Parallel Programming for Everyone

Naftali D. Schwartz

New York University, 251 Mercer St., New York, NY 10012-1185

Abstract. This article proposes a novel architectural model which augments the latest developments in automatic program parallelization and distributed systems to achieve a level of practicality as yet unknown to either field. Today's premier automatic parallelization model is well suited to implementation on a network of commodity workstations (NOW) using only a very thin layer of software support. We describe a parallelizing compiler framework which greatly simplifies the parallelization of even highly complex sequential applications while producing extremely effective parallelizations for the NOW. We further show how our model greatly enhances programmer productivity through the use of minimally invasive C++ transformation techniques, aiding both debugging and portability.

1 Introduction

There has always been a dim awareness of some common motivation between the fields of Automatic Parallelization and Distributed Systems. After all, the *raison d'etre* of both is to aid programmer and execution efficiency through the construction of elaborate software tools. These tools unburden the programmer of the substantial programming investment needed for effective speedups on some parallel platform.

Naturally, the focus of the two fields is not identical. In Automatic Parallelization, emphasis is placed on deriving maximum performance from elaborate multiprocessing hardware with little or no programmer intervention. Distributed Systems, on the other hand, stresses effective utilization of existing commodity hardware, even if some programming convenience must be sacrificed.

Recently, researchers from either field have begun to consider the goals of the other. Eigenmann, et al.[EPV97], looking for a more economically feasible target architecture, investigated parallelizing transformations for parallel workstations. Baratloo, et al.[BDK95], attempting to ease the complexity of writing distributed applications, devised an environment to provide efficient network execution within a simplified programming paradigm. What has been overlooked, however, is that the time is ripe for a convergence of the fields where each can benefit from the methodologies of the other.

Such a convergence has been made possible by complimentary discoveries. The parallelization community has learned the value of data-locality over efficient data-distribution[EHP98] and the significance of coarse-grain loops[HAM+95]. The distributed systems community has found that fault masking can be more effectively handled at the systems level than at the application level[DKR95].

Interestingly enough, a synthesis of all these ideas leads to an extremely efficient and effective mapping of sequential applications onto a NOW, without any run-time hardware or software sophistication of any kind. In this paper, we set forth the details of this mapping. We also show that a C++ implementation of this mapping is especially attractive, not only because of its increasing popularity, but because it incorporates features which support a completely platform-independent program transformation methodology. Furthermore, debugging is eased by the simplicity of the parallelization model, and also by performing only the most minimally invasive transformations for parallelization.

The rest of this paper is organized as follows. In Section 2, we describe a novel technique for efficient demand-driven parallelization. In Section 3, we describe how to effectively limit the high cost of communications over a NOW. Then, in Section 4, we show how C++ can serve as a universal parallel language, neatly sidestepping the parallel versioning problem. The following Section 5 details the mechanics of sharing memory via NFS for simplicity and portability. In Section 6, we show how our unification of the concepts of static and dynamic memory has simplified the handling of both. Section 7 outlines the different scheduling issues relevant to workstation network execution. Section 8 discusses caveats and some extensions and limitations of our method. Section 9 mentions the limited related work, and Section 10 concludes.

2 Demand-driven Parallelization for NOW

As outlined in the Introduction, NOW is ideally suited for the execution of coarse, independent grains. However, finding the ideal loop which can provide these grains is a non-trivial task, requiring an elaborate interprocedural analysis. Here we offer such an analysis, building on work done in dynamic compilation.

Autrey[AW97] outlines an interprocedural analysis for discovering the frequency of modification of particular program variables. Global Recursion Level Analysis (GRLA) is a Value Specific Optimization (VSO) formulated as a forward data-flow problem over the program's complete call graph (whose non-trivial construction in C++ is described in [CHS95]). GRLA partitions the computations of a program by labeling them with Code Stating Levels (CSL) which are the monotonically increasing elements of a single-chain stage-level lattice. More deeply nested code is associated with higher lattice values. Once CSLs have been assigned, Glacial Value Propagation (GVP) labels variable definitions and expressions with elements of the same lattice, indicating the frequency with which each value changes.

