Efficient Machine-Independent Programming of High-Performance Multiprocessors

Chau-Wen Tseng
Department of Computer Science
University of Maryland
College Park, Maryland 20742
(301) 405-8010
tseng@cs.umd.edu
http://www.cs.umd.edu/~tseng/

Summary

A major component of the success of scientific computing is the rapid increase in computing capability. Parallel computing can provide the next great leap in the computation power scientists and engineers need to solve many important problems. The proliferation of parallel architectures, however, discourages users from writing parallel applications. Recent advances in automatic parallelization and parallel languages provide a means for users to write portable programs that protect software investment. Unfortunately, current systems frequently experience poor performance because they fail to fully exploit the features of the underlying parallel architecture.

For uniprocessors, compilers have been quite successful in mapping machine-independent programs (e.g., Fortran, C) down to the complex features of today's microprocessors. I believe that compilers are also well-suited for customizing portable parallel programs. The objective of my research is to *develop compilation techniques to support efficient machine-independent programming of high-performance multiprocessors.* I plan to implement and evaluate these effectiveness of these techniques in COSMIC, a Communication-Optimizing, Shared-Memory Integrated Compiler. COSMIC will target a shared-memory programming model because it provides the best combination of flexibility and performance.

Experiments show that simply exploiting parallelism is no longer sufficient for achieving the best performance, mainly because the cost of interprocessor communication is too great compared to computation and local memory accesses [74, 77]. To achieve high performance, COSMIC will perform *communication analysis* and apply optimizations for locality, synchronization, communication, and memory system effects. COSMIC 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. A novel characteristic of COSMIC will be its ability to take advantage of the multiple coherence protocols and hybrid message-passing support found in software Distributed-Shared-Memory (DSM) systems and Flexible-Shared-Memory (FSM) machines.

To evaluate the impact on the performance on scientific applications, I will test COSMIC on programs from benchmark suites such as Perfect, NAS, SPEC, and SADIE, as well as representative applications (e.g., CHARMM, MOLDYN, DSMC, EULER, NRL Flame) from fields such as computational chemistry, computational fluid dynamics, sparse matrix codes, and computational combustion. I will measure the impact of the optimizations on application performance for the IBM SP-2, DEC Alpha multiprocessor, and clusters of Alpha workstations. My goal is to significantly improve on current compilers for shared-memory architectures and match or exceed the performance of message-passing programs on distributed-memory architectures.

This project will make several important contributions, including: 1) a sophisticated optimizing compiler for shared-memory machines, 2) a prototype compiler for software DSM systems, 3) a prototype compiler for FSM machines, 4) extensive experimental evaluation of advanced compiler optimizations, 5) memory-coherence protocols useful for compiler-parallelized codes, 6) insights towards efficient parallelization of advanced scientific applications. The compilation techniques developed by this project will be of great interest to computer vendors. My findings on useful memory-coherence protocols will also help computer architects and operating system writers. Most importantly, by reducing the penalty of writing portable parallel programs, my compilation techniques will make parallel computing more useful for scientists and engineers.

My educational goals are to provide undergraduate and graduate students with the skills they need to succeed in either industry or academia, and to introduce them to the potential of high-performance computing. I plan to teach a number of courses in compiler construction, with programming projects that make students appreciate the role of compilers in achieving high performance in both sequential and parallel architectures. I maintain slides used in my lectures and other course materials on-line, accessible via a course home page on the World-Wide-Web. Doing so keeps the course materials accessible to students as well as faculty at other institutions interested in teaching the same material. I hope to encourage faculty in undergraduate teaching universities to use these course materials to introduce their students to activities at research universities.

Contents

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. Despite their promise, however, parallel computers are not likely to be widely successful until they are easy to use.

One of the major obstacles to widespread usage of parallel computers is the large number of parallel architectures available. Parallel machines include multiprocessor workstations (e.g., SUN Sparcstation 20MP, SGI PowerChallenge, DEC Alpha Sable) and 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.

The variety of architectures is problematic for two reasons: programmability and performance. First, programming models for architectures may differ dramatically. Shared-memory machines are typically programmed using a global name space, where processors communicate through accesses to shared data. Distributed-memory machines, on the other hand, are typically programmed using a message-passing model, where processors can only communication through explicit messages.

The second problem with the variety of architectures is performance. 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.

Due to these differences in programming models and efficiency, 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.

2 Limitations of Existing Approaches

As we have seen, a key problem in parallel computing is to provide a portable yet efficient means of programming parallel machines. There have been many attempts to overcome this important difficulty, as shown in Figure 1(a). However, these proposals are limited in either usability or performance.

2.1 Parallel Programs and Message-Passing Libraries

One approach is to write explicitly parallel programs. Parallel dialects of Fortran exist for shared-memory machines which provide DOALL and PARALLEL DO annotations that users can easily add to indicate parallelism. However, performance can be poor unless users extensively rewrite programs to take advantage of machine-specific features of the underlying memory system and avoid problems such as poor spatial locality and false sharing [68]. For distributed-memory machines, users can write message-passing programs that use a standard message-passing library such as PVM, P4, PARMACS, or MPI. Users can achieve high performance because they have total control over interprocessor communication and data layout. Message-passing programs, however, must deal with separate address spaces, index translation, and explicit interprocessor communication. Writing efficient parallel programs thus require too much effort for most scientists and engineers.

