKEL{T}.

numbers:

$$\underbrace{\overbrace{0, 0, \ldots, 0}^{k}, \overbrace{1, \ldots, 1}^{k}, \ldots, \overbrace{(p-1), \ldots, (p-1)}^{k}}_{r}$$

(If r is not exactly divisible, the first (r mod p) groups each has $\lfloor r/p \rfloor + 1$ nodes, while the other groups each has $\lfloor r/p \rfloor$ nodes.) As `partition` performs a depth-first traversal of the skeleton, whenever it encounters a leaf node, it invokes the cursor to get the next cluster location to assign to the leaf. When it encounters an internal node, it first recursively calls `partition` on the four children. After the recursive calls, we set the parent's location to be the same as that of the majority of its children.[32]

After building a skeleton for either a uniform or a non-uniform quadtree, the `create` routine in `QUAD_TREE{T}` uses a `QUADTREE_SKEL{T}` object to allocate the nodes of a quadtree according to the computed distribution.

Figure 4.54 gives the interface of `QUAD_TREE{T}`. The `create` routine uses the routines `internal_node_create` and `leaf_create` to allocate internal and leaf nodes respectively. Since the implementation of `QUAD_TREE{T}` is inherited by the classes representing the regions (Figure 4.48), the `internal_node_create` and `leaf_create` routines are redefined by the descendents if they have other attributes which are not common to all quadtrees. For example, any `UNIF_REGION` has an interaction list, while any `NON_UNIF_REGION` has an x-list.

[32] Because of the way we assign cluster locations to leaf nodes, the only ways when there is a tie are (i) location(child_1) = location(child_2) $\neq$ location(child_3) = location(child_4), or (ii) each child has a different location. In (i), we simply use the location of children 1 and 2. In (ii), we put the internal node on the same cluster as the child with the smallest number of points. The reason is that a child with more points has a deeper subtree, so we avoid putting more nodes onto the same cluster as that subtree.

```
class QUAD_TREE{T} is
  create(skel:QUADTREE_SKEL{T}):$SELF_TYPE is ...end;
  -- Create a quadtree according to distribution in the skeleton.

  internal_node_create(p:$SELF_TYPE; l,i,j:INT;
        min_x, max_x, min_y, max_y:DOUBLE):SELF_TYPE is ...end;
  -- Create an internal node.

  leaf_create(p:$SELF_TYPE; l,i,j:INT;
        min_x, max_x, min_y, max_y:DOUBLE):SELF_TYPE is ...end;
  -- Create an leaf node.

  node_at(d,i,j:INT):$SELF_TYPE is ...end;
  -- The node at given depth, and (i,j) coordinates.  Returns a
  -- smallest covering node when the exact node is missing.

  exact_node_at(d,i,j:INT):$SELF_TYPE is ...end;
  -- The node at given depth, and (i,j) coordinates.
  -- Returns "void" when the exact node is missing.

  is_a_neighbor_of(r:$SELF_TYPE):BOOL is ...end;
  -- Returns true if "r" is a neighbor of "self".
  -- Assume that "r" is at the same depth as "self".

  is_adjacent_to(r:$SELF_TYPE):BOOL is ...end;
  -- Returns true if "r" is adjacent of "self", independent of
  -- of the difference between the depths.  Two regions are also
  -- adjacent if one is included in the other.

  is_ancestor_of(r:$SELF_TYPE):BOOL is ...end;
  -- Returns true if "self" is an ancestor of "r".

  is_leaf:BOOL is ...end;
  -- Returns true if "self" is a leaf node.

  height:INT is ...end;
  -- Height of a tree.  (-1 when height is unknown.)
```

Figure 4.54: Public interface of QUAD_TREE{T}.

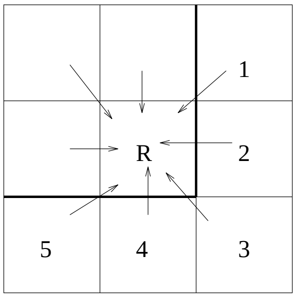


Figure 4.55: Repeated access to `R`'s attribute results in extra communication.

The other routines in `QUAD_TREE{T}` are more general and are used to determine the relationships between different nodes. For example, `node_at` gives the node at depth `d`, with position (`i`, `j`). It returns a smallest node covering that region if no exact node is found. The functionalities of other routines are described in Figure 4.54.

We have mentioned earlier that there is a second set of classes which are used to build the actual quadtree for the N-body program. Their routines can be divided into two categories: those that build the interaction lists etc., and those perform the actual potential field calculation. The `UNIF_REGION` class therefore contains routines:

```
get_neighbors is ...end;
get_interaction_list is ...end;
```

while `NON_UNIF_REGION` has:

```
get_neighbors is ...end;
get_u_list is ...end;
get_v_list is ...end;
get_w_list is ...end;
get_x_list is ...end;
```

In the next subsections, we introduce the use of caches to improve data locality and then describe how the multipole and local expansion calculations are parallelized in these classes.[33]

Software Caches

During the potential field calculation, a node's multipole expansion, local expansion and list of points may each be accessed by several other nodes, some or all of which may be in a different processor. For example, during the near field calculation, a node has to get the list of points of all its neighbors. In Figure 4.55, if regions 1 – 5 are all located on the same processor (cluster), then repeated access to `R` results in unnecessary and extra communication.

[33]The computation of interaction list, x-list etc. have also been parallelized in a similar way.

```
class CACHES is

  shared m_exp_cache:REPL_OB_HASH_MAP{MULTIPOLE_EXPANSION};
  shared l_exp_cache:REPL_OB_HASH_MAP{LOCAL_EXPANSION};

  shared neighbor_points_cache:REPL_OB_HASH_MAP{LIST{POINT}};
  -- Cache for list of points from neighbors or adjacent nodes which
  -- are located in remote processors.

  init_class is
    sync_m_exp_cache(REPL_OB_HASH_MAP{MULTIPOLE_EXPANSION}::create);
    sync_l_exp_cache(REPL_OB_HASH_MAP{LOCAL_EXPANSION}::create);
    sync_neighbor_points_cache(REPL_OB_HASH_MAP{LIST{POINT}}::create);
  end; -- init_class

end;
```

Figure 4.56: A class implementing caches for either adaptive or non-adaptive programs.

To improve data locality, we reuse the idea of a replicated hash table. While Section 4.3 describes the implementation of `REPL_INT_HASH_MAP{T}` that maps from `INT` to `T`, what we need in the N-body program is a replicated hash table that maps from `$OB` to `T`. The code for this `REPL_OB_HASH_MAP{T}` class is similar to `REPL_INT_HASH_MAP{T}` and is shown in Appendix E.

There is little difference between `REPL_INT_HASH_MAP{T}` and `REPL_OB_HASH_MAP{T}`, except that the latter has an additional `get_cached_ob(k:T):T` routine:

```
get_cached_ob(k:T):T is
  res := k;
  if (k.is_far) then
    s:PROTECTED_OB_HASH_MAP{T} := local_set;
    res := s.get(k);
    if (res = void) then
      res := k.make_copy_to(local_cid);
      s.insert(k, res);
  end; end;
end; -- get_cached_ob
```

Given an object `k`, we check whether it is on the local cluster. If not, then we check the local table for any mapping from `k` to a local object. If no such mapping exists, we make a local copy of `k` and store the mapping from `k` to the new object. This routine assumes that `T` is a subtype of `$OB`. (It is not applicable to `REPL_INT_HASH_MAP{T}` because `T` in `REPL_INT_HASH_MAP{T}` can never be a subtype of `INT`).

The replicated hash table is used to implement caches for the following objects (Figure 4.56): multipole expansions, local expansions and list of points in neighboring or adjacent regions. We give an example of how the caches are used. For example, in the near field computation, we want to get

```
foreach (child region r)
  Create a thread to compute r's multipole expansion.
end
Wait till all children's computations complete.
foreach (child region r)
  Translate r's multipole expansion to parent region's center.
end
Result ← Summation of translated expansions.
```

Figure 4.57: Pseudo-code for an internal node during upward pass.

the list of points from an adjacent region $p \in U_{\mathrm{R}}$. If we have uniform shared-memory, we would write:

```
until (i >= u_list.size) loop
  p:NON_UNIF_LEAF_REGION := u_list[i];
  remote_pts:LIST{POINT} := p.points;
  compute_near_field_from_region(remote_pts);
  i := i+1;
end;
```

On a distributed-memory machine, we call the cache to make local copies of any remote lists and keep the mapping from a remote list to a local copy. We only need to alter the code slightly to take into account data locality.

```
until (i >= u_list.size) loop
  p:NON_UNIF_LEAF_REGION := u_list[i];
  remote_pts:LIST{POINT} :=
    CACHES::neighbor_points_cache.get_cached_ob(p.points);
  compute_near_field_from_region(remote_pts);
  i := i+1;
end;
```

The multipole and local expansion calculations use the other caches in a similar manner.

Code Parallelization

Figure 4.57 shows how the upward pass can be parallelized. Each internal node creates threads to compute the multipole expansions of its children in parallel. The leaf nodes can compute their multipole expansions directly from the particles. Each internal node waits for its children to finish computing their multipole expansions. It then translates these expansions and sums them up.

On a CM-5, we can reduce the amount of thread creation by forking a thread only when a child is on a remote cluster (Figure 4.58). The upward pass uses a `cobegin-end` programming style for the parent to synchronize with the child threads. A child node gets a new thread to compute its multipole expansion only when it is located on a remote cluster (lines 5 – 20). For each child node that is located on the same cluster as the parent, we make recursive calls to compute their multipole

```
compute_multip_exp is
    fork_tl, fork_tr, fork_bl, fork_br:BOOL;
    mp_tl, mp_tr, mp_bl, mp_br:MULTIPOLE_EXPANSION;
    cobegin
      if (top_left.where /= where) then
        :- top_left.compute_multip_exp @ top_left.where;
        fork_tl := true;
      end;
      if (top_right.where /= where) then
        :- top_right.compute_multip_exp @ top_right.where;
        fork_tr := true;
      end;
      if (bottom_left.where /= where) then
        :- bottom_left.compute_multip_exp @ bottom_left.where;
        fork_bl := true;
      end;
      if (bottom_right.where /= where) then
        :- bottom_right.compute_multip_exp @ bottom_right.where;
        fork_br := true;
      end;

      if not (fork_tl) then top_left.compute_multip_exp;
        mp_tl := top_left.mp_exp; end;

      if not (fork_tr) then top_right.compute_multip_exp;
        mp_tr := top_right.mp_exp; end;

      if not (fork_bl) then bottom_left.compute_multip_exp;
        mp_bl := bottom_left.mp_exp; end;

      if not (fork_br) then bottom_right.compute_multip_exp;
        mp_br := bottom_right.mp_exp; end;
    end;

    if (mp_tl = void) then mp_tl := CACHES::m_exp_cache
      .get_cached_ob(top_left.mp_exp); end;
    if (mp_tr = void) then mp_tr := CACHES::m_exp_cache
      .get_cached_ob(top_right.mp_exp); end;
    if (mp_bl = void) then mp_bl := CACHES::m_exp_cache
      .get_cached_ob(bottom_left.mp_exp); end;
    if (mp_br = void) then mp_br := CACHES::m_exp_cache
      .get_cached_ob(bottom_right.mp_exp); end;

    mp_exp := mp_tl.translate(center)
      .plus(mp_tr.translate(center))
      .plus(mp_bl.translate(center))
      .plus(mp_br.translate(center));
  end; -- compute_multip_exp
```

Figure 4.58: Parallel computation of multipole expansion.

expansions (lines 21 – 28). We wait for all child threads to terminate, then get the child node's multipole expansion, caching local copies when necessary (lines 30 – 37). Lines 38 – 41 compute the parent's multipole expansion using the `translate` and `plus` routines from `MULTIPOLE_EXPANSION`.

In the downward pass, we parallelize the computation by observing that any node R (internal or leaf) has two components for the far field. In the non-adaptive case, they are *F2* (from the interaction list) and *F1* (parent's local expansion) (Section 4.8.1). In the adaptive case, they are Γ_{R} and the parent's far field (Section 4.8.1); the Δ_{R} component of the leaf nodes' far field are added in a later stage. So, in either case, when a node waits for its parent's far field to be available, it can continue to compute *F2* (or Γ_{R}). A parent signals completion of its far field, by setting its children's `GATE`{`LOCAL_EXPANSION`}.

Figure 4.59 shows how the downward pass is parallelized for an internal node. A zero depth node (root) does not get any far field from its parent; both its far and very far fields[34] are set to 0 (lines 2 – 4). A node of depth 1 gets a zero-valued very far field from its parent (lines 5 – 6).

A thread is created for each remote child node (lines 8 – 24). If a child node is local, its far field is computed in a recursive (lines 30 – 33), after we have computed the current node's far field. The routine to compute a node's far field (line 25) is shown in Figure 4.60; the non-adaptive program replaces `v_list` by `interaction_list`. Lines 26 – 29 signal to the child node that the parent's far field is ready.

A leaf node's `compute_far_field` code is similar, except lines that act on the children (lines 8 – 24, 26 – 33) are not necessary.

The point to note about Figure 4.60 is that each node uses a gate object to synchronize with its parent. The operation `very_far_field.read` (line 12) automatically suspends a thread when the parent's far field is not yet available. The rest of the code (lines 3 – 10, 13) simply computes the components of the far field and sums them up.

