Communication Analysis for Shared and Distributed Memory Machines

Chau-Wen Tseng
Department of Computer Science
University of Maryland
College Park, MD 20742
E-mail address : tseng@cs.umd.edu

Abstract

Advances in programming languages and parallelizing compilers are making parallel computers easier to use by providing a high-level portable programming model that protects software investment. However, experience has shown that simply finding parallelism is not always sufficient for obtaining good performance from today's multiprocessors, largely because the cost of interprocessor communication is much greater than computation or local memory accesses. To overcome this problem, I believe compilers need to perform communication analysis *to locate and optimize interprocessor communication. I show how communication analysis has been used to improve performance for both shared and distributed memory machines, and describe a new project to apply these techniques to compilers for software distributed-shared-memory (DSM) systems.*

1 Introduction

High-performance computing is rapidly becoming an important component of scientific research and development. Today's scientists and engineers depend on the computing power of supercomputers and fast workstations to solve many important problems in fields such as aeronautics, physics, biology, and medicine. Though microprocessor speeds are continuing to increase, most observers agree that parallel computing represents the only plausible way to significantly increase the computational power available. Multiprocessor workstations are becoming widely available, and almost all large computer vendors offer or are developing scalable multiprocessor systems.

One of the major obstacles to widespread usage of parallel computers is the large number of parallel architectures available. Parallel machines include multiprocessors (e.g., KSR-2, Convex Exemplar) that have physically distributed memories but a shared address space. Other architectures include large, loosely-connected message-passing multiprocessor systems (e.g., Intel Paragon, IBM SP-2) and clusters of workstations connected by a network. For brevity, I shall call these groups *shared-memory* and *distributed-memory* machines, respectively.

Each parallel machine has its own machine-specific features that must be considered in order to achieve efficient use of the underlying hardware. Shared-memory machines typically communicate through coherent caches maintained through an invalidation-based protocol. Accounting for the cache line size and associativity to avoid contention, cache conflicts and false sharing while improving cache line utilization is key to good performance. Distributed-memory machines, on the other hand, typically communicate through explicit messages with software buffering. Aggregating messages to amortize communication overhead, overlapping communication and computation, and exploiting specialized communication patterns become vital for achieving high performance. Despite the many difference, one quality shared by all parallel architectures is the high cost of interprocedural communication relative to local computation and memory accesses.

Due to the performance differences between architectures, a great deal of programming effort is required to make an application perform well for any given parallel architecture. Because parallel machines typically have short lifespans, scientists are understandably reluctant to invest their time in writing and tuning parallel applications, since they have no way of protecting their software investment. In order to encourage users and make parallel computing successful, users need an *easy-to-use, portable, yet efficient* method of programming parallel machines.

Fortunately, recent developments in programming language and parallelizing compilers have made it easier to write *machine-independent* programs that can execute across a number of parallel architectures. Users can either write sequential program for a parallelizing compiler (e.g., SUIF) or write data-parallel programs with some data decomposition hints (e.g., HPF). These systems make parallel machines more usable; however, they may produce programs which execute much slower than hand-optimized codes. Usually, the loss in performance is caused by high interprocessor communication or synchronization.

2 Communication Analysis

Since reducing interprocessor communication is critical to achieving good parallel performance, compilers for parallel architectures must manage locality and communication. Just as for uniprocessors, precise compile-time analysis is key to enabling and guiding compiler optimizations fo parallel machines. I believe future parallelizing compilers need to perform communication analysis to locate and optimize interprocedural communication. Communication analysis is a set of techniques used to track the flow of data between processors. It was first developed in compilers for distributed-memory machines in order to identify where messages must be inserted into a program to access data on other processors [31]. However, I will show communication analysis is applicable to all multiprocessor architectures.

Communication analysis relies on some knowledge of how computation is partitioned across processors. The actual identity of the processor assigned to each piece of computation is not needed, just which pieces of work are assigned to the same processor. This information allows the compiler to distinguish accesses to local data from accesses to nonlocal data; only the latter causes communication.

The compiler represents collections of loop iterations containing computation as *iteration sets*. The set of loop iterations assigned to a processor is then its *local iteration set*. Given the iteration set for a piece of computation, the compiler can calculate the data accessed for those iterations by translating them into the data space based on the subscript functions in each array reference. The resulting pieces of data are represented as *index sets* for each variable. By intersecting the index sets of data defined by each processor with the index sets of data accessed by other processors, the compiler can determine the *nonlocal index set*, data that needs to be communicated between processors, as well as the identity of the processors doing the communication. These data items can then be mapped back into the computation space by applying the inverse of the array subscript functions, yielding the *nonlocal iteration set*, the set of iterations accessing nonlocal data.

