A Compiler-Assisted Approach to SPMD Execution

Ron Cytron*
Jim Lipkis†
Edith Schonberg‡

Abstract

Today, two styles of scientific parallel programming prevail. In the SPMD style, all processors execute the same program, with sequential code executed redundantly and parallel code executed cooperatively. In the fork-join style, a sequential thread of control spawns multiple threads to execute a portion of the code concurrently. In this paper we describe an automatic method for approaching the efficiency of SPMD-style execution for programs written in the more structured fork-join style.

1 Introduction

In literature and in common practice, two models have evolved for the expression of parallelism in source-parallel programs. As an example, consider the same program expressed in each of these models in Figures 1 and 2.

Fork-Join In this model, an instruction stream (or *thread*) executes sequentially until a parallel construct is encountered. In the program of Figure 1, the `doall` construct causes the otherwise sequential control to *fork* into P processes that execute the N iterations of the loop, allocating any private variables declared in the loop. The `endall` acts as a *join* point for the P processes: a single processor proceeds past the `endall` after all P processes are finished with the loop.

Other parallel constructs are admissible, as long as the beginning and end of the construct are lexically visible, and the construct is entered only at its official lexical entry (no jumps are allowed into the construct).

```
doall i=1 to N
   A(i) = B(i) + C(i)
   if (i .eq. N) x = 0
endall
x = x + 25
call foo(x)
doall i=1 to N
   A(i) = A(i) + x
endall
```

Figure 1. Fork-Join-style program.

SPMD In this model [7, 8], all processors execute the same instruction stream. During "sequential" portions of a program, computations are actually performed redundantly, with each processor computing the same result. When a "parallel" construct is encountered, the processors' execution ceases to be redundant, with each processor cooperating on a portion of the parallel work.

For example, consider the program shown in Figure 2. As processors encounter the `dopar` loop, the iterations are distributed to achieve parallel execution. The next assignment statement is protected by a `serbegin` and `serend` pair, so that the increment of `x` is performed just once, rather than once by each processor. Finally, the procedure call is executed redundantly by all processors, and the iterations of the next `dopar` are again distributed.

The fork-join model has many appealing characteristics; the semantics are clear and the parallel constructs *nest* in a straightforward manner. Because the lexical scope of variables corresponds to the fork-join structure, a compiler can more easily determine the liveness of variables than in the less restrictive SPMD model. If efficiency were not an issue, fork-join would be the model of choice. Unfortunately, a naive translation of programs written in this model could result in inefficient parallel execution. For example, consider the program shown in Figure 3. The trace for the program would show a pattern in which a *join* operation is followed immediately by a *fork*. If executed naively, the overhead of releasing and reallocating processors in between the parallel constructs becomes excessive. Even when very "light-weight" threads are used as virtual processors, this scheduling typically entails swapping registers, allocating control structures and

*IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.

†Ultracomputer Research Laboratory, New York University, 715 Broadway, 10th Floor, New York, NY 10003. This research was supported in part by the Applied Mathematical Science subprogram of the office of Energy Research, U.S. Department of Energy under contract number DE-FG02-88ER25052. Author's current address is Chorus systèmes, 6, Avenue Gustave Eiffel, 78182 Saint-Quentin-En-Yvelines CEDEX, France.

‡IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.

```
/* x is a shared variable */

dopar i=1 to N
   A(i) = B(i) + C(i)
   if (i .eq. N) x = 0
endpar
serbegin
   x = x + 25
serend
call foo(x)
dopar i=1 to N
   A(i) = A(i) + x
endpar
```

Figure 2. SPMD-style program.

```
doall i = 1, N
   stuff
endall

doall i = 1, N
  stuff
endall

doall i = 1, N
  stuff
endall
```

Figure 3. An inefficient fork-join program.

storage, and copying context. This overhead is mostly eliminated with SPMD, because processors do not really disappear at the end of a parallel construct. Instead, they execute redundantly until the next parallel (or serializing) construct.

Although efficient, programming in the SPMD model is cumbersome. In Figure 2, the `serbegin` section *must* end before the call to `proc`, if any parallelism is to be exploited within `proc`. If the procedure call appeared within the `serbegin..serend` pair, then only one processor would actually invoke `proc`, and no cooperative execution could be achieved until `proc` returned. Furthermore, redundant execution is conceptually difficult for programmers and writing code that is executed redundantly can be error-prone.