The final phase computes the near fields and the far fields, and sums them up. The parallelization strategy is similar — we fork off new threads for remote nodes and make recursive calls for local nodes.

4.8.3 Performance of Fast Multipole

This section discusses the performance of the fast multipole N-body program. We start the timer after the skeleton is built, and before the quadtree is constructed. To be more precise, the timings are for lines 5 – 9 in Figure 4.43. By including some of the sequential phases in the timings, we ensure a fairer picture of the overall performance gain. The calculations use polynomials with 17 terms. The programs have been fine-tuned to treat `COMPLEX` and `POINT` as value classes.

Table 4.16 (a) shows the timings of the non-adaptive version (i.e. uniform input). In these cases, each quad-tree has, on average, ≤ 100 points per leaf node. We note that for 32 processors,

[34] A node's *very far* field is its parent's far field.

```
compute_far_field is
     if (depth = 0) then
        very_far_field.set(<zero polynomial>);
        far_field := <zero polynomial>
     elsif (depth = 1) then
        very_far_field.set(<zero polynomial>);
     end;

     fork_tl, fork_tr, fork_bl, fork_br:BOOL;
     if (top_left.where /= where) then
        :- top_left.compute_far_field @ top_left.where;
        fork_tl := true;
     end;
     if (top_right.where /= where) then
        :- top_right.compute_far_field @ top_right.where;
        fork_tr := true;
     end;
     if (bottom_left.where /= where) then
        :- bottom_left.compute_far_field @ bottom_left.where;
        fork_bl := true;
     end;
     if (bottom_right.where /= where) then
        :- bottom_right.compute_far_field @ bottom_right.where;
        fork_br := true;
     end;

     compute_own_far_field;
     top_left.very_far_field.set(far_field);
     top_right.very_far_field.set(far_field);
     bottom_left.very_far_field.set(far_field);
     bottom_right.very_far_field.set(far_field);

     if not (fork_tl) then top_left.compute_far_field end;
     if not (fork_tr) then top_right.compute_far_field end;
     if not (fork_bl) then bottom_left.compute_far_field end;
     if not (fork_br) then bottom_right.compute_far_field end;
end; -- compute_far_field
```

Figure 4.59: Parallel computation of local expansion for an internal node.

```
compute_own_far_field is
   far_field := <zero polynomial>;
   i:INT;
   until (i >= v_list.size) loop
      r:$NON_UNIF_REGION := v_list[i];
      mp:MULTIPOLE_EXPANSION :=
         CACHES::m_exp_cache.get_cached_ob(r.mp_exp);
      far_field := far_field.plus(mp.local_expansion(center));
      i := i + 1;
   end;

   parent_far_field:LOCAL_EXPANSION
      := CACHES::l_exp_cache.get_cached_ob(very_far_field.read);
   vff:LOCAL_EXPANSION := parent_far_field.translate(center);
   far_field := far_field.plus(vff);
end; -- compute_own_far_field
```

Figure 4.60: Computing a node's local expansion.

No. of processors	$N = 10000$	$N = 15000$	$N = 20000$	$N = 30000$
1	149.42	237.88	365.32	513.47
2	74.64	118.42	180.03	254.99
4	38.94	61.65	93.82	134.72
8	21.55	34.35	52.88	69.57
16	12.75	19.67	28.66	36.68
32	10.68	15.30	19.48	20.11

(a) Grain size is ≤ 100 points per region (on average).

No. of processors	$N = 10000$	$N = 15000$	$N = 20000$	$N = 30000$
1	150.17	372.99	412.39	538.78
2	74.80	187.30	207.16	255.18
4	38.60	93.16	103.28	132.91
8	21.58	50.70	55.59	69.56
16	14.77	27.32	31.64	37.93
32	10.91	14.38	16.43	19.07

(a) Grain size is ≤ 50 points per region (on average).

Table 4.16: Computation time (in seconds) of fast multipole N-body program for uniform input, N = number of points.

	$N = 10000$	$N = 15000$	$N = 20000$	$N = 30000$
≤ 100 points/leaf	94.68	146.49	224.06	336.93
≤ 50 points/leaf	93.70	244.99	270.89	335.04

Table 4.17: Sequential computation time (in seconds) of non-adaptive fast multipole N-body program (pSather/V3) for uniform input, N = number of points.

the execution times increase only two-fold from 10000 to 30000, so we obtain better speedups for larger inputs. When we reduce the granularity to ≤ 50 points per leaf node (on average), we get the timings in Table 4.16 (b). The effect of reducing the granularity is that the quad-tree's size increases, while for each leaf node, the amount of near field computation decreases.

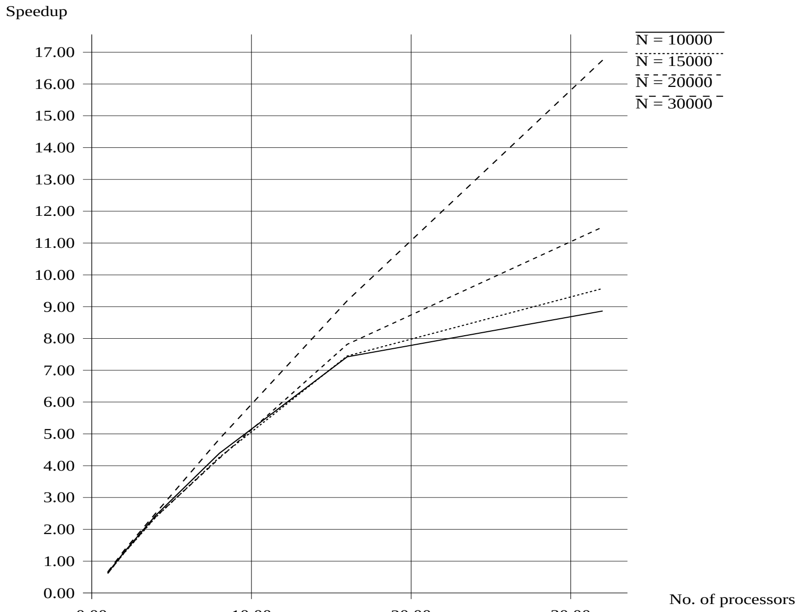
To get the speedup graphs for Tables 4.16 (a) and (b), we compile the parallel program without generating polling points and measure its execution time on a CM-5 node. This is the version we referred to as pSather/V3 in Sections 4.2.1, 4.4.1 and 4.6.1. The results are shown in Table 4.17. We think that it is justifiable to use these timings as the base times, because from other measurements (e.g. N-queens, primes, SOR), the speedup graphs based on sequential pSather/V3 timings have been similar to those calculated relative to the C timings. The speedups relative to C are worse than those relative to pSather/V3, but not significantly so. Figure 4.61 (a) and (b) show the speedup graphs for Tables 4.16 (a) and (b) respectively. Again we note that the major runtime overhead is incurred by the polling points.

Next we look at the timings of the adaptive version with non-uniform input. We consider two kinds of uniform inputs — a normal distribution and a distribution in which the points are distributed along several fixed lines.

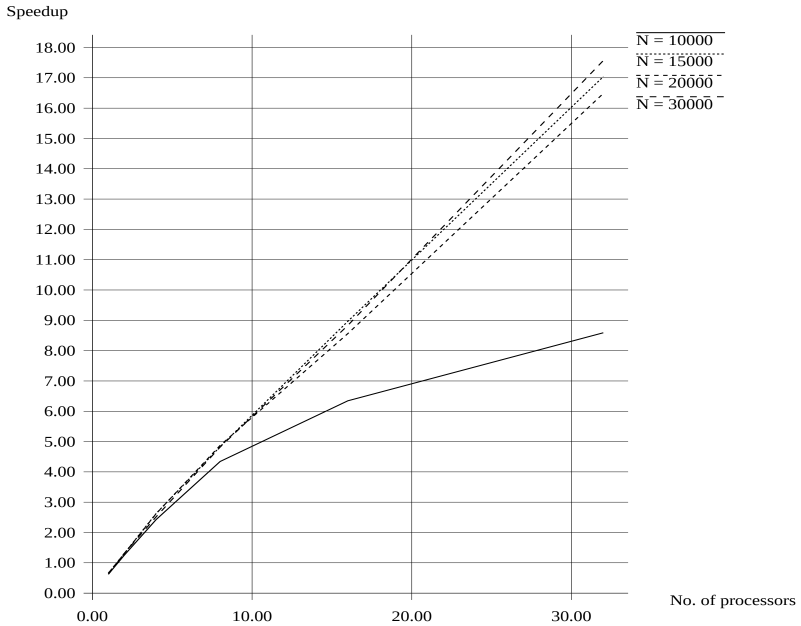
The normal distribution is generated by using two random normal generators, one for the x-axis, the other for the y-axis. (We discard any point that falls outside the square bounded by (0,0), (0,1) (1,0) and (1,1).) The line distribution is given in Figure 4.62. Each segment has the same number of particles. Within a segment, the particles are even distributed.

The parallel timings for line and bell inputs are shown in Tables 4.18 and 4.21 respectively. The corresponding pSather/V3 timings are given in Tables 4.19 and 4.21. These timings are used to construct the speedup curves given in Figures 4.64 and 4.65.

One possible reason that the non-uniform distributions get lower speedups than the uniform distribution is because the points are not evenly balanced among all the processors. This may arise because our quadtree node placement strategy was based on trying to place an almost equal number of nodes among all processors, assuming that the nodes have approximately the same number of points. But for non-uniform inputs, the number of points in the nodes may vary a lot. For example, consider Figure 4.63, the distribution of points in two 1000 point inputs (one uniform and one with normal distribution). The number in a box gives the number of points within that box. A black boundary groups nodes on the same processor (assuming that there are 8 processors). We only



a.



b.

Figure 4.61: Speedups of non-adaptive fast-multipole N-body program w.r.t. pSather/V3: (a) $\leq$ 100 points per leaf (on average), (b) $\leq$ 50 points per leaf (on average).

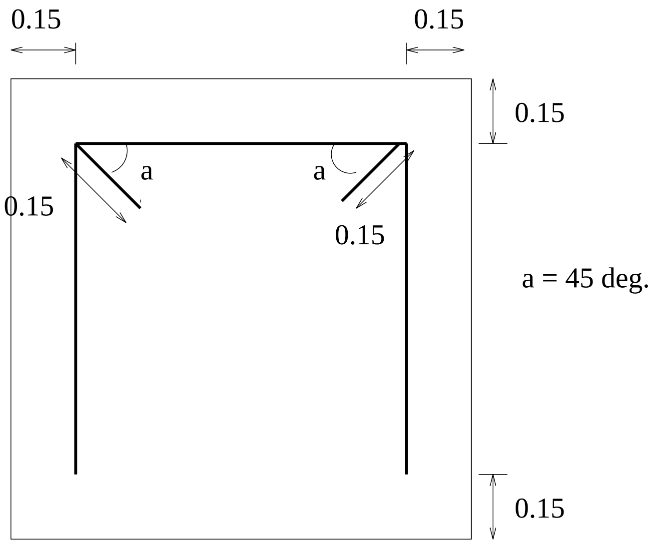


Figure 4.62: Distribution of particles for line input.

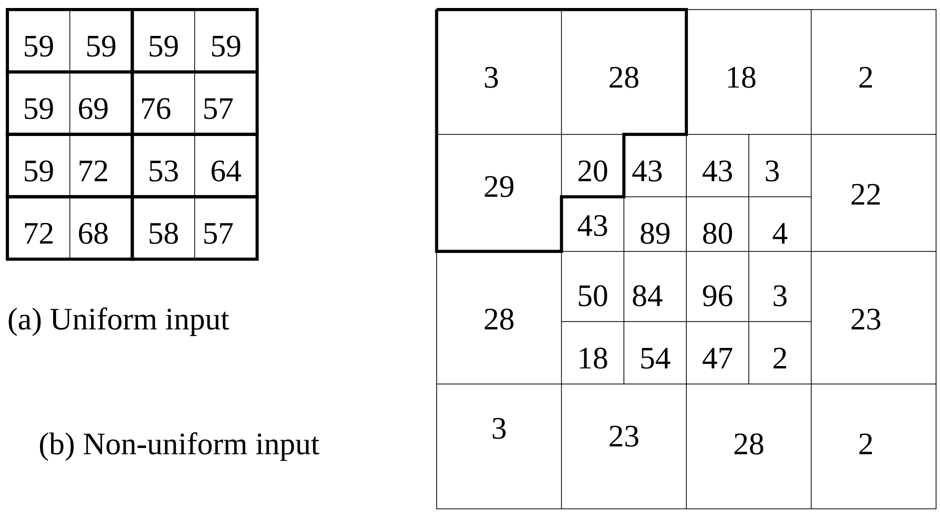


Figure 4.63: Distribution of particles for (a) 1000 point uniform input and (b) 1000 point normal distribution.

No. of processors	$N = 10000$	$N = 15000$	$N = 20000$	$N = 30000$
1	106.19	159.13	219.72	327.83
2	79.80	123.63	164.47	244.58
4	40.12	55.90	79.29	114.03
8	23.55	33.68	46.16	72.50
16	14.44	19.11	26.03	39.84
32	8.41	12.90	14.82	20.37

(a) Grain size is ≤ 100 points per region (on average).