We modify this analysis to assign labels in the other direction, with lowest values assigned to simple loops without function calls and increasing for each loop nesting. For non-recursive programs, this can be done in a single pass. Recursive cycles need to be iterated over. While in GRLA functions are evaluated as the MAX meet of their callers, we assign values to functions based on the MAX meet of their constituent loops. At the end of the analysis, a clear picture of the global program loop nesting emerges from which a coarse-grain parallelization scheme can be devised.

In general, this analysis will discover a number of “main” functions if there are several unused functions included in the program (this is usually the case). Although it is possible to distinguish the real one by matching to the `main` identifier, this would prevent application of the system to the parallelization of libraries. A truly general solution simply takes the highest-labeled function as the main function.[1]

The structure of the chosen main function is then examined. In general, it will consist of a sequence of statements, function calls and loops. Individual statements and low-numbered function calls and loops (below a parallel-practical threshold) are ignored. Loops with high level are considered for parallelization, and functions with high level are recursively examined. In this manner the main function is (recursively) divided up into an alternating sequence of sequential code and parallelizeable loops. [2]

Each coarse grained loop is then analyzed for parallelizeability. This involves testing for a number of disqualifying conditions:

– Loop-carried flow-dependence or control-dependence
– Input operations or other operating system calls
– Non-local transfers of control such as `longjmp` or C++ exceptions
– Allocation of memory without corresponding deallocation

Notice that loop-carried output- and anti-dependences do not disqualify a loop, because these are memory-related and each iteration will use a private copy of the shared memory. Notice also that as special cases, writes to output and error streams are also permitted within a parallel loop.

The final disqualifier, allocation of memory which will be used outside of the parallel loop, can in the general case not be checked statically, but this error condition can be checked at run-time by intercepting and matching up all calls to memory allocation and deallocation at the client. If unmatched allocations are discovered at client termination, a special error condition would have to be returned to the server to have it inform the user and abort.[3]

If one or more of the main program loops do not satisfy the parallelization prerequisites, the internal loop structure is further analyzed in a recursive manner as performed on the original main program function. [4] At the end, we arrive

[1] This works as long as the loop depth of the unused code is lower than the loop depth of the main program—probably a safe assumption.

[2] Attempt is made to split up long sequences of sequential code into parallel sections as well, but the best division in this case is less well defined and needs further study.

[3] In this unlikely circumstance, the user can insert a more obviously illegal allocation inside of the would-be parallel loop to discourage the compiler from subsequent parallelization attempts on the loop.

[4] As an aid to the user, we can also indicate which loop was a desirable parallelism candidate and why it had to be disqualified. This should be a great deal more effective than current interactive parallelization systems, because it quickly focuses attention

at the maximal set of parallelizeable coarse-grained loops present in the input program. We accomplish this in a simple two-phase algorithm: first bottom up, assigning labels to loops and functions, and then top-down, finding the largest-grain parallelizeable loops. This procedure is very efficient, and focuses attention quickly on the loops of interest.

Once qualifying loops have been identified, we use an analysis similar in spirit to GVP to determine which variables must be shared. Our criteria, however, is not frequency of execution, but access both outside and inside (in a recursive sense) of any parallel loop, or within two or more parallel loops. Each variable meeting this criterion is considered shared, and is transformed as laid out in Section 4. Variables which are accessed exclusively outside of any parallel loop need no action, and variables accessed only inside of one particular loop should be defined only in the client application which performs that loop's iterations. The mechanics of this shared variable test, as well as the construction of each client, is described next.

3 Controlling Communication Costs for NOW

Once this loop is located and its parallelism verified, we must be particularly meticulous in minimizing the communications costs, which on a NOW are expected to be rather high. In the following two subsections we show a methodology for the custom generation of a scheduling server and cooperative clients which minimize both data and code communication, assuring maximal performance within the constraints of our model.

