# Automatic Parallelization for Non-cache Coherent Multiprocessors *

Yunheung Paek David A. Padua

Department of Computer Science
University of Illinois at Urbana-Champaign,
1304 West Springfield Avenue,
Urbana, IL 61801, USA
{paek,padua}@csrd.uiuc.edu

**Abstract**

Although much work has been done on parallelizing compilers for cache coherent shared memory multiprocessors and message-passing multiprocessors, there is relatively little research on parallelizing compilers for non-cache coherent multiprocessors with global address space. In this paper, we present a preliminary study on automatic parallelization for the Cray T3D, a commercial scalable machine with a global memory space and non-coherent caches.

## 1 Introduction

Of the three main classes of today's parallel computers, namely, message-passing multiprocessors, cache coherent multiprocessors, and noncoherent cache multiprocessors with a global address space, parallelizing compilers [2, 9, 11, 12] have been extensively studied for only the first two. In this paper, we present a preliminary study on the automatic parallelization of Fortran programs for the third class machine. Our translation algorithms were implemented in the Polaris restructurer [2], which was developed by the authors and others at Illinois. Important advances in automatic parallelization for cache coherent multiprocessors have been demonstrated recently with the Polaris restructurer. For the work reported in this paper, we have extended Polaris to generate code for the Cray T3D, the only noncoherent cache machine commercially available today. Our current implementation does a straightforward translation involving only a few optimizations beyond parallelism detection. However, our long-term objective is to develop more sophisticated techniques, such as those necessary for loop scheduling, data distribution, and communication minimization.

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

This paper is organized as follows. In Section 2, we briefly introduce the Polaris restructurer. The Cray T3D and CRAFT, the target language of our code generator, are discussed in Section 3. In Section 4, we discuss the compiler techniques we have implemented. In Section 5, we present the results of applying these techniques to ten programs from the SPEC and Perfect Benchmarks. We also discuss some of the factors that limit the performance of the target programs. In Section 6 we discuss a few advanced techniques we plan to implement in the near future to deal with these factors.

# 2 Polaris

The main objective of the Polaris project [2, 7] is to develop and implement effective parallelization techniques for scientific programs. One important characteristic of Polaris is its powerful internal representation [1]. It includes an extensive collection of program manipulation operations to facilitate the implementation of compiler transformations. After a program is converted into Polaris' internal form, it is analyzed and transformed by a sequence of compiler passes which add annotations to identify the parallelism detected by the compiler. Passes currently implemented in Polaris include: symbolic dependence analysis, inlining, induction variable substitution, reduction recognition, and privatization [3, 5, 8]. To support data dependence analysis, Polaris applies range propagation techniques based on symbolic program analysis [10]. After the input program is restructured into the internally-represented parallel program, a final pass or *backend* applies machine-specific transformations and outputs the target parallel program. As mentioned above, we have implemented two backends in Polaris: one for shared-memory multiprocessors, and one for the Cray T3D which is discussed in this work.

Our main focus during the past two years has been on accurately identifying parallelism. Very little effort has been devoted to the backend. However, even without a sophisticated code generation algorithm, Polaris has been quite successful. On an extensive collection of programs gathered from the Perfect Benchmarks, SPEC, and other sources, Polaris substantially outperforms the native parallelizer of the SGI multiprocessor. Work on parallelism detection continues on several fronts, including the study of efficient interprocedural analysis techniques, the parallelization of loops containing complex recurrences, and run-time dependence analysis [16].

# 3 The Cray T3D and CRAFT

While the lack of cache coherence in the Cray T3D helps make the machine affordable and scalable, it also introduces some difficulties in the development of efficient programs [18]. The experimental results presented in this paper give us hope that these programming difficulties can be overcome with effective compiler techniques. In this section, we very briefly describe the Cray T3D and CRAFT.

More detailed descriptions of the machine can be found in [14, 15, 21].

The Cray T3D was designed mainly for large-scale parallel scientific applications. It consists of up to 1024 processing nodes, each containing 2 processing elements(PE) and a local memory. The PEs are 150 MHz DEC Alpha 21064 microprocessors. The interconnection network is a 3-D torus network with high throughput and low latency. Remote memory latency on the T3D ranges from 90 to 130 cycles [33]. Local memory latency is 22 cycles. The T3D also contains a special tree-like network for global barrier synchronization. Various communication primitives, including single-sided communication primitives such as GET and PUT, are supported in hardware. The local memory is logically partitioned into private and shared address spaces. The shared memory in the T3D is no more than the collection of the shared address portions of all local memories. Every shared memory access is manipulated by off-chip components before sending the request to the appropriate module. This off-chip manipulation requires 20-30 cycles. The T3D has a first level on-chip cache which is used only for data in the private address space of the local memory.

There are implementations of the PVM and MPI message passing libraries for the Cray T3D. The machine is often programmed using these primitives to facilitate portability to other scalable multiprocessors. However, programming and compiling for the T3D following the shared-memory model is simpler for most problems and is, therefore, the approach we followed in our work. This can be done using CRAFT, an extension of Fortran for the T3D which has several features in common with other languages for distributed shared-memory machines [19, 25, 28]. CRAFT follows the Single-Program Multiple-Data(SPMD) model and contains a shared address space. In our experiments, each CRAFT process was allocated to a separate physical 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 High Performance Fortran, Vienna Fortran and other similar languages [4, 6, 13, 20]. CRAFT uses `:block` for block distribution and `:block(`$N$`)` for block-cyclic distribution.