No. of processors	$N = 10000$	$N = 15000$	$N = 20000$	$N = 30000$
1	132.34	175.61	275.91	363.54
2	94.53	132.76	199.02	262.85
4	48.53	65.18	101.52	133.04
8	27.34	41.36	59.87	77.71
16	14.87	20.39	32.99	39.25
32	8.91	11.38	19.58	25.49

(a) Grain size is ≤ 50 points per region (on average).

Table 4.18: Computation time (in seconds) of fast multipole N-body program for non-uniform line input, N = number of points.

	$N = 10000$	$N = 15000$	$N = 20000$	$N = 30000$
≤ 100 points/leaf	74.99	113.39	154.14	228.11
≤ 50 points/leaf	88.69	121.50	189.39	254.16

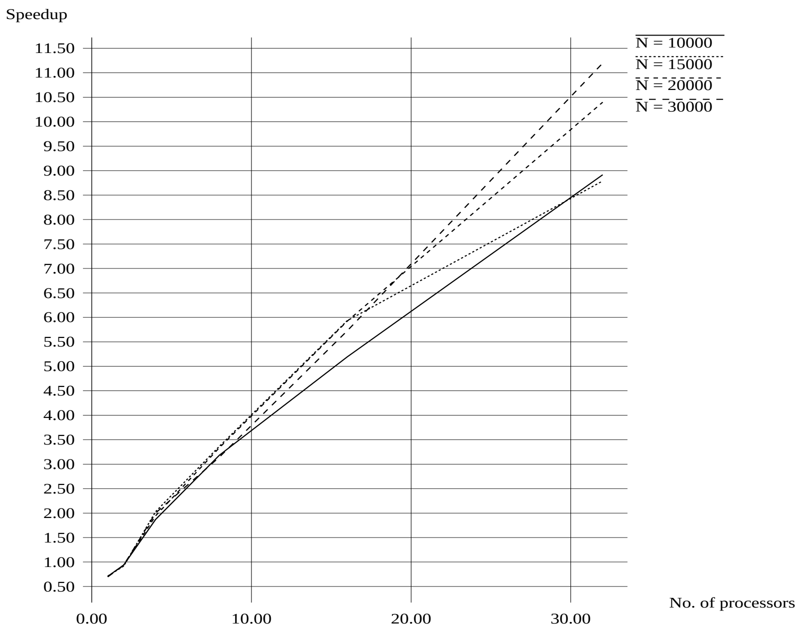
Table 4.19: Sequential computation time (in seconds) of adaptive fast multipole N-body program (pSather/V3) for non-uniform line input, N = number of points.

show one grouping in (b) because most of the groupings are non-contiguous and difficult to outline. In the uniform input, the number of points per processor range from 115 to 140. In the normal distribution, the range is from 52 to 193 which can lead to greater load imbalance, and thus smaller speedups.

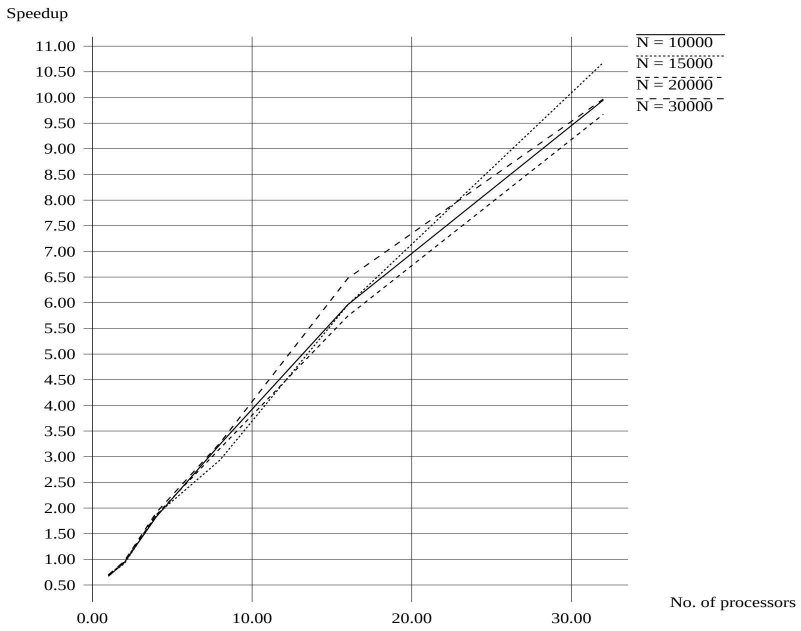
4.9 Improvements with 1.0 Constructs

The use of library code shows that the goal of encapsulating parallelism and synchronization in library classes is not a complete pipe dream. The shared address space aspect of pSather's cluster model allows the programmers to write their programs in a relatively clear and succinct manner; the distributed memory aspect of the model motivates programmers to take into account data locality for efficiency.

Our use of pSather 0.1 has also uncovered some deficiencies of the language which will be overcome with 1.0. It is however encouraging to note that these deficiencies are in the sequential



a.



b.

Figure 4.64: Speedups of adaptive fast-multipole N-body program w.r.t. pSather/V3 (non-uniform line input): (a) $\leq$ 100 points per leaf (on average), (b) $\leq$ 50 points per leaf (on average).

No. of processors	$N = 10000$	$N = 15000$	$N = 20000$	$N = 30000$
1	151.76	229.81	316.99	482.36
2	78.04	122.31	164.96	239.34
4	45.31	64.91	88.71	127.33
8	26.87	46.15	51.68	81.76
16	14.46	26.62	30.18	45.17
32	10.18	18.63	18.23	27.54

(a) Grain size is ≤ 100 points per region (on average).

No. of processors	$N = 10000$	$N = 15000$	$N = 20000$	$N = 30000$
1	181.75	294.14	377.33	562.75
2	92.87	147.39	186.31	283.92
4	51.44	79.63	98.91	160.60
8	31.38	50.53	65.10	96.36
16	16.37	27.17	43.11	56.39
32	10.43	15.82	20.64	31.49

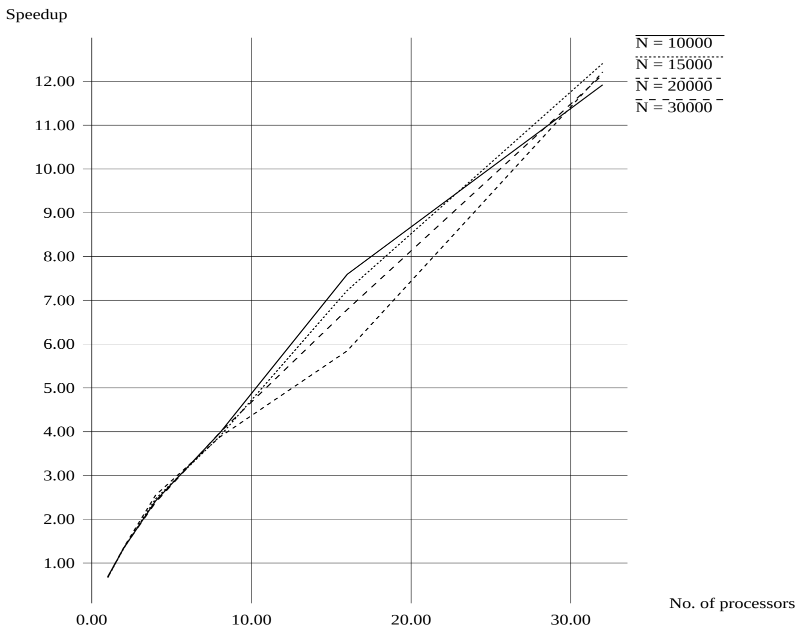
(a) Grain size is ≤ 50 points per region (on average).

Table 4.20: Computation time (in seconds) of fast multipole N-body program for non-uniform normal-distribution input, N = number of points.

	$N = 10000$	$N = 15000$	$N = 20000$	$N = 30000$
≤ 100 points/leaf	106.32	165.57	222.50	332.88
≤ 50 points/leaf	124.33	196.30	252.07	382.77

Table 4.21: Sequential computation time (in seconds) of adaptive fast multipole N-body program (pSather/V3) for non-uniform normal-distribution input, N = number of points.

a.



b.

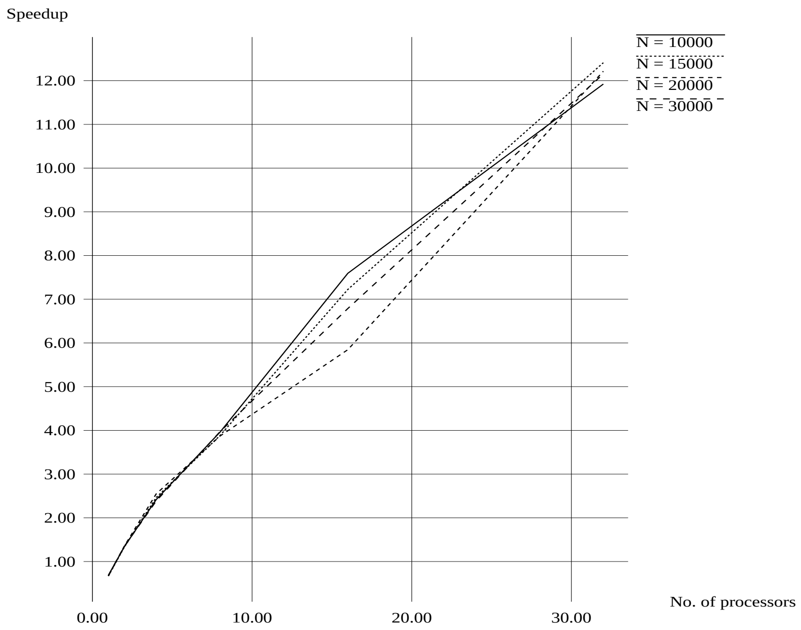


Figure 4.65: Speedups of adaptive fast-multipole N-body program w.r.t. pSather/V3 (non-uniform normal-distribution input): (a) $\leq$ 100 points per leaf (on average), (b) $\leq$ 50 points per leaf (on average).

language and not in the parallel constructs. We examine how some of the earlier code can be improved in 1.0.

4.9.1 Workbag Revisited

A Single Class for Different Strategies

Section 4.1.5 describes how to construct workbags with different strategies by altering some routine calls in the workbag implementation (shown in Figures 4.6– 4.8). The problem is that we then get different classes with almost similar code. We would achieve more code reuse if these routine calls can be treated as "variables".

In 1.0, we can generalize the `DISTRIB_BAG{T}` implementation by making the routine calls that insert/retrieve elements from the sub-bags be bound routines. The class has three additional attributes:

```
insert_local:ROUT{BAG{T},T};
get_local:ROUT{BAG{T}}:T;
get_remote:ROUT{BAG{T}}:T;
```

These attributes are initialized at object creation time, and allows workbags with different strategies to be constructed from the same class. We redefine the distributed workbag's `insert` routine (defined earlier in Figure 4.7) to show how the `insert_local` bound routine can be used.

```
insert(e:T) is
  local_subbag:BAG{T};
  with self, local_subbag near
    local_subbag := [local_id];
    l:GATE0 := locks[local_id];
    -- Protect the insertion of local sub-bag using gate.
    lock l then
      insert_local.call(local_subbag, e)
  end; end;
end; -- insert
```

The `get_local` and `get_remote` bound routines can be similarly used in the workbag's `get` routine. Using this strategy, we can create a mixed strategy workbag (Section 4.2.1) as follows:

```
#DISTRIB_BAG{T}(insert_local:=#ROUT(_.BAG{T}.insert_back(_)),
  get_local:=#ROUT(_.BAG{T}.get_back(_)),
  get_remote:=#ROUT(_.BAG{T}.get_front(_)))
```

(This creates a workbag which works similarly to the code shown in Figures 4.6– 4.8.)

Master-Worker Class

A common code skeleton that has been used repeatedly is that of a worker thread using a workbag (Figure 4.5):

```
1 :   exec_workers(compute:ITER{T}) is
2 :      i:INT; cobegin
3 :         until (i >= CONFIG::current_num_clusters) loop
4 :            :- worker(compute) @ i;
5 :            i := i+1;
6 :      end; end;
7 :   end;

8 :   worker(compute:ITER{T}) is
9 :      bag:DISTRIB_BAG{T} := workbag.dir_copy;
10:      loop
11:         w:T := bag.get;
12:         if (w /= void) then
13:            loop
14:               new_work:T := compute.call!(w);
15:               bag.insert(new_work);
16:            end;
17:         else
18:            bag.signal_quiescent;
19:            if (bag.is_quiescent) then break end;
20:      end; end; end;
21:   end;
```

Figure 4.66: A master-worker class that further shields the user from the use of workbag.

```
1 :   loop
2 :      w:T := bag.get
3 :      if (w exists) then
4 :         <Compute with w, and insert
5 :         new work generated into the bag.>
6 :      else
7 :         bag.signal_quiescent;
8 :         if (bag.is_quiescent) then break end;
9 :   end; end; end;
```

We can provide further encapsulation using a master-worker class so that the user does not need to know about the forking of worker threads and the threads' placement. He/she only needs to write an iter definition that accepts a evaluated-once argument of type `T` and executes the code in lines 4–5. Every time a new work is to be inserted into the bag, the iter assigns the work to the `res` variable and executes a `yield` statement.

