Parallel Processing Letters

Compiling for Scalable Multiprocessors with Polaris *

YUNHEUNG PAEK
Department of Computer Science
University of Illinois at Urbana-Champaign,
1304 West Springfield Avenue, Urbana, IL 61801, USA
y-paek@cs.uiuc.edu

and

DAVID A. PADUA
Department of Computer Science
University of Illinois at Urbana-Champaign,
1304 West Springfield Avenue, Urbana, IL 61801, USA
padua@cs.uiuc.edu

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Due to the complexity of programming scalable multiprocessors with physically distributed memories, it is onerous to manually generate parallel code for these machines. As a consequence, there has been much research on the development of compiler techniques to simplify programming, to increase reliability, and to reduce development costs. For code generation, a compiler applies a number of transformations in areas such as data privatization, data copying and replication, synchronization, and data and work distribution. In this paper, we discuss our recent work on the development and implementation of a few compiler techniques for some of these transformations. We use Polaris, a parallelizing Fortran restructurer developed at Illinois, as the infrastructure to implement our algorithms. The paper includes experimental results obtained by applying our techniques to several benchmark codes.

Keywords: Parallelizing Compiler, Multiprocessors, Communication

1 Introduction

Scalable shared memory multiprocessors with physically distributed memories, such as the Cray T3D and the Convex SPP, provide a scalable and affordable solution for high performance scientific computing. Accessing data efficiently in these machines often requires somewhat complex program transformations based on accurate analysis of the memory access pattern [1,11]. Some of these distributed

*Research supported in part by Army contract #DABT63-95-C-0097. This work is not necessarily representative of the positions or policies of the Army or the Government.

memory machines have noncoherent caches [3,16]. This multiplies the difficulty of programming these machines because, to compensate for the lack of hardware cache coherence mechanisms, explicit control of cacheability and coherency is critical for fast execution. This programming work can be done manually; however, automatic techniques not only facilitate program development, but also allow the source program to be more readable without sacrificing efficiency. Additionally, more readable programs are easier to debug and maintain. In this paper, we discuss our work at Illinois on the development of compiler techniques for scalable shared memory multiprocessors with noncoherent caches. Specifically, we have developed techniques to translate conventional Fortran programs for efficient parallel execution on the the Cray T3D, the only commercial machine of this class available today.

In fact, we have found in this work that multiprocessors with noncoherent caches have important advantages over their cache coherent counterparts. For example, non-cache coherent machines are easier to scale and are more economical [3]. To optimize communication costs in multiprocessors, it is often necessary for software to have explicit control over data movement. On cache coherent machines, controlling data movement can be cumbersome unless the machine includes mechanisms to override the hardware cache controller. Non-cache coherent machines, by contrast, allow the programmer or the compiler to have explicit and direct control over communications through explicit data movement operations. Having explicit communication control results in other advantages [12], such as a substantial reduction in communication costs from prefetching, data pipelining, and aggregation [11]. In this work, we profit from the fact that the Cray T3D supports fast single-sided communication in the form of PUT/GET primitives. The target language of our translator is CRAFT [6] augmented with libraries that provide single-sided communication primitives. Our work extends the parallelization techniques implemented in the **Polaris** restructurer [4], which was developed by the authors and other researchers at Illinois.

This paper is organized as follows. In Section 2, we briefly introduce the Polaris restructurer. In Section 3, we discuss shared-memory programming in the T3D. In Section 4, we discuss several basic and advanced transformation techniques. We applied these techniques to three benchmark programs, the results of which are presented in Section 5.2. Finally, we conclude our discussion with future research goals in Section 6.

2 Polaris Restructurer

Parallelizing compilers [1,4,9] have been extensively studied during the past twenty years or so. Polaris was developed to overcome limitations in the analysis and transformation techniques implemented in other systems and to be robust enough to allow serious experimental studies. As shown in Figure 1, Polaris contains two distinct phases. The *frontend*, which identifies implicit parallelism, consists of several passes implementing advanced techniques for dependence analysis, induction variable substitution, reduction recognition, and privatization [8,19,20].

