Parallelization of Benchmarks for Scalable Shared-Memory Multiprocessors[*]

Yunheung Paek
New Jersey Institute of Technology
paek@cis.njit.edu

Angeles Navarro Emilio Zapata
University of Málaga
{angeles,ezapata}@ac.uma.es

David Padua
University of Illinois
padua@cs.uiuc.edu

Abstract

This work identifies practical compiling techniques for scalable shared memory machines. For this, we have focused on experimental studies using a real machine and representative codes. In the experiments, we transformed conventional codes to shared memory codes using several existing techniques and ran the output on the target machine to evaluate those techniques and to identify where improvement is needed. Based on the analysis of the results, we developed a few new techniques, and experimented again on the target machine to measure the effectiveness of each one. The results reported in this paper were quite positive, lending useful information to future research on compiler optimizations for existing SSM machines.

1 Introduction

For the past decade, compiler techniques for NUMA machines have been studied extensively, mainly in the context of the well-known Fortran extensions, such as HPF and Vienna Fortran. The best understood techniques for these compilers target *message passing models* [9]. In this context, Send/Receive primitives are generated to access remote data. There has been active research on compiler techniques for determining the best data/work partitioning and for generating efficient Send/Receive operations because these tasks are a tremendous burden for the user.

During the last few years, it has become evident that high-end NUMA architectures are converging towards *scalable shared memory* machines [8, 16, 17], which provide programmers and compilers with more attractive features than traditional message passing machines. In this paper, we discuss a compiler strategy for SSM machines that capitalizes on their salient features.

One of our goals in this project is to study how effectively parallel machines could be programmed when all parallel constructs are generated by the compiler from conventional codes. We believe it is crucial that the experimental evaluation of any compiler study be based on a real machine and real codes. Thus, an important part of this paper is the experimental evaluation of the effectiveness of the translator using six scientific benchmark codes.

Section 2 describes the techniques we used to generate parallel code for SSM machines. Section 3 shows our experimental results on the Cray T3D.

2 Techniques Tested in This Work

The techniques we have tested in this work can be classified largely into three groups: parallelism detection, parallel thread generation, and communication overhead reduction. They are briefly described in this section.

2.1 Detection of Parallelism

Finding parallelism in the source code is the key ingredient of successful compilers for all types of multiprocessors. This section describes the techniques we used to tackle the problem of finding parallelism.

2.1.1 Reduction and Induction Variables

In our work, induction and reduction variables are recognized and eliminated before any other technique for parallelism detection is applied. To handle these variables in this work, we used the existing techniques [14]. Figure 1 shows how the induction variable `D(M)` is replaced by loop indices.

```
SUBROUTINE SUB(M,N)            SUBROUTINE SUB(M,N)
REAL M(100)                    REAL M(100)
M = 0                          M = 0
DO I = 1, N                    DO I = 1, N
 DO J = 1, I                    DO J = 1, I
  M = M + 1                      D(M+J+(I*I-I)/2) = ···
  D(M) = ···           ⇒        ENDDO
 ENDDO                         ENDDO
ENDDO                          M = M+N+(N*N-N)/2
DO K = 1, N                    DO K = 1, N
 ··· = D(K)                     ··· = D(K)
ENDDO                          ENDDO
```

Figure 1. Induction variable substitution

[*]This work is supported in part by U.S. Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract ASC-96-12099; a Partnership Award from IBM; and Spanish M.E.C. under contract TIC96-1125-C03-01. This work is not necessarily representative of the positions or policies of the Army or Government.

2.1.2 Privatization Based on Array Access Analysis

In scientific programs, arrays used as temporary storage often cause memory-related dependences, which usually can be eliminated by *array privatization*. Furthermore, array privatization can help reduce communication costs in SSM machines by placing data into local memory.

In the initial phase of our work, we used the technique developed by Tu [18]. But, we soon found it necessary to find another technique to enable the generation of efficient parallel code for SSM machines in some important cases. In our search for a better technique, we observed that the most important ingredient for the success of array privatization is the accuracy of *array access analysis*. This is, of course, also true for many existing compiler techniques.

The accuracy of array access analysis relies mostly on the *array region descriptor* (ARD) used to summarize the array locations accessed within a section of code. In [13], we discussed the limitations of existing ARDs, which motivated us to develop a new descriptor, called the *linear memory access descriptor* (LMAD).

In the LMAD, accessing an array is viewed as traversing a linear memory space. For example, in the code

```
REAL A(N-1,N)
DO J = 1, M
  DO I = 0, K
    ··· A(2*I+J,J+3) ···
  ENDDO
ENDDO
```

the two-dimensional array access is, in reality, the traversal in a linear memory space starting from the *base address* τ (= the memory location for A(1,4)) all the way to τ+2·K+(M−1)·N (= the location for A(2*K+M,M+3)). From this example, it can be seen that memory traversals are driven by the loop indices I and J. In the LMAD, the memory access driven by a single loop index is characterized by a *stride/span* pair. The stride is the distance in the number of array elements between accesses generated by consecutive values of the index. In the above example, the stride for index I is 2 because the access moves across two elements of A on each iteration of I. Similarly, we can see the stride for J is N. The span is the total element-length which the access traverses when the index iterates its entire range. Again in the example, the span for index I is 2·K, which is the entire distance traversed between iterations 0 and K for a fixed value of J. Similarly, the span of J is calculated to (M−1)·N for the iteration of J from 1 up to M.