3.1 Minimal Shared Memory by Memory Classification Analysis

The minimal set of shared memory locations are those which are accessed both within and without a parallel step. We call these locations collectively *shared memory*, and in the following section we introduce a transformation methodology to ensure that the values in these locations are made available at thread startup and new values are returned to the server at thread termination. It turns out that the information needed to determine this set of memory elements precisely is computed by existing parallelization analysis techniques.

Memory Classification Analysis (MCA) is a recent technique for precisely (up to conservative approximation) describing the memory reference behavior of any code section. It was first introduced for Fortran arrays by Hoeflinger[Hoe98], and recently extended by the author[Sch99] to deal with the nested and recursive structures characteristic of C code.

MCA is a symbolic analysis which operates in a single pass over the program, constructing Read-Only, Write-First and Read/Write sets for each statement. Access summaries are generated for straight-line code, conditionals, loops and

to the most important loops in the program and their problems, instead of making the user decide which loops should be parallelized. It should then be a fairly easy task for the user to remove or modify the offending code.

procedure calls by the appropriate merging operations defined on these sets. In order to precisely determine the shared data, then, all that is needed is a final intersection of these sets for code inside the parallel loop with the merged sets of the rest of the program code. Of course, since this is an interprocedural analysis, any code executed through sequences of function calls will also be considered.

Furthermore, our extensions of the basic MCA technique include a methodology for the classification of heap allocated as well as static data. Using this technique, we can classify every allocation statement as to whether it creates shared or unshared blocks, and modify it as appropriate to allocate memory from a shared segment.

3.2 Minimal Shared Code by Code Classification Analysis

We define a Code Classification Analysis, analogous to MCA, but vastly simpler. The general idea is to sweep through the program, associating every statement with the functions that it calls, and summarizing appropriately. With this information, we can generate custom server and client functions which contain only the functions relevant to each. This is particularly critical for clients, because today's networks don't implement demand-paging for code as they do for data, which means that large clients can take a substantial performance hit at the start of a parallel step.

It is true that some caution must be exercised for both virtual functions and functions whose addresses have been taken to ensure that these function addresses are consistent between the server and all clients. However, these functions tend to be quite small, so that inserting all of these as a common program preamble into the server and all client files should be quite inexpensive.

All datatype declarations, of course, can be copied indiscriminantly, as they take up no room in the (stripped) executable.

4 C++: A Universal Parallel Language

A persistent difficulty associated with the use of all parallelizing compilers is that the output code is generated for a specific parallel platform, whether it be a particular multiprocessor, network DSM, or popular message-passing library. This scheme has several disadvantages. Firstly, users' choice of parallel platform is constrained by the model(s) supported by the parallelizing compiler. This encourages the incorporation of support for a multiplicity of parallel models within the compiler, already a highly complex (and often fragile) piece of software. Furthermore, users wanting to run the code on several different targets are forced to generate (sometimes at high cost) and maintain separate versions for each platform, an onerous task. In C++, at least, there *is* a better way.

There are two primary concepts which need to be specified in the parallelization of a sequential code: the loops or code sections which are to be executed in parallel, and the declarations and uses of shared data. We show how each of these may be done *completely portably* in C++. Furthermore, we show that the various scheduling and memory sharing policies can be effectively encapsulated in a library, providing users with unlimited options in the parallel execution of their programs, simply by linking in different support libraries. To be sure, some memory models perform best when associated with particular program parallelization styles, but the value of the portability and modularity of this approach is indisputable.

4.1 Parallelism Specification

Specification of parallelism is done by wrapping up the relevant sections of code into independently executable functions which are then packaged together with all code and data on which the sections depend into independently executable "mini-apps." The names of these concurrently executable mini-apps are then written into the program as arguments to a scheduling call. If the code is the body of a parallelized loop, a specification of the index variable sequence is also passed to the scheduler, which can then call the proper mini-app with appropriate arguments. Stack variables and parameters are transformed into global variables (with appropriate name clash resolution) which are assigned to before the scheduler is called, and (perhaps) read from after the scheduler terminates.