The `do shared` directive of CRAFT is used to mark parallel loops. To illustrate its semantics, consider the loop

```
cdir$ shared A(:block(1)), B(:block)
cdir$ do shared (I) on A(I)
      do I = 1, N
         A(I) = B(I)
      enddo
```

Its $I$-th iteration is executed by the PE that owns `A(`$I$`)` in its local memory. Since the elements of `A` are local to the PE accessing them, the compiler enables the caching of `A`. The elements of `B`, on the other hand, are not cached because some of them may be accessed remotely. Therefore, due to the `on` clause, CRAFT users can partition the computation according to the data distribution and thereby enable the caching of some shared data structures. In the example, the computation happens to be distributed according to the owner computes rule, as is done in HPF; however, other distributions could also have been used.

Table 1 summarizes the differences between CRAFT and HPF, as discussed in [14].

| | CRAFT | HPF |
|---|---|---|
| Memory Classes | shared/private | implicitly all shared |
| Data Distribution | *W*:block(*N*) | block/cyclic/align |
| Computation Distribution | programmer-controlled | owner-computes rule |
| Redistribute statement | NO | YES |
| Explicit Communication and Synchronization | YES | NO |

Table 1: Comparison of CRAFT and HPF

Although the machine-supported shared memory model facilitates programming on the T3D, inefficiencies could arise if the program is not carefully designed for the following reasons:

- Unlike cache coherent machines, a remote memory access to a single array element does not take advantage of spatial locality.
- Shared data objects are not cached, even when they are accessed from local memory. Therefore, shared data objects have to be fetched from the remote location every time the program references them. This significantly increases average memory latency and network contention.

To avoid these inefficiencies it is necessary to explicitly control caching and data transfer. We plan to do so in future versions of the translator. To this end we will use the SHMEM [21, 22] communication library which contains various single-sided communication primitives, such as PUT and GET, and explicit cache control routines. For example, SHMEM enables message aggregation which is not possible in ordinary CRAFT programs. The PUT/GET communication [19] allows asynchronous non-blocking access to any memory location. This means that with the SHMEM primitives, the whole memory either private or shared can be shared by all processors. Therefore, all data can be cached without losing the ability to share it. Furthermore, the SHMEM library can be used in programs containing shared data, whereas conventional message passing libraries, such as PVM and MPI, assume all data to be private. SHMEM is more efficient than current implementations of other message-passing models. In fact, in experiments conducted on a 16-processor partition [17], the latency of a SHMEM PUT operation was measured at 2 $\mu$sec and the peak throughput was measured at 116.8 MB/s, while the equivalent figures for PVM send/receive operations are 63 $\mu$sec and 26 MB/s respectively.

# 4 The Cray T3D backend

In this section, we describe the transformations applied by the Cray T3D backend module of Polaris. The backend applies four classes of transformations which we describe in separate subsections below.

Before invoking the backend, Polaris marks all loops it identifies as parallel. It also marks all the induction variables and reductions that have to be substituted, as well as the variables that have to be privatized to make the loop parallel.

## 4.1 Translation into SPMD form following the master/slave model

The first step of the T3D backend is to translate the parallelized Fortran program into SPMD form. We follow a relatively simple approach. One of the processes, designated as the master, executes a program that, outside parallel loops, is identical to the original sequential Fortran program. The master and slave processes cooperate in the execution of parallel regions, while the slaves do not participate in the execution of sequential regions.

We use barriers to enforce synchronization in the master/slave model. Barriers are inserted around parallel loops and calls to subprograms containing parallel regions. Barriers are also needed for some other statements that read or initialize privatized, and therefore replicated, variables.

Barriers do not have a significant impact on overall program execution time in the T3D. One reason is the efficient hardware implementation of barriers in the machine. Table 2 shows the performance of the T3D barrier [33].

| PEs | Barrier Time($\mu$sec) |
|---|---|
| 4 | 1.73 |
| 32 | 1.81 |
| 256 | 1.90 |

Table 2: Barrier Performance. Times are the average of 5000 barrier executions.

Furthermore, in the experiments reported in Section 5, barriers are executed infrequently. The execution time increases by less than 1% due to barriers in all cases. For instance, in FLO52, the dynamic counts of barrier calls are about 50,000. The total overhead due to barriers is approximately 0.1 sec on a 64-processor partition, which is less than 0.2 % of the overall execution time of FLO52.

We, therefore, apply only a few simple techniques to minimize barrier overhead. One of techniques is to identify the places where *implicit* barriers are inserted by CRAFT and avoid placing *explicit* barriers there. For instance, CRAFT automatically places a barrier at the entrance and exit of shared subprograms in order to synchronize allocation and deallocation of shared data and data redistribution. As a consequence, our code generation algorithm does not need to insert explicit barriers at the boundaries of shared routines. Also, we eliminate the in-between barriers when parallel regions are adjacent. We found that Polaris' strategy of preferring outer loops for parallelization further contributes to reducing barriers.