The LMAD is the collection of stride/span pairs of all indices involved in the array access and the base offset, which has the general form: $\mathcal{A}^{stride_{i_1}, stride_{i_2}, \cdots, stride_{i_d}}_{span_{i_1},\, span_{i_2}, \cdots,\, span_{i_d}} + \tau$, assuming the access to array $\mathcal{A}$ is driven by d loop indices, i_1, i_2, $\cdots$, i_d. Here, τ is set to the offset of the first access from the beginning of the array. As can be seen in the above example, a single stride/span pair for a loop index (*e.g.*, {1,2·K} for I and {N,(M−1)·N} for J) characterizes the independent access pattern generated by the index iteration. Thus, the LMAD defines the overall pattern of the access driven by all indices. The LMAD summarizing the overall access in the example would be $\mathcal{A}^{1,\mathtt{N}}_{\mathtt{2K},(\mathtt{M}-1)\mathtt{N}}$+3N. The accesses in Figure 1 can be likewise summarized as:

$\mathcal{D}^{1}_{\mathtt{N}-1}$+0 for the region read in the loop, and

$\mathcal{D}^{1,\mathtt{I}}_{\mathtt{I}-1,(\mathtt{N}^2-\mathtt{N})/2}$+M for the region written.

From the LMAD for the write, we can see the memory traversal structure: starting from $\tau = \mathtt{M}$, the assignment statement visits I elements of array D consecutively with stride 1 and jumps the elements with stride I to the element right next to them until it reaches the element at address $\tau+\frac{\mathtt{N}^2+\mathtt{N}}{2}$. Despite the apparent complexity of subscript expressions, this write access is actually a simple stream of accesses in memory from the element D(M) to D(M+(N*N+N)/2) with stride 1, which can be summarized with a simpler LMAD: $\mathcal{D}^{1}_{(\mathtt{N}^2+\mathtt{N})/2}$+M. Using this simpler one, we can compare the write region with the read region for privatization and, additionally by proving M=0, declare array D private to SUB.

In our experiments, we used the LMAD in array access analysis for array privatization, and found that the LMAD indeed helped improve the effectiveness of the technique.

2.1.3 Dependence Analysis

After eliminating memory-related dependences with privatization, we tested dependences in the loops to determine whether the loops were parallel. Most subscript expressions occurring in scientific programs are simple. Thus, primitive dependence tests are usually efficient and accurate at disproving dependence in practice. For this reason, to find parallel loops with those simple access patterns, the *Equality test* and *GCD test* were first applied in our work. Then, for more complex loops, we applied the *Region test* [7, 12]. To find parallelism in a loop, the Region test summarizes the regions of array accesses within the loop, and intersects the regions to determine cross-iteration dependences between the regions. Such region summaries for the Region test are represented with the LMAD.

To handle the situations where insufficient information exists to carry out the dependence testing at compile time, the Region Test enables the generation of constraints for run-time dependence testing. For example, consider the loop intraf_1000 in Figure 2.a. The array elements accessed in the loop are:

1, 2, 3, N, N+1, N+2, 2N, $\cdots$, M+2−N, M, M+1, M+2.

To disprove cross-iteration dependences for the loop, we must show that there is no overlap between array accesses on different iterations of the loop. From this sequence, it is clear that no overlap exists if $\mathtt{N} > 2$. Because the value N is not known, the constraint cannot be proven at compile time. However, using the Region test, a parallel loop can be generated for intraf_1000 under the constraint, $\mathtt{N} > 2$, for run-time testing.

Although parallelizing loops under run-time constraints is not a new idea [2], little research has been done on how to efficiently collect the predicates from the program. The LMAD representation facilitated gathering the predicates. In this example, to generate the LMAD for the accesses to array F, first notice that the accesses are driven by two indices: I and an implicit index, say I', to represent +1 and +2 in the loop. As discussed in Section 2.1.2, the accesses driven by indices I' and I can be characterized by two stride/span pairs: {1,2} for I' and {N,M-1} for I, from which we build the LMAD $\mathcal{F}^{1,\mathrm{N}}_{2,\mathrm{M}-1} + 0$. From the LMAD, we extract a necessary condition for parallelization, namely:

$$\mathtt{N} > 2 \quad \vee \quad 1 > \mathtt{M} - 1,$$

which is obtained by simply comparing the stride of one pair with the span of the other pair. We call this the *non-overlap constraint*. Inside the loop, $1 > \mathtt{M} - 1$ is clearly false, resulting in $\mathtt{N} > 2$ as the non-overlap constraint for this loop.

```
DO I = 1, M, N
  ...
 F(I) = F(I) ···
 F(I+2) = F(I+2) ···
 F(I+1) = F(I+1) ···
ENDDO
```