So far, we have not appealed to any specific C++ features; all of the foregoing works just as well in C. In the next subsection, we outline the special C++ property which makes ultra-portable parallel programming possible.

4.2 C++ Shared Memory Declaration and Use

The general procedure for sharing data is the mapping of shared variables to a special shared segment which is protected through operating system page fault mechanisms. This provides the flexibility to implement a wide variety of shared memory semantics.

The problem with this scheme is that there is no language-independent way of mapping particular variables to particular storage areas, forcing all parallelizing compilers to generate platform-dependent code. Indeed, this has brought about the design of a great variety of incompatible syntaxes for the specification of shared data.

We solve this problem by adding one level of indirection to every shared variable and initializing it to point to an appropriately-sized chunk allocated from the shared memory segment. This works fine in C, as long as no shared variables happen to be static. But global and static variables in C must have static (constant) initializers! To be sure, some hackery is possible, such as collecting all the proper initializations in a preamble to `main()`, but this is not a general solution. Problems would arise when the values or addresses of global shared variables are used in other global definitions, not to mention the complications of dealing with shared variables defined in multiple files.[5]

[5] There is certainly no general solution for C++, which does not specify the order in which the global initializers defined in different source files are executed. Any scheme

This is where the special feature of C++ comes to play. In C++, *any variable can have dynamic initialization*. Which means that any variable can be initialized at definition to point into shared memory. Furthermore, the language provides direct support for this chicanery with the placement `new` operator which can direct the allocation to a particular *shared arena*[Str97]. Accesses to this memory use an additional indirection, and address operators are simply removed. Now that the basic infrastructure is in place, we just need a way of properly handling initializations that may be associated with shared variables.

Any variable which is either scalar or initialized with a constructor has a simple mapping to a shared declaration by the simple insertion of the placement `new` operator in front of the constructor call. The only difficulty is with aggregates initialized using the aggregate initialization syntax. This, however, is easily dealt with (if at the minor expense of some program namespace pollution) by appeal to the general expressional nature of C ad C++ initializers. Basically, a garbage declaration is inserted right after the shared variable declaration, whose initialization expression is a parenthesized, comma separated list of assignments to the individual aggregate elements, and terminated with a zero (or any value of the appropriate type for the garbage declaration).

We have described a natural and platform-independent method of specifying parallelism and defining, initializing and accessing shared memory in C++. Using this methodology, many different scheduling and shared memory policies can be implemented by linking in particular versions of the placement `new` operator and the relevant scheduling functions. In the next section, we describe a surprisingly simple and effective scheme based on the Network File System[PJS+94] (NFS) and memory-mapped files.

5 Sharing Memory Over NFS

The shared arena used in the placement `new` shared memory definitions is actually mapped to a disk file. When the scheduler spawns off a client, a copy of this file is created over NFS. As the client starts up, it will execute a similar sequence of shared memory allocation calls to set the shared variable pointers to the identical addresses in shared memory, which is likewise mapped to this shared memory file copy, but without any initialization. As execution proceeds, some or all of the data pages in the disk file will be paged in. Any page which is written to is backed up beforehand, and at the client's termination a set of diffs can be sent back to the scheduler. When the set of diffs from each client has been collected, they are committed to the official copy of the shared memory, with conflicts resolved in favor of the logically later writer.

In the past, many different groups have tried unsuccessfully to harness NFS as a DSM manager. In fact, Minnich has developed his own souped up version of NFS, Mether-NFS[Min93], which fills the gaps in NFS to make it a powerful communication and synchronization subsystem. Minnich contends that NFS

which tries to collect global definitions from different files together must take a stand on this issue, thereby introducing a platform dependence.

was never designed to support shared memory, and presents several compelling arguments for the necessity of a more powerful drop-in replacement:

- NFS client code does not differentiate between read and write faults.
- NFS server code does not ensure that only one process is writing a page at a time.
- The only coherency mode supported is completely incoherent.
- More than one page size is needed.
- A mechanism for delivering a writable copy of a page to holders of read-only copies is needed.
- A mechanism for direct application-to-application synchronization is needed.