2.2 Data-parallel Languages

Another solution is to use data-parallel languages such as High Performance Fortran (HPF) [45]. HPF is an enhanced Fortran 90 extended with annotations that specify how data should be partitioned across processors. Compilers have

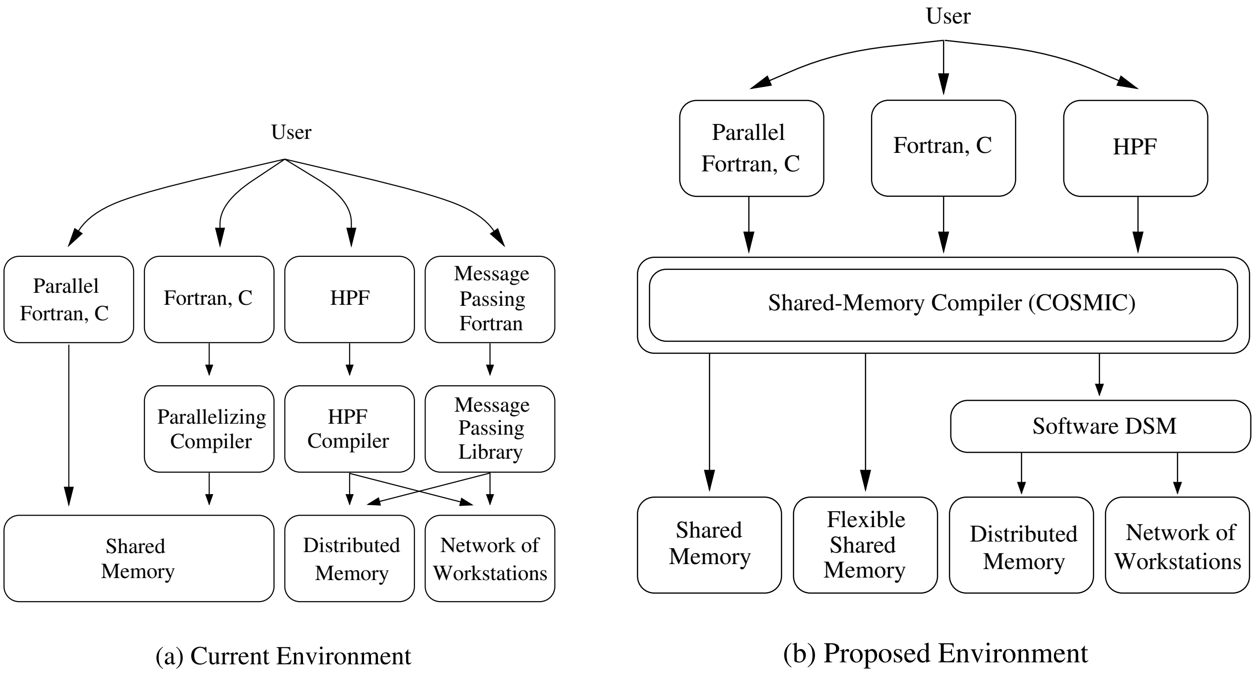

Figure 1 Current and Proposed Parallel Programming Environments

been developed (e.g., Fortran D [36], IBM HPF [27], Kali [47], Paradigm [69] Vienna Fortran [16]) that can translate HPF programs into message-passing programs for distributed-memory machines. Extensive compiler and run-time support (e.g., Chaos [18], Fortran D [31], Kali [46], Parti [66]) have also been developed to handle programs with complicated reference patterns, such as those found in adaptive sparse applications. By ignoring data decomposition annotations, HPF programs can be treated as explicitly parallel Fortran programs and run on shared-memory machines as well. However, this route is unlikely to yield good performance since locality and communication are important even for shared-memory machines.

Though HPF is a good solution for data-parallel applications, there are still a number of disadvantages to using HPF. First, users are forced to rewrite their application using the data-parallel constructs found in Fortran 90. Though the process is not as difficult as writing message-passing code, it may still be laborious for large legacy codes. Some compilers for distributed-memory machines can avoid this problem by automatically detecting data-parallelism in sequential programs (e.g., Fortran D [36]). More problematic is the fact that some applications may contain parallelism not expressible in the data-parallel constructs found in HPF, such as trees and linked lists found in C.

Another limitation is that HPF compilers usually determine the computation decomposition and address translation in the compiler. Much of this information can be calculated at compile time, but requires precise analysis that can easily fail for poorly written or unstructured code. When a HPF compiler cannot analyze the decomposition and address translation at compile time, it must extensive modify the output code to determine ownership and communication at run time. Even when the HPF compiler succeeds in its analysis, it may be forced by the nature of the application (e.g., sparse matrix) to produce code that explicitly translates between global and local addresses or rely on extensive libraries to do so at run time. As a result of its restrictions on parallelism and software overhead, HPF is not usually suitable for programming small multiprocessor workstations.

2.3 Parallelizing Compilers

An alternative approach for achieving portability that minimizes user effort is to use parallelizing compilers. Such compilers can create parallel programs automatically from the original sequential program with little or no user interaction. Parallelizing compilers for distributed-memory machines are limited in their applicability, since precise information is needed. In comparison, parallelizing compilers for shared-memory machines are beginning to mature. Several research prototypes have been developed with powerful symbolic and interprocedural analyses that can automatically exploit parallelism in many numeric programs (e.g., SUIF [28, 79], Polaris [9]). These compilers generate shared-memory programs with parallel constructs such as DOALL loops and REDUCTION routines. However, experiences with shared-memory compilers suggest that they lose performance because they do not take into account features of the underlying architecture, particularly interprocessor communication [74, 77].

3 My Approach

The problem with high communication costs points out a key obstacle to obtaining performance from a machine-independent parallel programming model. Simply stated, parallel architectures possess many machine-dependent characteristics that must be exploited to achieve high performance. Examples of such features include communication overhead and latency, false and true sharing of cache lines and pages, and network and memory contention. Exploiting architecture and operating system support for efficient interprocessor communication is especially important, due to the high cost of interprocessor communication. Having users to provide this information is undesirable since it required extra work, detailed knowledge of computer architecture, and is specific to a particular parallel machine.