(a) intraf_1000 from mdg

```
DO J = 1, (M-1+N)/N, 1
 DO I = 1, K, 1
  ··· = D(1-M+I*M+(J-1)*N)
  D(1-M+I*M+(J-1)*N) = ···
 ENDDO
ENDDO
```

(b) ftrvmt_103 from ocean

```
DO I = 0, 2**(M-M/2)-1, 1
 DO J = 1, 2**(M/2), 1
  Y(J+2*I*2**(M/2)) = ···
  Y(J+2**(M/2)+2*I*2**(M/2)) = ···
 ENDDO
ENDDO
```

(c) cfftz_#2 from tfft2

Figure 2. Abbreviated versions of loops from our testing codes in the Perfect and NASA benchmark suites. The notation intraf_1000 stands for the loop with label 1000 within the subroutine intraf. All the loops are obtained after interprocedural value propagation, induction variable substitution, and forward substitution. In particular, inlining is applied to obtain cfftz_#2.

Likewise, we can calculate the LMAD for the accesses of the loop ftrvmt_103 in Figure 2.b:

$$\mathcal{D}^{\mathrm{M,N}}_{(\mathrm{K}-1)\mathrm{M},(\lfloor\frac{\mathrm{M}-1+\mathrm{N}}{\mathrm{N}}\rfloor-1)\mathtt{N}}+0.$$

Using this descriptor, we generate the following necessary condition for parallelization:

$$(\mathtt{K}-1)\cdot\mathtt{M} < \mathtt{N} \quad \vee \quad (\lfloor\frac{\mathrm{M}-1+\mathrm{N}}{\mathrm{N}}\rfloor - 1)\cdot\mathtt{N} < \mathtt{M},$$

which is to be tested at run-time for overlap checking. Parallelizing this loop and other similar loops in ocean, such as ftrvmt_106, ftrvmt_108, ftrvmt_109, is important because they are very time-critical loops. Some loops (e.g., ftrvmt_109) provide enough information for M and N to statically prove the non-overlap constraint so that they can be parallelized without run-time tests. This example also shows the importance of *symbolic range analysis techniques* [2]. In fact, with powerful analysis techniques, it is possible to disprove dependence at compile time by showing that the condition above is always true even though the current implementation of the Region test can not yet perform such an accurate symbolic analysis.

The simplification operations on the LMAD help disprove dependence in loops with very complex subscript expressions. The subscripts in Figure 2.c show complex access patterns that can be simplified. This access pattern is typically found in FFT applications: array accesses with the strides of powers of two. No existing symbolic range analysis technique can handle this pattern due to its complex subscript expressions. To handle this, the Region test first summarizes the accesses for the references as follows:

$$\mathcal{Y}^{1,2c}_{c-1,2^{\mathtt{M}+1}-2c}+0 \quad \text{and} \quad \mathcal{Y}^{1,2c}_{c-1,2^{\mathtt{M}+1}-2c}+c$$

for $c = 2^{\mathrm{M}/2}$. The similarity of these accesses becomes more evident in the LMAD forms: both accesses are *interleaved* and have the same stride/span pairs only with different base addresses. This similarity allows their LMADs to be aggregated into a single LMAD form: $\mathcal{Y}^{1,2c}_{2c-1,2(\mathrm{M}+1)-2c}+0$, which, as shown in Section 2.1.2, is futher simplified into $\mathcal{Y}^{1}_{2(\mathrm{M}+1)-1}+0$. From this simplified access description, it is straightforward to disprove dependence on array Y.

For dependence analysis, we initially used the *Range test* [2, 3], a symbolic extension of Triangular Banerjee's Inequalities test. But, we soon discovered several cases, such as those illustrated in Figure 2, where the test was not effective. The Region test was developed to improve on the accuracy of the Range test.

2.2 Generation of Parallel Threads

Once parallel loops were identified in the source code, a shared-memory program was generated as the target parallel code. Since only loop-level parallelism was exploited in our work, the parallel loops were the only sections of the target code run across parallel threads. For parallel threads, iterations of a parallel loop were stripmined by using the following simple strategy: (1) if several loops in a loop nest were parallel, then the outermost loop was parallelized; (2) if the loop nesting structure was square, a block schedule of iterations was chosen for it; and, (3) if the structure was triangular, a cyclic schedule of iterations was chosen. Serial sections were executed by a designated single thread.

