Compiler-directed Shared-Memory Communication for Iterative Parallel Applications*†

Guhan Viswanathan James R. Larus
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
Telephone: (608) 262-9519
{gviswana,larus}@cs.wisc.edu

August 6, 1996

Abstract

Many scientific applications are iterative and specify repetitive communication patterns. This paper shows how a parallel-language compiler and a predictive cache-coherence protocol in a distributed shared memory system together can implement shared-memory communication efficiently for applications with unpredictable but repetitive communication patterns. The compiler uses static analysis to identify program points where potentially repetitive communication occurs. At runtime, the protocol builds a communication schedule in one iteration and uses the schedule to pre-send data in subsequent iterations. This paper contains measurements of three iterative applications (including adaptive programs with unstructured data accesses) that show that a predictive protocol increases the number of shared-data requests satisfied locally, thus reducing the remote data access latency and total execution time.

1 Introduction

Many scientific applications are iterative with each iteration simulating the evolution of a physical system along one dimension of the problem domain (typically time). Each iteration of a problem is usually divided into multiple phases of parallel execution separated by synchronization. Communication within a parallel phase may include both structured communication (e.g., nearest-neighbor

*This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF NYI Award CCR-9357779, NSF Grant MIP-9225097, DOE Grant DE-FG02-93ER25176, and donations from Digital Equipment Corporation, Sun Microsystems, and The Portland Group. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School.

communication in Jacobi iteration) and unstructured communication (e.g., using indirection arrays or pointer dereferences). Many of these applications have communication patterns that show little or no change **between** iterations. As a result, even for irregular programs, for which static analysis is imprecise, a run-time system can detect communication patterns in one iteration and use them to predict communication in the subsequent iterations.

There are many examples of programs with repeated patterns of communication. In static mesh calculations, nearest-neighbor communication is repeated in each iteration. In some irregular problems, such as molecular dynamics codes [16], communication changes infrequently, perhaps once every 20-30 iterations. In adaptive problems, communication changes frequently, but incremental changes between iterations are small. For example, structured adaptive meshes gradually add mesh nodes for greater accuracy in each iteration [9], and gravitational N-body problems represent bodies in a quad-tree, which undergoes small structural changes between iterations.

This paper shows that a compiler for a data-parallel language can cooperate with a predictive cache-coherence protocol in a distributed shared-memory (DSM) system to implement shared-memory communication efficiently for applications with dynamic, but repetitive communication patterns. The compiler uses static analysis to place protocol directives at points in a program at which potentially repetitive communication takes place. A two-part predictive protocol in the runtime system extends the standard memory coherence protocol. The first part of the protocol identifies communication patterns at runtime and builds a communication schedule. The second part uses a schedule to pre-send data that anticipates data requests in subsequent iterations. As a result, the protocol can reduce the number of remote data requests, total remote memory access latency, and program execution time. The predictive protocol optimizes communication for repetitive producer-consumer or migratory patterns; it does not target other sharing patterns (e.g., reductions, for which high-level language support is available in data-parallel languages).

This paper describes this combination of two techniques — a predictive cache coherence protocol, and simple compiler analysis — for optimizing shared-memory communication. The predictive protocol relies on and exploits customizable cache-coherence protocols in a cache-coherent DSM system. User-level control over protocols is available in a number of systems, including Blizzard [15] and FLASH [10]. The predictive protocol builds dynamic incremental communication schedules — new requests not satisfied by the pre-send phase are added to the schedule for subsequent iterations. This approach has the advantage that it can be applied to adaptive applications with repetitive dynamic communication patterns that a compiler cannot analyze.

The second technique, simple compiler analysis, automatically applies a predictive protocol for applications with repetitive producer-consumer sharing patterns for which a sequentially-consistent memory coherence protocol would incur large overheads [2]. By contrast, compilers targeting message-passing machines must identify and fully analyze run-time communication patterns in applications. Our simple analysis only identifies program points at which potentially repetitive communication takes place, but need not identify the patterns themselves.