Current literature exhibits proponents for each model. For hand-translation of sequential code to parallel code, some believe the SPMD model is easier, because when all processors execute the same code, only synchronization instructions must be executed [7]. However, this style of programming may be more prone to race conditions, and users must be aware of overheads and process traffic through the code. Analysis and further automatic translation is simpler for fork-join programs, but SPMD undeniably has the lower overhead. Because of these conflicting needs, the emerging Fortran standard [10] supports both models: nested fork-join constructs are available, as well as a closed nestable *parallel region* construct that encapsulates SPMD execution.

We do not address the issue of which model is best, because such judgement is largely a matter of taste. We seek to eliminate the argument that parallel programs should be constructed in the SPMD model for reasons of efficiency. Our approach is to accept a fork-join program and *automatically* introduce SPMD regions into the program where appropriate. Because such decisions require intimate knowledge of run-time, operating system, and architecture overheads, we believe that programs will become more portable when written in a general fork-join model. It then falls on the compiler to restructure certain regions into SPMD style for efficiency.

2 Framework

We assume as input a *fork-join* program structure, either originally supplied by a programmer and subsequently analyzed, or generated from a program parallelization system such as PTRAN [1, 4]. In such programs, parallelism can be created and terminated by closed, nestable parallel constructs. We call the beginning of a parallel construct the *fork point* and the end of the parallel construct the *join point*. We assume that the execution of parallel loops is deterministic: there are no data dependences among the parallel threads, so that no race conditions are present. For simplicity, we assume there are no other synchronization primitives; however, our algorithm can be extended to handle other primitives such as DOACROSS.

Private variables may be defined within a parallel construct, declared by the keyword `private`. Semantically, a separate copy of a private variable exists for each thread of a parallel construct. Standard block structure rules apply to private variables: an inner construct may reference a private variable defined in the scope of an outer construct, in which case it is shared with respect to the inner construct. Figure 4 illustrates the use of a private variable `Z`.

Our goal is to automatically introduce SPMD execution where profitable without violating the semantics of the original fork-join program. More specifically, we insert processor *entry points* and *exit points* that delimit *redundant execution regions*, such that:

- Parallelism is created at entry points and terminated at exit points rather than at parallel constructs;
- Code between an entry and exit point is executed in SPMD mode.

Profitability of this transformation is obtained by the following considerations:

- To ensure that processor allocation/deallocation is in fact reduced, our algorithm attempts to maximize the number of parallel constructs executed in the same redundant region.
- Further savings can be obtained within redundant regions through the *privatization* of computations to exploit target architecture memory hierarchies. By

converting shared variables into variables that are private per processor, a single computation of a shared value is replaced by multiple local computations.

Figures 4 and 5 show a fork-join program before and after being transformed for SPMD execution.[1] In

```
do I = 1, N
    X = I * pi
    doall J = 1, N
     private Z
      Z = X ...
      if (J .eq. N) Y = ..Z..
    endall J
    = Y
 end do I
 doall K = 1, M
 ...
 endall K
```

Figure 4. Fork-Join program before SPMD transformation.

```
processor-entry
private I, X, Z

do I = 1, N
    X = I * pi
    dopar J = 1, N
       Z = X ...
       if (J .eq. N) Y = ..Z..
    endpar J
     = Y
 end do I
 dopar K = 1, M
 ...
 endpar K

 processor-exit
```

Figure 5. SPMD Program with Private Variables.

Figure 5, the `dopar J` and `dopar K` loops are contained in a single redundant region, in which the `do I` loop and statement `X = I * pi` are executed redundantly. There is only one processor allocation and deallocation point instead of `N + 1` as in the original program. Since `X` and `I` are private, the number of shared variable accesses is decreased. Furthermore, storage allocation for all private variables, including `Z` is only performed once at the entry point.

[1] We use `doall` for fork/join parallel loops and `dopar` for SPMD parallel loops.

Variable `Y`, on the other hand, is not made into a private variable. Since its updated value must be accessed by all processors after `endpar J`, there is no significant savings in making `Y` private.

SPMD execution enables other low level implementation optimizations. In this example, it is possible to keep the value of `N` in a register throughout the entire redundant region. Similarly, SPMD enables more extensive code motion making inner loops more efficient.