Using the parallelism identified by the frontend, the backend generates parallel

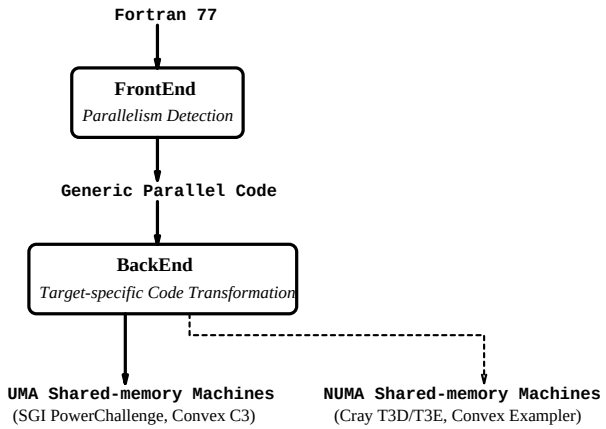

Fig. 1: Components of Polaris

programs for a variety of Uniform Memory Access(UMA) shared-memory machines. We have obtained good speedups for these machines by applying in the backend a relatively simple code generation algorithm that tries to distribute as evenly as possible the parallel work across the processors [18]. In fact, on an extensive collection of programs gathered from the Perfect Benchmarks, SPEC, and other sources, Polaris substantially outperforms the native parallelizer of the SGI [4].

However, as discussed below, for non-cache coherent machines a simple code generation algorithm is not sufficient to achieve reasonable performance. To achieve this goal, it is necessary to deal with data distribution, data movement, and other issues.

3 Shared-memory Programming in the T3D

Like most other distributed-memory machines, the Cray T3D supports the message-passing programming models through libraries such as MPI and PVM [15]. However, unlike the true message-passing machines, the T3D also has several powerful hardware features to efficiently support shared-memory programming [16] on top of physically distributed memory structures. The 3-D torus interconnection network provides high-throughput and low-latency remote memory access. The remote access is only five to six times slower than the local access. Its prefetch queues are designed to fetch remote data, especially scalars, and are effective at hiding the memory access latency. A special barrier network is very useful for global synchronization. In order to compensate for its lack of hardware cache coherency, the T3D has hardware mechanisms to facilitate efficient software control of local memory coherency. Thus, its shared memory access library makes data movement across the system faster than any other software-based implementation [7,14]. In fact, the bandwidth of data copy operations on the T3D is about 1.5 Mwords/sec.

It is possible to access the special hardware features of the T3D through CRAFT, an extension to Fortran 77 that includes several data parallel programming features

from Fortran 90 as well as directives to control parallelism and data placement. CRAFT follows the Single-Program Multiple-Data(SPMD) model and operates on a shared address space. In the work reported below, each CRAFT process was allocated to a physically separate processor. Data objects can be declared as shared or private. Shared data can be distributed across memory using directives similar to those made popular by Fortran D and Vienna Fortran [5,11]. CRAFT uses `:block` for block distribution and `:block(`N`)` for cyclic distribution. CRAFT also includes several explicit synchronization mechanisms and provides a direct interface to the SHMEM library containing various global memory operations to control caches and single-sided data transfer operations using the PUT/GET model. CRAFT allows the programs to control the hardware very efficiently.

4 Basic Transformations

The T3D backend module we have implemented in Polaris uses the parallelism identified by the frontend to transform the source code into parallel form. This transformation consists of two phases that we call basic and advanced, for lack of better terms. The basic transformation phase generates SPMD parallel code. Using rather simple data flow analysis, the basic transformation phase identifies some data as private. By default, data not identified as private are assumed to be shared. Also, many important semantical differences between CRAFT and Fortran are resolved in this phase. Once the first phase is complete, several advanced techniques are applied in the second phase to improve code quality.

In this section we discuss the basic transformations. An earlier version of the material in this section was presented in [17].

4.1 Shared Subprograms

A *shared subprogram* is a subroutine or a function whose execution requires all processors to participate. A shared subprogram contains a *parallel construct* (e.g., a `doall`), an I/O statement with private variables, a call to other shared subprograms, or any other statements marked with parallel assertions.

As an initialization step of our code transformation, we first identify shared subprograms by traversing the call tree. Then, we isolate all calls to shared functions; that is, we place each invocation to each shared function on the right-hand side of a different assignment statement. We do this for several reasons. Let's take as an example the following code segment:

```
X = A(I) + f(A) + g(A).
```

If `X` is a shared variable, then this assignment must be executed serially. When `f` is a shared subprogram but `g` is not, we have to isolate the call to `f` from the original assignment statement as follows:

```
tmp = f(A)                (parallel)
X = A(I) + tmp + g(A)     (sequential).
```

This helps us to control the execution flow of different processors. Only after `f` is isolated is it possible to generate the statements needed to guarantee that only one processor will execute the serial parts and that all processors will execute `f`.

