Experimental Study of Compiler Techniques for NUMA Machines

Yunheung Paek
Dept. of Computer & Information Science
New Jersey Institute of Technology
paek@cis.njit.edu

David A. Padua
Dept. of Computer Science
University of Illinois at Urbana-Champaign
padua@cs.uiuc.edu

Abstract

This study[1] explores the applicability of fully automatic parallelizing techniques for parallel computers. In this study, we capitalize on a variety of traditional compiling techniques as well as new techniques developed specifically for distributed memory architectures. Combining these traditional and new techniques, we conducted experiments with several benchmark programs on the Cray T3D.

1. Introduction

Much work has been done in recent years on automatic parallelization of conventional codes for uniform memory access (UMA) machines. Significant progress has been made and several new strategies currently under study have been developed, which also has given us the foundations needed to tackle the study of compilers for nonuniform memory access (NUMA) machines.

This paper reports our recent work in the development of compiler algorithms to transform conventional Fortran 77 codes into parallel form for NUMA machines, which are NUMA machines with a global memory space [4, 6, 8, 16]. We have implemented these algorithms in the compiler infrastructure provided by **Polaris** [2], a parallelizing compiler developed by the authors and others. The code transformation is fully automatic requiring no user intervention. The compiler gathers from the source code all the information needed to expose parallelism, distribute data, and control data movement. This fully automatic parallelization is very important not only because it facilitates the development of new parallel programming models [15] in the familiar sequential paradigm, but also because it may help to port the vast amount of legacy code to new multiprocessor systems.

In this study, we use the techniques already implemented in Polaris to identify parallelism. In addition, we have implemented several new techniques to generate code for NUMA architectures. To evaluate the effectiveness of our techniques, we have conducted experiments on one of our target machines using six sequential codes: `bdna`, `mdg`, and `trfd` (Perfect benchmarks); and `swim`, `tomcatv`, and `tfft2` (SPEC benchmarks).

The rest of this paper is organized as follows. Section 2 presents an overview of our approach. Sections 3 and 4 describe the approach in detail. Section 5 discusses our experiments.

2. Compiling for NUMA Machines

Speedups on NUMA machines depend mainly on two factors: *parallelism* and *communication*. Good performance is possible only when both of these factors are properly handled. One approach that is frequently followed when programming NUMA machines we call the *data-partition oriented* approach. The approach consists of finding an optimal data partition [9, 1]. Work partitioning or computation assignment is done usually as a function of the data layout, following strategies such as *the owner-compute rule*. Sometimes parallelism is reduced to obtain better data locality.

Often this approach involves complex data/work partitioning and occasional data redistribution [14] when the access patterns change. Finding an optimal data/work partition, however, is often problematic because many scientific applications consists of several different parts that have conflicting distribution requirements for fast execution. For example, in Figure 1, an array `X` is accessed horizontally in the first loop and vertically in the second. Conventional techniques may require redistribution of `X` from by rows to by columns somewhere between the two loops. If `M` is less than `N`, this redistribution will cause redundant communication to move data unrelated to this computation. Also, it is impossible to have a communication-free data partition for `Y` because the access regions overlap across loop iterations. Aliasing is another common factor that makes the problem difficult. In the example, if the dummy arrays `X` and `Y` are

[1] This work is supported in part by Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract ASC-96-12099; and a Partnership Award from IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

aliased when h is called, then it becomes difficult to have different data distributions for them. Furthermore, to determine the proper data placement, we need additional information about all arrays aliased in other subroutines, which requires a very complex interprocedural analysis. Complex subscripting patterns are also well-known factors that prevent the identification of optimal solutions, as in the case of the array Z.

```
subroutine h(X,Y,Z,M,N)
real X(N,N), Y(*), Z(*)
   ...
doall I = 1, M
   X(I,1) = Z(2**(M-I))
   do J = 1, M
      X(I,J+1) = X(I,J) + Y(I+J)
   enddo
enddo
   ...
doall J = 1, M
   do I = 1, J
      X(I+1,J) = X(I,J) + Y(J-I+1)
   enddo
enddo
```

Figure 1. Code example to illustrate the data distribution issue

The data-partition oriented approach is particularly effective when remote memory accesses in the target machine are quite expensive. This is the case in some systems, such as loosely-coupled multicomputers and PC clusters. However, the NUMA machines with a global memory space we target in this work is generally tightly-coupled in that they support very low remote memory latency[2]. The study of LeBlanc and Markato [10] indicated that the data partition issue is less important to compiling for these machines, which implies that it is not necessary to follow the same approach to obtain reasonably good performance for the NUMA machines with a global memory space.