To see how communication analysis works, consider the following example:

```
REAL A(100), B(100)
N$PROC = 4              → P ∈ {0...3}
DIST (BLOCK)::A,B       → Local data = [1:25]
DOALL i = 2,100         → Local computation = [1:25]
  A(i) = B(i–1)         → Nonlocal accesses = B(0)
ENDDO                   → Send B(25) to P+1
```

The two arrays A and B are distributed block-wise among four processors with ids from 0 to 3. This distribution means each processor is assigned 25 contiguous elements, resulting in local index set of [1:25] for A and B. Assuming work is partitioned by a heuristic such as the *owner-computes* rule, each processor is assigned one quarter of the work, resulting in local iteration sets of [1:25].

By applying the subscript function (i–1) for B, the compiler finds out that the data accessed on local iterations is B[0:24]. By intersecting the index set with the local index set, the compiler finds almost all accesses are local. B(0), the sole remaining nonlocal access in the index set, represents communication. By comparing the elements in the nonlocal index set against the local index sets of the other processors, the compiler discovers that the identity of the sending processor for B(0) is the processor to the left. It also determines that B(0) is equivalent to the B(25) on the sending processor.

Communication analysis thus can identify data that needs to be communicated, the loop iterations defining and accessing the nonlocal data, and the producing and consuming processors. This information can then be combined with traditional data dependence and data flow analyses to guide compiler optimizations to reduce interprocessor communication. This paper summarizes results from a number of papers [33, 32, 58, 59] to demonstrate how communication analysis is useful for both shared and distributed memory machines to improve performance. In addition, I discuss how these techniques may be extended for more advanced multiprocessors.

3 Compiling for Distributed-Memory Machines

Distributed-memory multiprocessors (e.g., Intel Paragon, IBM SP-2) have disjoint address spaces. As a result, the compiler must insert explicit send and receive messages in the program to communicate data between processors. Fortran D is a version of Fortran extended to allow users or automatic tools to specify how data may be partitioned onto processors. It is designed to provide a simple yet efficient machine-independent data-parallel programming model, shifting the burden of optimizations to the compiler [28, 31, 57].

Given a data decomposition, the Fortran D compiler automatically translates sequential programs into efficient parallel programs, using communication analysis to determine what data needs to be communicated and inserting the necessary messages. The two major compilation steps are partitioning the data and computation across processors, then introducing communication for nonlocal accesses where needed. The compiler partitions computation across processors using the "owner computes" rule—where each processor only computes values of data it owns [15, 24, 49].

The Fortran D project contributed to the development of High Performance Fortran (HPF), an informal Fortran standard adopted by researchers and vendors for programming massively-parallel processors [37]. The success of HPF hinges on the development of compilers that can provide performance satisfactory to users. In this section we illustrate how communication analysis can be used to reduce communication costs for distributed-memory machines.

3.1 Communication Optimizations

The Fortran D compiler performs a large number of communication and parallelism optimizations based on combining communication analysis with standard data flow and dependence analyses [33]. Communication analysis is key because it identifies data that must be communicated, as well as the iterations defining and accessing the data and the processors involved. The compiler can then use this information for optimizations, some of which are shown in the figure at the top of the next page.

The most important communication optimization is *message vectorization*, a loop-based optimization fundamental to the Fortran D compilation process [9, 24]. It uses the results of data dependence analysis [2, 39] to extract communication identified by communication analysis from within loops, combining element messages per loop iteration with one vectorized message preceding the loop. Because overhead for messages is high, combining messages can significantly reduce communication cost.

Once messages for each variable have been extracted out of loops, they may also be coalesced to eliminate redundant data and reduce the number of messages further. Message aggregation combines messages from different arrays to the same processor, at the expense of copying them to a single contiguous buffer. By looking at the sending and receiving processors, the compiler may also detect communication patterns such as broadcast that can efficiently supported through collective communication routines.

Compiler optimizations can also seek to hide communication overhead by overlapping messages with computation. Vector message pipelining attempts to hide message

Fortran D Communication Optimizations

```
post B,C,D
send B,C,D
{ computation, communication }
recv B,C,D
DO i = 1,100
  A(i) = B(i+10) + B(i+11) + C(i+10) + D(100)
ENDDO
```

unbuffered messages
vector message pipelining
iteration reordering
message vectorization

message coalescing
collective communication
message aggregation

- Implemented in prototype compiler
- Evaluated on iPSC/860 and CM-5

transit time by separating *send* and *recv* primitives for messages. Iteration reordering extracts local loop iterations to increase the amount of computation that can be overlapped. Unbuffered messages rely on architectural support to eliminate extra copying of messages by posting a destination address in the program before messages arrive.

3.2 Experimental Results

The prototype Fortran D compiler incorporates a large number of communication optimizations. It is implemented as a source-to-source Fortran translator in the context of the ParaScope parallel programming environment [18]. It utilizes existing tools for performing dependence analysis, program transformations, and interprocedural analysis.

3.2.1 Hand-Optimized Code Comparison

To evaluate the status of the prototype Fortran D compiler, the output of the Fortran D compiler was compared with hand-optimized programs on the Intel iPSC/860 and the output of the CM Fortran compiler on the TMC CM-5 [32]. The comparisons with hand-optimized programs are presented in the figure on the top half of the next page. Solid and dashed lines correspond to speedups for hand-optimized and Fortran D compiler-generated programs, respectively. Results show that the Fortran D compiler results varied for the smaller kernels on the top row of graphs, but were more consistent for larger programs in the bottom row.