## 4.2 Work partitioning

This transformation uses the `on` clause to partition the iteration space of a parallel loop across processors. We currently use the same code generation algorithm for the T3D as we use for cache coherent machines. Thus, parallel triangular loops are given cyclic schedules and square loops are given block schedules. Only one loop in a nest is parallelized. If several loops in a nest are parallel, then the outermost loop is parallelized unless its number of iterations is small relative to the number of processors. In this case, loop interchanging will be applied to move more practical inner loops to the outer level so that an inner loop can be parallelized.

For the case of loops containing reductions, it was necessary to modify the simple strategy applied by Polaris for cache coherent machines with a few processors. Consider, for example, the loop

```
do J = 1, M
   do I = 1, N
      ...
      A(I) = A(I) + ...
      ...
   enddo
enddo
```

Loops such as this arise frequently in real codes. Assuming that the loop is parallel except for the reduction on `A`, the backend for cache coherent machines generates the loop

```
cdir$ preamble
      A_priv(1:N) = 0.0
cdir$ parallel loop(J)
      do J = b*my_PE+1,(my_PE+1)*b
        do I =1, N
          ...
          A_priv(I) = A_priv(I) + ...
          ...
        enddo
      enddo
cdir$ postamble
      call set_lock(lock)
      A(1:N) = A(1:N)+A_priv(1:N)
      call clear_lock(lock)
```

Here, `A_priv` is a private array of the same size and type as `A`, and `b` is a block size; that is, $\mathtt{M}/P$, where $P$ is the number of processors. The `preamble` and `postamble` code segments are executed once by each processor cooperating in the execution of the loop. The `loop(J)` code is strip-mined to distribute computation across the processors.

This parallel version of the loop has the disadvantage that the `postamble` is executed serially because it is in a critical section. This works well for a few

processors, but a different strategy is needed for a large number of processors. In our current implementation, `A_priv` is divided into $P$ sections within each processor. The postamble consists of two phases. First, for $1 \leq i \leq P$, the $i$-th section of all processors is copied into the $i$-th processor. Then all processors add the $P$ sections copied into them. As expected, this approach of parallelizing the postamble has an important impact on performance. This is illustrated in Figure 3.

| PEs | Preamble | Middle Loop | Serial Postamble | Parallel Postamble |
|---|---|---|---|---|
| 2 | 0.014 | 260 | 0.39 | 0.1 |
| 64 | 0.017 | 11 | 2.7 | 0.129 |

Table 3: Times for parallel version of loop INTERF_do1000 in MDG

## 4.3 Data Distribution and Privatization

In the T3D, if a variable is declared `shared`, it is not cached even when the reference is to a local memory module. In fact, performance difference between the best and worst distributions in a loop of the form

```
cdir$ shared A(distribution descriptor)
      do I = 1, N
         ... A(f(I)) ...
      enddo
```

is only a factor of 2 on 16 processors. On the other hand, declaring data as `private` has an important impact on performance since private data is cached: approximately one order of magnitude.

Our implementation of the T3D backend follows a simple strategy. Polaris identifies loop-private data using the existing privatization algorithm. Furthermore, the T3D backend marks as procedure-private those scalar variables local to a procedure and never used in parallel loops, and other temporal variables exclusively used by individual processors. Both the loop level and the procedure level privatizable variables are declared as `private` in the target program. All other variables are declared `shared` and are given `:block` distribution in all their dimensions.

```
cdir$ private(procedure-level privatizable variables)
      X = 3
          ⋮
      do I = L, U
cdir$ private(loop-level privatizable variables)
            ⋮
      enddo
```

The implementation of this simple strategy involves some complex issues. For example in CRAFT, the `private/shared` attribute of formal arguments has to

be explicit and match that of the actual arguments. This is discussed in more detail in Section 4.4.

In future implementations, the local sections of logically shared arrays will be privatized to enable the use of the cache. Routines from the SHMEM library could be used to fetch and store remote sections of these logically shared arrays.

## 4.4 Compatibility problems between Fortran and CRAFT

Many MPP Fortran extensions, such as CRAFT and HPF [20], help the user attain high performance through distribution of data and other directives, while maintaining some degree of compatibility with conventional Fortran 77. However, total compatibility has not been achieved because these languages impose several restrictions in the name of performance. The following list shows the major restrictions imposed by CRAFT:

- Fortran's sequence and storage association rules [23] do not apply to shared data.
- Shared data may not be in `EQUIVALENCE` or blank `COMMON`.
- Shared data may not be of type `CHARACTER`.
- The dimension size of shared arrays must be a power of two[1].
- Shared formal parameters may not be associated with private actual parameters, and their size and shape must match those of the corresponding actual parameters.

We have endeavored to develop translation techniques to overcome these limitations. Three of these techniques are discussed below.

### 4.4.1 Renaming

Aliasing has always been an important issue in program analysis in general, and in automatic parallelization in particular [32]. Aliasing is also one of the most difficult problems in automatic parallelization for the T3D. Consider, for example, the following segment extracted from HYDRO2D:

```
subroutine X1
real A, B, C, D, E, F
common /SCRA/ A, B, C, D, E, F
   ⋮
subroutine X2
real G(5)
integer I
common /SCRA/ I, G
```

[1] In the Cray T3E, this restriction is no longer imposed.