Based on all these observations, we decided to follow a different approach that is not data-partition oriented, which is illustrated in Figure 2. First, parallelism is exposed from the input program. Then, using the parallelism information, our algorithm tries to evenly distribute parallel work prior to any data distribution being considered. Given the work distribution, the data regions accessed by each individual processor are identified. Based on the memory access pattern, data are privatized to reduce communication overhead and to improve locality, and all non-privatized arrays are simply block-distributed across the distributed memories. Finally, most of the non-local accesses in the code are localized by using data prefetching and poststoring strategies. We discuss this transformation procedure in Section 4 in more detail.

[2]For instance, the difference in speed between local access and remote access ranges far less than one order of magnitude for the Cray T3D and the SGI Origin.

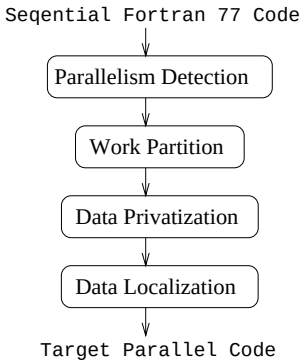

Figure 2. Code transformation in Polaris

3. Target Parallel Programming Model

The target code generated by our transformation is shared-memory code, which is of a thread-based single program multiple-data (SPMD) form with the following characteristics: (1) data is explicitly declared as `private` or `shared`, (2) thread control is explicitly synchronized, and (3) threads communicate with PUT/GET primitives [11, 7, 5].

The parallel program follows the master/slave model in which one of the threads, *the master*, executes all regions, and the others, *the slaves*, participate only in the computations of parallel regions. A private variable in a program is replicated to create a copy for every local memory. In contrast, only one object for each shared variable exists in the system and is accessible to all threads. Shared arrays can be given `BLOCK` or `CYCLIC` distribution [15]. Barriers and locks are used to control and synchronize the flow of execution of threads. PUT/GET operations work well with the shared-memory programming paradigm in that they allow the threads to asynchronously access any data object in the system whether the object is private or shared.

Our techniques can be applied to any NUMA machine with a global address space that supports all these characteristics by either hardware or software. However, for our techniques to be more effective, we specifically target the machines that provide special H/W mechanisms to support fast synchronization and asynchronous remote memory access for global address space operations.

4. Code Transformation

In this section, we describe the compiler modules shown in Figure 2. Among these modules, parallelism detection, data privatization, and data localization rely on the analysis of array access patterns in a program. Array access patterns must be expressed in some standard representation for which Polaris has used the traditional *triplet notation*. However, we have found that the triplet notation does not always accurately capture complex access patterns that are

commonly encountered in real programs and, thus, prevents our complier modules from carrying out their techniques in some important cases. For this reason, we have developed a new region representation, called the *Access Region* [13], which increases the accuracy of the analysis and improves the effectiveness of the compiler. In the work presented here, we make use of both the traditional techniques based on the triplet notation and the new techniques based on the Access Region.

4.1. Parallelism Detection

The techniques implemented in Polaris to detect parallelism include: dependence analysis, inlining, induction variable substitution, reduction recognition, and privatization [2]. The loops that are identified as parallel by these techniques are marked with `parallel` directives as shown in the loop

```
cdir$ parallel (I)
      do I = 1, N
         X(I) = ···
      enddo
      print *, X(N), N
       ···
```

The dependence analysis techniques in Polaris have been developed to handle several important loop patterns that commonly occur in scientific programs. This strategy has worked well on medium- and small-scale UMA multiprocessors, where a few important loops in a program dominate the overall performance [2]. In large-scale multiprocessors, however, this strategy is inappropriate because small unimportant loops become important, as illustrated in Table 1.