Although the beginning and end of a `dopar` construct are not true fork and join points, processor synchronization is still necessary for enforcing data dependences and for coordinating execution of `pardo` loops. Placing a *barrier synchronization* at the start and end of a `dopar` construct is always safe. However, often the compiler can determine that it is possible to omit either of these barriers. For example, by omitting the start barrier, some processors can begin executing a `dopar` while other processors are still executing the parent code of the `dopar`. Safety issues pertaining to barrier placement are discussed in Section 4.

3 Algorithm Overview

Let R_1 and R_2 be parallel constructs. We call a program path C between the join point of R_1 and the fork point of R_2 a *connecting path* if R_1 and R_2 are at the same parallel construct nesting level, are nested within the same outer parallel construct, and there is no other parallel construct along the path C. For example, in Figure 4 there is a connecting path from `endall J` to `doall J` and `endall J` to `doall K`. The constructs R_1 and R_2 can be merged in the same SPMD region by considering all connecting paths from R_1 to R_2 for safety and profitability.

More precisely, the algorithm consists of the following steps:

1. Construct the *fork-join graph*, which is an abstract graph containing F (fork) nodes, J (join) nodes, and edges between them representing connecting paths of sequential code. Accordingly, each procedure begins with a J node and ends with an F node. Our goal is to merge, where possible and profitable, a J node with a subsequent F node, causing the contained sequential code to be executed redundantly.
2. Construct the depth-first spanning tree and numbering of this graph. We will consider nodes in their reverse postorder listing, so that a J node is considered before any F node on a connecting path from J to F. (The tree and numbering provide a topological ordering for the nodes, without having to sort. Reverse postorder allows for more accurate cost estimates (see Section 6), since variables already privatized in the same region are known.)

   For each J node in this order, and for every connecting path between this node and an F node,

   (a) Flag variables as privatizable, according to whether privatization increases safety and profitability.

   (b) Determine whether redundant execution of each connecting path is safe and profitable. If so, merge private variables into a single storage frame.

Issues of safety and privatization are discussed in Sections 4 and 5. The decision to transform code into SPMD is guided by a cost function, discussed in Section 6.

3. Insert processor entry and exit points (discussed in Section 7) and declarations of private variables.

4 Safety Conditions

Redundant execution of a section of code C is safe only if it produces the same result as non-redundant execution of C. More specifically, we define the *P-trace* of C to be the sequence of values assigned to shared variables by a processor P executing C. Redundant execution of C is *safe* if the P-trace of C is the same for all processors P that execute C redundantly for all execution interleavings, and the P-trace is independent of the number of processors that execute C.

For example, consider the following code fragment executed redundantly by processors P_1 and P_2:

```
S = X * 2
X = A + Y
R = S
```

If P_1 finishes execution before P_2 starts, then the values assigned to S by P_1 and P_2 are different. This is because P_1 modifies X before P_2 uses it. On the other hand, suppose we remove the assignment to `X`. Then redundant execution is safe, because the values assigned to `S` and `R` are the same for all processors executing the two remaining assignments, for all possible interleavings.

We refer to standard definitions of data dependence for a variable X [9]:

Flow: A flow dependence exists if X is modified and subsequently accessed.

Storage-Related: A storage-related dependence exists if X is accessed and subsequently modified (an anti-dependence), or if X is modified and subsequently modified (an output dependence).

The safety of redundant execution is given by the following theorem:

Theorem 1: Redundant execution of a sequential code segment C is safe if there are no storage-related dependences in C.

Proof: Suppose there are no storage-related dependences in C, but redundant execution is not safe. Then some P_1-trace and P_2-trace for C are different. Let F be the first statement that assigns a variable X where P_1-trace and P_2-trace are different. Then some variable Y used in F has a different value when P_1 accesses it than when P_2 accesses it. Suppose P_1 accesses Y first. There must be an assignment to Y after P_1 accesses Y in F and before P_2 accesses Y in F. If the assignment to Y is performed by P_1, then there is a storage-related dependence. If the assignment to Y is performed by P_2, then this assignment precedes F, and computes a different value for Y in both P_1 and P_2. Therefore, redundant execution must be safe.