CRAFT cannot distribute a shared data object if it is aliased to other objects with different shape or type. In the code above, the shared variables within `SCRA` cannot be distributed because of the aliasing of `A` and `I`. In order to address this problem, we interprocedurally check the life times associated with each variable and apply renaming. In the previous example, it can be proven that the life time of the values of `SCRA` in `X1` and in `X2` are disjoint. As a result, we could rename either occurrence of the common block without affecting the outcome of the program.

### 4.4.2 Linearization

Variable linearization and renaming are the most common techniques used to solve problems related to storage association rules. For example, linearization is needed to inline a subroutine call when an actual parameter differs in shape from the corresponding formal parameter.

We use linearization to deal with array equivalences because of the restriction that the size of all shared array dimensions must be a power of 2. When two arrays of different shapes are equivalenced, we linearize the array before dimension expansion. Linearization also helps to save memory. For example, a shared array `A(9,9,9)` is expanded to `A(16,16,16)` without linearization, and `A(1024)` with linearization. Linearization, in this case, saves 3K words. However, we do not want to apply linearization to all multidimensional arrays because linearization makes the program less readable and program analysis more difficult due to the complex subscript expressions.

### 4.4.3 Array Reshaping and Procedure Cloning

The following code, where we assume that `A` and `B` are shared arrays, illustrates one important difference between Fortran 77 and CRAFT:

```
real A(8,8)
call foo(A(1,7))
   ⋮

subroutine foo(B)
real B(4,4)
... B(2,2)
```

If interpreted as Fortran 77, `B(2,2)` in this code is aliased with `A(6,7)`; but, if interpreted as CRAFT, it is aliased with `A(2,8)`. This is because, in CRAFT, aliasing between shared arrays takes place at the submatrix level, whereas in Fortran, a linear storage sequence is often assumed for parameter aliasing. Many real Fortran codes rely on such association rules.

We address this problem by changing the subscript expressions within the subroutine to conform to the CRAFT semantics. *Procedure cloning* is applied whenever the same routine is called with different submatrices as actual parameters.

For example, our algorithms would translate codes like the previous one using the following transformation pattern:

```
real A(L1:U1,...,Ln:Un)          real A(L1:U1,...,Ln:Un)
call foo(A(f1,...,fn))           call foo_clone(A)
   :                       ⇒        :

subroutine foo(B)                subroutine foo_clone(B)
real B(L'1:U'1,...,L'm:U'm)      real B(L1:U1,...,Ln:Un)
... B(g1,...,gm)                 ... B(h1,...,hn)
```

Notice that in the resulting code the origin of the parameter array is passed to the subroutine rather than the address of one of its elements. The actual and formal parameter arrays are forced to have the same size and shape. Let $X$ be the offset of `A(`$f_1,\ldots,f_n$`)` from the first address of `A`, and $X'$ be that of `B(`$g_1,\ldots,g_m$`)`. Then, the array index $h_k$ is defined as

$$h_k = X_k \texttt{ mod } N_k + L_k$$

where $N_k = U_k + L_k + 1$, $X_1 = X + X'$ and $X_i = \lfloor X_{i-1}/N_{i-1} \rfloor$ for $i > 1$.

Our experiments show that cloning does not increase the size of the original programs by more than a factor of 2. This is the case because actual and formal parameters usually have the same shape, size and dimension.

# 5 Preliminary Results

Figure 1 presents the speedups obtained by Polaris on the T3D for five programs from the Perfect Benchmarks (ARC2D, BDNA, FLO52, MDG, and TRFD), and five programs from the SPEC collection (APPSP, HYDRO2D, SU2COR, SWIM, and TOMCATV).

We report speedups for processor numbers that are powers of two between 1 and 64. During the sequential execution of an original program, we measured the percentage of overall running time of the program for each loop, and added the percentages of each parallelizable loop to obtain the total sum. We call this total sum *parallel coverage* [2]. Two dotted lines in Figure 1 plot the ideal speedups for programs with parallel coverage of 99% and 90% respectively. The ideal speedup, $S$, is calculated by using the Amdahl's equality: $S = 1/(c/P + 1 - c)$ where $P$ is the number of processors and $c$ is a parallel coverage.

After all ten analyzed programs were transformed by Polaris, their parallel coverages were between 90 and 99%. Therefore, the speedup curves for these programs should, under ideal conditions, lie between the two dotted lines. Real program speedups are much lower than the ideal for several reasons, including synchronization and scheduling overhead, communication costs, and shared data bypassing the cache. These speedups should improve once we implement in Polaris optimizations to deal with these issues.