By generating output for SHALLOW and DISPER that virtually matched their hand-optimized versions, the Fortran D compiler has demonstrated its effectiveness for parallel stencil computations, despite not producing the most efficient communication. The compiler succeeds because it does a sufficiently good job that communication costs become a minor part of the overall execution time. Scalability is excellent because performance improves as the problem size increases. Implementing additional optimizations is desirable for achieving good speedup for small programs or many processors, but is not crucial.

In comparison, there is considerable room for improvement when compiling communication-intensive codes such as linear algebra and pipelined computations. Results for DGEFA and ERLEBACHER show that the prototype Fortran D compiler only attains limited speedups. Noticeable performance gains can be achieved through advanced communication optimizations; they are important because communication is performed much more frequently than in parallel stencil computations. The effect of these optimizations on overall execution time increases in importance as the problem size and number of processors increases.

3.2.2 CM Fortran Compiler Comparison

I also compared the results of the Fortran D compiler on the Thinking Machines CM-5 against the CM Fortran compiler available at that time. The output of the Fortran D compiler was easily ported to the CM-5 by replacing calls to Intel NX/2 message-passing routines with equivalent calls to TMC CMMD message-passing routines. I converted program kernels into CM Fortran by hand for the CM Fortran compiler, inserting the appropriate LAYOUT directives to achieve the same data decomposition.

The CM Fortran compiler can generate two versions of output. The first uses CM-5 vector units, the second only uses the Sparc node processor. Unfortunately, the current TMC Fortran 77 compiler does not generate code to utilize CM-5 vector units, and the node-level CMF compiler was insufficiently robust to perform experiments. Fortran D message-passing programs are thus forced to rely on the Sparc processor. For the purpose of comparison, timings for CM Fortran programs are provided using either Sparc or vector units.

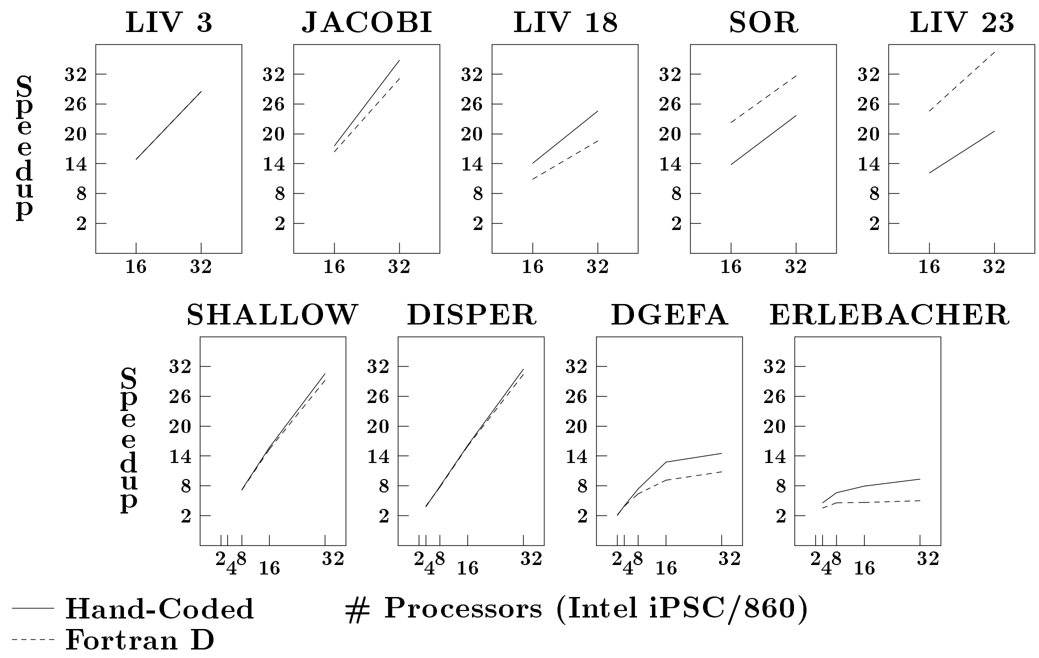
The results are shown in the figure on the bottom half of the next page. Execution times in seconds are plotted logarithmically along the Y-axis. The problem size is plotted logarithmically along the X-axis. Solid, dotted, and dashed lines represent the CM Fortran using Sparc, CM Fortran using vector units, and Fortran D using Sparc, respectively. All parallel execution times are for 32 processors.

Results indicate that when utilizing only Sparc processors, the CM Fortran compiler produces code that is significantly slower than the corresponding message-passing programs generated by the Fortran D compiler. Direct comparisons are somewhat misleading, since the CM Fortran compiler (2.1 *beta*) directly generates Sparc code instead of using the Sparc Fortran 77 compiler.