A barrier at the beginning or end of a parallel construct in a redundant region can hurt performance. Dependence analysis can also determine when barrier synchronization is not needed. For example, if C is a connecting path terminating at R_2, and there is a storage-related dependence from C to R_2, then redundant execution of C is not safe if there is no barrier at the beginning of R_2. However, a barrier at the beginning of a parallel construct R_2 might obviate the need for a barrier at the end of the previous parallel construct R_1. For example, in Figure 6, a barrier is required at the `dopar` to break the storage dependence for `A(I)`, but consequently no barrier is required at the corresponding `endpar`.

```
processor-entry
do I = ...
     = A(I)....
  dopar J =                 /* barrier */
    if (foo) A(I) = ...
  endpar                    /* no-barrier */
enddo
processor-exit
```

Figure 6. Storage Dependence That Requires Barrier

While storage-related dependences are unsafe, flow dependences do not cause safety problems for redundant execution. Fortunately, storage-related dependences can always be removed by using additional storage [5]. Making redundant execution safe via *privatization* is described in the next section.

5 Privatization

While the lifetime and scope of a private variable in a fork-join parallel construct is semantically *per thread*, within an SPMD execution region, each private variable is allocated once per processor at the time a processor enters the region and deallocated when a processor exits the region.[2] In Figure 7, the private variable `Z` is allocated at the processor entry point, In each iterate of the `dopar`, `X` is assigned to the variable `Z` that is private to the processor executing the iterate.

Privatization is a transformation, similar to renaming [5], that replaces a shared variable by a private variable [3]. In Figure 8, the assignment to the shared variable `X` is replaced by an assignment to the private variable `X`, which eliminates all shared references in the redundant region.

Privatizing a shared variable X used in a connecting path C is always possible. However, it is not always

[2]It is unsafe to allocate a private variable in a fork-join program once per-processor in this manner *only* if threads are allowed to block and context switch. Otherwise, a given location can be reused in each thread executed by the same processor.

```
processor-entry  /* Private Z allocated */
private Z
...
X =             /* Shared X  */
dopar i =
  Z = X         /* Private Z = Shared X */
  ...
endpar
processor-exit   /* Private Z deallocated */
```

Figure 7. Per processor private variable Z.

```
processor-entry  /* X, Z allocated */
private X, Z
...
X =            /* Private X  */
dopar i =
   Z = X     /* Private Z = Private X   */
   ...
endpar
processor-exit   /* X, Z deallocated */
```

Figure 8. Per processor private variables X and Z.

profitable. Copying is sometimes necessary from shared variables to the private variables, and vice versa, to preserve the original program semantics. For example, if there is an upwardly exposed use of a privatized variable X in a redundant region, then the private variable X must be initialized before it is used. For this reason, it is often not profitable to privatize arrays: the cost of copying an entire array is too high. Unless it is possible to determine that all array elements are needed in the redundant region, or that copying of the array is not necessary, arrays are not privatized.

More precisely for each variable X that occurs in a connecting path C, X is flagged *privatizable* when either:

- Privatization eliminates storage dependences so that connecting paths can be executed redundantly safely.
- Privatization eliminates the need for a barrier at the beginning of a parallel construct.
- Privatization improves the efficiency of redundant execution, (as described in Section 6).

When processor entry and exit points are added (step 3 in Section 3), variables flagged privatizable are made private. To preserve correctness, declarations and statements are added to copy values to and from shared variables, according to the following rules. Let X be a use or definition on a connecting path C in a redundant region D:

1. If X is an upwardly exposed use in D then X must be copied in from shared memory at the processor entry point.
2. If X is a definition that is live on exit from D, then X must be copied out to shared memory at the processor exit point.
3. If there is already a definition of a *private* variable X in a parallel construct R that reaches a use of X in C, then a renaming transformation must be applied to the existing private variable.
4. If there is a definition of X in a parallel construct R that reaches a use of X in C, then the value assigned to X in R must be transmitted to all processors executing C.

The first three rules are illustrated by Figures 9 and 10, which show a program before and after applying the algorithm. Variables Y and the outer X are privatized.

```
X = Y ...
doall J = 1,N
  private X
    X =
    ... = X
    if (J .eq. N) Y = ...
endall
...
print X, Y
```

Figure 9. Original fork-join program.

Variable Y is privatized to break the anti-dependence, so that no barrier synchronization is needed at the `dopar J`.