Lines 1–6 in Figure 4.66 shows the code that sets up the worker threads to use the workbag. Each worker thread makes a local copy of the directory (line 9) and looks for work from the workbag (line 11). If work is found (line 12), then we go into a loop that performs the actual computation (lines 13–16). Otherwise, we indicate that the current thread is inactive and is ready to terminate.

4.9.2 Replicated Hash Table Revisited

Section 4.8.2 has a subsection on software caches which describes how the N-body programs use a `REPL_OB_HASH_MAP{T}` class to map from `$OB` to `T`. This class is similar to another `REPL_INT_HASH_MAP{T}` class (described in Section 4.3). We can improve code reuse if, instead of building two different classes, we implement a more generic class `REPL_HASH_MAP{T1,T2}` that maps from `T1` to `T2`. This encapsulation will be more easily accomplished in 1.0 because 1.0 allows all classes to be subtypes of `$OB`.

```
class REPL_OB_HASH_MAP{T} is
  include REPL_HASH_MAP{$OB, T}; end;

class REPL_INT_HASH_MAP{T} is
  include REPL_HASH_MAP{INT, T}
    get_cached_ob(T) ->; end;
```

We note that `REPL_INT_HASH_MAP{T}` has to remove the `get_cached_ob` routine from its implementation, because the routine assumes that the key `T2` is a subtype of `T1`, (but `T` in `REPL_INT_HASH_MAP{T}` can never be a subtype of `INT`).

The type parameter of the replicated hash table classes must have a `make_copy_to` routine in its interface. This routine, however, is not part of the "natural" interface of many classes. In 1.0, it would be better to include the copying routine as one of the parameters of the hash table routines. In `REPL_OB_HASH_MAP{T}`, instead of:

```
get_wait(k:$OB):T
get_cached(k:$OB):T
get_cached_ob(k:$OB):T
```

we get the following:

```
get_wait(k:$OB; make_copy:ROUT{T}:T):T
get_cached(k:$OB; make_copy:ROUT{T}:T):T
get_cached_ob(k:$OB; make_copy:ROUT{T}:T):T
```

The bound routines for `make_copy` may be defined in other classes; `T` is no longer required to include a possibly "unnatural" interface routine.

4.9.3 Improving Fast Multipole N-Body Programs with 1.0

The implementations of the N-body algorithms show how several language features in pSather 1.0 will allow cleaner encapsulation. The first is user-defined value classes. By implementing `COMPLEX` as a value class, `POLYNOMIAL{COMPLEX}` automatically assumes the compact representation (Figure 4.67) that programmers would use if writing the code in C. The other benefit of having `COMPLEX` as a value class is that it eliminates a pointer indirection and reduces the amount of memory used.

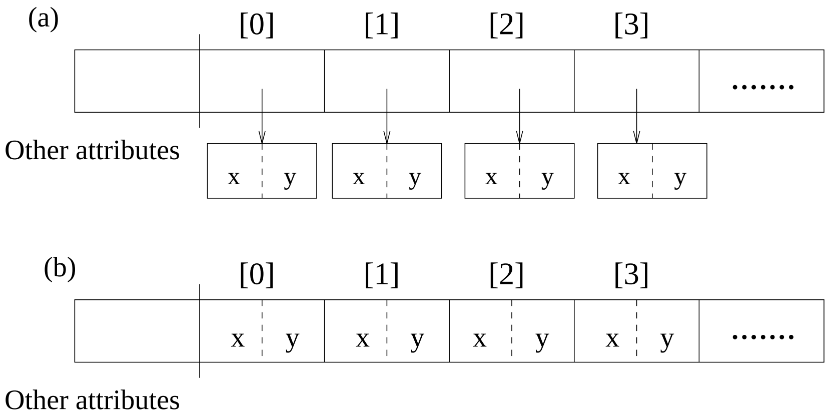


Figure 4.67: Representation of POLYNOMIAL{COMPLEX} when (a) COMPLEX is a reference class, and when (b) COMPLEX is a value class.

The next useful feature is the syntactic sugar for operations such as `plus`, ... (Appendix B). In 0.1, when adding the ith coefficients of two polynomials, we have to write:

```
res[i] := [i].plus(p[i]);
```

This means that in order for INT to be used as a type parameter of POLYNOMIAL{T}, it has to include routines like `plus` (which simply executes the actual `+`). In 1.0, we are able to write "`res[i] := [i] + p[i];`" which is clearer and more succint and.

Both MULTIPOLE_EXPANSIONS and LOCAL_EXPANSIONS contain the routine `make_copy_to`, which is not in the natural interface of the expansions. This routine is included because these two classes are used to construct the caches whose types are REPL_OB_HASH_MAP{MULTIPOLE_EXPANSION} and REPL_OB_HASH_MAP{LOCAL_EXPANSION}, and REPL_OB_HASH_MAP{T} requires its parameter T to define a `make_copy_to` routine. In 1.0, with bound routines, this restriction on T can be removed (Section 4.9.2).

Chapter 5

Future Directions and Conclusions

This chapter first summarizes the experience and results for the current pSather prototype (Section 5.1). We discuss what we have achieved and also the shortcomings in our work. The shortcomings lead us to think about several possible research directions for pSather (Section 5.2).

5.1 Conclusions

The previous chapters have gone through the design process of pSather, its implementation on a distributed-memory and our programming experience using pSather. One of the main goals in the design is that the language be suitable for future multiprocessors which will have distributed memory. Thus, pSather supports a distributed-memory shared-address model and has constructs that allow programmers to take data locality into account. Another characteristic of pSather is its combination of various parallel paradigms in one consistent framework. This will be useful for supporting research on multiple parallelizations within an algorithm [79].

The current protoype supports all the parallel constructs. Our implementation demonstrates that even if a distributed-memory machine does not support a shared-address space at the hardware/operating system level, the compiler can still provide the shared-address abstraction relatively well. It also suggests that pSather's model can also be realized on networks of workstations if the network latencies are ≤ 2 orders of magnitude slower than the fastest local access. (e.g. [214]).

Although active messages were developed as support for dataflow languages, we find that they are also useful as part of the runtime support for pSather, a high-level object-oriented language with a completely different paradigm. Our experience also suggests that when selecting a low-level message library for language runtime, it would be more efficient to use an interrupt-driven library (as opposed to one based on polling).

Our prototype has been used by other researchers to implement other non-trivial applications (e.g. a multiprocessor architecture simulator using the time-warp mechanism [145]). The main shortcoming of the system is the absence of debugging support and performance monitoring; we briefly discuss these possible areas of future research in Section 5.2.

We have implemented a few abstractions which turn out to be reusable by applications with very different characteristics (Chapter 4). These are preliminary examples of how software can be reused in both small and medium-sized parallel applications. The performance of our applications shows reasonable speedups even when compared to similar C programs executing on a CM-5 node. The pSather programs on CM-5 have been able to make absolute performance gains over comparable sequential processors. We think that pSather will serve as a practical and efficient platform for experimenting with new parallel algorithms and data structures.

5.2 Possible Research Directions

This section discusses possible directions of future research on pSather. In terms of language design, although there are open questions of how exception handling works in pSather and how the language might support vector code generation, we feel that the current design of pSather 1.0 is reasonably expressive and useful. Therefore we will focus our discussion on other aspects of implementation — performance monitoring (Section 5.2.1), debugging (Section 5.2.2), garbage collection (Section 5.2.3), improving compilation techniques (Section 5.2.4), and issues in the portability of the implementation (Section 5.2.5). We also need more pSather applications to further demonstrate its usefulness and/or expose its limitations. Another aspect of further research is to design and implement more parallel algorithms; Section 5.2.6 discusses possible candidates.

5.2.1 Performance Monitoring

There are two aspects in performance monitoring (or performance debugging). The first is to gather information about the performance of the underlying language implementation (e.g. average costs of synchronization operations), so that the implementer can improve its efficiency. The second is to collect data about the performance characteristics of the user programs (e.g. number and total overhead costs of remote calls). In the latter case, although low-level information may be useful to programmers who are familiar with the guts of the system, a general user prefers to have the data presented in a high-level manner. So the lower-level performance characteristics should be related to the contexts of the program (e.g. which part of the program is generating a large number of remote calls).

There are various ongoing research efforts in performance debugging. [65] has a summary of various software performance monitoring tools for multiprocessors. For example, Pablo [197, 185] is a high-level instrumentation system for sequential and (distributed-memory) multiprocessors; it requires the programmer to insert trace calls in the program. In Prism [77], the instrumenting code is automatically generated by the tool.

Although multiprocessor vendors normally provide a performance tool for their machine, much research still needs to be done. It is important to keep the instrumentation overheads low, so that the measurements reflect the actual behavior of an un-monitored parallel program. This is because a parallel program's synchronization pattern and/or overheads may depend on the relative execution speeds of different threads; adding instrumentation code may perturb the relative execution speeds so much that the measurements do not reflect the original execution. In shared-memory multiprocessors, the memory access latency (on a cache miss) may make up a high proportion of a parallel program's execution time. Thus, in addition to compute time, parallel programs need other statistics e.g. synchronization and memory overheads. While there are research efforts that addresses such issues (reducing the monitoring overheads and measuring the memory overheads) in shared-memory multiprocessors (e.g. [108]), the research in examining such problems for distributed-memory machine remains even more open. Furthermore, for languages which support irregular and dynamic structures (e.g. pSather), a performance tool should not only show program behavior in terms of program execution, but also provide information about the distribution of objects and their access pattern (e.g. showing hot-spots). To reduce learning effort, users would like to have a consistent model of their tools across different architectures. So another question is whether there is a core set of metrics which apply for different architectures and/or programming models.

While many research efforts on performance monitoring concentrate on software tools, another way to find out about the behavior/performance of parallel programs is by simulation. While simulation can be slow, it has the virtue of being arbitrarily precise and repeatable. If a simulator can be configured to simulate a wide range of architectures, the programmer also benefits from being able to examine his/her program's performance on different machines (which in turn indicates the program portability). Examples of multiprocessor simulation systems include PROTEUS [46] and Tango [112]. The systems allow the user to record various low-level events (e.g. lock acquisition/release), but does not support the mapping of such information to high-level program components. It would be interesting to build a high-level performance debugging tool based on a precise, repeatable simulation system which would allow the programmer to measure statistics on different models of multiprocessors (including distributed-memory).

5.2.2 Debugging

Another programming tool closely related to performance monitoring, is program debugging. There are two ways to look their possible link. First, many common implementation mechanisms may appear in both tools, e.g. fast data break points [150]. Another link is that they are both trying to help the programmer visualize program behavior. The difference is in how the knowledge of program behavior is used; in a performance monitoring tool, it is used to improve program performance while in a debugger, it is to find errors in program semantics. Although the exact kinds of information may be different, there are strong similarities in their user models. For example, setting breakpoints may be to measure the frequency of routine calls (performance monitoring), or to check whether the calls satisfy a certain predicate (debugging). This idea of providing a tool that observes the program behavior is not new, e.g. Parasight [16] was a research effort that aims to support "programming for observability". Because performance monitoring and debugging share common aims ("provide information on program behavior"), they also share common research issues such as parallel program visualization [177] (or even auralization).

In addition to software tools, other common approaches in parallel debugging is by static analysis (e.g. [89]), or a combination of logging information during execution with post-mortem analysis (e.g. [69]).

One dimension of the debugging problem is the methodology (e.g. software tracing vs. static analysis). Another dimension is the nature of the bugs. In a parallel object-oriented language like pSather, the nature of program errors (e.g. data races, deadlock detection) becomes more difficult to pinpoint because the "shared variables" are no longer static, and are stored as attributes in dynamically created objects whose links may change during program execution.

5.2.3 Garbage Collection

Garbage collection (GC) is a important research area that is not addressed in the current pSather prototype (although Sather 0.2 does use a conservative garbage collector developed by Boehm and Weiser [41]). The compiler analysis used to identify short-lived objects (Section 3.5.4) is insufficient for many application programs; hence the importance of a garbage collector so that the user can focus on the high-level program/algorithm design, instead of the implementation details such as identifying unreachable objects.

A garbage collector for pSather on a distributed-memory machine like CM-5 should neither use centralized control or stop-the-world synchronization. A stop-the-world collector can easily cause load imbalance due to different amounts of garbage in different clusters. Thus, the GC scheme should probably consist of a number of asynchronous local collectors which *incrementally* reclaim local garbage while user threads (or mutators) continue to execute on other processors. In the cluster machine model, collectors and mutators may execute in parallel on processors in the same cluster.

The collector should also be able to reclaim unreachable cycles/graphs of objects which are distributed among different clusters. Furthermore, on a distributed-memory machine, to keep the garbage collection overheads low, the communication and synchronization costs must be kept low. So a local collector on processor p cannot afford to spend too much time communicating with other processors (e.g. collecting references to objects in p from other clusters). Instead the local collectors might conceivably collect/exchange GC information when the runtime system is idle (e.g. no local threads to be scheduled, waiting for acknowledgements etc.), or piggyback GC information on other system messages. Whatever the case, the GC scheme probably has to be tightly integrated with the runtime support system for good performance.

[155, 172] are examples of research efforts on design and implementation of distributed GC.