We implemented this new approach in a compiler for the data-parallel language C** [11], which uses Blizzard [15], to implement a shared address space. Blizzard implements the Tempest parallel programming interface [14] on the CM-5, providing fine-grain distributed shared memory, and allows user-level shared-memory coherence protocols to customize communications to fit an application's needs. The C** compiler inserts run-time directives to invoke a single predictive protocol in the run-time system to optimize repetitive communication for data-parallel applications.

We analyze the performance of cooperative communication optimization by comparing the performance of optimized and non-optimized versions of three applications on a 32-processor CM-5. They include Adaptive, an adaptive structured mesh relaxation, Barnes, a gravitational N-body code and Water, a molecular dynamics code. In all cases, the predictive protocol reduced total remote access latency. In two cases, the optimized version is faster than the best non-optimized version (1.5x for Adaptive and 1.07x for Water). For Barnes, which shows excellent spatial locality, the optimized and non-optimized versions are comparable.

This paper proceeds as follows. Section 2 compares the approaches in this paper to previous work. Section 3 describes C**'s predictive protocol for communication optimization, building on an outline of Blizzard's default Stache coherence protocol. Section 4 describes compiler analysis for C** programs to identify repetitive communication patterns and place runtime system directives. Section 5 shows how these optimizations can be applied to improve the performance of three different applications. Section 6 concludes the paper.

2 Related Work

Related work for repetitive communication support falls into four broad categories: libraries, compilers, memory coherence protocols, and hardware.

A number of run-time libraries provide communication support for specific classes of applications, and require explicit programmer actions to structure the application using abstractions provided by the library. LPARX [8] and its adaptive-mesh extension [9] provide a software infrastructure to support structured static and adaptive mesh methods on message-passing machines. By contrast, our approach implements automatic communication optimization for programs written in a data-parallel language running on customizable cache-coherent DSMs (which also run on message-passing machines).

The most closely related work is the compiler-based Inspector-Executor approach that targets irregular communication patterns using the CHAOS [4] communication library. For each parallel loop that specifies irregular communication (e.g., using indirection arrays), the compiler generates an inspector and an executor. The inspector identifies non-local accesses at runtime and builds a communication schedule, which the executor uses to transfer data before executing the loop. A number of optimizations attempt to reduce the cost of the inspector phase, which is typically expensive, and must be executed whenever the indirection array changes. Ponnusamy et al. [12] note that if indirection arrays do not change between iterations, the communication schedule need not be rebuilt. Agrawal et al. [1] describe two optimizations that apply to distinct parallel loops whose schedules overlap: coalescing, which merges the two schedules, or incremental schedules, which subtracts the common part from the second schedule. Our work differs from the Inspector-Executor approach in three significant ways. First, CHAOS targets message-passing multiprocessors while C** targets customizable cache-coherent DSMs, which again can run on message-passing machines. Second, our approach requires no separate inspector and executor code, because the default protocol handles the problem of obtaining a copy of remote data which is absent when the loop is executed. Third, our approach includes incremental communication schedules, which are necessary for adaptive applications. Although the CHAOS group has looked at means to build incremental schedules, we are unaware of published descriptions of their approach or results.

Many DSM systems provide mechanisms to control a memory system to provide better support for parallel applications. Falsafi et al. [5] show that application-specific protocols can significantly improve application performance, especially for repetitive producer-consumer sharing patterns for which write-invalidate policies are inefficient. Their implementation included hand-written custom protocols for each application. By contrast, C** uses a single protocol that is automatically invoked by compiler directives.

Ramachandran et al. [13] propose additional hardware coherence primitives (e.g, update and prefetch) to help the programmer optimize common sharing patterns. Their SEL_WRITE primitive provides functionality very similar to our predictive protocol. Our approach adds compiler analysis for automatic predictive protocol usage.

3 A Predictive Protocol for Repetitive Communication Schedules

The C** system relies on Blizzard, a cache-coherent DSM sytem, to implement a global address space for data-parallel programs. Blizzard's default Stache coherence protocol provides sequentially consistent, transparent shared memory using a write-invalidate protocol [14]. Shared memory provides a high level of abstraction, which makes compiler development easier, but the write-invalidate policy incurs large overheads for producer-consumer sharing patterns (which occur repeatedly in many iterative applications).