For efficient parallel programming on SSM machines, many commercial and academic shared-memory languages [4, 5, 11] have been developed recently and implemented in existing machines. They contain several constructs that were not supported in languages [6] originally designed for message-passing machines. First, they include Put/Get statements [5, 9, 11] or assignment statements operating on a global (or shared) memory space. These statements are provided to control *asynchronous communication* at the software-level because the SSM machine supports fast asynchronous communication directly in hardware for efficient global memory operations. Second, they include synchronization statements because synchronization is essential to ensure memory coherency when processors access shared memory asynchronously and to control the flow of execution of parallel threads in the target code. Third, they usually provide a mechanism to allow the programmer to explicitly choose memory classes between private data and shared data. For shared data, they provide data distribution directives similar to those in Fortran D and Vienna Fortran because their shared memory is actually built on top of physically distributed memory. Our target code is designed to provide all these programming constructs.

2.3 Communication Optimizations

As discussed above, data objects in our target parallel code are either private or shared. Using privatization, some data objects in the source code are declared as private in the target code. All non-private objects are, by default, declared as shared. Unless all objects are private in our shared-memory program, parallel threads need to communicate to access shared data distributed across the machine. In this work, therefore, our efforts for communication optimization focus on shared arrays accessed in parallel loops.

Asynchronous communication in the SSM machine provides a very efficient way to transfer a large data block between processors. The basic concept of our technique is simple: all shared array accesses within a loop are replaced with private accesses by copying at loop entries all elements of the shared arrays used within the loop into private memory, and by copying at loop exits the updated results in private memory to the original shared arrays. By using this copy-in/copy-out strategy [12], we can take advantage of low-latency high-throughput block data transfer when we copy data between private memory and shared memory. In our target code, Put/Get statements are used to asynchronously transfer data blocks in the machine.

As discussed in [10], the communication patterns in a parallel program greatly influence communication overhead. We classify the patterns into three types: *local*, *frontier*, and *global* communications.

2.3.1 Local Communications

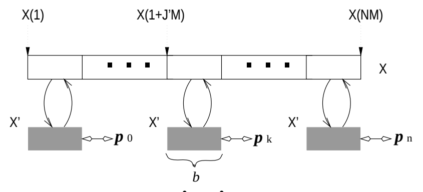
We say a program section, such as a subroutine or a loop, has *local communications* to mean it has arrays that can be distributed such that most of their references can be local to each processor, thereby minimizing the need of communication caused by the arrays in that program section. Figure 3 shows a typical example of local communications in a loop nest. Suppose the J-loop is parallel and array X is declared as shared and block-distributed. That is, subarrays of X, each with b·M consecutive elements, are allocated to every processor's private memory. Then, to match the block data distribution, the compiler would stripmine the loop with block schedule; thereby, allowing all the work to be be done on private array X′ only with no communication.

```
REAL X(N*M)
 ...
DO J = 0, N-1
  ··· = X(1+J*M)
  DO I = 2, M
    ··· = X(I+J*M)
    X(I+J*M) = ···
  ENDDO
ENDDO
```

source sequential code

```
<initialize X′ >
DO J = J′, J′+b-1
    ··· = X′(1+(J-J′)*M)
  DO I = 2, M
    ··· = X′(I+(J-J′)*M)
    X′(I+(J-J′)*M) = ···
  ENDDO
ENDDO
<update X>
```

target parallel code



communication pattern

Figure 3. Source code and its target code in SPMD form, and the illustration of a data distribution for the array for processors p_k's: White boxes represent the shared array sections accessed by processors in the loop, and gray boxes represent their private counterparts where processors work.

Once private references are substituted for shared ones in the loop, Put/Get statements are generated to copy-in/out data between the private and shared arrays. The analysis of array access patterns in loops plays an essential role in the generation of Put/Get statements because the efficiency of Put/Get operations hinges on the accuracy of the analysis that summarizes the array regions to be copied at loop boundaries. We summarize the array regions accessed in the parallel loop after stripmining in the same way that we did for finding parallelism in Section 2.1. For instance, in Figure 3, the read regions for two references to X are summarized and simplified [13] to generate the array region of X in the GET statement as follows:

$$\begin{tikzcd}
X^{1,M}_{M-2,(b-1)M}+J'M+1 \arrow[dr, "\text{aggregate}"'] & & & \\
& X^{1,M}_{M-1,(b-1)M}+J'M \arrow[r, "\text{coalesce}"] & X^{1}_{bM}+J'M \arrow[r, bend left] \arrow[r, bend left=50] \arrow[dr] \arrow[dr, bend right] & X(1+J'M:(J'+b)M:1) \\
X^{M}_{(b-1)M}+J'M \arrow[ur] & & & \oplus \arrow[u]
\end{tikzcd}$$

The region of X′ can be computed by simply shifting the lower limit of X region to 1. Then, *initialize* X′ is implemented with a Get statement to move a single data block:

```
GET(X(1+J′*M:(J′+b)*M:1),X′(1:b*M:1)).
```

For *update* X, we used the LMAD representing the write access to X: $\mathcal{X}^{1,M}_{M-2,(b-1)M}$+J′M+1, which has two stride/span pairs. The region cannot be transferred with a single PUT operation because Put/Get operations are implemented to transfer blocks of data with constant strides one at a time. Thus, we need to convert one of the pairs into an enclosing loop in which a vector of M-1 elements is transferred with stride 1 on every iteration, which results in:

```
DO J = 0, b-1
  PUT(X(2+(J+J′)*M:M+(J+J′)M:1),X′(2+J*M:M+J*M:1))
ENDDO
```

for *update* X. Notice that the number of iterations is computed from the second pair, and the size and stride of the vector are from both pairs.