5.2.4 Better Compilation Strategies

There are two issues in the compiler's intermediate representation (IR) which we have not examined so far. The first is in studying IR's that would map to more efficient code, while the second is the portability of the IR. We discuss the second issue in the section on portability (Section 5.2.5). Here we would like to suggest that some variation of continuation-passing may be relevant to a better IR.

In the description of pSather's implementation on CM-5 (Section 3.3), we explained how *continuation states* are used to implement remote routine calls, thread creation and gate operations. But these techniques are employed in the runtime but not in the compiler. The essential idea is that a continuation state contains all the information necessary for the next stage of execution E (e.g. execution of routine on remote cluster). The continuation after E may be in the continuation state (and hence determined at execution time), or statically fixed in E's exit code.

Continuation-passing is a well-known compilation technique [15], but to our knowledge, it has not been extensively applied in compiling parallel object-oriented languages. We feel that it may be worthwhile to explore the use of continuation-passing techniques in compiling a parallel object-oriented language such as pSather. While the interaction of threads within a cluster may be handled by conventional procedure calls/thread creation, continuation-passing may be a useful mechanism for thread interactions between clusters. For example, in a call such as:

```
foo(a1@c1,a2@c2,a3@c3);
```

a smart compiler should represent the execution as:

```
r1           is  r2(a1)          @c2   end;
r2(x1)       is  r3(x1, a2)      @c3   end;
r3(x1, x2)   is  r4(x1, x2, a3)  @orig_cluster  end;
r4(x1, x2, x3) is foo(x1, x2, x3) end;
```

The original `foo` call is replaced by `r1@c1`. This will result in four messages (to `c1`,`c2`,`c3` and back to the original cluster), instead of the current six messages (to and back from `c1`,`c2`,`c3`). The number of context-switches is also reduced. The sizes of the four messages however may be larger than the sizes of six messages, so there are some tradeoffs here.

5.2.5 Portability

There are two different aspects to an implementation's portability – compiler and runtime system.

Compiler

To reduce porting efforts, it is important to explore compilation strategies that allow the compiler to generate the same intermediate form for different architectures. The intermediate representation must still lead to relatively efficient code. Using our current strategy of using C as the intermediate language, the main difference between different multiprocessors which the compiler must deal with appears to be the dichotomy between machines that support a shared address space and those that do not.

The structure of an address space mainly affects the memory operations (i.e. reads/writes of variables). Therefore it would seem that if the compiler can categorize the kinds of variables in a language and identify the specific kinds of read/write operations in a program, then a certain amount of portability is achievable. For example, the CM-5 compiler distinguishes between writing a local variable and writing an object attribute. The compiler generates different code templates (in the form of C macros) for these two different write operations. Porting simply entails re-mapping the code templates to the more primitive operations (e.g. message send/receive or memory store).

The above briefly addresses the issue of support a shared address space on different architectures. What seems to be more problematic is how pSather's shared-address model affects its runtime implementation. One example is the implementation of gate operations on CM-5. A straight-forward strategy that simply maps a shared-memory implementation would result in disastrous performance, because during a remote gate operation (e.g. `read`), every access to the gate's internal attribute will require a message send/receive. An alternative approach is to implement remote gate operations on top of remote routine calls. While the gate operations may become more portable (due to the portability of remote calls/thread module), they can remain inefficient. We currently use continuation states to implement gate operations (Section 3.3.4), but this is not portable when we want to move to a multiprocessor that supports shared-address space. A portable approach to implement gates efficiently might be a two-pronged one as follows. First, we implement the `GATE` classes in pSather by building upon runtime operations (which are encapsulated in primitive pSather classes). Second we develop a general compilation strategy (e.g. one that uses some variation of continuation-passing technique) such that the compiler's intermediate representation is more general/useful than the current abstract syntax tree representation. This compiler will then automatically generate the appropriate codes for gate operations on shared- and distributed-memory machines.

Runtime Support

The other aspect of portability is pSather's runtime of which the threads and remote operations appear to be the most system-dependent. To ensure portability, one approach is for pSather to implement its user-level threads using a common standard (e.g. POSIX threads [143, 179]) and to use the common message library (e.g. PVM [28]) for it runtime messages.

However a POSIX thread may still be too expensive (e.g. to implement remote calls). And our prototype suggests that a light-weight, low-latency active message package is more suitable than message libraries such as PVM.

An indirect approach would be to build an execution model which uses the functionalities of a POSIX thread package, plus an active message library. Such a model might map several user-level threads to a single POSIX thread, similar to the way we have reused the same runtime thread objects for non-suspendable remote calls (Section 3.3.2). We would also need to identify the primitive runtime operations which can be implemented efficiently on different platforms. PSather-level operations (e.g. remote calls) would be built on top of these primitives; the code for pSather operations can be either inline-generated by the compiler, or implemented in the runtime.

5.2.6 More Applications

Although we have implemented several small and medium-sized pSather programs, they are obviously not an exhaustive proof of pSather's usefulness and/or limitations. It would be interesting to build pSather applications which are larger and even more dynamic. There are several reasons for more application programs: to study the actual behavior of algorithms (e.g. load imbalance in irregular and dynamic problems), to further try out the expressiveness of pSather language, and to identify performance bottlenecks in the runtime system. We think that interesting applications include algorithms for the max-flow problem [66, 110], various kinds of simulators (e.g. ICSIM [205] a neural network simulator), numerical algorithms (e.g. linear programming), and parallel constraint satisfaction systems ([117]).

We have mentioned that one of pSather's goals is to provide a platform for building reusable classes. In Chapter 4, we have discussed some abstractions, but there are other data structures (e.g. parallel priority queues [193]) which we did not look at. To further develop the vision of a programming environment with a well-designed parallel class library, much work remains to be done to examine the implementation (e.g. encapsulation and specification) and theory (e.g. scalability, performance model) of parallel and distributed data structures.

Appendix A

Presentation Convention

To ensure a consistent presentation, we adopt the following conventions.

- When we refer to a piece of code in a sentence, the code will be in typewriter font and enclosed in double-quotes, e.g. "`x := y;`".

- When a piece of code is displayed separately from any paragraph, the code will be in typewriter font but will not be enclosed in double-quotes. Reserved words will be in bold, as in the following:

```
loop
  x := y;
end
```

- Class names will be in typewriter font but not enclosed in any quotes, (e.g. `ARRAY{INT}`, or `ARRAY`). Similarly for names of variables (e.g. `local_var`), parameters, keywords of languages and other short expressions.

- Italics are reserved for emphasis or for highlighting new unfamiliar terms.

- A language construct is described using the typewriter font and displayed separately from any paragraph. For example, the syntax for `if`-statement in Sather 1.0 is as follows.

```
if <expr> then
  <statement-list-1>
[else <statement-list-2>]
end;
```

  The square brackets denote an optional part of the statement.

- We use the notation `func(INT,ARRAY{INT})` to refer to a function `func` which has two parameters of type `INT` and `ARRAY{INT}`.

- We use the san serif font to refer to generic syntactic and runtime entities, (e.g. using $\mathsf{T0}$ to refer to a user-level thread, and O_1 for object).

We keep the distinction among "users", "library designers" and "programmers". But we ignore this distinction when "user" is used in other contexts (e.g. a "user-defined" class may be written by a "user" or "library designer").

Appendix B

Syntactic Sugar in p/Sather 1.0

Sugar form	*Translation*
expr1 + *expr2*	*expr1*.plus(*expr2*)
expr1 - *expr2*	*expr1*.minus(*expr2*)
expr1 * *expr2*	*expr1*.times(*expr2*)
expr1 / *expr2*	*expr1*.div(*expr2*)
expr1 ^ *expr2*	*expr1*.pow(*expr2*)
expr1 % *expr2*	*expr1*.mod(*expr2*)
expr1 /= *expr2*	not (*expr1*=*expr2*)
expr1 < *expr2*	*expr1*.is_lt(*expr2*)
expr1 <= *expr2*	*expr1*.is_leq(*expr2*)
expr1 > *expr2*	*expr1*.is_gt(*expr2*)
expr1 >= *expr2*	*expr1*.is_geq(*expr2*)
- *expr*	*expr*.negate
[*expr_list*]	aget(*expr_list*)
expr1[*expr_list*]	*expr1*.aget(*expr_list*)
(*expr*)	*expr*

Table B.1: Syntactic sugar expressions and their translations.

Table B.1 shows the sugar expressions in Sather 1.0 [189].

Appendix C

Sequential SOR Programs

```
for (i = 1; i < s; i++) {
  for (j = 1; j < s; j++) {
    matrix[i*ncols+j] = (omega1 * matrix[i*ncols+j]) +
      omega * (matrix[i*ncols+(j-1)] + matrix[i*ncols+(j+1)] +
            matrix[(i-1)*ncols+j] + matrix[(i+1)*ncols+j]);
  }
}
```

Figure C.1: C code in SOR (which updates the variables in each iteration), used in speedup measurements.

```
with arr near
  i:INT := 1;
  until (i >= size_minus_1) loop
    j:INT := 1;
    until (j >= size_minus_1) loop
      arr[i,j] := (omega1 * arr[i,j])
          + omega * (arr[i,j-1] + arr[i,j+1]
          + arr[i-1,j] + arr[i+1,j]);
      j := j+1;
    end;
    i := i+1;
  end;
end;
```

Figure C.2: PSather code for a sequential SOR program, used for timing comparisons.

Appendix D

A Support Class for Replicated Hash Table

We implement the PROTECTED_INT_HASH_MAP{T} class (Figure D.1– D.2) using a serial INT_HASH_MAP{T} class written by Steve Omohundro.

```
class PROTECTED_INT_HASH_MAP{T} is
   -- Hash tables which associate elements of type 'T' to
   -- non-negative 'INT' keys.  The table is always at least
   -- twice as large as number of ints (plus the number of
   -- deletes since last expansion).  The key '-1' is used
   -- for empty entries, '-2' for deleted ones.  The size
   -- of the table is always one greater than a power of two
   -- and nearly doubles on each expansion.  The last entry
   -- is a sentinel element which always holds '-1' (to avoid
   -- testing for end of array in loop).  The hash function
   -- is just the last 'n' bits of the 'INT' key.  The extra
   -- serial search this simple function sometimes entails
   -- is still cheaper than more typical functions which use a
   -- multiply and a modulo operation.

   INT_HASH_MAP{T};
   gate:GATE0; -- Protected by a gate object.

   alias old_create create;
   alias old_clear clear;
   alias old_get get;
   alias old_insert insert;
   alias old_size size;
```

Figure D.1: Part 1 of definition of PROTECTED_INT_HASH_MAP{T} implemented in current prototype.

```
create:SELF_TYPE is
  -- An empty hash table.
  res := old_create;
  res.gate := GATE0::new;
end; -- create

clear is
  -- Clear the tables.
  lock gate then old_clear end;
end; -- clear

get(k:INT):T is
  -- The entry associated with key `k' or `void' if absent.
  lock gate then res := old_get(k) end;
end; -- get

insert(k:INT; e:T) is
  -- Insert key `k' with entity `e'.  Overwrite if present.
  lock gate then old_insert(k,e) end;
end; -- insert

size:INT is
  -- Number of entries stored, (not fast).
  lock gate then res := old_size end;
end; -- size
```

Figure D.2: Part 2 of definition of PROTECTED_INT_HASH_MAP{T} implemented in current prototype.

Appendix E

Miscellaneous Classes in Fast Multipole N-Body Programs

```
class REPL_OB_HASH_MAP{T} is
  -- This class associates entities of type 'T' to keys of type
  -- '$OB' in a distributed memory machine.  'T' has routine
  -- 'make_copy_to(cid:INT):T' in its interface.
  -- The hash table is replicated on every cluster.

  SPREAD{PROTECTED_OB_HASH_MAP_HEADER{T}};

  constant local_cid:INT := CONFIG::current_cluster_id;

  private local_set:PROTECTED_OB_HASH_MAP{T} is
    -- Return the local cache table.
    res := [local_cid].local_set;
  end; -- local_set

  private shared_set:OB_HASH_MAP{T} is
    -- Return the shared table.
    res := [local_cid].shared_set;
  end; -- shared_set
```

Figure E.1: Part 1 of definition of REPL_OB_HASH_MAP{T} implemented in current prototype.