The C** predictive protocol optimizes shared-memory communication for repetitive producer-consumer and migratory sharing patterns in data-parallel programs. It augments Stache to build communication schedules in one iteration and to pre-send data using a schedule in subsequent iterations. If the application's communication pattern is repetitive, the predictive protocol reduces the number of high-latency, non-local shared data accesses. The predictive protocol builds incremental communication schedules — new requests not anticipated previously are identified through access faults and are added to the schedule for subsequent iterations. The predictive protocol was developed using Teapot, a domain-specific language that reduces the complexity of specifying and developing cache-coherence protocols [3].

This section describes two parts of C**'s predictive protocol, the first part that build a communication schedule, and the second part that pre-sends data. Before describing the predictive protocol, we outline Stache's mechanisms and policies, and briefly describe why a write-invalidate protocol is inefficient for producer-consumer sharing patterns.

3.1 The Stache Shared-Memory Protocol

Stache implements sequentially-consistent shared memory using a directory-based write-invalidate protocol[14]. Stache is built on Tempest, which is a parallel programming substrate that supports fine-grain access control, i.e., at the cache block granularity (32–128 bytes). Each cache block may be in one of three states: **Invalid**, **ReadOnly**, or **ReadWrite**. Inappropriate accesses to a block (e.g., a read access to an **Invalid** block) generate faults that are vectored to a user-level handler in the Stache protocol.

Each shared-memory cache block in the system is mapped to its **home** node, where it resides initially. The home node also maintains a block's directory information, which lists multiple readers or a single writer, and is used to maintain consistency.

A read access to an invalid block invokes a user-level Stache fault handler, which sends a message to the home node requesting a copy of the block. The home node updates its directory information and sends a read-only block back to the requesting processor. On a write access to an invalid or read-only block, the home processor invalidates all outstanding read-only copies (to maintain sequential consistency) and sends a writable block to the requestor.

3.2 Inefficiencies in a Write-invalidate Protocol

It is widely known that write-invalidate protocols are inefficient for iterative producer-consumer communication patterns (see, for example, [2]). Each data transfer between producer and consumer involves four messages if a data item's home location is different from the producer and consumer:

1. The consumer requests a readable copy from the home node
2. The home node invalidates the producer's copy
3. The producer returns its copy to the home node
4. The home node sends the consumer a readable copy

The producer follows a similar protocol to acquire a writable copy when it generates new values.

When producer-consumer sharing patterns can be identified in an application, a write-update protocol can transfer a data item with one or two messages [5]. However, update protocols do not ensure sequential consistency and cannot be used in general.

3.3 Building Communication Schedules in the Predictive Protocol

C**'s predictive protocol augments Stache to collect communication information within a parallel phase. The protocol identifies, for each cache block requiring communication due to faulting accesses, whether the block was read or written (and the processors that read or wrote the block). The protocol relies on the compiler to demarcate parallel phases in the program (Section 4).

Since all requests to a block are routed through the home node, the predictive protocol augments Stache handlers at the home node. At run time, when the home node receives a read (or write) request from a remote node for a cache block, the augmented handler updates the communication schedule to mark the block as read (or written) in that phase. If a block is read and written within the same phase, it is marked as a "conflict" block. This can occur if there is false sharing (i.e., when two processors access distinct parts of the block), or if parallel tasks conflict.

The predictive protocol builds schedules incrementally, starting from an empty schedule. During the first iteration, the protocol identifies faulting cache block accesses and extends the schedule. In subsequent iterations, changes in the communication pattern may cause faulting accesses to additional blocks, which are identified and added to the original schedule. This allows the protocol to track evolving sharing patterns characteristic of adaptive applications.

The predictive protocol works well for incremental additions to a schedule, but does not track deletions. When a processor no longer accesses a block, the protocol transfers the block unnecessarily. For applications whose pattern changes include a significant number of deletions, the schedule must be rebuilt often by flushing the old schedule and building a new one.

3.4 Using Communication Schedules to Presend Data

At the beginning of a subsequent iteration of the parallel phase, compiler directives invoke the pre-send phase of the predictive protocol on all processors to transfer data according to the communication schedule. The goal of the pre-send phase is to anticipate block requests and execute anticipated actions early.