2.3.2 Frontier Communications

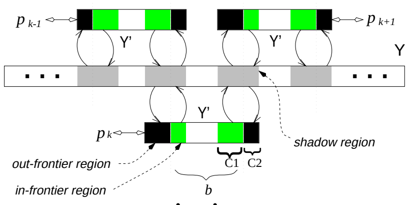
In real programs, a loop often contains an array whose references are only of the form: $i \pm c$, where i is a loop index and c can be arbitrary constants, as shown in Figure 4. In the example, array X is accessed only with three references of the forms: X(I-C1), X(I), and X(I+C2). This pattern of array references results in another communication pattern, which we call *frontier communications*, when the code is compiled and run in parallel on SSM machines.

```
DO J = 1, M
  DO I = C1, N-C2
    ··· = Y(I-C1)+Y(I)
          +Y(I+C2)
  ENDDO
  DO I = C1, N-C2
    Y(I) = ···
  ENDDO
ENDDO
```

source sequential code

```
<initialize Y′ >
DO J = 1, M
  DO I = I′+1, I′+b
    ··· = Y′(I-I′)+Y′(I-I′+C1)
          +Y′(I-I′+C1+C2)
  ENDDO
  DO I = I′+1, I′+b
    Y′(I-I′+C1) = ···
  ENDDO
  <update shadow>
ENDDO
<update Y>
```

target parallel code



communication pattern

Figure 4. Frontier communication pattern: Y is a shared array, and Y′ is a private array that contains the region of Y accessed in the J-loop. Dark areas in Y′ represent out-frontier regions and gray areas represent in-frontier regions. Gray areas in Y represent shadow regions.

Suppose the two inner loops in Figure 4 are both parallel whereas the J-loop is serial. If the inner loops are block-scheduled, neighboring processors have overlapped access regions (called *shadow regions*); thus, they need to communicate to keep their neighbors updated on the results of every iteration of J. The standard data distribution type that handles frontier communications is the *shadow* distribution [6]. In the shadow distribution, an array is first block-distributed so as to allocate a private array to each processor. Extra consecutive private space, which we call *out-frontier* region (OFR), is allocated for the elements that are shared by neighboring processors. The *in-frontier* region (IFR) refers to the elements within the local block that are to be moved to neighboring processors after they are updated. The shadow distribution for array Y would be written as follows (borrowing the syntax of HPF2.0):

```
DISTRIBUTE(BLOCK), SHADOW(C1:C2) :: Y.
```

To generate Put/Get, we first summarize access regions the same way we did in Section 2.3.1. In *initialization*, the initial values of all portions of the shared array corresponding to a private array are transferred by Get statements. In this example, we need a single Get for initializing Y′:

```
GET(Y(I′+1-C1:I′+b+C2:1),Y′(1:b+C1+C2:1)).
```

The *update shadow* operation deals with the intermediate results generated during the loop execution. These results are stored in the frontier areas. In Figure 4, the change in the IFR of processors during the current iteration of the loop J must be copied to the OFR of their neighbors before starting the next iteration. This operation involves two steps, as illustrated in Figure 4. First, all IFRs are written back to the corresponding shadow regions simultaneously. Then, all OFRs are updated with the new results in shadow regions. These two steps must be separated by an explicit barrier to ensure that all writes to shadow regions will complete before any processor tries to read the region. For *update shadow*, we generate the following sequence of statements:

```
PUT(Y(I′+1:I′+C2), Y′(1+C1:C1+C2))
PUT(Y(I′+b+1-C1:I′+b), Y′(1+b:b+C1))
barrier synchronization
GET(Y(I′+1-C1:I′), Y′(1:C1:1))
GET(Y(I′+b+1:I′+b+C2), Y′(C1+b+1:C1+b+C2)).
```

A similar procedure is followed when changes take place in the OFR during the loop execution.

When the loop execution ends, the final results in private arrays are *put* to the shared regions, as shown in Section 2.3.1. For *update* Y above, we would generate

```
PUT(Y(I′+1+C2:I′+b-C1:1),Y′(1+C1+C2:b:1)).
```

2.3.3 Global Communications

We say a loop or subroutine contains *global communications* if it has shared data that needs to be accessed by all processors, thereby requiring data movement under the intervention of the processors. The typical data transfer operations for global communications are *broadcast*, *reduction*, and *gather*/*scatter* operations. In programs involving irregular or dynamic data access patterns and where the majority of data tend to be frequently transferred between processors, we have seen that simple block distribution can be relatively efficient when it is used with those global data transfer operations. One reason is that array accesses in scientific programs are usually contiguous and, thus, block distribution increases the chances of finding a long data block in fewer remote memory modules. Another reason is that the calculation of the target location of the data transfer can be simplified when arrays are distributed in contiguous blocks.