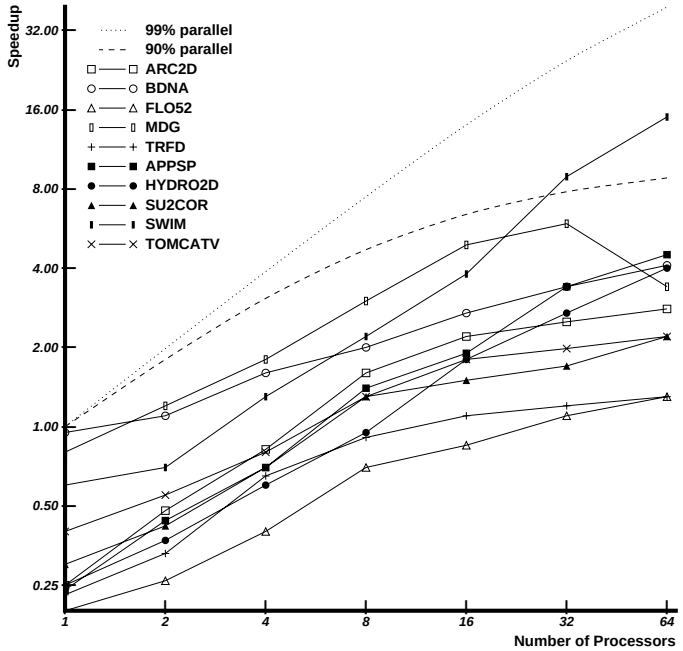


Figure 1: Preliminary Results for the Perfect and SPEC Benchmarks

The effect of bypassing the cache, which we call *cache bypassing penalty*, is reflected in the speedups for one PE, where the speedup is below one in all cases, as shown in Figure 1. The largest values are 0.9 for BDNA and 0.8 for MDG. This occurs because both programs use several large privatized reduction arrays and, thus, have very few accesses to shared data. To the contrary, FLO52 and TRFD have lowest speedups for one PE because they repeatedly access large sections of shared arrays in the loops.

In addition to the cache bypassing penalty, as we increase the number of processors, other factors become significant, such as communication, amount of parallelism, and the number of iterations of parallel loops. Such factors tend to decrease the efficiency as the number of processors grow. In this paper, *efficiency* is defined as the ratio between speedup and the ideal speedup made possible by the parallel coverage of the program. For example, the efficiency of TRFD, whose parallel coverage is 90%, goes from 0.23 to 0.13 as the number of processors grows from 1 to 64.

In the rest of this section, we discuss the behavior of three of the ten programs presented in Figure 1: SWIM, MDG, and TRFD.

## 5.1 SWIM

SWIM is a finite difference solver of the shallow water equation on a 512x512 grid. Its serial execution time on the T3D is 2378 seconds. SWIM performs

most of its operations on fourteen 513x513 arrays. In the parallelized version, these arrays are expanded to 1024x1024 shared arrays because of the CRAFT restrictions discussed in Section 4.4. Fortunately, for 64 processors, the total amount of extra memory required is relatively small: 0.1M words per processor.

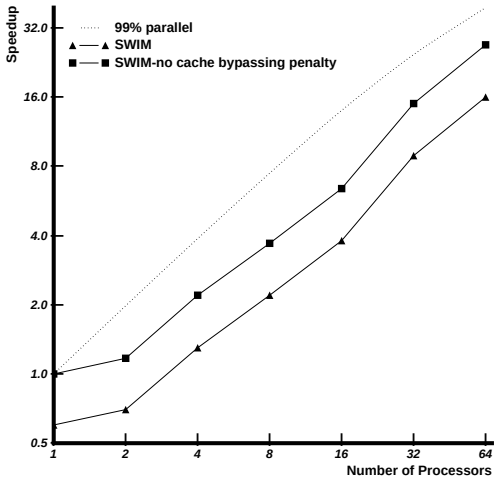


Figure 2: Speedup Analysis for SWIM

SWIM shows the best speedups in Figure 1. The main reason for these speedups is that Polaris parallelizes all the outermost loops, which results in a parallel coverage of almost 100%. Most parallel loops in SWIM are doubly-nested loops with 512x512 iterations. This is large enough to saturate 64 processors. Furthermore, each processor accesses different regions of the shared arrays in parallel loops, so there are few memory access collisions during the execution of the parallel loops. Above all, our simple block distribution policy happens to match computation distribution in SWIM, thus reducing a large amount of remote memory access overhead. As a result, we see that the speedup grows with the number of processors.

Despite the good characteristics of SWIM, we notice that the efficiency, for a single processor is still 0.55. As mentioned earler, this number mainly refects the impact of the cache-bypassing penalty in the T3D. In other words, if caching is allowed for shared data, we would get a speedup of 1.8(= 1/0.55). We plot this predicted speedup line in Figure 2. Therefore, we conclude that one of the major optimization efforts in SWIM should focus on increasing the fraction of cacheable data.

## 5.2 MDG

MDG is a molecular dynamics model of water. Its sequential execution time on the T3D is 330 seconds. The most important loop in MDG is `INTERF_do1000`, which accounts for 92% of sequential execution time. This loop is parallelized be-

cause of several advanced techniques applied by Polaris, including inlining, array privatization, induction variable recognition, and histogram reduction recognition. The parallel version of `INTERF_do1000` has many privatized reduction arrays instead of shared arrays. Figure 3 shows that the speedup will not grow significantly even after eliminating all the cache bypassing penalty for the parallel loops. Thus, the cache bypassing penalty is not as influential in MDG as it is in SWIM.