1. Since Y is upwardly exposed in the region, Y must be initialized from shared memory. This is indicated by the declaration keyword `copyin`.
2. Variables X and Y are live on exit from the redundant region, so that they must be copied out to shared memory at the exit point. This is indicated by the declaration keyword `copyout`. Since all processors exiting the redundant region have the same value of X, only one processor need perform the actual copying. For Y, the processor that executed the last J-loop iterate must copy Y to shared memory.
3. The private variable X in the `doall J` loop in Figure 9 must remain distinct from the privatized outer variable X. Therefore, it is renamed as Z in Figure 10.

Privatizing shared variables that are modified in parallel constructs pose a special problem (Rule 4). Consider Figure 11, in which Y is used in the connecting path from `endall J` to `doall K`. It is still desirable to privatize Y to break the anti-dependence. However, since the assignment to Y inside the `doall J` loop is performed by only one processor, the final value must be transmitted to all processors executing the subsequent code redundantly. This is accomplished in Figure 12 by storing back into the shared variable SY, and using SY to reinitialize the private variables Y after the `dopar`.

```
processor-entry
private Z
private Y (copyin, copyout)
private X (copyout)

X = Y ...
dopar J = 1,N
    Z =
    ... = Z
   if (J .eq. N)  Y =
endpar
...
processor-exit
print X, Y
```

Figure 10. Final SPMD program.

```
  = Y ...
doall J = 1,N
   if (J .eq. N)  Y =
endall
...
   = Y
...
doall K ...
endall
```

Figure 11. Program with Transmission Problem

```
processor-entry
private Y (copyin)

  = Y ...

dopar J = 1,N
   if (J .eq. N)
       Y =
       SY = Y   /* Update shared variable */
   endif

endpar
Y = SY  /* Transmit to private variables */
...
   = Y
...
dopar K ...
endpar

processor-exit
```

Figure 12. Program with broadcast.

6 Profitability

Let C be a connecting path from R_1 to R_2. The profitability of SPMD execution is a function of:

- $I(C)$ - the cost of executing redundantly from R_1 to R_2, and
- $CS(R_2)$ - the cost of a true fork (creating parallelism) if the start of R_2 is a processor entry point.

Redundant execution is profitable if $I(C) < CS(R_2)$. In other words, for a processor leaving R_1 we compare the cost of executing C redundantly vs. exiting and reentering at R_2.

The value of $I(C)$ depends on

- the cost of executing the path C,
- additional bus/network contention for redundant execution of C, and
- shared variable accesses in R_2, assuming SPMD execution.

These costs can be estimated by analysing instruction sequences [11] and by counting the number of shared memory accesses in C and R_2. Even when the connecting path C is short, redundant execution may increase the bus/network contention significantly if there are too many shared memory references. For example, consider the code sequence:

```
A = 5
X = A + B * C + D
doall i = ...
   = X
endall
```

If `A, B, C, D,` and `X` are all shared variables, memory contention for redundant execution is going to be a problem.

The value of $CS(R_2)$ depends on

- the cost of saving and loading local context for R_2,
- fork and synchronization overhead (e.g. queue operations), and
- shared variable accesses in R_2, assuming no SPMD execution.

Both costs $I(C)$ and $CS(R_2)$ are generally architecture and implementation dependent, according to, for example, whether processors have local memory, caches, or can access the local memory of other processors.

Privatization can be used to reduce the cost function $I(C)$. For example, it is generally beneficial to privatize a variable that is defined in C and subsequently used in either C or R_2, reducing the overhead of multiple processors accessing the same shared memory location. In the example above, the cost function $I(C)$ is improved by privatizing `A` and `X`, assuming each processor has a cache. However, nothing is saved by privatizing variables that are upwardly exposed, such as `B, C` and `D`. If these variables are privatized, their initial value will have to be copied from shared memory anyway.

7 Determining Entry and Exit Points

With respect to the fork-join graph described in Section 3, our analysis has determined which F nodes may be merged with a given J node. However, analysis may not allow all connecting paths from a given J node to become redundant. Consider the example shown in Figure 13. Suppose the

```
dopar I

endpar I
if ( )....
  dopar J

  endpar J
else
   ...
   ...
   dopar K
   endpar K
end if
```

Figure 13. Before analysis.

`dopar I` and `dopar J` are mergable, but `dopar I` and `dopar K` are not.