4.2 Parallel Loop Decomposition

We use the same loop decomposition algorithm for the T3D as we use for cache coherent machines. Thus, parallel loops are given block or cyclic schedules where the same number of loop iterations are allocated to each processor element(PE). If several loops are parallel in a multi-nested loop nest, then only one loop in a nest, usually the outermost parallel loop, is transformed into parallel form. The following example illustrates block scheduling.

```
cdir$ parallel (I)          b = N/number_of_PEs
      do I = 1, N           do I = b*my_PE+1, b*(my_PE+1)
         ⋮            ⇒        ⋮
      enddo                 enddo
```

For the case of loops containing reductions, it is necessary to modify the simple strategy we used for shared-memory machines, as shown in the following code example:

```
                                    cdir$ loop preamble
                                          b = M/number_of_PEs
                                          A'(1:N) = 0.0
cdir$ parallel (J)                  cdir$ loop body
      do J = 1, M                         do J = b*my_PE+1,b*(my_PE+1)
         do I = 1, N                         do I =1, N
            ...                                 ...
            A(I) = A(I) + ···     ⇒             A'(I) = A'(I) + ···
            ...                                 ...
         enddo                               enddo
      enddo                               enddo
                                    cdir$ loop postamble
                                          call set_lock(lock)
                                          A(1:N) = A(1:N)+A'(1:N)
                                          call clear_lock(lock)
                                          ...
```

In the example, A′ is a private array of the same size and type as shared array A. The transformed loop consists of three parts: the loop preamble, the loop body, and the loop postamble. In the preamble, A′ is initialized by all processors before the loop execution. After the loop body execution completes, the partial results stored in A′ are gathered into A in the postamble. This parallel version of the loop generally works well for a few processors, but has the intrinsic drawback that the postamble is executed *serially* because it is in a critical section. Table 1 presents an example illustrating that the overall execution time of the parallelized loop with reductions is dominated by its serial postamble on a large number of processors.

To address this problem, we use a different strategy in our implementation. In the previous code example, the private array A′ within each processor is (conceptually) divided into $number\_of\_PEs$ sections. The postamble consists of two phases. First, for $1 \leq i \leq number\_of\_PEs$, the i-th section of all processors is copied into the i-th processor. Then all processors add the $number\_of\_PEs$ sections copied into them. As expected, this approach of parallelizing the postamble has an important impact on performance. This is demonstrated in the example in Table 1.

PE	Preamble (sec)	Loop Body (sec)	Serial Postamble (sec)	Parallel Postamble (sec)
2	0.014	260	0.39	0.10
64	0.017	11	2.7	0.13

Table 1: Measurements of the execution times of the components of loop `INTERF_do1000` in MDG

4.3 SPMDization

We implemented a relatively simple method to generate parallel threads for the T3D, as shown in the example code:

```
                                        subroutine foo(···)
                                        <declarations>
                                        Initialization for all
subroutine foo(···)                     if (slave) goto wait1
<declarations>                          serial regions
serial regions                  wait1:  call barrier()
[a parallel construct]          ⇒       [a parallel construct]
serial regions                          call barrier()
[a parallel construct]                  if (slave) goto wait2
serial regions                          serial regions
···                             wait2:  call barrier()
                                        [a parallel construct]
                                        call barrier()
                                        if (slave) goto ···
                                        serial regions
                                        ···
```

One of the threads, designated as the master, executes all sequential regions. Others, the slaves, wait at barriers while the master is in a sequential region. When the master hits a barrier preceding a parallel region, the slaves are released to participate in the computation. The barriers used to control the flow of execution of masters and slaves are illustrated in the following code.

Due to the T3D's special hardware for global barrier synchronization, the total overhead incurred by barriers has little impact on the performance, according to our experiments [17]. Thus, in this paper, we do not discuss any techniques to reduce barrier overhead.

4.4 Data Declaration

In CRAFT, a variable must be declared as either shared or private. Based on simple inter-procedural data flow analysis, we identify as procedure private the variables that are not used in parallel regions. Also, the privatizer, one of the frontend passes in Polaris, identifies loop private data [20]. Both classes of private variables have to be declared as private in CRAFT, as shown in Figure 2.