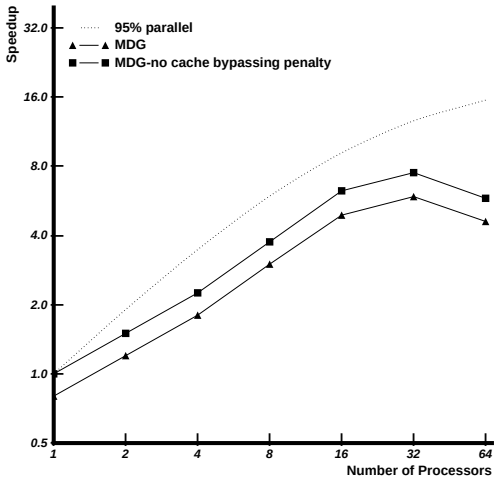


Figure 3: Speedup Analysis for MDG

The main cause of the drop in speedup is a doubly-nested sequential loop, `INTRAF_do1000`, which accounts for just 1% of sequential execution time. This loop accesses shared data and, when the number of processors grows, so does the execution time of this loop, as shown in Table 4. This shows that, in some cases, communication costs in a serial loop grow much faster than the same costs in a parallel loop. Hence, we need to reduce these costs in sequential loops even when their execution time is relatively small. In fact, the loop `INTRAF_do1000` is parallel and, thus, the speedup will improve when Polaris is extended to parallelize this loop.

| PEs | INTERF_do1000 (sec) | INTRAF_do1000 (sec) |
|---|---|---|
| 2 | 260 | 4.6 |
| 4 | 140 | 5.8 |
| 8 | 76 | 8.4 |
| 16 | 41 | 14 |
| 32 | 21 | 26 |
| 64 | 11 | 59 |

Table 4: INTERF_do1000 and INTRAF_do1000 on the T3D

### 5.3 TRFD

TRFD is a small kernel for quantum mechanics calculations. It has two major loops, `OLDA_do100` and `OLDA_do300`, both of which correspond to 90% of the sequential execution time. All the other parallel loops in TRFD take up less than 1% of sequential execution time. Judging from the case on a single processor in Figure 1, TRFD suffers from bypassing the cache for shared data more than SWIM and MDG. Removing such a penalty would drastically boost the performance by a factor of 4.6, as projected by the ideal speedup shown in Figure 4.

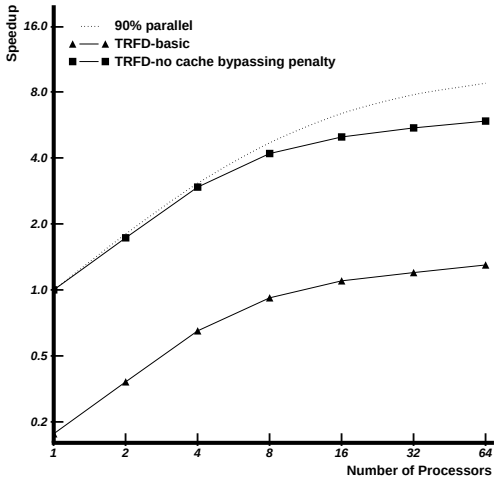


Figure 4: Speedup Analysis for TRFD

Even after we eliminate the cache bypassing penalty, the efficiency of TRFD would be too small still, as indicated in Figure 4. This is primarily due to the small data set size for TRFD [24, 26].

## 6 Program Optimizations and the Data Copying Strategy

From the discussion above, it is clear that the main factors influencing parallel performance are:

- the parallel coverage;
- the parallel loop structure of a program;
- the amount of shared data used in local computations; and,
- the data access pattern.

In [7], we discuss several on-going efforts in the Polaris project to increase the parallel coverage and, in Section 4, we briefly deal with the parallel loop structure of a program. In this section, we discuss the last two factors.

From the evaluation outlined in Section 5, we concluded that to subtantially improve performance it is necessary to privatize shared data. One strategy that we plan to explore, the *shared data copying scheme*, uses shared memory as a source/repository of values for private variables. In this strategy, most of the work is done on private variables. Before a program section starts, the processors copy all that is used in the computation from shared memory into private memory. At the end of the section, the processors copy the results back to shared memory so that all processors have access to the results. Shared memory coherence is maintained by explicit synchronization.

PUT/GET primitives will be used for data copying in our future work. One reason we have chosen to use PUT/GET is that these primitives match the shared-memory programming paradigm assumed by Polaris. Another reason is that PUT/GET is rapidly gaining widespread acceptance. In fact, several portable shared-memory programming models supporting PUT/GET are already implemented on ordinary message-passing machines, such as the IBM SP-1/2, Intel Paragon and TMC CM-5 [19, 25, 28]. Furthermore, several existing and newly proposed large-scale machines, such as the T3E, directly support these primitives in hardware [21, 27], which reduces the effect of the increased communication overhead resulting from the extra data copy operations.