We must place "instructions" in the code to allow processors to exit a given region of SPMD-style execution, as shown in Figure 14, so that processors do not exit until the `else` branch is chosen. Now, we sketch a simple algorithm

```
processor-entry
dopar I

endpar I
if ( )....
  dopar J

  endpar J
  processor-exit
else
  processor-exit
   ...
   ...
   dopar K
   endpar K
end if
```

Figure 14. After analysis.

that determines placement of the `processor-entry` and `processor-exit` instructions.

We have thus far described how to determine mergings of J and F nodes in the abstract fork-join graph described in Section 3. Suppose that initially, each F-J pair is colored distinctly. Subsequently, whenever a J node is merged with an F node, the F node (and its closing J node) assume the color of the merged J node. So, in the above example, the `dopar J` loop assumes the same color as the `dopar I` loop, while the `dopar K` loop retains a distinct color.

With respect to the fork-join graph, `processor-exit` instructions should be placed as early as possible on connecting paths between J and F nodes of differing color. (`Processor-entry` instructions should be placed as late as possible.) This problem is mappable to the placement of `post` instructions for software-controllable caches, which has a simple flow solution [6].

8 Issues in Processor Scheduling

Under the SPMD execution model, every processor entering a redundant-execution region at a processor entry point executes all the way to the next exit point. Hence a processor never enters or exits a region at an individual parallel construct. It is this requirement which makes it possible to minimize coordination overhead in the initiation and termination of parallel constructs. Many systems that support SPMD execution further avoid problems of processor scheduling by supporting only single-level parallelism.

Although we introduce SPMD execution over portions of a fork-join program, we still wish to maintain some aspects of the more flexible fork-join runtime environment, nested parallelism and `doall`s outside of redundant regions at runtime [2]. If the strict SPMD execution model is adhered to, performance will suffer in some cases because of difficulties in scheduling processors to the available work within a program. Figure 15 demonstrates a load-balancing problem that can occur with nested parallelism under standard SPMD. Suppose P processors are executing

```
processor-entry
...
dopar i = 1,3
    ...
    doall j = 1,200000
         ...
    endall j
    ...
endpar i
...
processor-exit
```

Figure 15. SPMD program with nesting.

redundantly before the `dopar i` statement. By the standard SPMD rules, at most 3 of these processors execute separate iterates of the `dopar i` loop, and the remaining $P-3$ processors wait at the barrier at the `endpar i`. Although work is available at the inner `doall j` loop, these $P-3$ processors are not used to help execute that loop. Ideally, only 3 processors should enter at the processor entry point, so that the other processors could enter at the `doall`. However, detection of that constraint would require analysis of all of the parallel constructs reachable from this entry point. It is not always possible to tell at compile time how many processors will be needed for any parallel construct, especially when the loop bounds are calculated at execution time.

A second scheduling anomaly is referred to as the "catch-up" problem. This problem arises when an extra processor becomes available and joins in a redundant region relatively late in its computation. In Figure 16, a processor

```
processor-entry
do  I = 1,IBOUND
   ...
   dopar  J = ...
      ...
   endpar
   ...
end do
processor-exit
```

Figure 16. Program with catch-up problem.

might enter the region after K iterations of the serial `do I` loop ($K <$ `IBOUND`) have been completed by other processors. This new processor begins execution with the `I = 1` iteration of the `DO` loop, repeatedly bypassing already-completed instances of the `dopar J` loop. Eventually, it may reach an `I =` L iteration ($K \leq L \leq$ `IBOUND`) in which the `doall J` loop has not completed and thus has work available. The catch-up process delays and reduces the useful contribution to be made by the late-arriving processor. Furthermore, the entire `do I` loop could complete, and the other processors exit the region, while the new processor is still catching up. In this case the new processor would serve only to delay the program, since the code after the `processor-exit` cannot execute until all processors in the region have exited.

In the remainder of this section we consider modifying the SPMD execution model to reduce or eliminate these anomalies in scheduling. Any resulting increase in run-time coordination overhead must be justified in terms of the overall performance improvement gained.

Our goal is to allow a `dopar` or `endpar` statement within a redundant region to function as a conditional or *adaptive* fork/join. Thus, while it is assumed at each `dopar` that processors are already executing within the region, we may want to make provision for new processors to enter. Likewise, a processor completing its assigned work (or finding no available work) in a parallel construct might go elsewhere in the program to look for further work rather than resuming redundant execution in the current region. Thus in each processor, a decision is required whether to execute a fork (i.e., save local context and advertise new parallelism through a broadcast or queue manipulation) at each `dopar`, and whether to exit the current region at each `endpar`.

