Compiler Techniques for Effective Communication on Distributed-Memory Multiprocessors

Angeles G. Navarro Yunheung Paek† Emilio L. Zapata David Padua†

Dept of Computer Architecture, Univ. of Málaga, Spain
{angeles,ezapata}@ac.uma.es

† Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign
{y-paek,padua}@cs.uiuc.edu

Abstract

The Polaris restructurer transforms conventional Fortran programs into parallel form for various types of multiprocessor systems. This paper presents the results of a study on strategies to improve the effectiveness of Polaris' techniques for distributed-memory multiprocessors. Our study, which is based on the hand analysis of MDG and TRFD from the Perfect Benchmarks and TOMCATV and SWIM from SPEC benchmarks, identified three techniques that are important for improving communication optimization. Their application produces almost perfect speedups for the four programs on the Cray T3D.

1 Introduction

The problem of compiling for distributed memory multiprocessors has been studied extensively in recent years [2, 4, 5]. One of the many projects on this subject centers around *Polaris* [1], a parallelizing compiler which automatically transforms sequential Fortran 77 programs into parallel form without programmer intervention. Unlike most other approaches, the techniques currently implemented in Polaris pay little attention to data distribution across processors. In fact, Polaris applies a simple data distribution strategy which block-distributes all shared data objects in the parallel programs regardless of their access patterns. The lack of data distribution strategies is compensated by the use of advanced techniques for privatization and communication optimization and for work distribution [3]. These techniques have proven quite effective on a collection of codes from the Perfect and SPEC Benchmarks when the target machine is the Cray T3D.

Despite these good results, there is still much room for improvement. This paper presents the results of a study on additional techniques needed to improve the effectiveness of Polaris for distributed-memory multiprocessors. Our study, which is based on the hand analysis of the four programs, TOMCATV and SWIM from SPEC benchmarks and MDG and TRFD from the Perfect Benchmarks, identifies three new techniques important to improving Polaris' effectiveness.

2 Automatic Parallelization

When generating code for distributed-memory machines with a global address space, Polaris applies five passes: a *parallelism detection* stage, a *work partitioning* stage, a *data privatization* stage, a *data distribution* stage, and a *data localization* stage. A more detailed description can be found in [3].

Starting with a conventional Fortran77 program, Polaris generates a parallel version for a global address space machine. The target code has a Single Program Multiple Data (SPMD) form. Barriers and locks are used to control explicitly the flow of execution of processors in the SPMD code. Program variables are declared explicitly as either `private` or `shared`. Shared arrays can be distributed by `BLOCK` and `CYCLIC` directives. PUT/GET operations are used to allow the processors asynchronous access to any data object in the system.

In the first phase of the transformation procedure, Polaris detects parallelism from the input program [1]. In the second phase, Polaris tries to distribute parallel work evenly while ignoring data distributions. In the third phase, Polaris analyzes the data regions that each individual processor will access and identifies the `private` data. In the fourth phase, Polaris declares all non-privatized arrays as `shared` and `BLOCK`-distributed across the target machine. Then, in the fifth phase, Polaris inserts PUT/GET operations (`polaris_put` and `polaris_get` routines) to localize the non-local accesses to these shared arrays by following the *shared data copying scheme* [3]. In this

scheme, shared memory is used as a repository of values for private memory.

3 Additional Optimization Techniques

As mentioned in Section 1, we have found that the techniques enumerated in Section 2 produce good speedups on the Cray T3D for the collection of programs we have evaluated. In fact, a previous study [3] showed that data privatization substantially increases the probability that the processors fetch their data from local memory, thus reducing the overall communication overhead. Furthermore, data privatization presents additional benefits in the T3D by providing more chances for processors to use data caches for their computations. The reason is that, in this machine, shared data are not cached even when they reside in local memory.

Through the hand analysis reported in [6], we identified some additional optimizations that are needed to further improve performance. Strategies to reduce the number of PUT/GET operations, which are based on the cross-loop access region analysis, are discussed in Section 3.1. A second optimization is to improve data locality by distributing arrays according to the data access pattern in a program. In Section 3.2, we discuss one data access pattern commonly encountered in an important class of scientific applications; we also present an automatic data distribution technique that minimizes the communication costs and memory requirements for these applications.

3.1 Communication Overhead

Because the communication optimization algorithm currently implemented in Polaris is relatively simple, the communication overhead is sometimes unnecessarily large and scalability is hindered. In the next two subsections we analyze where this overhead arises and propose additional techniques based on access region analysis to reduce the overhead.

3.1.1 Placing Communication Operations

In the shared data copying scheme, it is crucial to avoid consuming an excessive amount of space for private data. Also, it is important to minimize communication overhead by reducing the calls to `polaris_put/get` primitives. In this scheme, the *copy-level* is a loop nest level at which the elements of the shared array are copied by using these primitives. There is an obvious trade-off between time and space when choosing the copy-level. Currently, Polaris usually chooses the innermost copy-level since this copy-level allows us to exploit the *data pipelining* technique if the implementation of the `polaris_put/get` routines is *non-blocking*.