3 Experiments with Benchmarks

In this section, we report the case studies in which six benchmark codes are transformed to the target parallel code written in *Craft* [4], a native Cray shared-memory language, and executed on a Cray T3D system, which consists of processor elements based on DEC 21064 64-bit μ-processor. Each processor contains 1 level on-chip 8 KB direct-mapped data cache and 64 MB local memory.

For these studies, two different parallel versions were generated as targets for each code, as shown in Table 1 where the techniques used for each version are compared. Version 1 was generated using the *Polaris* compiler [3] at Illinois, where most of the techniques discussed in Section 2 are implemented [†]. Version 2 was manually generated by researchers at Malaga [1], based on their past work on data distribution for message passing codes.

optimizations	version 1	version 2
parallelism	privatization w/ LMAD, GCD/equality/region test	by hand analysis
loop scheduling	cyclic,block	cyclic,block
communications	N/A	local,frontier,global
data distribution	block	block,cyclic,shadow, block-cyclic,replicate

Table 1. Comparison of the major optimization strategies used in the two versions of parallel target codes.

We found that the techniques for finding parallelism (see Section 2.1) were effective for the six codes; almost the same loops were identified as parallel in both versions. In particular, the loops with complex access patterns, such as `cfftz_#1/2/3`, `intraf_1000`, and `predic_1000`, can be parallelized by the Region test even though they were originally serialized by the Range test. Also, in both versions, the same loop scheduling strategies (see Section 2.2) were applied.

The only major difference between the two versions was in the distribution of shared arrays and the generation of Put/Get. In Version 1, the compiler block-distributes all shared arrays, and Put/Get generation is intra-loop based; that is, given parallel loops, it analyzes the shared array regions accessed in individual loops, and generates Put/Get statements around each loop independently. In Version 2, communication patterns (see Section 2.3) in the codes were manually analyzed. The analysis results were used to determine the most appropriate data distribution strategies for shared arrays and to generate Put/Get statements accordingly. Therefore, we conclude that by comparing the optimization strategies of these two versions for each code, we can estimate the impact of data distribution and explicit communication control on the T3D.

BDNA: The `bdna` code uses the BIOMOL package to perform molecular dynamic simulations of biomolecules in water. The major routine in the program is the `actfor` subroutine, which consumes 96% of the total program execution time. To parallelize the major loops in `actfor`, Polaris applied privatization (for `actfor_240`) and reduction variable substitution (for `actfor_240/320/500/700`). In particular, the Polaris dependent test parallelized all the important loops in the subroutine. Array privatization and reduction recognition techniques in Polaris were quite effective for `bdna` because they privatized a large number of arrays and scalars, which eliminate substantial communication overhead. All these factors made it possible to maintain good scalability on up to 64 processors, as shown in Figure 5.

[†]In fact, because the LMAD was not fully implemented when this experiment was conducted, some hand transformations additionally had to be applied to the loops where more accurate LMAD-based array analysis was necessary. Its implementation was recently completed [7, 13].

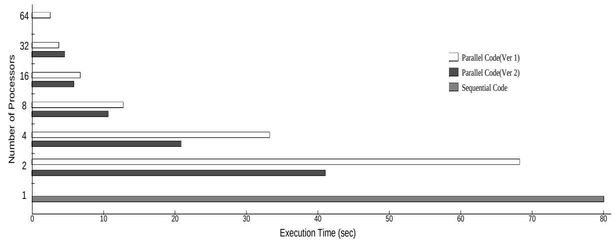


Figure 5. Executions of the sequential `actfor` routine in `bdna` code and its parallel code generated by Polaris and by hand. The accurate speedup result for Version 2 was unavailable on 64 processors.

The main effort in Version 2 was to optimize communication by finding a suitable data distribution. By hand analysis, we found that for arrays `FX`, `FY`, `FZ`, `FSX`, `FSY`, `FSZ`, `TX`, `TY` and `TZ`, cyclic distribution is better because the arrays are accessed within the triangular loops, and for the arrays `FSX`, `FSY`, and `FSZ`, which are aligned as four element blocks with the other, a block-cyclic(4) distribution was suitable.

MDG: The `mdg` code is a driver for molecular dynamic simulation of flexible water molecules in the liquid state at room temperature and pressure. The computations in `interf` subroutine are controlled by a parallel triangular nesting: `interf_1000-interf_1100`. Thus, a cyclic schedule of iterations in this nesting was used to balance the workload. With this cyclic loop scheduling, `interf_1000` accesses sixteen arrays in total. In addition, ten arrays out of the sixteen cause memory-related cross-loop dependences, preventing the compiler from parallelizing this most time-critical loop in `mdg`. The superlinear speedups on up to 16 processors are ascribed to the privatization and reduction recognition techniques that privatized the ten arrays, which not only greatly contributed to reducing communication overhead in the loop, but also parallelized it, along with other major loops such as `poteng_2000` and `intraf_1000`.