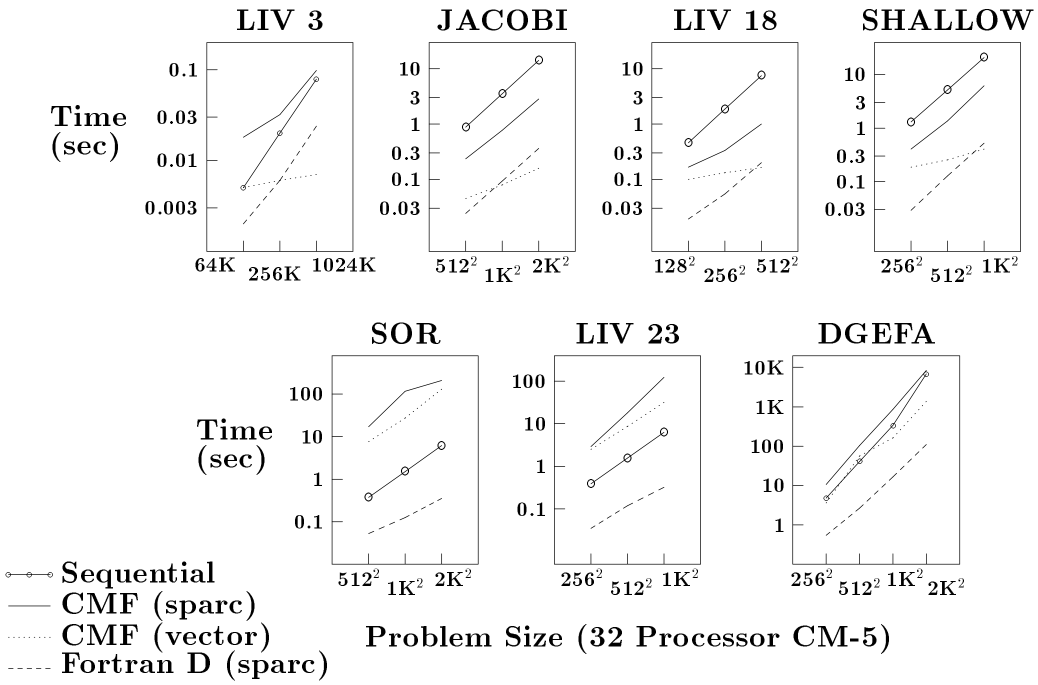
What is more interesting is the fact that for small to medium problems the Fortran D compiler is actually faster than the CM Fortran compiler using vector units. There are a number of reasons for this disparity. First, the code generated by the CM Fortran compiler uses virtual processes, causing extensive run-time calculation of addresses and much unnecessary data movement even for purely local computation. Second, it utilizes a *host-node* model, where a host processor synchronizes global computation on each node. In comparison, the Fortran D compiler utilizes a *hostless* model, eliminating global host-to-node synchronization. Finally, the compiler performs few communication optimizations.

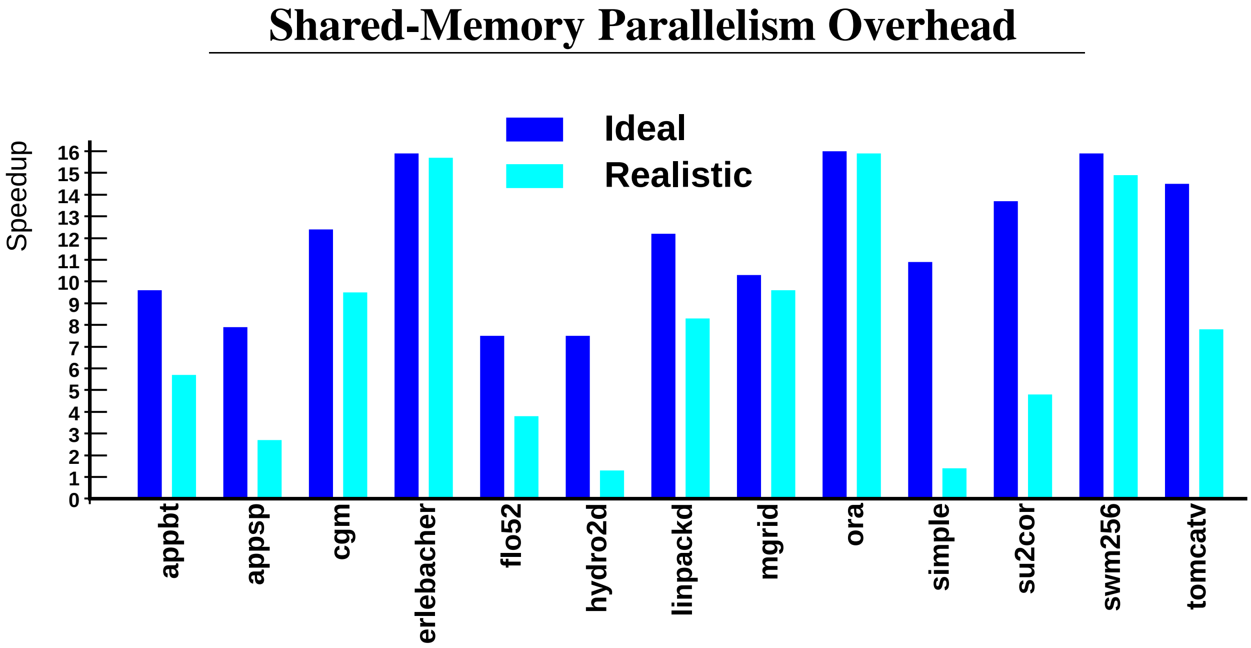
Note that it is not completely fair to compare a research tool like the Fortran D compiler against a commercial product like the CM Fortran compiler. Nonetheless, these experiments prove that severe performance penalties result if communication analysis is not used to reduce communication costs at compile time.