PEs	Parallel Loops (sec)	Serial Loops (sec)	Serial Coverage (%)
2	120	3.2	2.7
8	31	2.8	8.2
64	6.1	2.7	31

Table 1. Loop execution times in `mdg` on the T3D

In the table, the *serial coverage* is the percentage of execution time consumed by serial loops. As can be seen, the serial loop execution time can be ignored for fewer than eight processors; but, in light of Amdahl's law, the serial loop execution time becomes significant as the number of processors increases. Due to this fact, we had to parallelize these small loops in the code generation for NUMA multiprocessors. To this end, it was necessary to develop a new dependence test, we call the *Region Test* [13]. The test is similar to the Range Test [2], the most powerful dependence test used by Polaris, in the sense that both are based on array access region analysis. However, the Region Test can parallelize many loops that the Range Test cannot because it is based on the Access Region representation, while the Range Test is based on the triplet notation. Capitalizing on the powerful expressiveness of the Access Region, the Region Test is especially useful for parallelizing loops with very complicated access patterns. Consequently, the test contributes significantly to the sustained increases of speedups on a larger number of processors, as displayed in Figure 6. The importance of the Region Test is shown in Table 2.

PEs	No Region Test (sec)	Region Test (sec)	Speedup Improvement
4	764	188	4.06
8	634	114	5.56
16	563	69.0	8.16
32	540	47.0	11.5
64	522	36.0	14.5

Table 2. Comparison of the parallel execution times of `tfft2` on the Cray T3D

4.2. Work Partitioning

We transform the original program annotated with `parallel` directives into the SPMD form described in Section 3. All parallel loop iterations are statically assigned to the processors. We use the same loop scheduling scheme for the NUMA architectures as we use for UMA architectures: *cyclic* schedules for triangular loops, and *block* schedules for square loops. For the programs we tested, these conventional static scheduling schemes provide relatively good load balancing at low loop scheduling costs in most cases. The following loop shows the result after the code in the loop in Section 4.1 is transformed by the work partitioning module.

```
        IL = N/P*my_pid+1
        IH = N/P*(my_pid+1)
        do I = IL, IH
           X(I) = ···
        enddo
        if (slave) goto wait
        print *, X(N), N
wait:   call barrier()
         ···
```

$\mathcal{P}$ is the number of processors and my_pid is the processor ID number between 0 and $\mathcal{P}$-1. Notice that the `print` statement, as is usually the case with most I/O statements, is in a sequential region and, therefore, the slaves skip the statement by jumping to a barrier where they wait for the master.

4.3. Data Privatization

Given a partition of the computation, the Polaris privatizer [2] analyzes the data regions accessed by each processor and declares data as private if the data is always accessed by the same processor. The benefits of data privatization are many. One is that it removes some types of *output* and *anti* dependences, as will be shown in Section 4.4. In particular, data privatization is important to NUMA architectures because it increases data locality, thus reducing

the overall communication overhead. Furthermore, in non-cache coherent NUMA machines [4, 6], shared data may not be cached which causes performance degradation whenever the computation uses shared data. Data privatization presents additional benefits in these machines by providing more chances for processors to use data caches for their computations.