In Version 2, `FX`, `FY` and `FZ` were distributed with the BLOCK-CYCLIC distribution type, which resulted in better scalability on more than 16 processors. However, we reached a point of diminishing returns beyond 16 processors. One reason is that `interf_1000` has global communication for reduction variables. In particular, the impact of reductions in small blocks of those three arrays are important because they can eat up to 27% of the `interf` execution time (or equivalently 21% of the `mdg` execution time) on 64 processors.

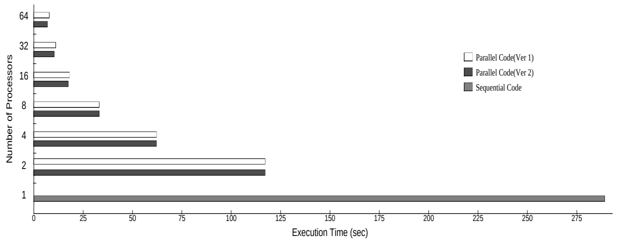


Figure 6. Executions of the sequential `mdg` code and its parallel code generated by Polaris and by hand

SWIM: The `swim` code is a benchmark program from `shalow`, a weather prediction program based on the dynamics of finite-difference models of the shallow-water equations. Figure 7 shows that the current implementation of Polaris can generate a quite efficient parallel code for `swim`. Overall, of the six programs, `swim` has the best scalability on up to 64 processors for two main reasons. First, 99.9% of sequential coverage is parallelized by Polaris. Second, its major loops do not have global communications, which is usually a main cause of scalability degradation on a large number of processors.

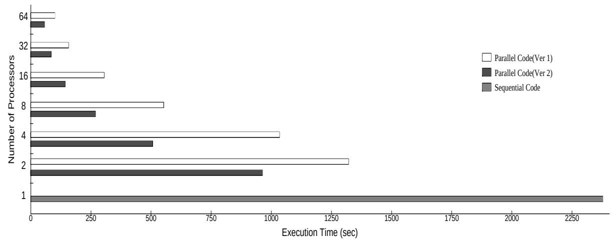


Figure 7. Executions of the sequential `swim` code and its parallel code generated by Polaris and by hand

The major routines contain doubly-nested parallel loops. Their loops access shadow regions, and their communication patterns are frontier communications. Therefore, their communication overhead can be reduced by the technique described in Section 2.3.2. In Version 2, a two-dimensional SHADOW distribution was chosen and the frontier communications in these subroutines were exploited, which allowed almost linear speedups in Version 2, even on 64 processors, as shown in Figure 7. In contrast, the current implementation of Polaris does not handle these communications and only exploits one dimensional parallelism, as stated earlier. This explains the increased performance gap between Version 1 and Version 2 on larger numbers of processors.

TFFT2: The `tfft2` code is an FFT program from NASA which performs real-to-complex, complex-to-complex, and complex-to-real FFTs. The repeated subroutine calls inside loops form a deeply nested loop structure. Polaris, using the dependence analysis based on the LMAD, parallelized more than 95% of sequential coverage of `tfft2`, which enabled the speedups of the parallel code, as shown in Figure 8. In particular, array privatization was highly effective for the two major loops, `cfftz_#1` and `cfftz_#3`. Loops in `tfft_120` and three subroutines `transa/b/c` have global communications for reductions and matrix transposition. This is a serious limitation for speedups of the parallel code because these loops account for about 14% of sequential execution. The results in Figure 8 confirmed this analytical conclusion. Also, we found that 5.2% of sequential time is spent in the loops that have frontier communications.

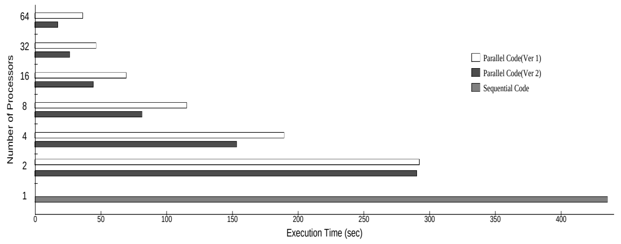


Figure 8. Executions of the sequential `tfft2` code and its parallel code generated by Polaris and by hand

Despite these difficulties of `tfft2`, we have achieved reasonable scalability in Version 2. We found this is mainly due to the selection of a BLOCK_CYCLIC data distribution, which gets local computation for the 77% of sequential coverage and provides the opportunity to exploit parallelism without communications. Also in Version 2, the performance improved further in several ways. First, the frontier communications were exploited. Second, three routines, `transa/b/c`, were manually optimized by using *tiling* [15] to exploit the small 8KB on-chip cache in the T3D. Third, array `S`, defined in the `randp` subroutine, was replicated for removing communication caused by it.

TOMCATV: The `tomcatv` code is a program that uses finite difference approximation for a fluid dynamics simulation. Frontier communications are the major communication pattern of the loop `main_140`, which is the computational kernel of `tomcatv`, accounting for more than 93% of the sequential execution time.