My approach is to *give an optimizing compiler the responsibility of achieving high performance by customizing programs to match the underlying parallel architecture*. Compilers for sequential programs have historically been quite successful in this role of optimizing programs for individual architectures, performing tasks such as instruction selection, register allocation, and instruction scheduling. I feel compilers can also successfully customize programs for parallel architectures by supporting different architectures, efficiently utilizing available communication and synchronization mechanisms, and making communication and computation tradeoffs to achieve high performance. However, to accomplish this goal compilers must perform additional analyses and optimizations not found in traditional sequential and parallelizing compilers. In particular, the compiler must do an excellent job of *communication analysis and optimization*, since interprocessor communication is so expensive. I call this advanced compiler COSMIC, for Communication-Optimizing, Shared-Memory Integrated Compiler, to emphasize the integration of communication analysis and optimization in a sophisticated shared-memory compiler.

3.1 Shared Address Space

COSMIC will generate code that utilizes the *shared address space* found in a shared-memory programming model because it provides the best combination of flexibility and performance. By choosing a shared-memory model, COSMIC can efficiently compile a much larger set of programs. Because a shared-memory compiler is not responsible for inserting communication or translating addresses between processors, it is easier to construct and much more flexible than distributed-memory compilers. Any program may be compiled, since sequential computation executes as before and parallel loops can be executed by assigning different iterations to each processor.

Shared-memory compilers are easier to use, since scientists are no longer required to write their entire programs in data-parallel languages such as HPF. Instead, they can focus on rewriting a few computation-intensive procedures, adding parallelism directives where necessary. These compilers also have the advantage that they produce programs that can run on the large-scale parallel machines as well as the low-end, but more pervasive workstations. This portability is important for scientists and engineers who want to develop applications that run well on their multiprocessor workstations, but who desire the ability to scale their applications up for larger parallel machines as needed. The combination of ease of use and scalability of software is a key appeal of shared-memory compilers.

3.2 Flexible-Shared-Memory Machines and Software DSM Systems

Two recent advances in computer systems have made shared-memory even more desirable as a programming model. First is the development of new Flexible-Shared-Memory (FSM) machines (e.g, Alewife [2], Flash [33, 49], Typhoon [64, 67]). These architectures 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.

Second is the development of software distributed-shared-memory (DSM) systems (e.g., Ivy [53], Treadmarks [19]) that support a shared address space on top of existing message-passing machines and networks of workstations. Software DSMs thus allow shared-memory compilers to produce code that executes on both shared and distributed-memory machines. The latest generation of software DSMs (e.g, Munin [7], Blizzard/Tempest [21], CVM [40]) also support multiple coherence protocols and explicit messages. Software DSM systems are both useful as portable programming systems in their own right, and as testbeds for developing and evaluating new coherence protocols that may be used on FSM machines. By exploiting the advanced features of FSM machines and software DSMs, COSMIC has the potential to match or exceed the performance of optimized message-passing programs on distributed-memory machines.

My proposed programming environment is pictured in Figure 1(b). The role of COSMIC, my shared-memory compiler, is to take parallel programs and customize them for different parallel architectures. Parallelism in the input programs may be either automatically detected or programmer-specified as parallel loops (PCF Fortran) or data-parallel operations (HPF). COSMIC is one of the first compilers to target software DSMs; it is the first compiler to take advantage of the multiple coherence protocols and message-passing in software DSMs and FSM machines. Because it will utilize HPF data decompositions to optimize the program, COSMIC can also be considered the first optimizing HPF compiler for shared-memory machines. COSMIC will thus be valuable both to scientists and as a research tool for shared-memory compilers.

3.3 Achieving High Performance

An optimizing shared-memory compiler is easy to use, flexible, and well-suited to shared-memory machines. The important question is whether shared-memory compilers can approach the performance of current distributed-memory compilers or explicit message-passing programs on distributed-memory machines. Based on preliminary experience [77, 60], I believe the answer is yes with advanced compiler analysis and optimizations, in particular communication analysis [76]. To achieve high performance, the compiler follows two basic guidelines:

- Adapt compilation techniques found in distributed-memory compilers, in order to retain the benefits of explicit messages in message-passing programs.
- Exploit architectural and operating system support found in FSM machines and software DSMs to improve on the flexibility and performance of message-passing programs.

To motivate my approach, I first examine the advantages and disadvantages of the message-passing programming model, then point out how COSMIC can retain most of the benefits of message-passing programs while capitalizing on the shared-memory model to avoid its disadvantages.

3.4 Message Passing Advantages

I previously pointed out some disadvantages of message-passing programs such as high user effort and software address translation overhead. Some additional problems are inflexibility due to static assignment of work to processors, overhead due to control flow in sequential code being replicated on each processor, and high software overhead due to the synchronization and buffering implicit in a message.

Despite these problems, message-passing programs can make very efficient use of the underlying hardware for distributed-memory machines. These benefits, listed below, are enjoyed both by hand-written message-passing programs and data-parallel languages (e.g., HPF) which have distributed-memory compilers that generate message-passing code. The advantages of message-passing programs motivate the need for communication analysis and optimization in shared-memory compilers.

Co-location of data and computation In message-passing programs, locality is clearly defined since data is explicitly distributed across processors. Heuristics such as the owner-computes rule can be used to assign computation to each processor in such as way that each processor accesses mostly local data [36]. By co-locating data and computation, interprocessor communication is significantly reduced.

Block data transfer Message-passing programs transfer data between processors through explicit *send* and *recv* operations. By aggregating data for many nonlocal references into a single large message, communication overhead and latency costs can be paid once and amortized over many nonlocal accesses. Messages are sent from the producer, without requiring a request from the consumer. Data in a message is packed into a contiguous buffer, so no unnecessary data is transferred. It may also be possible to overlap communication and computation during a message.

Collective communication Message-passing programs can take advantage of opportunities for collective communication such as broadcast, gather, all-to-all, or global reductions. Such communication patterns are extremely efficient because they combine synchronization, data transfer, and computation.

Synchronization Explicit synchronization in the form of locks or barriers is not needed for message-passing programs. Instead, synchronization is integrated with data transfer by requiring processors receiving a message to block until the message arrives. Processors thus only synchronize when communication occurs, and then only with the exact processors required.