The privatizer tries to find parts of a data object that are always written prior to being read in each iteration of a loop. Figure 3 shows how the original code is transformed after data privatization is applied. We have five privatized variables in `f`. To privatize the array `W`, the privatizer first determines the region `W(IL:IH,1:2)` read in the `I`-loop and the region `W(JL:JH,1:2)` written in `g`. Then, it determines that the write region covers the read region in all processors by proving that `JL` $\leq$ `IL` and `JH` $\geq$ `IH`, from which `W` is identified as privatizable. It is clear from the code that each processor accesses at most `LEN/`$\mathcal{P}$ elements in the first dimension of `W`; hence, the privatizer redeclares `W` to have `W(LEN/`$\mathcal{P}$`,2)` and changes the subscript expressions of `W` in `f`. Notice that the formal parameter `B` in `g` also must be declared as private in order to match the private attribute of the actual parameter `W` in `f`. Also, notice that the formal parameter `V` is not privatized. One reason for this is that the actual parameter corresponding to `V` in the subroutine which calls `f` could be shared. We will discuss how to deal with these *non-privatized* arrays in the next section.

```
 subroutine f(V,L)
 real W(LEN,2), V(*) ···
  ···
 call g(V,W,L)
  ···
 IL = L/P*my_pid+1
 IH = L/P*(my_pid+1)
 do I = IL, IH
    T = W(I,1) + W(I,2)
    V(I) = T ···
 enddo
   :                          ⇒
   :
 subroutine g(A,B,L)
  ···
 JL = L/P*my_pid+1
 JH = L/P*(my_pid+1)
 do J = JL, JH
    B(J,1) = ···
    B(J,2) = ···
 enddo
   :
   :
    subroutine f(V,L)
    real W(LEN/P,2), V(*) ···
cdir$ private T, I, IL, IH, W
     ···
    call g(V,W,L)
     ···
    IL = L/P*my_pid+1
    IH = L/P*(my_pid+1)
    do I = IL, IH
       T = W(I-IL+1,1)
          + W(I-IL+1,2)
       V(I) = T ···
    enddo
      :
      :
    subroutine g(A,B,L)
     ···
    JL = L/P*my_pid+1
    JH = L/P*(my_pid+1)
    do J = JL, JH
       B(J-JL+1,1) = ···
       B(J-JL+1,2) = ···
    enddo
```

Figure 3. Simplified code example from the `su2cor` benchmark after work partitioning, and code result after applying data privatization

4.4. Data Distribution and Localization

We declare all non-privatized arrays as shared and `BLOCK`-distribute all their dimensions. Then, we apply the *shared data copying scheme* [13] to localize most of the accesses to these shared arrays that are made by the processors. In the scheme, shared memory is used as a repository of values for private memory. In most of current NUMA machines, the shared memory is usually a logical collection of the shared address portions of all local memories. In order to apply the scheme, we first change references to shared arrays to the corresponding private arrays in each loop. Before a loop starts, the processors copy all portions of shared arrays that are read in the loop into private memory. After the loop execution completes, the processors copy the updated results back to shared arrays so that all the processors have access to the new results. By doing so, most of the work is done on private data objects.

The copying operations are implemented with PUT/GET primitives, which are specified as

`polaris_put/get(`*SA*`,` *PA*`,` *SS*`,` *PS*`,` *L*`)`

where `polaris_put` transfers *L* words of data from *PA* in private memory space to *SA* in shared memory space. The *SS* and *PS* are, respectively, the strides of the shared and private arrays being transfered, and `polaris_get` works the same way except that the source and destination of data transfer are reversed. For example, in the code of Figure 3, `V(IL:IH)` is identified as a write region and, thus, the appropriate PUT/GET primitives are generated to copy the region.

Figure 4 shows the code that results after applying the copying scheme to the subroutine `f` in Figure 3. The private array `v` is dynamically allocated here. Notice that the `I`-loop no longer contains any remote memory access.

The copying scheme brings us several benefits. One is that we can take advantage of prefetching strategies. Another is that we can reduce communication overhead by copying data blocks instead of single data items. This scheme is particularly useful when the data redistribution is frequently required, as in the case of array `X` in Figure 1: instead of total redistribution of `X`, the processors can prefetch and poststore the exact portions of `X` that they access in each loop. Similar to data privatization, the scheme pro-

```
      subroutine f(V,L)
      real W(LEN/P,2), V(*), v(*) ···
cdir$ private T, I, IL, IH, W, v
cdir$ shared V(BLOCK)
  ···
      call g(V,W,L)
   ···
      IL = L/P*my_pid+1
      IH = L/P*(my_pid+1)
      call alloc(v(1:IH-IL+1))
      do I = IL, IH
         T = W(I-IL+1,1) + W(I-IL+1,2)
         v(I-IL+1) = T ···
      enddo
      call polaris_put(V(IL),v(1),1,1,IH-IL+1)
      call dealloc(v)
        .
        .
        .
```