However, all of these features go unused in the coarse, independent-grain parallelization model which characterizes some of today's most powerful systems. So it appears that the NFS feature set, once dismissed as insufficient for the needs of DSM, needs another look in light of today's DSM needs.

6 Unified Shared Memory Allocation

We have designed our shared variable definition transformation in a manner which does not in any way interfere with original program semantics. We have developed a simple and effective methodology for allocating shared memory and communicating it over workstation networks. Here we describe in detail how memory allocation is done at the server and each client.

6.1 Shared Memory Address Assignment at the Server

As described above, shared memory operations may take place either implicitly in the initialization of shared variables, or explicitly when an ordinary allocation statement whose heap blocks have been determined to have shared memory semantics is modified to use the same placement `new` operator. With this model, we unify both static and dynamic memory allocation with regard to shared memory. All of this memory, because it is mapped to the same disk file, can easily be copied and supplied to every client.

6.2 Shared Memory Address Assignment at the Client

Our task in this subsection is to describe in detail how the shared variables used at the client side are correctly mapped to the identical addresses originally assigned to them at the server side. This would be trivial if we could arrange for identical sequence of shared variable allocation calls to be executed at the client as were done at the server. (Although the actual placement `new` operator itself could not be used because it would insist on calling a constructor, a similar allocator is called.)

However, we do encounter some complications. For one thing, the original collection of shared variables could have been defined in multiple source files. In this case, the order of their initializations is undefined, and therefore cannot be duplicated within the client application. (This problem is a consequence of our insistence on creating custom client executables for each parallel step.[6])

A different problem occurs even when all shared variables are defined in a single source file. This has to do with shared heap allocations occurring in between shared variable allocations. Because these shared allocations only occur within the server program, they will throw off future shared variable address calculations which must occur within the client as well.

One way of solving these problems is to use a different shared arena/file for each of the source files in which shared variables are defined. One additional arena/file would also be allocated for objects on the shared heap. However, as this significantly multiplies file management operations and/or introduces artificial limits on the size of each of these segments, we choose instead to tabulate within the parallelizer the shared address to which each shared variable is to point.

The basic idea is to statically compute every shared address and explicitly initialize every shared variable with the appropriate address, both in the server and in the client applications. (Shared heap allocations, however, would still be allocated dynamically from the shared arena which logically begins following the memory occupied by the statically shared variables.) In order to prevent platform dependences from being manifest in the generated code, however, these "hard" addresses could be specified symbolically in terms of the `sizeof` the various constituent datatypes.

7 Scheduling

The abundance of node faults characteristic of real NOW platforms could easily cripple any large-scale parallel computation. This problem can be effectively dealt with through Idempotent Eager Scheduling[AKPR93]. In this model, multiple copies of idempotent computations are started on all available nodes, and the first successfully terminated client has results committed; all others are killed. In our independent parallel grain model, all clients are of course idempotent. Therefore, this technique may be directly applied.

Idempotent Eager Scheduling not only gracefully supports node faults, but the introduction of new nodes as well. One way of providing this feature is this is to keep a list of machines which can be used in a disk file which is reread by the server when changed, or providing some type of GUI for introducing the new nodes.

Furthermore, for the scheduling of parallel loops, we can draw on the work of Yan, et al.[YJZ97] for an optimal scheduling policy. They describe how to adaptively adjust loop scheduling granularities to match the particular runtime workload characteristics of the iterations.

[6] There seems to be no guarantee even that within the identical compiler and platform the linkage order will be the same for two different applications[Str97]

8 Caveats and Extensions/Limitations

Although we expect our model to support the parallelization of a wide variety of codes, there are some points which must be kept in mind.