Each processor executes one of two actions for blocks in the communication schedule for which it is the home node. For a block marked "read", the processor sends invalidations to any current writer, and forwards readable copies to all processors marked as readers. For a block marked "write", the processor invalidates current readers or writers, and forwards a writable copy to the marked writer. Currently, there is no action for blocks marked "conflict", since they occur very rarely in programs with independent parallel threads of execution. One possible action for such blocks is to anticipate the first stable block state (read or write) before the conflict occurred.

```
class Grid(float) [][]) {   /* Member functions */    };
```

Figure 1: Aggregate definition syntax in C**

Pre-sent copies are cached at remote nodes with appropriate access control tags (**ReadOnly** or **ReadWrite**). Accesses to cached copies are handled transparently by Tempest, usually at full hardware speeds, without invoking the protocol or other software intervention.

After all blocks in the schedule have been transferred, the protocol enforces a global barrier synchronization to ensure that all protocol cache blocks states are stable and match those expected by the default protocol. For efficiency, the predictive protocol coalesces neighboring blocks and transfers them using bulk messages to amortize message startup costs.

4 Compiler Analysis to Identify Potentially Repetitive Patterns

The predictive protocol relies on directives from the C** compiler to identify points in the program where potentially repetitive communication patterns exist. In C**, as in other data-parallel languages, data-parallel operations clearly divide a program's execution into sequential and parallel phases. The C** compiler uses data-flow analysis to identify repetitive parallel phases that require communication, and augments these phases with directives to invoking the predictive protocol.

Our simple compiler analysis is optimistic and conservative and does not attempt to identify actual patterns of communication in the program (e.g., nearest-neighbor communication), or even that the pattern is really repetitive in the sense that data items requested in a previous iteration will be requested again in a following iteration. While such analysis is routine for programs with mostly static communication patterns, it is infeasible for programs with dynamic communication patterns such as adaptive applications. Our analysis can wrongly identify a non-repetitive pattern as a repetitive one, leading to slower (but still correct) execution of the program with the predictive protocol.

This section describes our data-flow analysis, which proceeds in two phases. First, parallel functions are analyzed to broadly classify their access patterns. Second, the sequential part of the program (which includes calls to parallel functions) is analyzed to identify where annotations for parallel phases must be placed. Before describing compiler analysis, we briefly introduce the data-parallel features of C**.

4.1 Parallel functions in C**

C** is a large-grain data-parallel language based on C++ [11]. It includes all the desirable features of the data-parallel programming model [7], such as a global namespace and nearly deterministic execution. It provides coarse-grain data-parallelism, much like HPF's `DO INDEPENDENT` loops [6], but enforces independent execution of coarse-grain tasks. A detailed description of C** can be found elsewhere [11].

Data-parallel programs express parallelism by invoking a data-parallel operation on a data collection. In C**, data collections are called Aggregates and look like arrays of classes. For example, Figure 1 declares a two-dimensional collection of floating point values. The size of an Aggregate may be specified at runtime when an Aggregate object is created.

Data-parallel operations in C** are user-defined functions denoted by the `parallel` keyword. A parallel function operates on a specific Aggregate argument, which is also denoted by the `parallel`

```
void stencil(parallel Grid &new_A, Grid A) parallel
{
    new_A[#0][#1] = (A[#0 - 1][#1] + A[#0][#1 - 1] + A[#0 + 1][#1] + A[#0][#1 + 1]) / 4.0;
}
```

Figure 2: Stencil in C**

```
void update(parallel Mesh &primal, Mesh &dual) parallel
{
    /* Loop over all in-edges */
    for (int i = 0; i < primal[#0].in_degree; i ++)
        primal[#0].value -= dual[primal[#0].edges[i]].value * primal[#0].coeff[i];
}
```

Figure 3: Unstructured mesh update in C**

keyword. For example, Figure 2 describes a 4-point stencil operation in C**. The pseudo variables `#0` and `#1` identify row and column positions within the Aggregate and allow access to neighboring elements.