Figure 4. Subroutine `f` after data localization

vides an additional benefit of helping processors to better utilize caches in non-cache coherent machines by enforcing more private arrays to be referenced.

```
      do I = 1, M
         do K = 1, N
            do L = 1, 1+M-I
               ··· = X(K+(I-1)*N+L*N)
            enddo
            X(K+(I-1)*N) = X(K+(I-1)*N) ···
         enddo
      enddo
                  ⇓
cdir$ private x, ···
cdir$ shared X(BLOCK), ···
   ···
    call alloc(x(1:(M-IL+2)*N))
    call polaris_get(X(1+(IL-1)*N),x(1),1,1,(M-IL+2)*N)
    do I = IL, IH
       do K = 1, N
          do L = 1, 1+M-I
             ··· = x(K+(I-IL)*N+L*N)
          enddo
          x(K+(I-IL)*N) = x(K+(I-IL)*N) ···
       enddo
    enddo
    call polaris_put(X(1+(IL-1)*N),x(1),1,1,(IH-IL+1)*N)
    call dealloc(x)
```

Figure 5. Loop `predic_do1000` from `mdg` with induction variables substituted, and its parallelized version after removing anti dependence with PUT/GET

The copying scheme is also of use to eliminate anti and output dependences. For instance, `predic_do1000` in Figure 5 originally has only anti dependences; hence, this loop can be parallelized and block-scheduled, and the array `X` is then renamed as a private array. Thus the *upwards exposed use* regions of `X` can be copied to the private array with `polaris_put/get`. In the parallel version of this loop, the processors that execute the block-scheduled loop iterations from `IL` to `IH` get the upwards exposed region `X(1+(IL-1)*N:N+M*N)`, and put the downwards exposed region `X(1+(IL-1)*N:IH*N)`.

5. Experiments

To measure the effectiveness of our transformation techniques, we are implementing code generators for various types of NUMA systems. Figure 6 shows the experimental results obtained on the Cray T3D [4]. In this section, we will briefly describe the T3D and analyze the experiments.

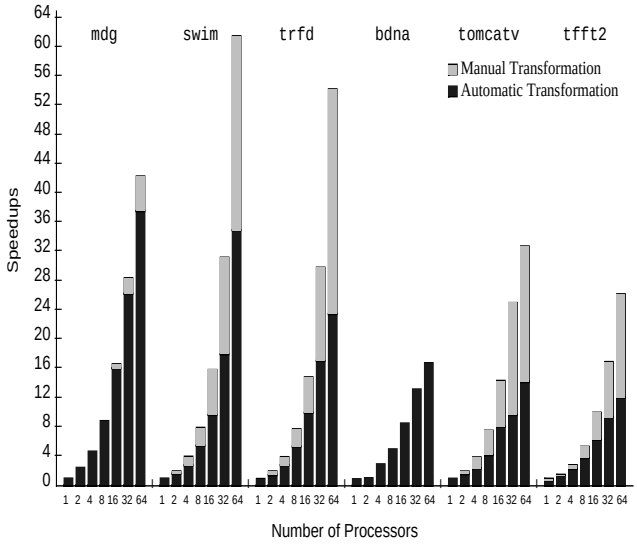

Figure 6. Comparison of parallel speedups on the T3D with six benchmarks: 'Automatic' represents the speedups obtained by Polaris, and 'Manual' represents those obtained by performing hand-optimization in addition to automatic ones

5.1. The Cray T3D

The T3D has several powerful hardware features to efficiently support shared-memory programming on top of physically distributed memory structures. The 3-D torus interconnection network provides high-throughput and low-latency remote memory access. Its prefetch queues are designed to fetch remote data and are effective at hiding the memory latency. A special barrier network is very useful for global synchronization. The T3D does not support hardware cache coherence, but it has hardware mechanisms to facilitate efficient software control of local memory coherency and PUT/GET operations through library primitives [5]. The shared-memory programming model [3] implemented in the T3D provides all the necessary characteristics that we described in Section 3.