Communication aggregation is useful in reducing the overhead. In most Fortran programs, the natural program section that organizes copying from/to global memory is the loop. The following code example shows how Polaris transforms the information about the range of arrays fetched and written by the loop into PUT/GET primitives.

```
cdir$ must_read (A(l1:u1:s1))          call polaris_get(A(l1:u1:s1),a(l1':u1':s1'))
cdir$ must_write (A(l2:u2:s2))         do I = l',u',s'
      do I = l,u,s                  ⇒      access a
         access A                          enddo
      enddo                                call polaris_put(A(l2:u2:s2),a(l2':u2':s2'))
```

In the transformation, a shared array `A` is replaced by a private array `a`. The `must_read` directive indicates the shared data sections that *must* be used in the loop, and `must_write` indicates the shared data sections that *must* be updated in the loop. We generate PUT/GETs to move the data sections specified in these directives around each loop. In addition to the two directives, `may_read` and `may_write` can be generated to mark shared data sections that *may* be used and updated in a loop.

We implemented high level data transfer routines, `polaris_put/get`, which make use of the T3D's lower level single-sided communication primitives. Micro-benchmarking experiments on a 16-processor partition show that the throughput of these routines is approximately 1.5 Mwords/sec with data over 1K words long. Table 5 shows the effect of our PUT/GET operations on the performance of loop `OLDA_do100` of TRFD. The sequential execution time of the loop is 23.6 sec.

| PEs | No Data Copying (sec) | Data Copying (sec) | |
|---|---|---|---|
| | | Loop | PUT/GET |
| 2 | 65 | 13 | 0.2 |
| 4 | 38 | 6.3 | 0.6 |
| 8 | 26 | 3.2 | 0.7 |
| 16 | 23 | 1.5 | 1.1 |
| 32 | 21 | 0.76 | 1.4 |
| 64 | 20 | 0.40 | 1.8 |

Table 5: Reduction of Cache Bypassing Penalty by PUT/GET on TRFD: The total amount of data transferred at a time for the loop execution is about 1 Mwords with sizes varing from 0.1 to 600 Kwords

Even without using any other techniques, the effect of the work on loop `OLDA_do100` increases the overall program speedup to 2. For the loop itself, with overhead ignored, the speedup is 59 on 64 processors. Although the PUT/GET overhead reduces the speedup to 11, this is still 9 times faster than the execution time before optimization. Another program that benefits from using PUT/GET is SWIM. Figure 5 shows the speedup obtained by hand on up to 64 processors. These and other test results have convinced us that the T3D's PUT/GET operations are efficient enough to improve performance across the board.

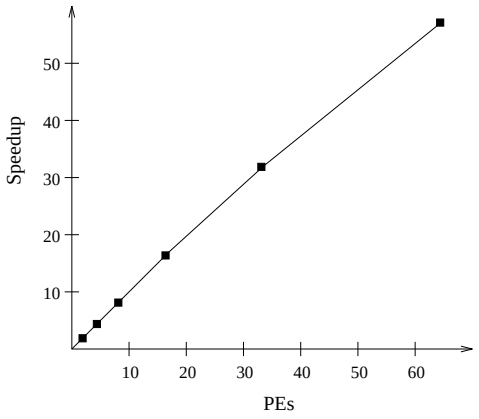


Figure 5: Hand-optimized SWIM using PUT/GET

An important strategy that can be used to complement shared data copying is prefetching and poststoring. Unlike cache memory, the data copied into private memory is fully software controllable; that is, the data would never be flushed out until explicitly done so by the program. Hence, as long as the local memory space is available, a processor can prefetch data anytime before it is needed and poststore it sometime after the computation. In [29], some other advantages of the compiler-directed communication over cache-coherence protocol driven

communication on the non-cache coherent architecture are discussed.

The shared data copying scheme is especially useful when the data distribution requirements of a program are dynamic. The conventional technique for such a case is *data redistribution* [30]. Data redistibution usually moves much of the array across the distributed memories. Unless most elements of the array are fully used for a long time, this total data movement is inefficient. In contrast, PUT/GET operations enable us to avoid this inefficiency by making each processor copy only the portions of an array that are needed for its local computations.

Shared data distribution is another important issue that we will have to consider. In general, as discussed above, data distribution by itself did not significantly influence performance in our experiments. However, we found that it is important to distibute shared data in order to minimize data copy overhead in the shared data copying scheme. One way to reduce the copy overhead is to aggregate the data as much as possible, taking advantage of the T3D's high-bandwidth low-latency network. Most GET/PUT primitives support contiguous data streams with regular stride more effectively. For these reasons, we simply block distribute shared arrays in the shared data copying scheme. This also helps simplify the calculation of the owner of the target data before issuing PUT/GET calls. In TRFD, for example, the data sections needed by `OLDA_do100` change every time we encounter the loop and thus, no single data distribution can match this changing data access pattern. The simple block distribution always guarantees no more than $P$ PUT/GET calls per data copy for a $P$-processor partition.

In any case, distributing data so that most target data are in local memory is our ultimate goal since local copy operations are faster than remote copy operations. We plan to investigate the possibility of improving data distribution for the data copying scheme.

# 7 Conclusion

We have reported some preliminary experience using the Polaris restructurer to parallelize Fortran codes for the Cray T3D. The strategy we used here was to extend the traditional parallelizing compiler techniques for cache coherent machines to deal with non-cache coherent multiprocessors. We discussed our translation algorithms and presented optimization techniques that should significantly improve the preliminary results once they are implemented. Our long-term objective is to develop and implement these advanced techniques in Polaris.