Fortran D vs. Hand-Optimized Code



Fortran D vs. CM Fortran Compiler





4 Compiling for Shared-Memory Machines

Traditionally, research in compilers for shared-memory machines (e.g., KSR-1, Convex Exemplar) has focused mainly on the extraction of parallelism in sequential programs. However, experience with parallelizing compilers has shown that parallelism detection alone is not sufficient for achieving scalable performance. To discover why speedups are lower than expected, consider a recent simulation study by Torrie *et al.* that evaluates the impact of advanced memory systems on the behavior of compiler-parallelized programs [56]. Results show that even if the compiler is able to detect significant parallelism, performance can be rather poor.

The simulation study analyzed the performance of a collection of programs that were successfully (88–100%) parallelized by the SUIF interprocedural parallelizer [27, 26]. Speedups were measured for both an ideal multiprocessor with 1-cycle nonlocal memory access and barrier synchronization latency, as well as a more realistic multiprocessor with 130-cycle access latencies and a 128 KB, 4-way associative cache. On 16-processor system, average application speedups drop from 12.2 to 7.7 when going from the ideal to the realistic machine, with processor cycles wasted due to nonlocal memory accesses and synchronization. These results motivate the need for additional compiler analysis and optimization.

To reduce communication costs on shared-memory machines, the SUIF compiler is being extended in a number of directions by adapting distributed-memory compilation techniques to manage synchronization and data layout at compile time. Probably the most important is performing analysis to automatically colocate data and computation, reducing the need for interprocessor communication. To achieve this goal, the compiler uses a hybrid fork-join / single-program multiple data (SPMD) programming model and relies more on compile-time scheduling of loop iterations. In addition, this section shows how the advanced SUIF compiler can apply communication analysis to reduce synchronization costs and apply data layout optimizations to reduce undesirable memory system effects [59].

4.1 Synchronization Optimizations

Parallelizing compilers for shared address space machines typically employ a *fork-join* model, where a single master thread executes the sequential portions of the program, assigning computation to worker threads when parallel loops are encountered. Typically, synchronization between processes is achieved by having processors participate in a barrier at the end of each parallel loop. Occasionally, point-wise synchronization operations are used to implement doacross loops. Interprocessor communication of data occurs implicitly whenever processors issue load and store operations to remote data.

Barriers required after each parallel loop in current shared address space programs can impose significant overhead for two reasons. First, executing a barrier has some run-time overhead that typically grows quickly as the number of processors increases. Second, executing a barrier requires all processors to idle while waiting for the slowest processor; this effect results in poor processor utilization when processor execution times vary. Eliminating the barrier allows perturbations in task execution time to even out, taking advantage of the loosely coupled nature of multiprocessors. Barrier synchronization overhead is particularly significant when attempting to use many processors, since the interval between barriers decreases as computation is partitioned across more processors.

The key observation used to eliminate barrier synchronization is that synchronization is needed only if processors are communicating data. If there is no interprocessor communication between two parallel regions separated by a barrier, then there is no need for the barrier [58]. Communication analysis can thus be quite useful for reducing synchronization on shared-memory machines. If it can identify the producers and consumers of all data shared between two regions to be identical (*i.e.,* the same processor), then data movement is local and no synchronization is necessary. Note that communication analysis succeeds where traditional scalar and array data dependence analysis fails because it is able to take the compile-time computation partition into account.

The barrier synchronization optimization algorithm is composed of three major components: constructing SPMD regions, eliminating unneeded barriers, and replacing needed barriers with less expensive forms of pair-wise synchronization based on counters. Because barriers are required after each parallel SPMD region, the first step for the compiler is to construct larger SPMD regions by merging adjacent regions or enclosing loops wherever possible, setting the stage for applying communication analysis. Computation added to a SPMD region may be replicated or guarded explicitly to execute on the appropriate processors.

Within a SPMD region, a barrier separating two groups is eliminated if the compiler can determine that there are no race conditions between the code separated by the barrier. The SUIF compiler represents communication and computation using symbolic linear inequalities consisting of array subscript expressions and the computation partition. Detecting communication is then equivalent to solving the system of symbolic linear inequalities, scanning the system using Fourier-Motzkin elimination [4, 5]. If a solution to the system exists, inter-processor data movement takes place and must be synchronized. The solution calculated by the scan can then be used to calculate the identities of the communication processors, the nonlocal data accessed, and the loop iterations where the data is defined [3].