5.2. Automatic Transformation

Our performance analysis reveals three main factors that affect parallel speedups in our approach: parallelism detected by Polaris, the amount of data being privatized, and

the amount of data being moved in the shared data copying scheme. It is usually difficult to accurately quantify the effects of each compiler technique on these factors because performance results from the combination of various techniques in most cases. For instance, the speedups in Table 2 of Section 4.1 are not possible without the Region Test's identification of the major loops in `tfft2` as parallel; yet, the Region Test could not detect those parallel loops without the Polaris privatizer's removal of anti and output dependences from the loops. Nevertheless, in Table 3, we try to single out the impact of each technique we used in the transformation. In the table, the rows correspond to the techniques and the columns to the benchmarks. If a technique is important for a program, the corresponding entry is marked.

	bdna	mdg	trfd	swim	tomcatv	tfft2
PD_1	×	×	×	×	×	
PD_2	×	×	×		×	×
LT					×	
DP	×	×	×			×
DL	×	×	×	×	×	×

Table 3. Impact of the techniques on the benchmarks

PD_1 and PD_2 stand for the parallelism detection techniques. We assume that PD_1 is applied first; then, PD_2 is applied to loops not parallelized by PD_1. PD_1 includes the traditional dependence tests, such as the GCD test and the Range Test. PD_2 corresponds to the Region Test and supplementary tests for loops with I/O operations. As already mentioned, the Region Test parallelizes the important loops in `tfft2` that PD_1 failed to do, and also enables us to deal with small loops in `mdg` and `trfd` so that we can obtain good speedups on a larger number of processors. We found that `bdna` and `tomcatv` need to read large files, causing the I/O time to take a significant portion of the total execution time. So, we developed a simple test to parallelize the I/O operations. In `swim`, PD_1 parallelizes most of the loops, thus leaving few loops to PD_2.

LT stands for the conventional loop transformation techniques, which were useful in our approach because they help neighboring loop nests to reuse data stored in local memory. In `tomcatv`, LT doubles the speedups. DP stands for the data privatization. The superlinear speedups in `mdg` are ascribed to DP, which privatizes most of the important arrays in the program, as well as to PD_1 and PD_2, which parallelize almost all the loops in it. DL stands for the data localization. Our experience with scientific codes reveals that block distribution generally works well with DL because many loops access arrays in a contiguous way. Also, block distribution facilitates the calculation of the target location of the data to be copied within `polaris_put/get` calls. To find the effectiveness of DL with our choice of block distribution policy for shared arrays, consider Table 4, which indicates that DL is effective across all the programs, especially for `trfd`.

PE	bdna	mdg	trfd	swim	tomcatv	tfft2
2	1.0	1.3	4.0	2.2	1.7	2.0
8	1.4	1.3	4.6	2.4	1.2	1.8
32	1.5	1.2	5.8	2.1	1.4	1.4

Table 4. Performance improvement due to DL: the entries represent the value $\frac{\textit{speedup with } DL}{\textit{speedup without } DL}$, which reflects the effectiveness of DL for each program.

5.3. Manual Transformation

To estimate how closely our compiler techniques approach the speedups that can be achieved manually, the six benchmarks were hand-optimized based on the work of Navarro, et al [12], and the performance results are shown in Figure 6. In the hand-optimized versions, the overall communication overhead is reduced by aggregating data to be copied in the shared data copying scheme based on cross-loop analysis. To illustrate this, consider Figure 7. For the shared array `X`, the automatic copying scheme generates two private arrays, `x0` and `x1`, and two GET operations because `X` is accessed in two different loop nests. However, cross-loop access region analysis can identify that the region of `X` read by the `I`-loop completely covers the region by the following `J`-loop, and thus the second GET operation is redundant.

```
cdir$ private x0, x1, ···
cdir$ shared X(BLOCK), ···
      ···
      call polaris_get(X(L),x0(1), 1, 1, U-L+1)
      do I = L, U, 1
         ··· = x0(I-L+1)
      enddo
       ···
      call polaris_get(X(L),x1(1), 2, 1, (U-L)/2+1)
      do J = L, U, 2
         ··· = x1((J-L)/2+1)
      enddo
       ···
```