Spatial locality Finally, message-passing programs enjoy high spatial locality (i.e., memory accesses are usually to contiguous locations). Because they explicitly distribute data across processors, each processor's portion of an array is laid out in contiguous memory locations. Block data transfer also tends to pack nonlocal data into contiguous buffers before being accessed. High spatial locality improves memory system behavior and avoids problems such as capacity and conflict cache misses, paging, false sharing, and wasted communication bandwidth.

4 Shared-Memory Compiler Design

In this section, I present the design of COSMIC. I describe how it adapts distributed-memory compilation techniques for shared-memory programs to retain the benefits of message passing programs. It also apply new optimizations to take advantage of extensible coherence protocols and message passing support found in software DSMs and flexible-shared-memory (FSM) machines. The full set of optimizations are listed in Table 1.

4.1 Communication Analysis

Many of the optimizations rely on *communication analysis*, a compilation technique 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 [36]. Communication analysis extends traditional data-flow and dependence analysis by incorporating information on how computation is partitioned among processors; it identifies when and what data needs to be transferred between processors during parallel execution. Communication analysis is vital for all advanced multiprocessors, because the cost of interprocessor communication greatly outweighs computation and local memory accesses.

4.2 Hybrid Programming Model

The programming model assumed by the compiler back end is important, since it forms the boundary between the compiler and run-time system. Parallelizing compilers for shared-memory machines typically employ a *fork-join* model, where the master processor assigns work to the worker processors when a parallel loop is encountered. The fork-join model is flexible but cedes control of work partition to the scheduler and implicitly assumes a *broadcast* before the loop body and a *barrier* after the parallel loop.

In comparison, message-passing programs use a *single-program, multiple-data* (SPMD) model, where all processors execute the entire program [36]. When a parallel loop is encountered, processors compute their portion of the computation based on their logical processor IDs. As as result, the compiler has control over computation decomposition and placement, but processors are forced to remain idle during sequential computation.

COSMIC attempts to blend the advantages of both approaches through a hybrid strategy. Sequential parts of the program are executed by a single processor. Parallel loops, however, are combined to form larger parallel regions that can be treated as small SPMD programs. As I show later, this hybrid programming model retains the flexibility of the fork-join model while providing greater control over program execution where necessary for other optimizations.

4.3 Co-location of Data and Computation

The goal of exploiting parallelism while preserving locality of reference is key to obtaining high performance. Computation must be assigned to processors in such a way that each processor has good locality of reference. On distributed-memory machines, the problem of locality is clearly defined since data is explicitly distributed across processors. One approach is to leave the primary responsibility to the user, letting the user specify how data should be decomposed using a language such as High Performance Fortran (HPF) [34]. The compiler then infers the computation mapping by using the owner-computes rule [36]. Alternatively, global analysis may be applied across different loop nests to determine the data and computation partition automatically [6, 8, 26].

For shared-memory machines, the problem of locality is not as clear since it is not necessary to explicitly allocate arrays to processors. However, the algorithm used to minimize interprocessor communication for distributed-memory machines can be directly applied to shared-memory machines. By using the SPMD programming model within parallel regions, COSMIC can control the computation partition and placement to co-locate data and computation. It is important

Shared-Memory Compiler Optimization	Shared-Memory Machines	Software DSMs and FSM Machines
Communication Analysis	X	X
Hybrid programming model	X	X
Co-location of data and computation	X	X
Memory layout optimization	X	X
Packing nonlocal data	X	X
Prefetch nonlocal data	X	X
Eliminating synchronization	X	X
Affinity scheduling	X	X
Customized run-time system		X
Optimized reductions		X
Update and broadcast		X
Enhanced release consistency		X

Table 1 Applicability of Shared-Memory Compiler Optimizations

to support not just the simple data decompositions found in HPF, but also irregular data decompositions needed for adaptive sparse computations [18]. Issues with integrating run-time data decomposition libraries that generate irregular data decompositions based on input data will be handled in COSMIC. One possible implementation approach is for the compiler to introduce an additional level of indirection through a map array [60], but the technique still needs to be evaluated.

Because the cost of nonlocal accesses is high, COSMIC may need to sacrifice some amount of parallelism in order to maintain data locality. These situations arise for *pipelined computations*, computations containing wavefronts across the data space [36]. For computations such as ADI integration and SOR, exploiting maximum parallelism through loop interchange or skewing to produce DOALL loops will degrade data locality. Instead, COSMIC can sacrifice some parallelism to exploit greater data locality.

4.4 Memory Layout Optimization

Message-passing programs have good spatial locality, since data assigned to each processor is placed in contiguous memory locations. The same may not be true for shared-memory programs, since the data assigned to each program may be scattered through the address space depending on how it has been partitioned.

Poor spatial locality for local data may cause problems because modern memory systems are composed of multi-word *coherence units* such as cache lines and pages. When only a fraction of the data in each unit are used, more units are needed, causing *capacity misses* in the cache and *paging* in the memory. Because of limited set associativity, the manner data has been partitioned may also cause local data to be mapped to the same cache locations, causing *conflict misses*. Finally, poor spatial locality increases the chance of multiple processors simultaneously writing to the same coherence unit. This problem is known as *false sharing* and may significantly degrade performance [12, 20, 73].

To improve spatial locality of local data and avoid these undesirable memory system effects, COSMIC cannot simply obey the data layout convention used by the input language (e.g. column-major in FORTRAN and row-major in C). Instead, it must customize the data layout for each program based on the data decomposition. To eliminate false sharing between a pair of shared scalar variables, COSMIC may place them on separate coherence units. To eliminate false sharing between data arrays, COSMIC may align each array so that they begin on separate coherence unit boundaries.

Eliminating false sharing and conflict misses for portions of a single array is a more complex task that may require extensive modification to the program. In the simplest case, array dimensions may be padded to eliminate false sharing [11, 72]. Another alternative is to transpose array dimensions. This optimization is simple to apply, but only works for certain cases of one-dimensional data distributions.