Even after communication analysis identifies interprocessor data movement, it may still be possible to lower synchronization cost by replacing barriers with less expensive forms of point-to-point synchronization such as counters. The compiler tests communication to see whether it fits simple patterns allowing efficient producer/consumer synchronization. If so, the barrier is replaced.

4.2 Data Layout Optimizations

Communication in shared-memory machines is typically implemented at the level of coherent multiprocessor caches. Memory system effects (e.g., conflict misses due to low associativity, replacement misses due to poor spatial locality) can thus significantly degrade cache performance, increasing interprocessor communication. Fortunately, shared-memory compilers can adapt distributed-memory compilation techniques to change data layout and improve performance [59]. The key is to identify data accessed by each processor and change the data layout to make it contiguous, improving spatial locality and reducing memory system effects [6]. In many cases spatial locality may be accomplished by adding an additional dimension to distributed arrays and indexing it according to the processor id. The address calculations for the transformed arrays may contain expensive modulo and division operations. Scalar optimizations are thus also needed to improve the performance of these address calculations [6].

4.3 Experimental Results

To evaluate their impact, the synchronization and data layout optimizations described were implemented in the SUIF parallelizing compiler [61]. On thirty programs from the PERFECT, SPEC, and NAS benchmark suites, the synchronization optimizations were able to eliminate about 40% of all parallel loop invocations (by merging SPMD regions). About 29% of all barriers encountered during program execution were eliminated or reduced, with several order of magnitude reductions in some programs. In comparison, standard data dependence analysis was only able to eliminate on average 2% of barriers encountered.

The effect of synchronization and data layout optimizations on speedups is presented in the figure on the next page. Versions of five kernels and programs with and without each optimization are compared on the Stanford DASH shared-memory multiprocessor. Results show ADI integration, LU decomposition, and Tomcatv significantly benefit from optimization. Using counter synchronization to exploit pipeline parallelism improves the colocation of data and computation in ADI integration. Tomcatv contains several of the parallel loops are inner loops requiring nested barriers that can be eliminated with synchronization analysis. The compiler is able to replace a nested barrier with a counter for LU decomposition, reducing synchronization overhead and improving processor utilization. Data layout optimizations improve spatial locality and eliminate many cache misses for the direct-mapped cache in DASH.

Two other programs, Erlebacher and Shallow, experience less improvement despite significant decrease in synchronization. This behavior demonstrates that once sufficient granularity of parallelism is achieved, further reductions in synchronization overhead has less impact. However, note that for all programs the effect of synchronization optimization grows as the number of processors increase. With enough processors, even these two programs will significantly benefit from optimizations.