As shown in Figure 9, the scalability of the speedups of Version 1 is limited because, similar to the case of `swim`, Polaris currently cannot specialize in the optimization of frontier communications. The significant improvement of the speedups for Version 2 was made by exploiting shadow distribution, and frontier communications for the rows of arrays `X` and `Y`.

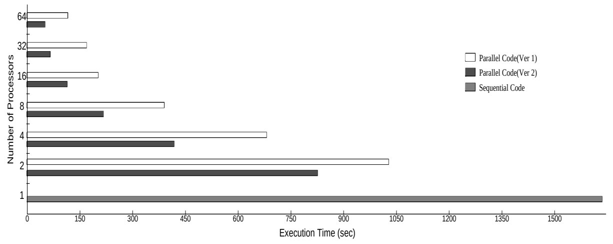


Figure 9. Executions of the sequential `tomcatv` code and its parallel code generated by Polaris and by hand

Further improvement was made in Version 2 with additional strategies. First, we exploited the data replication strategy for data distribution for other arrays, such as `RXM` and `RYM` which are small and are used as reduction variables. Also, in the first loop nested inside `main_140` we chose `main_50` as parallel instead of the outer loop `main_60`. By doing this, the arrays `X`, `Y`, `RX`, and `RY` were accessed in parallel by rows during the whole execution.

TRFD: The `trfd` is a quantum mechanics kernel where the major time-consuming subroutine `olda` accounts for 90% of the total sequential execution time.

Array privatization is a very crucial technique for optimization of the major loops, `olda_100/300`. In particular, Polaris successfully handled non-affine subscript expressions to parallelize these loops, as discussed in Section 2.1.

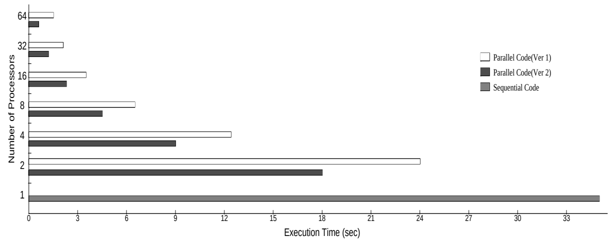


Figure 10. Executions of the sequential `trfd` code and its parallel code generated by Polaris and by hand

In Version 2, the work array `X` was partitioned into three matrices, (`X(I00), X(I20), X(I30)`), and a vector (`X(I10)`). One key aspect was that `X(I20)` and `X(I10)` and `X(I00)` were replicated on each processor to avoid any communications. The dimensions of these matrices and vectors are very small compared with `X(I30)`, the main matrix in `trfd`. So the only matrix that was distributed across processors in Version 2 is `X(I30)`. Choosing a BLOCK_CYCLIC data distribution for it improved the performance of Version 1 by approximately 50%, as illustrated in Figure 10.

4 Conclusion

In this paper, we presented experimental results that provide some promise that it would be possible for the compiler to automatically generate parallel code for SSM machines with relatively simple techniques, while still producing reasonable speedups for some actual applications on real machines.

In this work, the comparison with manual optimizations showed that data distribution or other communication minimization issues are still important for the T3D, but also that these techniques are not necessarily too sophisticated. With only a few extra techniques, such as reduction of communication overhead by recognizing communication patterns and the array access pattern analysis, we manually achieved nearly ideal speedups for several applications.

References

[1] G. Bandera, M. Ujaldn, M. Trenas, and E. Zapata. The Sparse Cyclic Distribution against its Dense Counterparts. *11th International Parallel Processing Symposium*, pages 638–642, April 1997.

[2] W. Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, June 1995.

[3] 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.

[4] Cray Research Inc. *CRAY MPP Fortran Reference Manual*, 1993.

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

[6] High Performance Fortran Forum. *High Performance Fortran Language Specification, ver. 2*, January 1997.

[7] J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, July 1998.

[8] HP CONVEX division. *HP-CONVEX Exemplar S-Class and X-Class Systems Overview*, 1996.

[9] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, January 12, 1996.

[10] A. Navarro, Y. Paek, E. Zapata, and D. Padua. Compiler Techniques for Effective Communication on Distributed-Memory Multiprocessors. *International Conference on Parallel Processing*, August 1997.

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

[12] Y. Paek. *Automatic Parallelization for Distributed Memory Machines Based on Access Region Analysis*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, April 1997.

[13] Y. Paek, J. Hoeflinger, and D. Padua. Simplication of Array Access Patterns for Compiler Optimizations. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[14] W. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *ACM International Conference on Supercomputing*, pages 444–448, July 1995.

[15] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines. *Supercomputing '95*, pages 111–120, 1991.

[16] S. Scott. Synchronization and Communication in the T3E Multiprocessor. *International Conference on Architechtural Support for Programming Language and Operating Systems*, pages 26–36, October 1996.

[17] SGI/Cray Research Inc. *Origin and Onyx2 Programmer's Reference Manual*, 1996.

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