To be practical, however, processor allocation decisions at `dopar` and `endpar` points must make use of only a small amount of global state information, and incur a small overhead. We describe some general heuristics.

The additional cost of adaptive forking at `dopar` points is mitigated by the fact that only one processor need pay this cost when the beginning of a `dopar` loop is not a barrier. Other processors entering the loop may proceed immediately with useful work. Nonetheless the fork should be executed only when it could have beneficial effect, i.e., when new processors are likely to be available. Similarly, a processor should exit a region prematurely only when useful work is likely to be available earlier elsewhere in the program than in the current region. Various mechanisms are possible:

- The run-time system can maintain counts of both the processors in the current team (executing in the current redundant section) and the processors assigned to the job. Along with information on the maximum degree of parallelism in a particular construct, these counts can be used to determine whether a fork is potentially worthwhile. If the current team already includes all available processors, or sufficient processors for the current parallel construct, the fork should be suppressed.
- A processor arriving at a barrier should remain in its current region until (1) it can observe that work is available elsewhere (e.g. by examining a queue head); and (2) it has waited for a minimum period at the barrier (so as to reduce context switching that is not clearly beneficial).
- Scheduling might be further improved if a processor waiting at a barrier or executing in catch-up mode could solicit a new fork by sending a signal to the processors active in a parallel construct. In this case the fork operation might require no more than an update to the saved context associated with an existing processor-entry point. This scheme relies on the availability of a signalling mechanism that does not incur substantial overhead in the processors already active.

Adaptive fork/join operations may also be useful for subroutines that contain parallel constructs and are invoked from both redundant sections and ordinary sequential code. Without this facility, it is necessary to create two copies of the code for these subroutines.

9 Conclusion

In this paper we have described how to unify two styles of parallel programming, while retaining the advantages of both. Programs can be specified using convenient, well-understood closed constructs. Analysis at compile-time and proper support at run-time yield execution efficiency that approaches the SPMD model. Moreover, a greater degree of portability is achieved by relegating the burden of deciding what should be in an SPMD parallel region to the compiler, which is probably more familiar with architectural detail than most programmers.

If our techniques are successful, then languages that currently offer both styles of expression can be significantly simplified. Future work will certainly include experimentation with these methods in the PTRAN system, comparing the results to hand-coded SPMD-style programs. Other future work includes extending the approach described to a more general model with explicit synchronization, and relaxing the rule that sequential code in SPMD regions is always executed redundantly. Additionally, it is desirable to automatically recognize and generate efficient code for reductions.

Acknowledgements

We would like to thank Fran Allen, Frederica Darema, Jeanne Ferrante, Vivek Sarkar, and Jan Stone for their comments and suggestions concerning this work.

References

[1] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the ptran analysis system for multiprocessing. *Proceedings of the 1987 International Conference on Supercomputing*, 1987. Also published in The Journal of Parallel and Distributed Computing, Oct., 1988, Vol. 5, No. 5, pp. 617-640.

[2] David Bernstein. *PREFACE-2, Supporting Nested Parallelism in Fortran.* Technical Report, IBM Research, 1988. Report RC14160.

[3] Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. *Automatic Determination of Private and Shared Variables for Nested Processes.* Technical Report, IBM T. J. Watson Research Center, 1987. Research Report RC13194.

[4] Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, 2(3):71–88, July 1989.

[5] Ron Cytron and Jeanne Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. *Proceedings of the 1987 International Conference on Parallel Processing*, 19–27, August 1987.

[6] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic management of programmable caches (extended abstract). *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988. Also available as CSRD Rpt. No. 728 from U. of Ill.-Center for Supercomputing Research and Development.

[7] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7:11–24, 1988.

[8] Harry F. Jordan. Structuring parallel algorithms in an mimd, shared memory environment. *Parallel Computing*, 3(2):93–110, May 1986.

[9] David J. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, 1978.

[10] PCF. *Parallel Computing Forum, Final Report.* Technical Report, Kuck and Associates, Incorporated, 1990. In preparation.

[11] Vivek Sarkar. Determining average program execution times and their variance. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):298–312, July 1989.