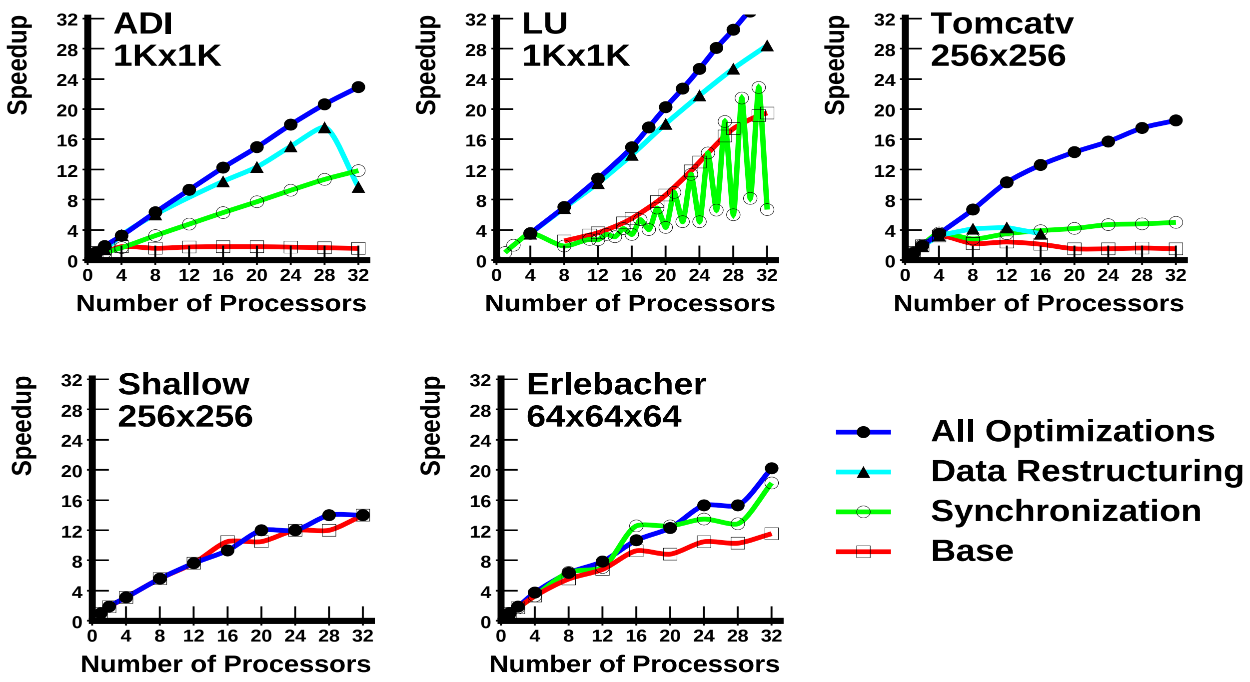
5 Compiling for Software DSMs

Finally, consider the role of communication analysis for future parallel architectures. One approach for building parallel machines is software distributed-shared-memory (DSM) (e.g., Ivy [43], Treadmarks [20]). These systems use operating systems support to provide a shared address space on top of existing message-passing machines and networks of workstations. The latest generation of software DSMs (e.g, Munin [10], Blizzard/Tempest [22], CVM [36]) also support multiple coherence protocols and explicit messages.

Communication in software DSMs is expensive because high software overhead are imposed on top of the hardware costs. Despite this penalty, software DSMs have desirable qualities because they provide a simpler programming interface for users and compilers. By taking advantage of software DSMs, shared-memory compilers can quickly compile a wide range of applications for message-passing machines. Reasonable performance should be obtainable for many coarse-grain parallel applications.

A recent development that makes compilers for software DSMs more desirable is the development of Flexible-Shared-Memory (FSM) machines (e.g, Alewife [1], Flash [30, 40], Typhoon [51, 53]). These architectures

Evaluating Shared-Memory Optimizations



Speedups on 32-processor Stanford DASH Multiprocessor

maintain a coherent shared address space on top of physically distributed memories, just like traditional shared-memory machines. In addition, FSM machines also support extensible memory coherence protocols and explicit messages where needed to achieve better performance. Because modern software DSMs also support the same features (in software), experience compiling for software DSMs may prove valuable in developing and evaluating new coherence protocols that may be used on new flexible-shared-memory machines. Software DSMs thus can serve as testbeds for future FSM machines.

The key question in compiling for software DSMs is whether acceptable performance is obtainable for a wide enough range of applications. I plan to answer this question by developing a compiler for software DSMs based on the SUIF parallelizing compiler. The compiler is based on the Stanford SUIF compiler, and targets the Coherent Virtual Machine (CVM) software DSM system under development at the University of Maryland [36].

By exploiting the operating system support and advanced features of software DSMs such as integrated messaging and multiple coherence protocols, I hope the resulting compiler can approach or exceed the performance of optimized message-passing programs on distributed-memory machines. To achieve high performance, the compiler will need to perform *communication analysis* and apply optimizations for locality, synchronization, communication, and memory system effects.

The compiler follows two basic guidelines. First, it uses compilation techniques for message-passing machines to retain most of the benefits of explicit messages. Second, it exploits architectural and operating system support available in shared-memory multiprocessors to improve flexibility and performance compared to message-passing compilers. I have a preliminary implementation working, and am working on incorporating new optimizations to exploit the multiple coherence protocols and hybrid message-passing capabilities of the CVM system.

6 Related Work

The Fortran D compiler is a second-generation distributed-memory compiler that incorporates and extends features from previous compilation systems [15, 24, 38, 42, 49]. Compared with other contemporary systems [8, 14, 16, 17, 34, 48], The Fortran D compiler is less flexible but performs deeper compile-time analysis, many more advanced optimizations, requires fewer language extensions, and relies on less run-time support.

Few researchers have published experimental results for large programs. Pingali & Rogers apply message vectorization, message pipelining, and reduction recognition in ID NOUVEAU to parallelize SIMPLE [52]. Koelbel & Mehrotra are able to parallelize ADI integration in KALI by implicitly applying dynamic data decomposition between computation phases [38]. Olander & Schnabel show that DINO programs can be significantly improved through iteration reordering and pipelining [47]. Bozkus *et al.* describe the a Fortran 90D compiler that translates Fortran 90 array constructs into calls to a portable run-time library [13]. Results for the first ten Livermore kernels show the compiler is usually within a factor of two of hand-coded programs.

The advanced SUIF compiler builds on shared-memory compiler algorithms for identifying parallelism [11, 27, 26, 60] and performing program transformations [62, 63], as well as distributed-memory compilation techniques to select data decompositions [7] and explicitly manage address translation and data movement [3, 31].

Many previous researchers have examined performance issues on shared-memory architectures. Singh *et al.* [54] applied optimizations similar to those in SUIF by hand to representative computations in order to gain good performance. Others evaluated the benefits of co-locating data and computation [23, 44], as well as false sharing [12, 21, 25, 41, 55]. These approaches focused on individual optimizations and were generally applied by hand.