A more powerful solution can be derived by looking at distributed-memory compilers. These compilers explicitly manage the address space of each processor based on the data decomposition specifications. In effect they lay out the local part of each array contiguously in the local address space. For regular data decompositions, COSMIC can apply distributed-memory compiler techniques for rearranging array layout to make local sections of each array contiguous. However, scalar optimizations are required to clean up modulo and division operations inserted into array subscripts [5].

For irregular data decompositions, COSMIC can add an additional level of indirection to the array by adding an index array to the subscript references.

4.5 Packing Nonlocal Data

Memory layout optimizations improve spatial locality for local data, but nonlocal data accesses may still have poor spatial locality. Accesses to scattered nonlocal data are problematic since data is moved in coherence units (e.g., cache lines or pages). First, communication bandwidth is wasted since only a fraction of the data transferred is accessed. Second, communication latency is increased due to more cache line or page misses. If data is moved in cache lines, capacity and/or conflict cache misses may occur since more cache lines are needed to hold the same amount of nonlocal data. If data is moved in pages, extra swapping of pages to disk may occur if the number of pages exceed available memory.

Message-passing programs avoid these problems by combining nonlocal accesses in a single message to reduce communication costs. Shared-memory compilers can obtain also benefit by copying nonlocal data with poor spatial locality into contiguous buffers. COSMIC must first apply communication analysis to detect nonlocal accesses. If the nonlocal data is not contiguous, then COSMIC must insert code to copy the data to contiguous buffers (one for each processor). The placement copy code is determined by data dependences using message vectorization [4, 36, 65, 83]. Finally, COSMIC must modify the code so data so nonlocal accesses are made to the buffers.

By making nonlocal data contiguous, COSMIC increases hardware prefetching, since entire coherence units of nonlocal data are communicated at once. Interprocessor latency penalties are reduced by the size of the coherence unit and less communication bandwidth is wasted. Since packing data introduces additional overhead and memory usage, its profitability is dependent on the ratio between copy overhead and cost of nonlocal accesses for the underlying architecture.

4.6 Prefetching Nonlocal Data

Message-passing programs can hide communication latency by separating matching *send* and *recv* operations to overlap communication with computation. Shared-memory machines can achieve the same effect through software prefetching, nonblocking hints to the system to prefetch data before it is actually required in order to hide access latency [13, 58, 59]. Communication analysis is needed to determine what data accesses are nonlocal, so they may be prefetched early enough in advance to overlap communication and computation latency. Hardware support is needed for software prefetching of cache lines. However, run-time system support in software DSMs is sufficient to prefetch pages.

4.7 Eliminating Synchronization

The fork-join programming model employed by shared-memory compilers typically require barrier synchronization following each parallel loop. These barriers 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.

Message-passing programs, by comparison, do not require barriers. Instead, they rely on the synchronization implicit in messages. Within SPMD regions, shared-memory compilers can use the results of communication analysis to determine when there is no interprocessor communication between loop nests, allowing barriers to be eliminated. When analysis reveals that communication does exist, barriers may be replaced by more efficient forms of pair-wise producer-consumer synchronization to directly synchronize the communicating processors [76].

4.8 Affinity Scheduling

One of the advantages of the fork-join programming model is its ability to easily dynamically balance work across processors. In comparison, distributed-memory compilers typically emphasize locality ahead of load balancing, partitioning all work at compile time. COSMIC will attempt to find the right tradeoff between load balance and locality. Doing so requires the compiler to partition work into smaller chunks and calculate information on locality preferences for the run-time affinity scheduler. The process of taking locality into account during scheduling is known as *affinity* *scheduling* [55]. To reduce memory effects, COSMIC can also consider the underlying granularity of coherence during scheduling [25]. For instance, COSMIC may choose a tile size that spans an integral number of cache lines but provides slightly worse load balance. Tiles may also be aligned with the proper offsets so that they span as few coherence units as possible.

4.9 Customized Run-time System

The optimization techniques described thus far require only a generic coherent shared memory, and are thus applicable to all shared-memory machines. The remaining optimizations in this section are applicable only on Flexible-Shared-Memory (FSM) machines and software Distributed-Shared-Memory (DSM) systems, since they require greater cooperation between COSMIC and underlying hardware and operating system.

I begin with optimizations to customize the compiler run-time system. Shared-memory compilers depend on a small run-time system to provide functions for synchronization, work dispatch, and reductions. These operations may be implemented as operations on shared memory locations, but are usually far more efficient if directly supported by the underlying hardware or operating system [48]. Calls to software locks and barriers are the simplest; they are replaced with the equivalent hardware operations or system routines based on explicit system-level messages.

Optimizing the work dispatch routine is more complicated. At the beginning of each parallel region, the master processor must dispatch work to the remaining processors by sending information about the work to be performed (usually a function pointer) and any parameters required. A simple approach is to assign these to global variables and require the workers to fetch the values. This approach imposes latency and contention. It is thus desirable to utilize broadcast capabilities in the underlying system, having the master broadcast the information to workers in a single message. If broadcast is not supported, the master can directly send data to the workers by maintaining coherence for work dispatch variables using an update protocol.

4.10 Optimized Reductions

Reductions are associative operations (e.g., SUM, MAX) on data that may be reordered and executed in parallel. Naive shared-memory parallelization of reductions guard individual reduction operations with locks to ensure mutual exclusion. COSMIC first performs the local portion of each reduction on privatized data, then combines the results through operations on global memory guarded by locks to ensure mutual exclusion. Privatizing reductions greatly reduces synchronization cost since mutual exclusion is needed only for the final accumulation step. The accumulation step can be made even more efficient by customizing the run-time system to use utilize reduction routines built into the underlying memory system. If explicit messages are supported, they can be used to integrate data transfer, synchronization, and computation, eliminating the need for locks.

Reductions over sparse data structures can be also be handled through privatization, but may be inefficient since the entire sparse data structure is replicated on each processor. A second solution is to explicitly buffer the reduction values for nonlocal references, then update the other processors using messages. COSMIC can use Chaos runtime routines [18] to efficiently combine results for sparse reductions.