In the model we describe, it is essential that the shared memory file be mapped to the identical memory address at all nodes. This is usually not problematic if the original address is "sufficiently high" not to interfere with anything which may already be mapped.

It must be noted that the reason we cannot parallelize loops which allocate memory which is not also deallocated within the loop is that we cannot coordinate the different threads of execution in our independent-grain model to guarantee non-overlapping allocations from the shared arena/file. At least two workarounds for this restriction are possible. One is actually synchronizing all shared memory allocations at the server, but this would detract from the simplicity of our model. The other is to allow shared allocations at the client, but to resolve them at the end of a parallel step, moving things around in shared memory, if necessary. A sweep of memory (such as is used in garbage-collectors) could then find and "correct" pointers into shared memory to their new values. However, while in the garbage-collector routine incorrect memory pointer identification could result at worse in uncleared memory, applying this procedure safely here could only be done in conjunction with sophisticated type inference[OJ96].

With some additional effort, this framework could be constructed on a heterogeneous NOW platform. A very precise datatype memory layout will be required to translate memory images across platforms. Data and function pointers should be able to be translated as well. Of course, type inference would be extremely helpful here as well.

Furthermore, although the semantics we have described for stack allocated shared variables turns them into globals and renders recursive calls of the parallelized function incorrect, recursion could be supported by the explicit incorporation of a shared variable stack into the shared segment. However, because these stack variables would then not have static addresses, the technique we described above for ensuring identical addresses for statically shared memory in the server and clients would need to be modified.

We have only described an architecture which can fairly accurately flag and transform parallelizeable loops. An important enhancement would be to use loop distribution[Wol96] or other sophisticated transformation techniques to automatically produce maximally sized parallelizeable loop subsections from nominally unparallelizeable loops. For example, a loop whose iterations are computationally complex but begin with an input statement can be split up into one input loop followed by one computation loop, the latter of which can be parallelized. Even loop-carried flow-dependences can sometimes be broken in this manner.

9 Related Work

This work unifies the fields of distributed systems and automatic parallelization. Previous distributed systems, such as Calypso[Bar99] and Chime[SD], require explicit programmer annotations for effective parallelization, making the porting of large existing codes impractical. On the other hand, Coelho[Coe94] has demonstrated that naive application of automatic parallelization techniques to the NOW setting leads to disappointing performance, with the network quickly emerging as the bottleneck. We show that by combining the best ideas in both fields, an extremely powerful parallelization model emerges.

10 Conclusion

We have developed a remarkably simple, flexible, effective and efficient model for the automatic and portable construction of parallel versions of sequential applications targeted for today's cheapest and most popular platform, networks of commodity workstations. We have demonstrated the independent value of each of these ideas and the particular utility of their combination.

We have shown how each of the three issues raised in Cook, et al.[CPW94] as impediments to widespread acceptance of parallelization are addressed in our model. Our model:

- minimizes the need for expert parallel programmers by focusing the user's attention on the parallelism disqualifiers of the most important loops
- demonstrates how the most popular hardware, software and language can be used together in an extremely simple system with unprecedented flexibility and power
- eases debugging by retaining simple sequential semantics in the generated parallel program as well as maximizing programmer recognition of the generated code by maintaining almost all original program structure

Furthermore, we have done all this without significantly mangling the original program source code, which leaves the programmer with a high degree of confidence in the transformed code[Ram98]. A common but dubious feature of parallelizing compilers, especially those that work with C, is indiscriminate "normalizing" of every difficult programming construct into lower-level code. In two companion papers[Sch98, Sch99], we show how proper analysis can proceed without this undesirable normalization for two of the most significant analysis issues that come up in C.

A further way in which we maintain original code integrity as compared with most parallelizing compilers is to concentrate solely on the coarse-grained optimizations, leaving any memory hierarchy tuning up to the subsequent compiler, which can be arbitrarily chosen to any level of sophistication.

The realization of this model will finally bring parallel programming within everyone's reach.