Shared arrays must be explicitly distributed by the software. For simplicity, our current implementation automatically applies block distribution to all dimensions of all shared arrays. In Section 5, we discuss further how to deal with these issues to improve performance based on data flow analyses.

```
      subroutine foo
cdir$ private(procedure private variables)
      ...
      X = 3
      ...
      do I = L, U
cdir$ private(loop private variables)
         ...
      enddo
```

Fig. 2: Declaration of privatizable variables in Polaris

4.5 Compatibility Problems

MPP Fortran extensions, such as CRAFT and HPF [10], help the user to attain high performance through distribution of data while maintaining compatibility with conventional Fortran 77 or Fortran 90. It is very difficult to achieve total compatibility, however. CRAFT, therefore, diverts from several conventional language features in the name of performance.

One of the major differences is that CRAFT does not follow Fortran's *sequence/storage association rules* for shared data. For instance, suppose a shared array `A` is declared as `A(N,N)`. In Fortran, `A(N,i)` and `A(1,i+1)` are adjacent in memory. This is not necessarily true in CRAFT. Another major restriction is that each dimension size of a shared array in CRAFT must be a power of 2. This implies that most shared arrays must be expanded by the compiler to the next power of 2 in every dimension. This causes serious compatibility problems, especially for multidimensional arrays that are aliased.

We have developed translation algorithms to overcome all these restrictions when we translate Fortran to CRAFT. The algorithms include linearization, renaming, data replication, array reshaping and procedure cloning. Details of these algorithms and other compatibility problems are given in [17].

5 Data Copying

The basic transformations try only to generate a correct parallel program and to apply a few optimizations. Thus, they do not do anything with cache exploitation and data distribution. To obtain reasonable performance, we have to deal with these issues.

Although the causes of performance degradation of parallel programs in the T3D vary depending on the applications, our analyses of several programs from the Perfect and SPEC benchmarks [17] show that there are two primary factors that prevent the achievement of maximum theoretical performance. The first factor is that shared data is not cached in the T3D, which results in loss of performance whenever the computation uses shared data repetitively. We call this the *cache bypassing penalty.* The second factor is communication overhead caused by remote memory accesses. In this section, we discuss the techniques we use to overcome these two main performance degradation factors and present some performance numbers.

5.1 *Shared Data Copying Scheme*

To deal with the cache bypassing penalty, we developed the *shared data copying* (also called *copy-in/copy-out*) scheme that was applied for earlier hierarchical global memory systems [13]. In the scheme, shared memory is used as a repository of values for private memory, as shown in Figure 3. Before a parallel loop starts, the processors copy all that is used in the loop from shared memory into private memory. After the loop execution completes, the processors copy the results back to shared memory so that all the processors have access to the results. By doing so, most of the work is done on private variables.

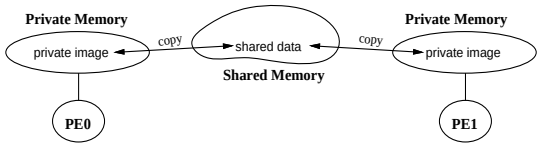

Fig. 3: Shared data copying scheme

Our strategy identifies the array regions to be copied and collects all array regions used in a loop. It classifies the regions into four classes: MUST_WRITE, MAY_WRITE, MUST_READ, and MAY_READ. The description of these regions is appended to the loop in the form of assertions that direct the generation of data copying operations. The following example shows how this region information is converted to PUT/GET operations. In the example, V is a shared array and V′ is the corresponding private array.

```
subroutine f(V,N)                  subroutine f(V,N)
real V(*)                             ...
   ...                             alloc(V'(1:N))
<MUST_READ V(1:N)>                 get(V(1:N),V'(1:N))
doall J = ...                      doall J = ...
   do I = 1, N          ⇒             do I = 1, N
      ... = V(I)                        ... = V'(I)
         ⋮                                  ⋮
```

5.2 *Experiments with Data Copying*

The shared data copying scheme in the T3D seems to be an effective way to deliver high performance because of the T3D's efficient hardware mechanisms to support PUT/GET primitives. In [17], we showed that the hand-coded version of this scheme works well in most cases, and that it works particularly well when the data distribution requirements of a program are dynamic. In this section, we report some of our recent experiments with the version of this scheme we implemented in Polaris.

5.2.1 Measuring Cache Bypassing Penalty

Figure 4 shows the speedups obtained after applying the basic transformation techniques only. As expected, the speedups are low in all cases due to the two factors discussed above.