Figures E.1 – E.5 describe a replicated hash table class used in the N-body programs.

```
private shared_set_gate:GATE0 is
  -- Return the gate for shared table.
  res := [local_cid].gate;
end; -- shared_set_gate

private waiting:OB_HASH_MAP{GATE0} is
  -- Return the hash table for waiting threads.
  res := [local_cid].wait_gates;
end; -- waiting

create:SELF_TYPE is
  -- Allocate an empty replicated hash table.
  res := new;
  num_clusters:INT := CONFIG::current_num_clusters;
  s:OB_HASH_MAP{T} := OB_HASH_MAP{T}::create;
  gs:OB_HASH_MAP{GATE0} :=
    OB_HASH_MAP{GATE0}::create;
  g:GATE0 := GATE0::new;
  i:INT;
  until (i >= num_clusters) loop
    -- Allocate a chunk header with the corresponding gate
    -- object and local table.
    res[i] := PROTECTED_OB_HASH_MAP_HEADER{T}::create(s,g,gs) @ i;
    i := i+1;
  end; -- loop
end; -- create

get_cached_ob(k:T):T is
  res := k;
  if (k.is_far) then
    s:PROTECTED_OB_HASH_MAP{T} := local_set;
    res := s.get(k);
    if (res = void) then
      res := k.make_copy_to(local_cid);
      s.insert(k, res);
  end; end;
end; -- get_cached_ob
```

Figure E.2: Part 2 of definition of REPL_OB_HASH_MAP{T} implemented in current prototype.

```
get_wait(k:$OB):T is
  -- Get the entry associated with key 'k'; current thread suspends
  -- if entry is absent.

  s:PROTECTED_OB_HASH_MAP{T} := local_set;
  with s near
    res := s.get(k);
    if (res = void) then
      g:GATE0 := shared_set_gate;
      -- Then wait on any insertion into the shared table.
      lock g then
        res := shared_set.get(k) @ shared_set.where;
        if (res /= void) then
          -- If we find something, then the value is cached.
          unlock g;
          if (res.is_far) then
            res := res.make_copy_to(local_cid);
            s.insert(k, res);
          end; -- if
        else
          wait_gate:GATE0 :=
            waiting.get(k) @ waiting.where;
          if (wait_gate = void) then
            wait_gate := GATE0::new;
            waiting.insert(k, wait_gate) @ waiting.where;
          end; -- if
          unlock g;
          wait_gate.read;
          -- Wait until it is set.
          lock g then
            res := shared_set.get(k) @ shared_set.where;
            if (res.is_far) then
              res := res.make_copy_to(local_cid);
              s.insert(k, res);
            end; end; end;
      end; end; end;
end; -- get_wait
```

Figure E.3: Part 3 of definition of REPL_OB_HASH_MAP{T} implemented in current prototype.

```
get_cached(k:$OB): T is
  -- The entry associated with key 'k' or 'void' if absent.
  s:PROTECTED_OB_HASH_MAP{T} := local_set;
  with s near
    res := s.get(k);
    if (res = void) then
      -- Then check the shared (up-to-date) table.
      lock shared_set_gate then
        res := shared_set.get(k) @ shared_set.where;
      end; -- lock
      if (res /= void) and (res.is_far) then
        -- If we find something, then the value is cached.
        res := res.make_copy_to(local_cid);
        s.insert(k, res);
      end; end; end;
end; -- get_cached

get(k:$OB): T is
  -- The entry associated with key 'k' or 'void' if absent.
  -- Unlike "get_cached", this does not have the side effect of
  -- copying objects.
  lock shared_set_gate then
    res := shared_set.get(k) @ shared_set.where;
  end; -- lock
end; -- get

insert(k:$OB; e:T) is
  -- Insert key 'k' with entity 'e'.  Ignore if present.
  s:PROTECTED_OB_HASH_MAP{T} := local_set;
  with s near
    s.insert(k, e);
    lock shared_set_gate then
      wait_gate:GATE0 := insert_help(k, e) @ shared_set.where;
      if (wait_gate /= void) then
        wait_gate.set end;
    end; end;
end; -- insert

private insert_help(k:$OB; e:T):GATE0 is
  shared_set.insert(k, e);
  -- "shared_set" is located on the same cluster as the hash table
  -- of waiting-gates.
  res := waiting.get(k);
end; -- insert_help

end; -- class REPL_OB_HASH_MAP{T}
```

Figure E.4: Part 4 of definition of REPL_OB_HASH_MAP{T} implemented in current prototype.

```
class PROTECTED_OB_HASH_MAP_HEADER{T} is
  shared_set:OB_HASH_MAP{T};
  -- Shared by all other replicated local sets.

  gate:GATE0;
  -- For the shared set, the gate is dissociated from the table.
  -- This is because we want each thread to be able to call the
  -- lock-statement from the thread's own cluster.  If the gate
  -- and table are combined, and the thread's locus of control is
  -- shifted, the remote call might never be scheduled.  (On the
  -- other hand, the semantics of gate operations ensure a certain
  -- amount of fairness.)

  wait_gates:OB_HASH_MAP{GATE0};
  -- For each key, associate a gate on which the threads are waiting.

  local_set:PROTECTED_OB_HASH_MAP{T};

  create(s:OB_HASH_MAP{T}; g:GATE0;
        gs:OB_HASH_MAP{GATE0}):SELF_TYPE is
    res := new;
    res.shared_set := s;
    res.gate := g;
    res.wait_gates := gs;
    res.local_set := PROTECTED_OB_HASH_MAP{T}::create;
  end; -- create

end; -- class PROTECTED_OB_HASH_MAP_HEADER{T}
```

Figure E.5: Definition of a header class used in REPL_OB_HASH_MAP{T}.

Bibliography

[1] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computation. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, June 1989.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge Massachusetts, 1986.

[3] Gul Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[4] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming.* The MIT Press, Cambridge, Massachusetts, 1987.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[6] Eugene Albert, Joan D. Lukas, and Guy L. Steele Jr. Data Parallel Computers and the FORALL Statement. *Journal of Parallel and Distributed Computing*, 13:185–192, 1991.

[7] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *ECOOP 1987, European Conference on Object-Oriented Programming LNCS 276*, pages 234–242. Springer-Verlag, June 15-17 1987.

[8] Pierre America. *Issues in the Design of a Parallel Object-Oriented Language.* Philips Research Laboratories, Eindhoven and University of Amsterdam, March 1 1989. Part of POOL2/PTC Distribution Package.

[9] Pierre America. *Programmer's Guide for POOL2.* Philips Research Laboratories, Eindhoven and University of Amsterdam, January 10 1991. Part of POOL2/PTC Distribution Package.

[10] Pierre America and Ben Hulshof. *Definition of POOL2/PTC, a Parallel Object-Oriented Language.* Philips Research Laboratories, Eindhoven and University of Amsterdam, March 15 1991. Part of POOL2/PTC Distribution Package.

[11] J.P. Anderson, J.A. Hoffman, J. Shifman, and R.J. williams. D825 – A Multiple-Computer System for Command and Control. *AFIPS Conference Proceedings*, 22:86–96, 1962.

[12] Thomas E. Anderson. Fastthreads user's manual. FastThreads software package manual, January 1990.

[13] Andrew W. Wilson, Jr. and Richard P. LaRowe, Jr. Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture. *Journal of Parallel and Distributed Computing*, 15:351–367, 1992.

[14] Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[15] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[16] Ziya Aral, Ilya Gertner, and Greg Shaffer. Efficient Debugging Primitives for Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 87–93, April 1989.

[17] Arvind and R.S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990. Also available as: Computation Structures Group Memo 271, March 1987, Laboratory for Computer Science, Massachusetts Institute of Technology.

[18] Krste Asanović, James Beck, Tim Callahan, Jerry Feldman, Bertrand Irissou, Brian Kingsbury, Phil Kohn, John Lazzaro, Nelson Morgan, David Stoutamire, and John Wawrzynek. CNS-1 Architecture Specification. Technical Report TR-93-021, International Computer Science Institute, April 1993.

[19] Colin Atkinson et al. Object-oriented concurrency and distribution in dragoon. Technical Report Research Report DoC 89/3, Imperial College, June 1989.

[20] Henry G. Baker. Inlining Semantics for Subroutines which are Recursive. *ACM SIGPLAN Notices*, 27(12):39–46, December 1992.

[21] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: A language for distributed programming. *SIGPLAN Notices*, 25(5):17–24, 1990.

[22] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17:746–757, 1968.

[23] J.G.P. Barnes. An Overview of Ada. *Software – Practice and Experience*, 10(11):851–887, 1980.

[24] Paul S. Barth. *Atomic Data Structures for Parallel Computing.* PhD thesis, MIT, MIT Laboratory for Computer Science, Cambridge, MA 02139, March 1992. Also available as technical report MIT/LCS/TR-532.

[25] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference Proceedings, Lecture Notes in Computer Science 523*, pages 538–568, August 1991.

[26] Joel F. Bartlett. SCHEME → C a Portable Scheme-to-C Compiler. Technical Report 89/1, Digital Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA 94301, January 1989.

[27] Bob Beck. Shared-Memory Parallel Programming in C++. *IEEE Software*, pages 38–48, July 1990.

[28] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. A Users' Guide to PVM – Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[29] G. Bell, R. Cady, H. McFarland, B. Delagi, J. O'Laughlin, R. Noonan, and W. Wulf. A New Architecture for Mini-Computers: The DEC PDP-11. In Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, editors, *Computer Structures: Principles and Examples*, pages 649–665. McCraw-Hill, Inc., 1982.

[30] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice Hall International, Inc., 1982.

[31] John K. Bennett. The Design and Implementation of Distributed Smalltalk. In *OOPSLA '87 Conference Proceedings*, pages 318–330, October 4-8 1987. Proceedings also published as: SIGPLAN Notices, Vol 22, No 12, December 1987.

[32] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Shared Memory for Distributed Memory Multiprocessors. Technical Report TR89-91, Department of Computer Science, Rice University, P. O. Box 1892, Houston, Texas 77251-1892, April 1989.

[33] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software – Practice and Experience*, 18(8):713–732, August 1988.

[34] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science,Carnegie Mellon University, Pittsburgh, PA 15213, September 1991.

[35] Andrew D. Birrell. An Introduction to Programming with Threads. Technical Report SRC Research Report 35, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, January 1989.

[36] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[37] Christian H. Bischof and Jack J. Dongarra. A Linear Algebra Library for High-Performance Computers. In Graham F. Carey, editor, *Parallel Supercomputing: Methods, Algorithms and Applications*, pages 45–55. John Wiley & Sons Ltd., 1989.

[38] Roberto Bisiani and Mosur Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proc. 17th Annual International Symposium on Computer Architecture*, pages 115–124, June 1990.

[39] G.A. Blaauw and F.P. Brooks, Jr. The Structure of SYSTEM/360: Part I – Outline of the Logical Structure. In Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, editors, *Computer Structures: Principles and Examples*, pages 495–710. McCraw-Hill, Inc., 1982.

[40] Andrew Black et al. Object Structure in the Emerald System. In *OOPSLA '86 Conference Proceedings*, pages 78–86, September 29 - October 2 1986. Proceedings also available as: Sigplan Notices, Vol 21, No 11, November 1986.

[41] H. Boehm and Weiser M. Garbage collection in an uncooperative environment. Software Practice & Experience pp. 807-820, September 1988.

[42] A.P.W. Bohm and R.R. Oldehoeft. SISAL 2.0: Overview and Rationale. Technical Report CS-92-141, Department of Computer Science, Colorado State University, November 1992.

[43] Shahid H. Bokhari. Multiprocessing the sieve of eratosthenes. *IEEE Computer*, pages 50–58, April 1987.

[44] James Boyle et al. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., New York, NY, 1987.

[45] T. Brandes. Efficient Data Parallel Programming without Explicit Message Passing for Distributed Memory Multiprocessors. GMD Internal Report No. AHR-92-4, September 1992.

[46] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[47] Jean-Pierre Briot and Akinori Yonezawa. Inheritance and Synchronization in Concurrent OOP. In *ECOOP 1987, European Conference on Object-Oriented Programming, LNCS 276*, pages 32–40. Springer-Verlag, June 15-17 1987.

[48] B. Buchberger. *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal.* PhD thesis, Univ. Innsbruck, 1965. In German.

[49] B. Buchberger. Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. In N. K. Bose, editor, *Progress, Directions and Open Problems in Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Company, 1985.

[50] Peter A. Buhr, Michael H. Coffin, and Michel Fortier. Monitor Classification and Priority-NonBlocking Monitor. Part of μC++ Distribution Package.

[51] Peter A. Buhr and Richard A. Stroobosscher. *μC++ Annotated Reference Manual Version 3.4.4*, August 18 1992. Part of μC++ Distribution Package.

[52] Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, Vivek Sarkar, and David Shields. Automatic Discovery of Parallelism: A Tool and an Experiment (Extended Abstract). In *Proceedings of the ACM/SIGPLAN Parallel Programming Experience with Applications, Languages and Systems (PPEALS) Symposium*, pages 77–84, 1988. Proceedings also published as: SIGPLAN Notices, Vol. 23, No. 9, September 1988.

[53] David Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[54] Denis Caromel. Service, Asynchrony, and Wait-by-Necessity. *Journal of Object-Oriented Programming*, 2(4):12–18, November–December 1989.

[55] Denis Caromel. Concurrency: An Object-Oriented Approach. In *Proceedings of TOOLS 1990*, pages 183–198, June 1990.

[56] Denis Caromel. Concurrency and Reusability: From Sequential to Parallel. *Journal of Object-Oriented Programming*, 3(3):34–42, September–October 1990.

[57] Denis Caromel. *Programmation Parallele asynchrone et imperative: Etudes et Propositions.* PhD thesis, Universite de Nancy 1 (France), Informatique, February 1991.

[58] Denis Caromel. Programming Abstractions for Concurrent Programming: A Solution to the Explicit/Implicit Control Dilemma. *ACM OOPS Messenger*, 2(2), April 1991. Journal is collection of articles from Workshop on Object-Based Concurrent Programming, (OOPSLA/ECOOP 1990), edited by Gul Agha et al.

[59] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course.* The MIT Press, Cambridge, Massachusetts, 1990.

[60] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. Technical Report TR91-150, Department of Computer Science, Rice University, P. O. Box 1892, Houston, Texas 77251-1892, March 1991.