In real cases, we have found that it is often better to perform copy operations at outer copy levels. Therefore, the strategy we propose here is to use access region analysis to gather the elements of each shared array accessed at all inner levels of nesting, and to perform copy operations at the outermost loop level possible. Although this strategy may consume more space for private data, there are several advantages to copying at the outermost levels: the amount of private memory needed for each processor decreases with the number of processors; the memory required to allocate private data can be managed using dynamic allocations functions; and, saving space is less important than optimizing time in general cases.

3.1.2 Redundant PUT/GET Elimination

The parallel programs generated by Polaris contain many redundant PUT/GETs due to the lack of a cross-loop analysis for the shared data copying scheme. The scheme currently implemented in Polaris generates `polaris_put/get` calls for each individual loop nest. The elimination improved the execution times and scalability of studied programs, as we will discuss with real programs in Section 4. In order to eliminate redundant PUT/GETs, we need to extend the access region analysis to a set of consecutive loops. Based on the cross-loop access region analysis, we try to determine the *upwards-exposed regions* and *downwards-exposed regions* for the shared arrays in the set of loops.

3.2 HALO Access Pattern

Data distribution is an important issue in the code generation for distributed memory multiprocessors. As discussed in Section 2, the current Polaris data distribution strategy is *access-pattern-insensitive* in that it simply chooses `BLOCK` distribution regardless of the data access patterns in a program. For very regular programs, however, previous work suggests that other data distribution policies improve performance. These strategies could be implemented in the data distribution stage in the Polaris transformation procedure to complement data privatization and localization techniques in Polaris.

In our studies, we focused on the regular data access pattern in a loop where all the subscript expressions for the array of interest are of the form: `X(I` $\pm$ `K)`, where `I` is the loop index and `K` is an arbitrary constant. We call this type of access pattern the *HALO access pattern*. The HALO access pattern is similar to a new data distribution pattern, called `SHADOW` region, which is included in one of the approved extensions of the new specification of High Performance Fortran.

We have found that the current strategy of Polaris causes excessive communication for arrays with that kind of access pattern. These communications cannot be eliminated by the techniques presented in Section 3.1. The proposed algorithm identifies when an array `X` has the HALO access pattern and is given a new distribution type `HALO` to denote this access pattern. The other arrays will still have the `BLOCK` distribution.

Whether the distribution type is `HALO` or `BLOCK`, both types of arrays are declared as shared and are block-distributed with the directive `BLOCK` in the data distribution stage. However, in the data localization stage, we allocate arrays with the `HALO` distribution to private memory of processors additional private space, called the *halo area* and the *frontier area*. To define the halo area and the frontier area, we refer to the example in Figure 1, which shows the `HALO` data distribution for array `X`.

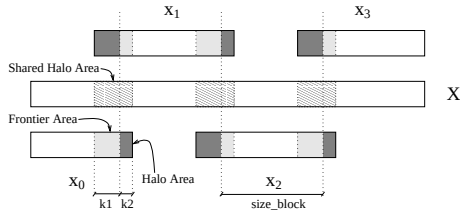

Figure 1: Halo Area and Frontier Area for array `X` with four processors

In the parallel code, for the array `X` with `HALO` distribution, we can hoist the PUT/GET operations out of the loop where `X` is accessed to copy the initial input and the final result for the loop computation. In addition, we now need extra operations, *create halo* and *update halo*, which are described as follows:

Create halo performs a `polaris_get` operation to copy the initial value to the halo area of processors from the portion of the shared array corresponding to the halo area, called the *shared halo area*. This create halo operation involves the communication between neighboring processors.

Update halo deals with the intermediate results generated during the loop execution. The changes in the frontier area of processors during the current iteration t of the loop must be copied to the halo area of their neighboring processors before starting the next iteration $t+1$. This operation involves two steps: first, all the frontier areas of the processors are written back to the corresponding shared halo area simultaneously; then, the processors update their halo area by copying the corresponding shared halo area.

The halo area and shared halo area can be generalized for multidimensional data arrays, where Polaris must build a halo area and a shared halo area for each dimension, and manage each as we have explained for one-dimensional arrays.

4 Case Studies

All the results presented in this section are based on experiments on the T3D, one of the commercial distributed-memory machines that Polaris targets. The T3D is a scalable machine with special hardware and software features to support a global address space efficiently.

The programs we have studied were parallelized automatically first with Polaris and manually later. The automatic versions were obtained from the transformation procedure described in Section 2. The speedups and efficiencies shown here have been calculated versus the sequential execution time. Based on the performance analysis of the automatic versions, we developed the optimization techniques discussed in Section 3, and applied the techniques to generate the manual versions.

Figure 2 shows the speedup comparisons from 1 to 64 processors, using the automatic and manual versions of TOMCATV, TRFD, SWIM, and MDG.

4.1 TOMCATV

The computational kernel of TOMCATV is the `MAIN_do140` loop. This loop is a multiply-nested serial loop with several inner loops parallelized by Polaris. Polaris generates PUT/GET calls around the inner parallel loops. Cross-loop analysis revealed that there are redundant copy operations in this automatic version. We, therefore, manually applied the techniques discussed in section 3.1.2. We detected that two important arrays used in the `MAIN_do140` loop have the HALO access pattern. Therefore, we applied the technique described in Section 3.2 to further reduce the communication overhead. If we consider only the loop `MAIN_do140`, we get superior efficiency of 98% for 32 processors and 95% for 64 processors. As shown in Figure 2 the manual version improves the speedups by approximately a factor of two.