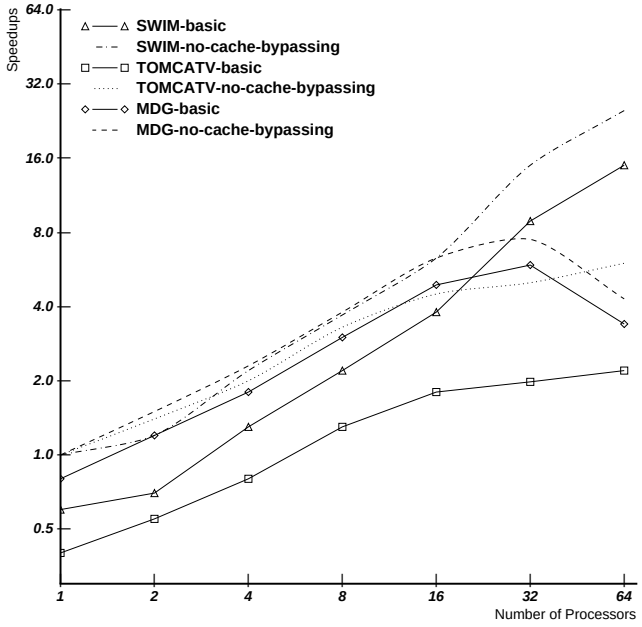

Fig. 4: Before applying the data copying scheme

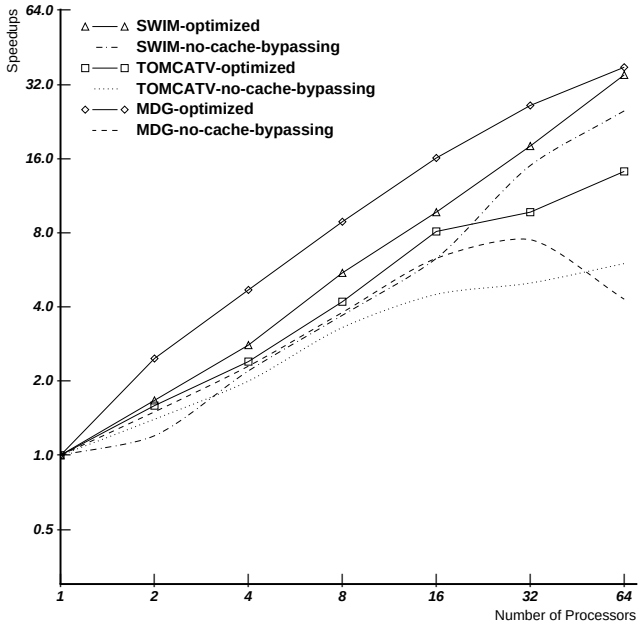

Fig. 5: After applying the data copying scheme

We can measure the effect of the first factor, the cache bypassing penalty, by looking at the speedups for one processor in the figure. In all cases, the speedup is below one, and we see that TOMCATV is most affected by the cache bypassing penalty. The dotted lines in the figure are the predicted speedups that we might achieve for each program after eliminating the penalty. In the following section, we show how we improve the speedups to reach the predicted lines.

5.2.2 Applying the Data Copying Scheme

Figure 5 shows the speedups obtained after applying the shared data copying scheme to the results in Figure 4 with other techniques already applied. It is usually difficult to accurately quantify the effects of each compiler technique on the speedups because performance results from the combination of various techniques in most cases. For instance, the speedups in Figure 5 for MDG are not possible without the dependence analysis techniques that identify the major loops in MDG as parallel; yet, the techniques could not detect those parallel loops without the privatizer's removal of anti and output dependences from the loops. The removal of these anti and output dependences is attributed to the copy-in/copy-out operations [20] using PUT/GET primitives; and, the primitives, in turn, are essential to the shared data copying scheme. Loop fusion has been found useful to remove most of the communication overhead for PUT/GET operations in TOMCATV. These additional loop transformation and parallelization techniques are very important for the data copying scheme.

Nevertheless, we tried to single out the impact of each technique we used in the transformation, and found that the greatest improvement in Figure 5 came from this data copying scheme; that is, the improvement of the speedups is mainly ascribed to the scheme that helps get rid of most of the cache bypassing penalty involved in the original speedups. Figure 5 also shows that the communication cost reduction through block data copying brings even better performance to the actual speedups than the predicted ones in Figure 4.