[61] Soumen Chakrabarti. A Distributed Memory Gröbner Basis Algorithm. Master's thesis, University of California at Berkeley, Berkeley, CA, 1992.

[62] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Integrating Concurrency and Data Abstraction in a Parallel Programming Language. Technical Report CSL-TR-92-511, Computer Systems Laboratory, Stanford University, February 1992.

[63] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, May 1989.

[64] Jeffrey S. Chase et al. The Amber System: Parallel Programming on a Network of Multiprocessors. Technical Report 89-04-01, Department of Computer Science, University of Washington, 1989.

[65] Doreen Y. Cheng. A Survey of Parallel Programming Languages and Tools. Technical Report RND-93-005, NAS Systems Development Branch, NAS Systems Division, NASA Ames Research Center, Mail Stop 258-6, Moffett Field, CA 94035-1000, March 1993.

[66] Joseph Cheryian, Torben Hagerup, and Kurt Melhorn. Can a Maximum Flow be Computed in o(nm) Time? In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, May 1990.

[67] Andrew A. Chien and William J. Dally. Concurrent aggregates (ca). In *Proceedings of the ACM SIGPLAN Conference on the Principles and Practice of Parallel Programming*, 1990.

[68] Andrew Andai Chien. *Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines*. PhD thesis, MIT, July 1990. Also available as: MIT Artificial Intelligence Laboratory, Technical Report 1248.

[69] Jong-Deok Choi, Barton P. Miller, and Robert H.B. Netzer. Techniques for Debugging Parallel Programs with Flowback Analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[70] Peter Christy. Software to Support Massively Parallel Computing on the MasPar MP-1. In *IEEE COMPCON Proceedings*, pages 29–33, 1990.

[71] W. D. Clinger. Foundations of Actor Semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, May 1981.

[72] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, Massachusetts, 1989.

[73] Jean-Francois Colin and Jean-Marc Geib. Eiffel Classes for Concurrent Programming. In *TOOLS '91, Technology of Object-Oriented Languages and Systems*, 4-8 March 1991.

[74] Communications of the ACM: Concurrent Object-Oriented Programming, September 1993.

[75] Thinking Machines Corporation. CMSSL for CM Fortran, Vols. I and II.

[76] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary.* Thinking Machines Corporation, Cambridge Massachusetts, October 1991.

[77] Thinking Machines Corporation. *Prism User's Guide.* Thinking Machines Corporation, Cambridge Massachusetts, December 1991.

[78] Lawrence A. Crowl. *Architectural adaptability in parallel programming.* PhD thesis, University of Rochester, May 1991.

[79] Lawrence A. Crowl, Mark E. Crovella, Thomas J. LeBlanc, and Michael L. Scott. Beyond Data Parallelism: The Advantages of Multiple Parallelizations in Combinatorial Search. Technical Report Technical Report 451, The University of Rochester, Computer Science Department, April 1993.

[80] W. Crowther, J. Goodhue, R. Gurwitz, R. Rettberg, and R. Thomas. The Butterfly (TM) Parallel Processor. *IEEE Computer Architecture Technical Committee Newsletter*, pages 18–45, September-December 1985.

[81] David Culler et al. LogP: Towards a Realistic Model of Parallel Computation. Technical Report UCB/CSD 92/713, Computer Science Division (EECS), University of California, Berkeley, CA 94720, 1992.

[82] Ole-Johan Dahl and Kristen Nygaard. Simula – an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

[83] Jack W. Davidson and Anne M. Holler. A Study of a C Function Inliner. *Software – Practice and Experience*, 18(8):775–790, August 1988.

[84] Jack W. Davidson and Anne M. Holler. Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Transactions on Software Engineering*, 18(2):89–102, February 1992.

[85] D. Decouchant et al. A synchronization mechanism for typed objects in a distributed system. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 105–107, September 26-27 1988. Workshop proceedings available as: SIGPLAN Notices, Vol 24, No 4, April 1989.

[86] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[87] Keith Diefendorff and Michael Allen. Orgnization of the Motorola 88110 Superscalar RISC Microprocessor. *IEEE Micro*, pages 40–63, April 1992.

[88] E. W. Dijkstra. The structure of the 'THE' multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[89] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, 1991. Proceedings also available as SIGPLAN Notices 26(12), December, 1991.

[90] Thomas W. Doeppner, Jr. and Alan J. Gebele. C++ on a parallel machine. Technical Report CS-87-26, Brown University, Department of Computer Science, Brown University, Providence, RI 02912, November 17 1987.

[91] Jack J. Dongarra, Jeremy du Croz, Sven Hammarling, and Iain Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathmatical Software*, 16(1):1–17, March 1990.

[92] A. Dubruille, R.G. Scarborough, and H. Kolsky. How to write good vectorizable Fortran. Technical Report G320-3478, Palo Alto Scientific Center, 1985.

[93] Anant Agarwal et al. Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[94] John E. Faust and Henry M. Levy. The Performance of an Object-Oriented Threads Package. In *ECOOP/OOPSLA Proceedings*, pages 278–288, October 1990. Proceedings available as: SIGPLAN Notices, Vol 25, No 10, October 1990.

[95] Michael J. Feeley and Henry M. Levy. Distributed Shared Memory with Versioned Objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 247–263, Oct 18 - Oct 22 1992.

[96] Jerome A. Feldman. High level programming for distributed computing. *Communications of the ACM*, 22(6):353–368, June 1979.

[97] Jerome A. Feldman, Chu-Cheow Lim, and Franco Mazzanti. pSather monitors: Design, Tutorial, Rationale and Implementation. Technical Report TR-91-031, International Computer Science Institute, Berkeley, Ca., 1991.

[98] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.0 final, May 24 1993.

[99] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D Language Specification. Technical Report COMP TR90-141, Department of Computer Science, Rice University, Houston, TX 77251-1892, December 1990. Revised April, 1991.

[100] Curt Freeland. Sun Machines Set the Standards: SPARCstation 2, ELC, IPX, and IPC. *SunWorld*, 5(1):62–73, January 1992.

[101] Svend Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. Technical Report UIUCDCS-R-92-1742, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2987, April 1992. Also in: Proceedings of ECOOP '92, 1992.

[102] Dennis Gannon, Vincent A. Guarna Jr., and Jenq Kuen Lee. Static Analysis and Runtime Support for Parallel Execution of C. Technical Report CSRD-918, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801-2932, August 1989.

[103] Dennis Gannon, Jenq Kuen Lee, and Srinivas Narayana. On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions. In *Proceedings of CONPAR 92 - LNCS 634*. Springer-Verlag, September 1992.

[104] R. Gebauer and M. Möller. On an installation of buchberger's algorithm. *Journal of Symbolic Computation*, 6:275 – 286, 1988.

[105] Kouroush Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors. In *Proceedings of the 18th Annual Symposium on Computer Architecture*, pages 245–257, June 1991.

[106] Kouroush Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, June 1990.

[107] Phillip B. Gibbons. A More Practical PRAM Model. Technical Report TR-89-019, International Computer Science Institute, Berkeley, Ca., April 1989.

[108] Aaron J. Goldberg. Multiprocessor Performance Debugging and Memory Bottlenecks. Technical Report CSL-TR-92-542, Computer Systems Laboratory, Stanford University, August 1992.

[109] Adele Goldberg and David Robson. *Multiple Independent Processes*, chapter 15, pages 249–266. Addison-Wesley, 1983.

[110] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988.

[111] B. Goldberg and P. Hudak. ALFALFA — Distributed Graph Reduction on a Hypercube Multiprocessor. In *Workshop on Graph Reduction*, Lecture Notes in Computer Science 279. Springer Verlag, 1986.

[112] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, Computer Systems Laboratory, January 1990.

[113] Christophe Gransart and Jean-Marc Geib. Reusability and Concurrency in Parallel Eiffel. In *10th International Eiffel User Conference*, April 3 1992. Also available as: Laboratoire D'Informatique Fondamentale De Lille Publication no. 112, March 1992.

[114] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Distinguished Dissertations. The MIT Press, Cambridge, Massachusetts, 1988.

[115] Andrew S. Grimshaw. An Introduction to the Parallel Object-Oriented Programming with Mentat. Technical Report TR-91-07, Department of Computer Science, University of Virginia, Charlottesville, Virginia 22903, April 1991.

[116] Andrew S. Grimshaw. A Software Environment for High-Performance Parallel Computing. In *Proceedings of the 1991 Minnowbrook Workshop on Software Engineering for Parallel Computing*, July 16-19 1991.

[117] Hans W. Guesgen, Kinson Ho, and Paul N. Hilfinger. A Tagging Method for Parallel Constraint Satisfaction. *Journal of Parallel and Distributed Computing*, 16:72–75, 1992.

[118] L. Gunaseelan and R.J. LeBlanc. Distributed Eiffel: A Language for Programming Multi-granular Distributed Objects on the Clouds Operating System. Technical Report 91/50, Georgia Institute of Technology, College of Computing, 1991.

[119] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[120] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–206, June 1975.

[121] Per Brinch Hansen. The n-body pipeline. Technical Report SU-CIS-91-09, School of Computer and Information Science, Suite 4-116, Center for Science and Technology, Syracuse, New York 13244-4100, March 1991.

[122] Per Brinch Hansen. Parallel Divide and Conquer. Technical Report SU-CIS-91-45, School of Computer and Information Science, Suite 4-116, Center for Science and Technology, Syracuse, New York 13244-4100, December 1991.

[123] Per Brinch Hansen. Monitors and Concurrent Pascal: A Personal History. In *Second ACM SIG-PLAN History of Programming Languages Conference (HOPL-II)*, Cambridge, Massachusetts, USA, April 20-23 1993.

[124] Samuel P. Harbison and Guy L. Steele, Jr. *C, A Reference Manual*. Prentice-Hall, Inc., 1991.

[125] Bryan Hastings and Dan Miller. Fastest Number-Crunching PCs for $3000. *PC World*, pages 132–165, May 1993.

[126] Philip J. Hatcher, Anthony J. Lapadula, Robert R. Jones, Michael J. Quinn, and Ray J. Anderson. A Production-Quality C* Compiler for a Hypercube Multicomputer. Technical Report TR 90-80-3, Oregon State University, Computer Science Department, 1990.

[127] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, Massachusetts, 1991.

[128] John Hennessy and David Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[129] John L. Hennessy and Norman P. Jouppi. Computer Technology and Architecture: An Evolving Interaction. *IEEE Computer*, 24(9):18–29, September 1991.

[130] Mark D. Hill et al. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-V Proceedings*, pages 262–273, Boston, Massachusetts, Oct 12 - Oct 15 1992.

[131] W. Daniel Hillis and Guy L. Steele, Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[132] Seema Hiranandani et al. An Overview of the Fortran D Programming System. Technical Report COMP TR91-154, Rice University, Houston, TX 77251-1892, March 1991.

[133] Kinson Ho. *High-level Abstractions for Symbolic Parallel Programming*. PhD thesis, University of California at Berkeley, Berkeley, CA, 1993. To appear.

[134] Kinson Ho, Paul N. Hilfinger, and Hans W. Guesgen. Optimistic discrete parallel relaxation. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, Chambery, Savoie, France, Aug 1993.

[135] W. Wilson Ho. Dld, A Dynamic Link/Unlink Editor, Version 3.2.3. Dld package, 1991.

[136] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17:549–557, October 1974.

[137] C.A.R. Hoare. *Communicating Sequential Processes*, pages 136–148. IEEE Computer Society Press, 1989. Also published in Communications of the ACM, August 1978, pp 666-677.

[138] Guido Hogen and Rita Loogen. A New Stack Technique for the Management of Runtime Structures in Distributed Implementations. Technical Report 93-03, Department of Computer Science, Aachen University of Technology, RWTH Aachen, Lehrstuhl für Informatik II, Ahornstraβe 55, W-5100 Aachen, Germany, 1993.

[139] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP '91 Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, July 1991.

[140] Chris Houck and Gul Agha. HAL: A High-Level Actor Languages and its Distributed Implementation. Technical Report UIUCDCS-R-92-1728, Department of Computer Science, Univeristy of Illinois at Urbana-Champaign, September 1992.

[141] Jin H. Hur and Kilnam Chon. Overview of a parallel object-oriented language clix. Technical Report CS-TR-87-25, Computer Science Department, Korea Advanced Institute of Science and Technology, Seoul, Republic of Korea, 1987.

[142] IEEE Spectrum Special Issue: Supercomputers, September 1992.

[143] IEEE. Threads Extension for Portable Operating Systems (Draft 6), February 1992. P1003.4a/D6.

[144] International Computer Science Institute. Sather 0.2i programming language and environment, 1992.

[145] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[146] Eric Jul et al. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[147] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor Based Concurrent Object-Oriented Languages. In *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 131–145, 10-14 July 1989.

[148] M. Karaorman and J. Bruno. Concurrent programming with Eiffel. Technical Report TRCS 91-12, University of California, Santa Barbara. Department of Computer Science, 1991.

[149] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. In *Proceedings of the Twenty-Fourth Annual ACM Symposium of the Theory of Computing*, pages 318–326, May 1992.

[150] David Keppel. Fast Data Breakpoints. Technical Report 93-04-06, Department of Computer Science and Engineering, University of Washington, April 1993.

[151] Tim Korson and John D. McGregor. Understanding Object-Oriented: A Unifying Paradigm. *Communications of the ACM*, 33(9):40–60, September 1990.