Figure 7. Redundant PUT/GET elimination

Another important issue in the hand-optimized versions is data distribution. Our current data distribution strategy is *access-pattern-insensitive*, meaning that we simply choose `BLOCK` distribution regardless of the data access patterns in a program. As discussed earlier, this naive distribution policy assists the copying scheme to some extent. For many programs, however, it is often suggested that more sophisticated data distribution policies improve performance. Therefore, more access-pattern-sensitive data distribution strategies were chosen in the hand-optimized code. The experimental results show improved performance for all the programs except `bdna`, whose most time-consuming loops contain many subscripted-subscript expressions, making it difficult for us to find better distributions statically by hand.

The machine-specific parameters, such as memory latency, network bandwidth, and cache size, are also important in code transformation. In the hand-optimized versions, these parameters were carefully handled when code generation decisions are made. This was another factor that accounts for gains in the performance of manual transformation. For instance, following the general rules, the shared data copying scheme preferred copying a large approximate data block instead of copying small exact fragments of the data, so as to reduce total memory access latency in the copy operation. Often the T3D's low latency network made our design choice less efficient, unlike other machines with high remote memory latencies.

6. Conclusion

We discussed our recent efforts to develop compiling techniques to automatically transform a sequential program into a shared-memory style parallel program for NUMA machines. We presented our transformation procedure and several new techniques we have developed. Combining these techniques with the traditional compiler techniques, we transformed sequential codes from the Perfect and SPEC benchmarks, and experimented with the parallelized codes on the Cray T3D.

We reported the experimental results in this paper. We profited from the fact that the Cray T3D supports fast barrier and PUT/GET operations based on low latency network. Since the T3D does not support hardware cache coherence. code is generated to copy shared data into private variables to enable the utilization of caches in the machine.

Finally, we presented and analyzed the experimental results. We believe that the results demonstrate the potential for the effectiveness of today's compiling techniques that help us obtain acceptable multiprocessor speedups for NUMA machines with a global space. The manual experimental results discussed in Section 5.3 provide ideas about other issues necessary to further improve and enhance existing compiling techniques for other types of NUMA machine.

References

[1] E. Ayguade, J. Garcia, M. Girones, J. Labarta, J. Torres, and M. Valero. Detecting and Using Affinity in an Automatic Data Distribution Tool. In *Lecture Notes in Computer Science*, pages 61–75. Springer Verlag, New York, New York, August 1994.

[2] 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.

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

[4] Cray Research Inc. *The CRAY T3D System Architecture*, 1993.

[5] Cray Research Inc. *SHMEM Technical Note for Fortran*, 1994.

[6] Cray Research Inc. *The CRAY T3E-900 Scalable Parallel Processing System*, 1996.

[7] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihara, and T. Shindo. AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. *Proceedings of 6th International Conference on Architechtural Support for Programming Language and Operating Systems*, pages 196–207, October 1994.

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

[9] K.Kennendy and U. Kremer. Automatic Data Layout for High Performance Fortran. *Proceedings of Supercomputing '95*, December 1995.

[10] T. LeBlanc and E. Markatos. Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. *4th Symposium on Parallel and Distributed Processing*, December 1992.

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

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

[13] 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.

[14] D. Palermo and P. Banerjee. Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. In *Lecture Notes in Computer Science*, pages 392–406. Springer Verlag, New York, New York, August 1995.

[15] R. Schreiber. High Performance Fortran 2.0. *Workshop on Challenges in Compiling for Scalable Parallel Systems* in conjuntion with *IEEE 8th Symposium of Parallel and Distributed Processing*, October 1996.

[16] SGI/Cray Research Inc. *Origin^{TM} and Onyx2^{TM} Programmer's Reference Manual*, 1996.