An alternative to privatization or explicit buffering is to exploit a reduction protocol in the underlying memory system. Such a protocol will allow processors to perform the reduction direction on the global memory addresses. The underlying memory system will record these values on the fly for each processor, then merge results with other processors at the end of the parallel region. Because an explicit reduction protocol supports multiple writers, no synchronization is needed until results have to be merged. The tradeoffs between the three choices for handling reductions on sparse data structures is heavily architecture dependent, and will be decided in COSMIC using the results of communication analysis and cost models.

4.11 Update and Broadcast

Most current shared-memory systems employ an invalidation-based coherence protocol, where processors invalidate cache lines (or pages for software DSMs) owned by other processors whenever they assign a value in the cache line. An invalidation protocol works well in most situations; however, it requires extra messages and latency between processors when data is repeatedly generated and communicated, since the consumer processor must request data from the producer processor.

An invalidation protocol performs particularly poorly when a processor needs to send data to multiple processors, such as when the master processor needs to dispatch work to the other processors. Under the invalidation model, the master first sends out invalidate messages to all the workers when new work is available and put into the work queue.

The workers attempt to access the work queue, realize data is no longer valid, and send data requests back to the master processor. The master processor then receives the requests and sends out the values of the work queue. Not only is the communication latency and number of messages tripled, contention at the master processor caused by incoming messages can cause a significant serial bottleneck. COSMIC can avoid problems with work dispatch in the run-time system by replacing it with explicit messages, but such patterns also occur in the user application (e.g., processors broadcasting pivots in Gaussian elimination).

Message-passing programs avoid this problem by having producer processors send data directly to the consumers using messages. When one processor needs broadcast data, special collective communication routines perform the broadcast very efficiently. To achieve the same effects, COSMIC can use communication analysis to detect locations in the program where processors will generate data needed by many other processors. The new values can then be sent to the other processors through explicit broadcast messages, or by maintaining coherence for the data using a update coherence protocol. By doing so, contention and the additional round trip for messages can be avoided.

4.12 Enhanced Release Consistency

Hardware shared-memory systems typically enforce a conservative form of release consistency, buffering a small number of invalidate and update messages to other processors until synchronization points are reached [15, 51]. Software DSMs, on the other hand, can enforce lazy release consistency, buffering all invalidate and update messages until a synchronization point is reached [42]. For compiler-generated parallel programs, COSMIC can use a tailored version of release consistency to achieve better performance.

The key observation is that compiler-parallelized programs do not have arbitrary data races between processors, since such data races would have prevented legal parallelization. As a result, any data defined by a processor within a parallel region does not need to be made visible to other processors until the end of the parallel region, unless the data is known to be needed by the compiler run-time system. The only data that fits in this category are lock variables, data passed to reduction routines, and data used in pipelined computations.

Relying on this observation, COSMIC can enforce an enhanced version of lazy release consistency for compiler-parallelized programs. For most data, all coherence messages can be buffered until the barrier at the end of the parallel region. Locks encountered with the parallel region thus do not force memory between processors to be made consistent, except for data used in reductions and pipelined computations.

In addition, COSMIC can use communication analysis to determine when data will be needed by other processors. By combining this information with standard data-flow and dependence analyses, COSMIC can calculate data that can be eagerly transferred between processors when the network is not busy, providing opportunities to overlap communication and local computation. With compiler analysis, COSMIC can thus customize the release consistency protocol to include selective eager updates for certain data while maintaining enhanced lazy consistency for all other data.

5 Related Work

My work builds upon shared-memory compiler algorithms for identifying parallelism [9, 29, 78] and performing program transformations [80, 81], as well as distributed-memory compilation techniques to select data decompositions [6] and explicitly manage address translation and data movement [4, 36].

Many previous researchers have examined performance issues on shared-memory architectures. Singh *et al.* [68] applied by hand optimizations similar to those I considered to representative computations in order to gain good performance. Others evaluated the benefits of co-locating data and computation [22, 56], as well as false sharing [12, 20, 25, 52, 73]. These approaches focused on individual optimizations and were generally applied by hand. In contrast, I have shown how they may be implemented in a compiler by adapting well-known techniques from distributed-memory compilers for shared-memory machines.

Cytron *et al.* were the first to consider mixing fork-join and SPMD models [17]. Their goal was to eliminate thread startup overhead and increase opportunities for privatization; they carefully considered safety conditions. In comparison, COSMIC uses local SPMD regions to enable compile-time computation partition and synchronization elimination through communication analysis. Researchers using the Alewife multiprocessor [2] compared the benefits of message-passing and shared memory [48]; they found message passing to be advantageous mainly for improving synchronization primitives by coupling synchronization and data movement.

Researchers have looked at the problem of exploiting parallelism through the use of data and event synchronization, where *post* and *wait* statements are used to synchronize between data items [70] or loop iterations [54]. 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 [32, 62, 63]. Simple data dependence analysis can be used to reduce barrier synchronization by orders of magnitude, greatly improving performance. In comparison, we begin with parallel loops discovered by parallelizing compilers, where the remaining synchronization is significantly harder to eliminate. Bodin *et al.* [10, 61] apply barrier elimination algorithms most similar to ours, but rely on heuristics to eliminate cross-processor dependences. In comparison, we perform full communication analysis.

Larus has compared implementing global address spaces in software using a distributed-memory compiler compared to hardware-based implementations [50]. He speculates that distributed-memory compilers are desirable because they can more closely exploit underlying architectural features in certain key cases; however, shared-memory hardware is desirable in the cases where the compiler fails.

Mukherjee *et al.* compared the performance of explicit message-passing programs with shared-memory programs [60] on Typhoon, a Flexible-Shared-Memory machine implemented on top of a CM-5 [64, 67]. Results show that with suitable extensions to the coherence protocol, the shared-memory program was able to match the performance of the optimized message-passing program utilizing Chaos [18]. The authors point out that a compiler like COSMIC can take advantage of the extensible coherence protocol to achieve performance.