[152] Monica S. Lam. *A Systolic Array Optimizing Compiler*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1989.

[153] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[154] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[155] Bernard Lang, Christian Queinnec, and José Piquer. Garbage Collecting the World. In *Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–50, January 1992.

[156] Sather language mailing list. Discussions on sather language, 1991-current. Notes available by anonymous ftp from icsi-ftp.berkeley.edu.

[157] Steven T. Lansdowne, Robert E. Cousins, and D. Clark Wilkinson. Reprogramming the sieve of eratosthenes. *IEEE Computer*, pages 90–91, August 1987.

[158] James R. Larus. *Restructuring Symbolic Programs for Concurrent Exectuion on Multiprocessors*. PhD thesis, University of California at Berkeley, 1989. Also available as: Report No. UCB/CSD 89/502, May 1989, Cmputer Division (EECS), Berkeley, California 94720.

[159] Jenq Kuen Lee and Dennis Gannon. Object Oriented Parallel Programming Experiments and Results. In *Proceedings of Supercomputing '91*. IEEE Computer Society Press and ACM SIGARCH, November 1991.

[160] A.L. Leiner, W.A. Notz, J.L. Smith, and A. Weinberger. PILOT – A New Multiple Computer System. *Journal of the ACM*, 6:313–335, 1959.

[161] Daniel Lenoski et al. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[162] Zhiyuan Li and Pen-Chung Yew. Efficient Interprocedural Analysis for Program Parallelization and Restructuring. In *Proceedings of the ACM/SIGPLAN Parallel Programming Experience with Applications, Languages and Systems (PPEALS) Symposium*, pages 85–99, 1988. Proceedings also published as: SIGPLAN Notices, Vol. 23, No. 9, September 1988.

[163] Chu-Cheow Lim, Jerome A. Feldman, and Stephan Murer. Unifying Control- and Data-parallelism in an Object-Oriented Language. In *Joint Symposium on Parallel Processing, Tokyo, Japan 1993*, May 17 - 19 1993.

[164] Chu-Cheow Lim and Andreas Stolcke. Sather Language design and performance evaluation. Technical Report TR-91-034, International Computer Science Institute, Berkeley, Ca., May 1991.

[165] Klaus-Peter Löhr. Concurrency Annotations. In *OOPSLA Conference Proceedings*, pages 327–340, October 1992. Proceedings available as: SIGPLAN Notices, Vol 27, No 10, October 1992.

[166] Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. In *Proceedings of International Conference on Parallel Processing*, pages 303–310. Pennsylvania State University Press, University Park, PA., 1988.

[167] Steven E. Lucco. Parallel Programming in a Virtual Object Space. In *Proceedings OOPSLA '87*, pages 26–34. ACM, December 1987. Proceedings available as: SIGPLAN Notice Vol 22, No. 12, Dec 1987.

[168] Steven E. Lucco and David P. Anderson. Tarmac: a language system substrate based on mobile memory. Technical Report UCB/CSD 89/525, Computer Science Division (EECS), University of California, Berkeley, CA 94720, November 1989.

[169] Mesaac Makpangou et al. Structuring distributed applications as fragmented objects. Technical Report 1404, INRIA, France, January 1991.

[170] Evangelos P. Markatos and Thomas J. LeBlanc. Load Balancing vs. Locality Management in Shared-Memory Multiprocessors. Technical Report 399, The University of Rochester, Computer Science Department, Rochester, New York 14627, October 1991.

[171] Satoshi Matsuoka. *Language Features for Extensibility and Re-use in Concurrent Object-Oriented Languages.* PhD thesis, University of Tokyo, 1993. In preparation.

[172] Satoshi Matsuoka, Shin'ichi Furuso, and Akinori Yonezawa. A Fast Parallel Conservative Garbage Collector for Concurrent Object-Oriented Systems. In *Proceedings of IEEE International Workshop on Object Orientation in Operating Systems*, pages 87–93, October 1991.

[173] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages, October 1990. (Proceedings unpublished).

[174] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, October 1992.

[175] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[176] Betrand Meyer. Sequential and concurrent object-oriented programming. In *Proceedings of TOOLS '90*, June 1990.

[177] Barton P. Miller. What to Draw? When to Draw? An essay on Parallel Program Visualization. Technical Report 1103, Computer Sciences Department, University of Wisconsin – Madison, July 1992.

[178] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. The MIPS R4000 Processor. *IEEE Micro*, pages 10–22, April 1992.

[179] Frank Mueller. A Library Implementation of POSIX Threads under Unix. In *1993 Winter USENIX*, pages 29–41, January 1993.

[180] Stephan Murer and Philipp Färber. A Scalable Distributed Shared Memory. In *Proceedings of CONPAR 92-VAPP V - Lyon, France, LNCS 634*. Springer-Verlag, September 1992.

[181] Stephan Murer, Jerome A. Feldman, and Chu-Cheow Lim. pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation. Technical Report TR 93-028, International Computer Science Institute, Berkeley, Ca., June 1993.

[182] Stephan Murer, Stephen Omohundro, and Clemens Szyperski. Sather Iters: Object-oriented Iteration Abstraction, 1993.

[183] Padmini Narayan, Sherry Smoot, Ambar Sarkar, Emily West, Andrew Grimshaw, and Timothy Strayer. Portability and Performance: Mentat Applications on Diverse Applications. Technical Report TR-92-22, Department of Computer Science, University of Virginia, Charlottesville, Virginia 22903, July 22 1992.

[184] O.M. Nierstrasz. Active Objects in Hybrid. In *OOPSLA 1987 Conference Proceedings*, pages 243–253, Oct 4 - Oct 8 1987.

[185] Roger J. Noe. Pablo Instrumentation Environment User's Guide (Final Draft Version), June 1992. Part of Pablo Performance Analysis Environment (Release 1.0).

[186] R.R. Oldehoeft et al. The SISAL 2.0 reference manual. Technical Report UCRL-MA-109098, Lawrence Livermore National Laboratory, Livermore, CA., December 1991.

[187] Stephen Omohundro and Chu-Cheow Lim. The Sather Language and Libraries. Technical Report TR-92-017, International Computer Science Institute, Berkeley, Ca., March 1992.

[188] Stephen M. Omohundro. The Sather Language. Technical report, International Computer Science Institute, Berkeley, Ca., 1991.

[189] Stephen M. Omohundro. The Sather 1.0 Specification. Technical report, International Computer Science Institute, Berkeley, Ca., 1993.

[190] Joseph Pallas and David Ungar. Multiprocessor Smalltalk: A Case Study of a Multiprocessor-Based Programming Environment. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 268–277, Atlanta, Georgia, June 22-24 1988.

[191] Joseph Ira Pallas. *Multiprocessor Smalltalk: Implementation, Performance and Analysis.* PhD thesis, Stanford University, June 1990. Also available as technical report CSL-TR-90-429.

[192] Cynthia A. Phillips. *Theoretical and Experimental Analysis of Parallel Combinatorial Algorithms.* PhD thesis, MIT, October 1989. Also available as: MIT VLSI Publications, VLSI Memo No. 90-577, June 1990.

[193] Maria Cristina Pinotti and Geppino Pucci. Parallel Priority Queues. Technical Report TR-91-016, International Computer Science Institute, March 1991.

[194] Carl Glen Ponder. *Evaluation of "Performance Enhancements" in Algebraic Manipulation Systems.* PhD thesis, University of California at Berkeley, 1988. Also available as Report No. UCB/CSD 88/438, August, 1988, Computer Science Division (EECS), University of California, Berkeley, CA 94720.

[195] Dick Pountain. A tutorial introduction to OCCAM programming. Inmos Occam Manual.

[196] Umakishore Ramachandran and M. Yousef A. Khalidi. An Implementation of Distributed Shared Memory. Technical Report GIT-ICS-88/50, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Ga 30332-0280 USA, December 1988.

[197] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. The Pablo Performance Analysis Environment, 1992. Part of Pablo Performance Analysis Environment (Release 1.0).

[198] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, May 1993.

[199] Kendall Square Research. Kendall Square Research: Technical Summary.

[200] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis vs. procedure integration. *Information Processing Letters*, 32:137–142, 1989.

[201] James Rothnie. Overview of the KSR1 Computer System. Kendall Square Research Report TR 9202001, March 1992.

[202] M. Sakkinen. Disciplined Inheritance. In *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 39–56, 10-14 July 1989.

[203] M. Satyanarayanan. Distributed File Systems. In Sape Mullender, editor, *Distributed Systems*, pages 149–188. ACM Press, 1989.

[204] H. W. Schmidt. Data-Parallel Object-Oriented Programming. In *Proc. of the Fifth Australian Supercomputer Conference, Melbourne (AUS): Univ. Melbourne*, December 1992.

[205] Heinz W. Schmidt and Ben Gomes. ICSIM: An Object-Oriented Connectionist Simulator. Technical Report TR-91-048, International Computer Science Institute, September 1991.

[206] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and Migration for C++ Objects. In *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 191–204, 10-14 July 1989.

[207] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA '86 Conference Proceedings*, pages 38–45, September 29 - October 2 1986. Proceedings also available as: SIGPLAN Notices, Vol 21, No 11, November 1986.

[208] W. Y. Stevens. The Structure of SYSTEM/360: Part II – System Implementations. In Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, editors, *Computer Structures: Principles and Examples*, pages 710–715. McCraw-Hill, Inc., 1982.

[209] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag, 1993.

[210] Bjarne Stroustrup. *The C++ Programming Language, Second Edition.* Addison-Wesley Publishing Company, September 1991.

[211] R.J. Swan, S.H. Fuller, and D.P. Siewiorek. Cm* – a modular, multi-microprocessor. In *Proceedings of the National Computer Conference*, volume 46, 1977.

[212] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *IEEE Computer*, pages 10–19, August 1992.

[213] David Tarditi, Peter Lee, and Anurag Acharya. No Assembly Required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

[214] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Efficient Support for Multicomputing on ATM Networks. Technical Report 93-04-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, April 1993.

[215] Robert H. Thomas and Will Crowther. The Uniform System: An Approach to Runtime Support for Large Scaled Shared Memory Parallel Processors. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, August 1988.

[216] Chris Tomlinson and Mark Sheeval. Concurrent Object-Oriented Programming Languages. In Won Ki and Frederick H. Lochovsky, editors, *Object-oriented concepts, databases, and applications*, chapter 5, pages 79–124. ACM Press, 1989.

[217] Chris Tomlinson and Vineet Singh. Inheritance and Synchronization with Enabled-Sets. In *OOPSLA 1989 Conference Proceedings*, pages 103–112, October 1-6 1989. Proceedings also as: SIGPLAN Notices 24(10), October, 1989.

[218] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill, Inc., 1989.

[219] Ping-Sheng Tseng. *A Systolic Array Parallelizing Compiler.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1990.

[220] Ping-Sheng Tseng. A Systolic Array Parallelizing Compiler. *Journal of Parallel and Distributed Computing*, 9:116–127, 1990.

[221] Jeffrey D. Ullman. *Computational aspects of VLSI.* Principles of computer science series. Computer Science Press, 1984.

[222] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA 1987 Conference Proceedings*, pages 227–241, 1987. Proceedings also available as SIGPLAN Notices 22(12), December, 1987.

[223] Reference Manual for the Ada Programming Language. United States Department of Defense, July 1982.

[224] Jan van den Bos and Chris Laffra. PROCOL: A Concurrent Object-Oriented Language with Protocols Delegation and Constraints. Technical report, Department of Computer Science, University of Leiden, December 6 1990.

[225] Jean-Philippe Vidal. The Computation of Gröbner Bases on a Shared Memory Multiprocessor. Technical Report CMU-CS-90-163, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, August 1990.

[226] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*. ACM Press, May 1992. Also available as technical report from University of California at Berkeley, CS Division, UCB/CSD 92/675, March 1992.

[227] Peter Wegner and Scott A. Smolka. Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives. *IEEE Transactions on Software Engineering*, SE-9(4):446–462, July 1983.

[228] L. Curtis Widdoes, Jr. and Steven Correll. The S-1 Project: Developing High-Performance Digital Computers. In Robert H. Kuhn and David A. Padua, editors, *Tutorial on Parallel Processing*. IEEE Computer Society, 1981.

[229] Katherine Anne Yelick. *Using Abstraction in Explicitly Parallel Programs*. PhD thesis, MIT, MIT Laboratory for Computer Science, Cambridge, MA 02139, December 1990.

[230] Y. Yokote and M. Tokoro. Experience and evolution of concurrentsmalltalk. In *Proceedings of OOPSLA*, pages 406–415, Orlando, Florida, December 1987. ACM.

[231] Akinori Yonezawa, editor. *ABCL – An object-oriented concurrent system*. MIT Press, Cambridge, Massachusetts, 1990.

[232] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In *OOPSLA 1986 Proceedings*, pages 258–268, September 1986.

[233] Akinori Yonezawa, Satoshi Matsuoka, Masahiro Yasugi, and Kenjiro Taura. Implementing Concurrent Object-Oriented Languages on Multicomputers. *IEEE Parallel & Distributed Technology, Systems & Applications*, 1(2):49–61, May 1993.

[234] Benjamin Zorn et al. Multiprocessing Extensions in Spur Lisp. *IEEE Software*, pages 41–49, July 1989.