Data and Computation Distribution The C** compiler relies on Stache to distribute all shared data at the granularity of a page. The C** system includes a number of simple computation distribution schemes, including block distributions on 1-dimensional Aggregates and row-block and tiled distributions on 2-dimensional Aggregates, but not other general distributions or user-specified data distributions.

4.2 Parallel Function Analysis - Identifying Access Patterns

Calling a parallel function on an Aggregate creates multiple function invocations, one for each element of the Aggregate. Each parallel function invocation "owns" the element of the Aggregate on which it operates. In addition to its "own" element, each invocation may also access neighboring Aggregate elements or elements from other global Aggregates. For example, the parallel function `update` in Figure 3 implements a simple unstructured mesh update on a bipartite mesh (partitioned into `primal` and `dual`). The edge descriptors (and their corresponding transfer coefficients) are stored with each mesh element (the data structure of a mesh element is not shown in the example).

The parallel function `update` in Figure 3 includes unstructured accesses to the `dual` mesh, some of which require inter-processor communication. For each parallel function, the C** compiler uses context-insensitive analysis to compile a list of all Aggregate member accesses that potentially require communication. Each access is (conservatively) categorized as a **Home** access (for example, access to the "own" element), or a **Non-Home** access (for all other accesses). For example, the summary access list of function `update` in Figure 3 contains two elements, (primal, Write access, Home), and (dual: Read access, Non-Home).

4.3 Compiler Analysis to Place Directives

The second step analyzes the sequential portion of the program which includes calls to parallel functions. First, the compiler builds a flow graph of the sequential program, mapping parallel