Mirchandaney *et al.* described the design of a compiler for Treadmarks, a software DSM [57]. They propose *section locks* and *broadcast barriers* to guide eager updates of data, integrating *send*, *recv* and *broadcast* operations with the software DSM, and reductions based on multiple-writer protocols. Their proposal is similar to portions of my proposal, but I have improved upon their approach based on my experiences with an existing shared-memory compiler implementation [3].

6 Plan of Work

Preliminary work with shared-memory compilers has concentrated on moderate-sized multiprocessors. I propose to extend these compilation techniques to target the entire spectrum of shared-memory systems, ranging from multiprocessor workstations to Flexible-Shared-Memory (FSM) machines and software Distributed-Shared-Memory (DSM) systems. To evaluate the impact on the performance on scientific applications, I will implement my compilation techniques in COSMIC, a prototype compiler, and evaluate its effectiveness. COSMIC will be based on the Stanford SUIF compiler [79], and will take as input Fortran or C.

I plan to target programs from benchmark suites such as Perfect, NAS, SPEC, and SADIE; these are mostly dense-matrix numeric computations. I also plan to test representative applications from fields such as computational chemistry, computational fluid dynamics, sparse matrix codes, and computational combustion, including the applications listed in Table 2. These applications operate on sparse data structures, reflecting the trend towards sparse codes in advanced scientific computations. I plan to measure the impact of my optimizations on application performance on the IBM SP-2, DEC Alpha multiprocessor, and clusters of DEC workstations. My goal is to significantly improve on current compilers for shared-memory architectures and approach or exceed the performance of message-passing programs on distributed-memory architectures.

My research plan is as follows:

1. Evaluate and improve compiler optimizations to reduce synchronization, communication, and memory system costs in shared-memory multiprocessors.

2. Build COSMIC by porting the existing SUIF shared-memory compiler. Tune the run-time system of the compiler by replacing key components with calls to efficient DSM routines that bypass the shared memory.

3. Explore architectural features, runtime support, or enhanced memory coherence protocols desirable for an flexible shared-memory architecture, using a software DSM as a target.

4. Improve overall application behavior by extending compiler analyses and adding DSM features to support more efficient interprocessor communication between processors for user data.

5. Experimentally evaluate COSMIC for benchmark programs from Perfect, NAS, SPEC, and SADIE, as well as representative scientific applications. Test compiler output on the IBM SP-2 as well as clusters of DEC workstations.

6. Evaluate properties of scientific applications that are either conducive or obstacles to successful compiler optimization for shared-memory machines.

Application	Field	Description
CHARMM	computational chemistry	model and simulate macromolecular systems
MOLDYN	computational chemistry	moecular dynamics N-body force calculation
GROMOS	computational chemistry	non-bonded force between molecules with cutoff
DSMC	computational fluid dynamics	direct-simulation Monte Carlo method
TLNS3D	computational fluid dynamics	multi-block Navier-Stokes' solver over a 3D surface
EULER	computational fluid dynamics	finite-difference approximations on a Eulerian mesh
CHOLSKY	sparse-matrix	implements sparse cholesky factorization
DAG	sparse-matrix	implements sparse triangular solves
NRL FLAME	computational combustion	structured finite volume methods

Table 2 Representative Scientific Applications

This project is being conducted in collaboration with research projects at Stanford University and the University of Maryland. COSMIC is based on the SUIF compiler developed by Prof. Monica Lam's group at Stanford [71, 79]. The SUIF compiler currently detects parallel loops and reductions, applies interprocedural parallelization and privatization [28], calculates data decompositions [6], and applies simple synchronization [76] and memory layout [5] optimizations. Plans are in progress to adapt the DEC HPF compiler front end. COSMIC will provide an optimizing back end for the SUIF compiler.

COSMIC will target Prof. Pete Keleher's Coherent Virtual Machine (CVM) software DSM system. CVM is designed to be extensible and widely portable, making it an ideal target for COSMIC [40]. Coherence protocol enhancements are easily added to the CVM system using the Sparks protocol construction library [41]. I already have a preliminary port of the SUIF compiler to target the CVM system.

I am also working with Prof. Joel Saltz's Chaos project to develop efficient run-time support for scientific applications [18]. My collaboration with Chaos project is to develop compiler support for parallelizing generalized reductions in C programs. The flexibility of COSMIC will serve as valuable infrastructure for Chaos. COSMIC will also take advantage of the extensive runtime support provided by the Chaos runtime system whenever appropriate.

7 Impact of Proposed Research

Though some of the compilation techniques described have been previously considered for shared-memory machines, this proposal is first to unify all the techniques in one compiler, as well as the first to combine compiler optimizations with the multiple coherence protocols and message passing support found in software distributed-shared-memory (DSM) systems (e.g., Blizzard, CVM) and flexible-shared-memory (FSM) machines (e.g., Flash, Typhoon). This project will make several important contributions:

- a sophisticated optimizing compiler for shared-memory machines,
- a prototype compiler for software DSM systems,
- a prototype compiler for FSM machines,
- extensive experimental evaluation of advanced compiler optimizations,
- memory-coherence protocol extensions useful for compiler-parallelized codes,
- insights towards efficient parallelization of advanced scientific applications.

The compilation techniques developed by this project will be of great interest to computer vendors. My findings on useful memory-model extensions will also help computer architects and operating system writers. Most importantly, by reducing the penalty of writing portable parallel programs, my techniques will make parallel computing more attractive for scientists and engineers.

8 Prior Research Accomplishments

I have pursued research in a number of areas in high-performance computing. My research emphasizes techniques that yield practical benefit. I demonstrate their effectiveness through prototype implementations and experiments on standard benchmark programs.