# Acknowledgements

We would like to thank the Cray Research Inc. for generously granting machine time for the experiments reported in this paper. We are especially grateful to Thomas MacDonald for helping us access and understand the Cray T3D.

# References


[1] K. Faigin, J. Hoeflinger, D. Padua, P. Petersen, S. Weatherford. The Polaris Internal Representation. *International Journal of Parallel Programming*, Vol. 22, No. 5, Oct. 1994, pp. 553–586

[2] B. Blume, et al., Polaris: Improving the Effectiveness of Parallelizing Compilers, *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing, OR. Lecture Note in Computer Science*, Aug. 1994, pp. 141-154

[3] B. Pottenger, R. Eigenmann, Idiom Recognition in the Polaris Parallelizing Compiler, *Proceedings of the 9th ACM International Conference on Supercomputing*, July 1995

[4] Z. Bokus, et al, Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers, Journal of Parallel and Distributed Computing, Vol. 21, 1994, pp. 15-26

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

[6] B. Chapman, P. Mehrota, H. Moritsch, H. Zima, Dynamic Data Distributions in Vienna Fortran, *Supercomputing '93 Proceedings*, 1993, pp. 284-293

[7] B. Blume, et al., Advanced Program Restructuring for High-Performance Computers with Polaris, Tech. Report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing R & D, 1996, CSRD Report No. 1473

[8] P. Tu, D. Padua, Automatic array privatization, *Proc. 6th Workshop on Language and Compilers for Parallel Computing, OR. Lecture Note in Computer Science*, Aug. 1993, pp. 500-521

[9] C. Polychronopoulos, et al., The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran, *Languages and Compilers for Parallel Computing*, MIT Press, 1990

[10] W. Blume, R. Eigenmann, The Range Test: A Dependence Test for Symbolic Non-linear Expression, *SuperComputing '94 Proceedings*, Nov. 1994, pp. 643-656

[11] P.Banerjee, et al., The PARADIGM Compiler for Distributed-Memory Multicomputers, *IEEE Computer*, Vol. 28, No. 10, Oct. 1995, pp 37-47

[12] S. Amarasinghe, et al., An Overview of the SUIF Compiler for Scalable Parallel Machines, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995, pp. 662-667

[13] C. Tseng, An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines, PhD Thesis, Rice University, Jan. 1993

[14] W. Oed, The Cray Reseach Massively Parallel Processor System CRAY T3D, Cray Research, Nov 1993

[15] CRAY T3D System Architecture Overview, Cray Research, 1993

[16] L. Rauchwerger, D. Padua, The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization, *Proceedings of the 8th ACM International Conference on Supercomputing*, July 1994, pp. 33-43

[17] R. Marcelin, Message Passing on the CRAY T3D, Massively Parallel Computing Group, NERSC, 1995

[18] D. Bernstein, et al., Solutions and Debugging for Data Consistency in Multiprocessors with Noncoherent Caches, *International Journal of Parallel Programming*, Vol. 23, No. 1, 1995, pp. 83-103

[19] M. Snir, Proposal for MPI-2, MPI meetings, 1995

[20] High Performance Fortran Language Specification, High Performance Fortran Forum, May 1993

[21] CRAY MPP Fortran Reference Manual, Cray Research, 1993

[22] SHMEM Technical Note for Fortran, Cray Research, Oct. 1994

[23] Programming Language FORTRAN, American National Standards Institute, ANSI X3.9-1978 ISO 1539-1980

[24] J. Gustafson, Reevaluating Amdahls Law, *Communications of the ACM*, Vol. 31, No. 5, May 1988, pp. 532-533

[25] D. Culler, et al., Parallel Programming in Split-C,*Supercomputing '93 Proceedings*, 1993

[26] A. Grama, A. Gupta, V. Kumar, Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures, *IEEE Parallel & Distributed Technology*, Aug. 1993, pp. 12-21

[27] K. Hayashi, et al., AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. Proc. 6th International Conference on Architechtural Support for Programming Language and Operating Systems, Oct. 1994, pp. 196-207

[28] J. Nielocha, R. Harrison, R. Littlefield, Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers, *Supercomputing '94 Proceedings*, 1994, pp.340-349

[29] J. R. Larus, Compiling for Shared-Memory and Message-Passing computer, *ACM Letters on Programming Languages and Systems*, 1996

[30] K. Kenney, Compiler Technology for Machine-Independent Parallel Programming, *International Journal of Parallel Programming*, Vol. 22, 1994, pp. 79-98

[31] R. Eigenmann, J. Hoeflinger, G. Jaxon, D. Padua, The Cedar Fortran Project, Tech. Report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing R & D, Apr. 1992, CSRD Report No. 1262

[32] H. Zima, B. Chapman, Supercompilers for Parallel and Vector Computers, ACM Press, 1992

[33] R. Arpaci, et al., Empirical Evaluation of the CRAY-T3D: A Compiler Perspective, *Proceedings of ISCA*, 1995, pp.320-331