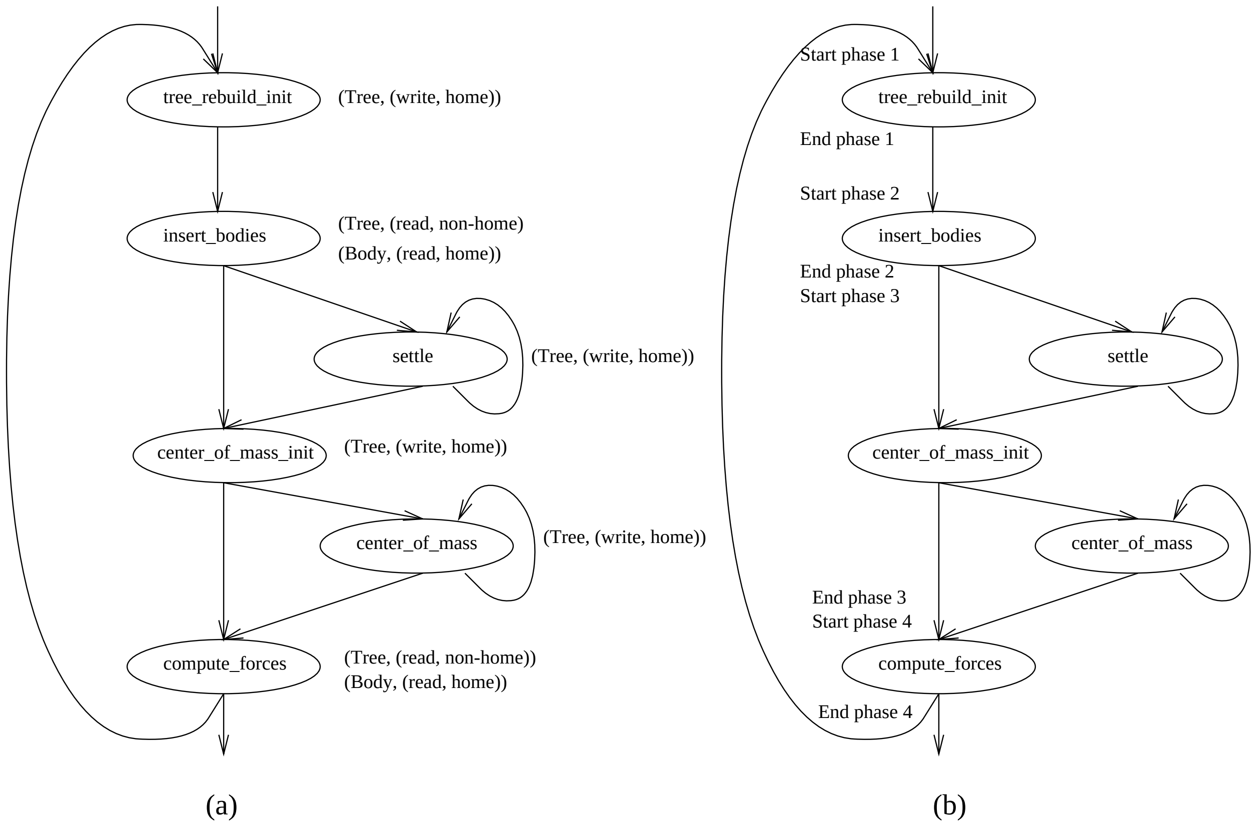

Figure 4: Control flow graph for the main sequential loop in Barnes-Hut. CFG (a) is annotated with parallel function access patterns. CFG (b) is annotated with runtime phase directives for the predictive protocol.

function data access lists back to function call sites. As our compiler currently does not support inter-procedural analysis, the sequential portion is restricted to the main function. For example, Figure 4 displays the control flow graph (CFG) for the main loop in the sequential portion of Barnes-Hut (Section 5.2) annotated with access lists.

We perform data-flow analysis on the sequential section of the program to determine, for each Aggregate at each program point, whether cached copies of Aggregate elements may exist on remote processors due to unstructured read or write accesses. If these copies cannot exist, a single copy of each element is present on its home processor, created by an owner write access. Analogous to reaching definitions, we define the reaching unstructured accesses property, which is true whenever cached copies of an Aggregate element may exist on remote processors.

The compiler uses a forward-flow, any-path data-flow analysis to compute reaching unstructured accesses for each Aggregate at each program point, using a framework identical to the reaching-definition problem. There are three transfer functions for parallel function data accesses:

1. Owner write accesses kill reaching unstructured accesses, because the remote copies are invalidated.

2. Unstructured write accesses kill reaching unstructured accesses, but generate potentially new unstructured accesses.

3. Unstructured read accesses do not kill reaching unstructured accesses (because the protocol allows multiple readers), and generate unstructured accesses.

The compiler computes reaching unstructured accesses using an iterative bit-vector based data-flow computation on the sequential control flow graph.

Results of the reaching unstructured access data-flow computation direct the placement of run-time protocol directives. A parallel function call requires a communication schedule and preceding predictive protocol phase if, for any Aggregate

1. The call is reached by unstructed accesses and includes owner write accesses, or

2. The call includes unstructured accesses itself, whether the reaching property includes unstructured accesses or not.

The placement algorithm also includes one optimization to coalesce multiple communication schedules. The compiler uses an inside-out pass on the CFG to coalesce neighboring phases that include only home accesses, and moves schedules out of loops that contain only home accesses (e.g., function `center_of_mass` in Figure 4). This optimization is analogous to communication schedule coalescing in the inspector-executor model [1], and amortizes the overhead of the predictive protocol over multiple parallel functions. In Figure 4, this optimization allowed a single directive for phase 3.

5 Measuring the Optimizations

In this section, we measure the effect of compiler-directed shared-memory communication on three iterative data-parallel scientific applications (Adaptive, Barnes, and Water) which are described briefly in Table 1. Adaptive and Barnes have dynamic repetitive communication patterns, and Water demonstrates a static repetitive communication pattern. All three applications spend a non-trivial fraction of execution time in remote access latency (Figures 5, 6, 7). We briefly outline the algorithm for each application, and compare the performance of C** versions with and without optimized communication. For Barnes, we also compare both versions against a hand-optimized SPMD version (written by others) that uses an application specific protocol for efficiency [5]. For Water, we compare both versions against the Splash-2 version [18] that is optimized for transparent shared memory.

Each performance graph compares the execution time of two or more versions of each benchmark application relative to the fastest version of that application. All execution times were measured on a 32-processor Thinking Machines CM-5 with the Blizzard [15] shared memory layer. Each bar in the graph is divided into three sections:

Remote data wait Time spent waiting for non-local memory accesses to complete

Predictive protocol Time spent in the pre-send phase of the predictive protocol

Compute+Synch Time spent in computation and synchronization. This portion of the execution time varies between different versions of the same program because of differences in synchronization time.