Researchers compiling for SIMD languages such as Fortran 90 and Modula-2* have been faced with a programming model that assumed the existence of barriers following each expression evaluation [29, 48, 50]. Simple data dependence analysis can be used to reduce barrier synchronization by orders of magnitude, greatly improving performance. In comparison, SUIF begin with parallel loops discovered by parallelizing compilers, where the remaining barriers are significantly harder to remove.

For barriers separating statements on the same loop level, Hatcher and Quinn use a two-dimensional radix sort to find the minimal number of barriers [29]. Philippsen and Heinz find the minimal number of barriers with an algorithm based on topological sort; they also attempt to minimize the amount of storage needed for intermediate results [48]. Cytron *et al.* were the first to explore the possibilities of exploiting SPMD code for shared-memory multiprocessors, concentrating on safety concerns and the effect on privatization [19].

O'Boyle and Bodin apply a classification algorithm to identify data dependences that cross processor boundaries, then apply heuristics based on max-cut to insert barrier synchronization and satisfy dependence [46]. Good results are reported for two programs. Finally, a number of researchers have developed techniques for analyzing synchronization in explicitly parallel programs [35, 45].

7 Conclusion

Communication analysis is a set of compilation techniques to identify interprocessor communication in parallel programs. Because of the high expense of interprocessor communication, I believe communication analysis need to be a fundamental part of parallelizing compilers for advanced parallel systems. Detecting parallelism must be considered to be just the first step in achieving high performance. Just as for uniprocessors, the key to high performance is precise compile-time analysis and optimization. This paper demonstrates the performance benefits of precisely analyzing and optimizing communication on both shared and distributed memory machines, and presents some techniques for applying communication analysis in program optimization. Preliminary experimental results for the prototype compiler implementations have been quite encouraging.

References

[1] A. Agarwal, R. Bianchini, D. Chiaken, K. Johnson, D. Kratz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[2] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[5] C. Ancourt and F. Irigoin. Automatic code distribution. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[6] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[7] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[8] Applied Parallel Research, Placerville, CA. *Forge 90 Distributed Memory Parallelizer: User's Guide*, version 8.0 edition, 1992.

[9] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[10] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[11] W. Blume et al. Polaris: The next generation in parallelizing compilers,. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[12] W. Bolosky and M. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, September 1993.

[13] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka,

and M. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

[14] T. Brandes. Efficient data parallel programming without explicit message passing for distributed memory multiprocessors. Internal Report AHR-92-4, High Performance Computing Center, GMD, September 1992.

[15] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[16] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[17] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. *Journal of Parallel and Distributed Computing*, 16(2):79–91, October 1992.

[18] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[19] R. Cytron, J. Lipkis, and E. Schonberg. A compiler-assisted approach to SPMD execution. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[20] S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel. An evaluation of software distributed shared memory for next-generation processors and networks. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[21] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[22] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

[23] J. Fang and M. Lu. A solution of cache ping-pong problem in RISC based parallel processing systems. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[24] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[25] E. Granston and H. Wishoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[26] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[27] M. W. Hall, S. Amarasinghe, and B. Murphy. Interprocedural analysis for parallelization: Design and experience. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[28] M. W. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[29] P. Hatcher and M. Quinn. *Data-parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.

[30] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J.-P. Singh, R. Simoni, K. Gharachorloo, J. Baxter, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[31] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[32] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[33] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, April 1994.

[34] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[35] T. Jeremiassen and S. Eggers. Static analysis of barrier synchronization in explicitly parallel systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Montreal, Canada, August 1994.

[36] P. Keleher. Multiple writers considered harmful. Technical Report CS-TR-3543, Dept. of Computer Science, University of Maryland at College Park, October 1995.

[37] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[38] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using

Kali. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[39] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[40] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simon, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[41] H. Li and K. Sevcik. NUMACROS: Data parallel programming on NUMA multiprocessors. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, September 1993.

[42] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[43] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *IEEE Transactions on Computer Systems*, 7(4):321–359, November 1989.

[44] E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[45] S. Midkiff and D. Padua. Compiler generated synchronization for DO loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 544–551, St. Charles, IL, August 1986.

[46] M. O'Boyle and F. Bodin. Compiler reduction of synchronization in shared virtual memory systems. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[47] D. Olander and R. Schnabel. Preliminary experience in developing a parallel thin-layer Navier Stokes code and implications for parallel language design. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[48] M. Philippsen and E. Heinz. Automatic synchronization elimination in synchronous FORALLs. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[49] K. Pingali and A. Rogers. Compiling for locality. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[50] S. Prakash, M. Dhagat, and R. Bagrodia. Synchronization issues in data-parallel languages. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[51] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[52] A. Rogers and K. Pingali. Compiling for distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):281–298, March 1994.

[53] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[54] J.P. Singh, T. Joe, A. Gupta, and J. Hennessy. An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[55] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[56] E. Torrie, C.-W. Tseng, M. Martonosi, and M. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.

[57] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[58] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[59] C.-W. Tseng, J. Anderson, S. Amarasinghe, and M. Lam. Unified compilation techniques for shared and distributed address space machines. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[60] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[61] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[62] M. E. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[63] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.