The Fortran D Distributed-Memory Compiler Automatically producing efficient message-passing programs for distributed-memory machines is a difficult problem. Previous research showed compilers can automatically generate *correct* message-passing code, but often with very high overhead compared to hand-written programs. To solve this problem I designed and implemented a compiler for *Fortran D*, a version of Fortran enhanced with data decomposition specifications [23, 36, 75]. I developed many advanced analyses in the prototype compiler, enabling sophisticated optimizations to reduce the number of messages, overlap communication with computation, parallelize reductions, and exploit pipeline parallelism [30, 38]. Experiments on common applications show it produces code that approaches the performance of hand-optimized programs and is several times faster than code output by commercial compilers [37]. The Fortran D compiler thus demonstrated compilers can automatically generate *efficient* code for message-passing machines; its success contributed to the development of High Performance Fortran (HPF) [45].

The SUIF Shared-Memory Compiler Many research compilers are only capable of compiling toy programs. With Prof. Lam's research group I built the SUIF shared-memory compiler into a robust tool capable of parallelizing programs from the major scientific benchmark suites [28, 79]. I designed and implemented its run-time system to be easily portable across a wide range of shared-memory machines. Obtaining scalable performance on shared-memory machines can be difficult due to memory access and synchronization overhead [3]. I implemented a hybrid programming model to exploit locality and apply communication analysis to reduce synchronization overhead [76]. Experimental results show barrier synchronization is reduced 29% on average over benchmark suites and by several orders of magnitude for certain programs. Combined with co-location and data layout optimizations, the SUIF compiler was able to obtain excellent speedups that were otherwise not achievable for two example programs [77]. Applying the SUIF parallelizer to the SPECfp benchmarks yields the best current SPECfp rating on a four processor DEC Alpha multiprocessor.

Memory Simulation Study With researchers at Stanford I performed the first extensive simulation study evaluating memory system behavior for applications amenable to compiler parallelization [74]. Results for compiler-parallelized programs from the SPEC, NAS, PERFECT and RICEPS benchmark suites show that compiler-parallelized codes achieve reasonable speedups on advanced memory systems, long cache lines can cause excessive false and true sharing, and coarse-grain parallelism correlates with good memory system behavior.

ParaScope Editor Because compilers are not infallible, it is important to develop programming environments capable of providing interactive user feedback. With researchers at Rice I built the ParaScope Editor, and interactive parallel programming tool [43]. It assists the knowledgeable user by displaying the results of sophisticated program analyses and by providing editing and a set of powerful interactive transformations. After an edit or parallelism-enhancing transformation, the ParaScope Editor incrementally updates both the analyses and source quickly [44].

D Editor Using the ParaScope Editor as a base, I built the D Editor to provide feedback for data-parallel languages such as HPF and Fortran D. The D Editor provides novel information on partitioning, parallelism, and communication based on compile-time analysis at the level of the original Fortran program [39]. It uses color coding and a collection of graphical displays to help the user to zoom in on portions of the program containing sequentialized code or expensive communication. The D Editor forms the centerpiece of the D System for data-parallel languages [1, 35].

ParaScope Dependence Analyzer Precise and efficient tests for data dependence are essential to the effectiveness of a parallelizing compiler. With researchers at Rice and OGI, I developed and implemented the Delta test [24], a dependence testing scheme based on classifying pairs of subscripted variable references, and the Power Test, a test based on Fourier-Motzkin elimination [82]. Empirical studies demonstrate that these tests are exact for the vast majority of array reference pairs in scientific Fortran codes.

Data Locality Optimizer One of the key performance issues for uniprocessors is data locality. With researchers at Rice, I developed compiler optimizations to improve data locality based on a simple yet accurate cost model [14]. The model computes both temporal and spatial reuse of cache lines to find desirable loop organizations. Experiments with kernels illustrate that our model and algorithm can select and achieve the best performance. We evaluated over thirty complete applications through simulations and timings. Performance improvements were difficult to achieve because benchmark programs typically have high hit rates even for small data caches; however, our optimizations significantly improved several programs.

References

[1] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren, and C. Tseng. Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58, Fall 1994.

[2] 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.

[3] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] B. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Montreal, Canada, August 1994.

[9] 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.

[10] F. Bodin, L. Kervella, and M. O'Boyle. Synchronization minimization in a SPMD execution model. Technical Report TR-94-863, INRIA, September 1994.

[11] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Litchfield Park, AZ, December 1989.

[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] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA, April 1991.

[14] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[15] R. Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance evaluation of hybrid hardware and software distributed shared memory protocols. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, Manchester, England, July 1994.

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

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

[18] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[24] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[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. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[27] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[28] 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.

[29] 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.

[30] 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.

[31] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

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

[33] 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.

[34] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[35] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[36] 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.

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

[38] 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.

[39] S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. Warren. The D Editor: A new interactive parallel programming tool. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

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

[41] P. Keleher. Sparks: Coherence as an abstract object. Technical Report CS-TR-3544, Dept. of Computer Science, University of Maryland at College Park, October 1995.

[42] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[43] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[44] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice and Experience*, 5(7):575–602, October 1993.

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

[46] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[47] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[48] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B. Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[49] 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.

[50] J. Larus. Compiling for shared-memory and message-passing computers. *ACM Letters on Programming Languages and Systems*, 2(1–4):165–180, March–December 1993.

[51] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[52] 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.

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

[54] Z. Li. Compiler algorithms for event variable synchronization. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[55] E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[56] 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.

[57] R. Mirchandaney, S. Hiranandani, and A. Sethi. Improving the performance of DSM systems via compiler involvement. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

[58] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[59] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, MA, October 1992.

[60] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[66] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

[67] 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.

[68] 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.

[69] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[70] P. Tang, P. Yew, and C. Zhu. Compiler techniques for data synchronization in nested parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[71] S. Tjiang, M. E. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating scalar optimization and parallelization. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[72] J. Torrellas, M. Lam, and J. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[73] 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.

[74] 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.

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

[76] 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.

[77] 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.

[78] 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.

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

[80] 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.

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

[82] M. J. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

[83] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.