4.2 TRFD

TRFD is another good example of the importance of access-pattern-sensitive data distribution strategies. Polaris does not exploit locality of data on loop iterations due to its lack of a more complete data distribution stage. We identified array `X` as the most important array in TRFD. By using access pattern information, the columns of `X` can be privatized. Also,

we found that the shared data copying scheme chooses a fourth level of the nested loops as a copy-level for the array X to generate PUT/GETs. By hoisting the PUT/GET operations out of the innermost loop nests, the manual version tried to minimize the communication overhead resulting from the copying operations. Using all of these tactics we can achieve a speedup of 30 on 32 processors and 55 on 64 processors for the whole program. These results show, as other researchers have indicated, the importance of the access-pattern-sensitive data distribution strategies to complement the data privatization and localization techniques in Polaris.

4.3 SWIM

SWIM contains an outermost serial loop that calls four subroutines. All these subroutines have the same loop nest structure and the same access pattern to data arrays. The major loop nests in SWIM are all doubly-nested and Polaris parallelizes all these loops, thus generating PUT/GETs for each loop nest. Similar to TOMCATV, the major overhead of SWIM in the automatic version of Polaris comes from redundant PUT/GET operations between consecutive loops that access the same data region. We found that these redundant operations could be eliminated by cross-loop analysis. Also, in SWIM, most arrays are found to have the HALO area patterns. Based on all these analyses of this program, we manually transformed the original SWIM to a parallel form for the T3D and we achieved a linear speedup for the whole program and a global efficiency of 98% for 32 processors and 96% for 64 processors.

4.4 MDG

In MDG we achieved superlinear speedups (on fewer than 16 processors), or almost the linear speedups. As can be seen in Figure 2, we were unable to develop a manual version based on conventional data distribution techniques that outperformed the automatic version. One reason is that MDG has irregular data access patterns and a single static distribution cannot be determined to satisfy all the data access patterns in the program. Also, there are frequent requirements for reduction operations which requires expensive global communication. As a result, we conclude that, in codes that need frequent global data sharing or that contain irregular data access patterns, the shared data copying scheme can be a better solution than more elaborate data distribution algorithms. In Figure 2, the manual version shows slightly better scalability in the speedup curves on more than 32 processors. This is made possible by the techniques for reduction of communication overhead discussed in Section 3.1.

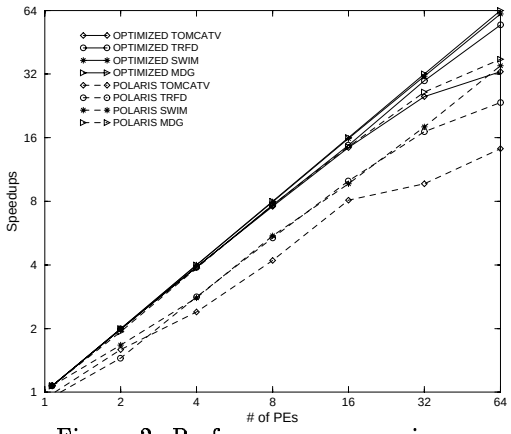

Figure 2: Performance comparison

5 Summary and Conclusions

In this paper, we analyzed several optimization issues for Polaris and outlined some new techniques to overcome limitations of the original techniques in Polaris. To measure the impact of these new techniques, we applied them manually in some benchmarks. We reached efficiencies of 98% for SWIM, 99% for computational kernel of TOMCATV (i.e., scorning the reading file), 93% for TRFD, and 84% for MDG for 32 processors on the T3D. Through our study, we concluded that if a few simple data distribution and communication techniques are combined with the compiling techniques in Polaris, it is possible to automatically generate the parallel code for the distributed memory multiprocessors and to obtain good parallel performance.

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, P. Tu, "Parallel Programming with Polaris", IEEE Computer, pp. 78-82, Dec. 1996

[2] S. Hiranandani, K.Kennedy, and C. Tseng. Compiler Optimizations for FORTRAN D on MIMD Distributed-Memory Machines. *Proc. of Supercomputing'91*, 1991.

[3] Y. Paek, D. Padua, "Compiling for Scalable Multiprocessors with Polaris", *To appear in Parallel Processing Letters*, World Scientific Publishing, UK, 1997

[4] B. Chapman, P. Mehrota, H. Moritsch, H. Zima, Dynamic Data Distributions in Vienna Fortran, *Supercomputing '93 Proceedings*, 1993

[5] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *Journal of Parallel and Dsitributed Computing*, 2(3):361-376, 1991.

[6] A. G. Navarro, Y. Paek, E.L. Zapata, D. Padua, "Performance Analysis for Polaris on Distributed Memory Multiprocessors", 3rd Workshop on Automatic Data Layout and Performance Prediction, Barcelona, Spain, Jan. 1997.