5.3 Communication Overhead

Although the data copying strategy generally reduces the overall communication overhead by aggregating data to be moved across the network, the strategy itself needs communication for data copying. Therefore, the heedless use of the strategy might not significantly reduce the total communication overhead; in fact, it may even increase the overhead, especially with a large number of processors.

Our current implementation of the data copying scheme is based only on intra-loop analysis. In many cases, a naive implementation of our strategy will lead to copying back and forth almost the same array regions many times between loop nests. This could slow down the program substantially despite the fast PUT/GET implementation in the T3D. Therefore, we have found that we need to apply other techniques to improve our copying scheme in the future. These techniques would be based on inter-loop data region analysis.

We may further reduce the overhead by using optimization techniques similar

to those used in message passing models [11]. For instance, we found that several small data blocks often can be merged into a larger block to reduce the number of transmitted data blocks. Although there are some technical problems in the current implementation of CRAFT, we hope to be able to pipeline the data blocks to hide the latency. Data distribution and work distribution [2] are additional important issues in this matter.

Other strategies to reduce the increased communication overhead include traditional loop transformation techniques, such as loop interchange, loop fusion, and loop distribution. We plan to use each of these in the near future to further improve the performance.

6 Future Work and Conclusions

We implemented in Polaris a pass to generate code for the T3D and we evaluated its effectiveness on three programs from the Perfect and SPEC benchmarks. The preliminary performance that we reported in [17] was unsatisfactory. But, the shared data copying scheme removed the most important performance degradation factors. The experiments presented in Section 5 revealed that this scheme is effective on these applications. However, it also revealed that there is much room for improvement. Our work is still in progress and this paper presented some of our early experience of this work. We will pursue research to extend our work to more general classes of applications.

The data copying scheme requires an accurate data access region analysis. We currently use the access region information generated by Polaris' frontend passes [20]. However, we need a more accurate data access region analysis to improve our opportunity to optimize the communication incurred by the copying scheme.

We do not yet have clear solutions for data and work partitions, which are among the most important issues for scalable multiprocessor machines. Our work will focus on solving these tasks. Eventually, all these solutions and methods will be implemented in Polaris.

Acknowledgements

We would like to thank the Cray Research Inc. and the Pittsburgh Supercomputing Center for granting machine times for the experiments reported in this paper.

References

1. P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.
2. R. Barua, D. Kranz, and A. Agarwal. Communication-Minimal Partitioning of Parallel Loops and Data Arrays for Cache-Coherent Distributed-Memory Multiprocessors. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1996.
3. D. Bernstein, M. Breternitz, A. Gheith, and B. Mendelson. Solutions and Debugging for Data Consistency in Multiprocessors with Noncoherent Caches. *International Journal of Parallel Programming*, 23(1):83–103, 1995.

4. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

5. B. Chapman, P. Mehrota, and H. Zima. Programming in Vienna Fortran 90. *Scientific Programming*, 1(1):51–59, Fall 1992.

6. Cray Research Inc. *CRAY MPP Fortran Reference Manual*, 1993.

7. D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parallel Programming in Split-C. *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

8. J. Grout. Inline Expansion for the Polaris Research Compiler. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1995.

9. M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maimizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

10. High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

11. S. Hiranandani, Kennedy K, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, pages 27–45, 1994.

12. R. Larus. Compiling for shared-memory and message-passing computer. *ACM Letters on Programming Languages and Systems*, 1996.

13. S. Lundstorm. A Controllable MIMD Architecture. *Proceedings of the 1980 International Conference of Parallel Processing*, pages 19–27, 1980.

14. J. Nielocha, R. Harrison, and R. Littlefield. Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers. *Proceedings of Supercomputing '94*, pages 340–349, November 1994.

15. Oak Ridge National Laboratory, Oak Ridge, TN. *PVM 3 User's Guide and Reference Manual.*

16. W. Oed. *The Cray Reseach Massively Parallel Processor System CRAY T3D.* Cray Research Inc., 1993.

17. Y. Paek and D. Padua. Automatic Parallelization for Non-cache Coherent Multiprocessors. In *Lecture Notes in Computer Science.* Springer Verlag, New York, New York, August 1996.

18. C. Polychronopoulos. *Parallel Programming and Compilers.* Academic Publishers, MA, 1988.

19. W. Pottenger. Induction Variable Substitution and Reduction Recognition in the Polaris Parallelizing Compiler. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1994.

20. P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, May 1995.