11 Acknowledgements

The author wishes to acknowledge the thoughtful comments of Zvi Kedem and Arash Baratloo on an earlier draft of this paper.

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grant number CCR-94-11590. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[AKPR93] Y. Aumann, Z. Kedem, K. Palem, and M. Rabin. Highly Efficient Asynchronous Execution of Large-grained Parallel Programs. In *34th IEEE Ann. Symp. on Foundations of Computer Science*, pages 271–280, 1993.

[AW97] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. *Lecture Notes in Computer Science*, 1239:120–??, 1997.

[Bar99] A. Baratloo. *Metacomputing on Commodity Computers.* PhD thesis, New York University, May 1999.

[BDK95] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, 1995.

[CHS95] P. Carini, M. Hind, and H. Srinivasan. Flow-Sensitive Interprocedural Type Analysis for C++. Technical Report 20267, IBM Corporation, November 1995.

[Coe94] F. Coelho. Experiments with HPF compilation for a network of workstations. *Lecture Notes in Computer Science*, 797:423–??, 1994.

[CPW94] C. R. Cook, C. M. Pancake, and R. Walpole. Are Expectations for Parallelism Too High? A Survey of Potential Parallel Users. In *Proceedings, Supercomputing '94: Washington, DC, November 14–18, 1994*, pages 126–133, November 1994.

[DKR95] P. Dasgupta, Z. M. Kedem, and M. O. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In *Proceedings of the 15th Intl. Conf on Distributed Computing Systems, to appear*, June 1995.

[EHP98] R. Eigenmann, J. Hoeflinger, , and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):5–23, January 1998.

[EPV97] R. Eigenmann, I. Park, and M. J. Voss. Are parallel workstations the right target for parallelizing compilers? *Lecture Notes in Computer Science*, 1239:300–??, 1997.

[GSW95] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form.

ACM Transactions on Programming Languages and Systems, 17(1):85–122, January 1995.

[HAM+95] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[Hoe98] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classificaiton Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 1998.

[Min93] R. G. Minnich. Mether-nfs: A modified nfs which supports virtual shared memory. In *Proc. of the Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS-IV)*, pages 89–107, September 1993.

[OJ96] R. O'Callahan and D. Jackson. Practical Program Understanding with Type Inference. Technical Report CMU-CS-96-130, School of Computer Science, Carnegie-Mellon University, May 1996.

[PJS+94] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. NFS version 3: Design and implementation. In USENIX Association, editor, *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, pages 137–151, Berkeley, CA, USA, Summer 1994. USENIX.

[Ram98] N. Ramsey. Unparsing Expressions with Prefix and Postfix Operators. *Software—Practice and Experience*, 28(8), August 1998.

[Rep94] T. Reps. Solving demand versions of interprocedural analysis problems. In *Proceedings of the Fifth International Conference on Compiler Construction*, pages 389–403, April 1994.

[Sch98] N. D. Schwartz. Steering Clear of Triples: Deriving the Control Flow Graph Directly from the Abstract Syntax Tree in C Programs. Technical Report TR1998-766, New York University, Department of Computer Science, June 1998.

[Sch99] N. D. Schwartz. Memory Classification Analysis for Recursive C Structures. Technical Report TR1999-776, New York University, Department of Computer Science, Feb 1999.

[SD] S. Sardesai and P. Dasgupta. Chime: A Versatile Distributed Parallel Processing Environment. Available as http://www.eas.asu.edu/ calypso/frames/research_papers/chime/.

[Str97] B. Stroustrup. *The C++ Programming Language*. Reading, Mass.: Addison-Wesley, third edition, 1997.

[SWG94] E. Stoltz, M. Wolfe, and M. P. Gerlek. Constant Propagation: A Fresh, Demand-Driven Look. In *Symposium on Applied Computing*, March 1994.

[Tu95] P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[Wol96] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[YJZ97] Yong Yan, Canming Jin, and Xiaodong Zhang. Adaptively scheduling parallel loops in distributed shared-memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(1):70–81, January 1997.