We also experimented with different cache block sizes for each application. In general, the predictive protocol worked best for small cache blocks (the smallest being 32-bytes), while the unoptimized or hand-tuned SPMD codes were able to exploit larger cache blocks effectively. In

Program	Brief Description	Data set
Adaptive	Structured adaptive mesh	128x128 mesh, 100 iterations
Barnes	Gravitational N-body simulation	16384 bodies, 3 iterations
Water	Molecular dynamics	512 molecules, 20 iterations

Table 1: Benchmark applications

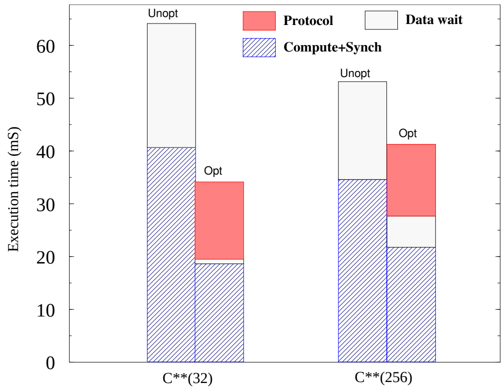

Figure 5: Execution time for 4 C** versions of **Adaptive** — C** versions with and without optimized communication at 2 different cache block sizes. Numbers in parentheses indicate cache block sizes.

addition to the 32-byte block comparison between unoptimized and optimized codes, we also present execution times using programs with larger cache block sizes which minimized execution time for unoptimized or hand-optimized codes.

5.1 Adaptive

Adaptive is a structured mesh calculation that computes electric potentials in a box. The program imposes a mesh over the box and computes the potential at each point by averaging its four neighbors. At points where the gradient is steep, finer detail is necessary and the program subdivides the cell into four child cells. This process iterates until the mesh relaxes. Initially, the mesh is represented by a two-dimensional array, and dynamically allocated quad trees capture cell subdivision. Each iteration of the program consists of a red-black sweep over the mesh computing averages. Within each sweep, each cell updates values in its quad tree, reading values from neighboring points. The predictive protocol optimizes data movement from neighbor reads in the quad tree.

Figure 5 shows that the predictive protocol successfully reduces shared-data wait time by presending data. The protocol also indirectly reduces synchronization time in Adaptive, resulting in significantly lower total execution time. Synchronization time is reduced because load imbalance in Adaptive implies that the shared-data wait time is distributed unevenly among processors, and differences in wait time contribute to synchronization time on lightly loaded processors. At a larger cache block size of 256 bytes (the best case for the unoptimized program), the predictive protocol is less effective because it transfers larger amounts of data, some of which may be redundant. The best optimized version of Adaptive is 1.56x faster than the best unoptimized version.

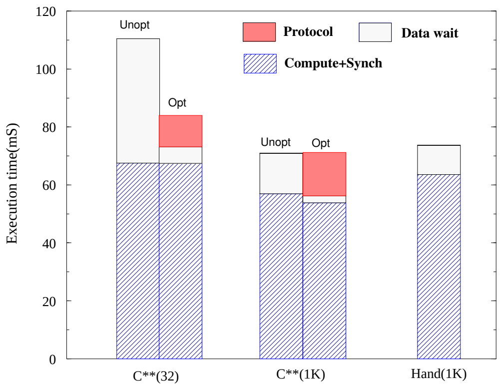

Figure 6: Execution time for 5 versions of **Barnes** — C** versions with and without optimized communication at 2 different cache block sizes, and hand-optimized SPMD. Numbers in parentheses indicate cache block sizes.

5.2 Barnes

Barnes [17] simulates the movement of bodies in a gravitational system over time. The bodies are modeled as point masses that exert gravitational forces on other bodies. The algorithm computes forces between bodies which are used to update body positions in each time step. Rather than computing all N^2 forces, Barnes approximates the force exerted by a distant collection of bodies by that of a point mass at the center of mass of the collection.

Barnes uses an oct-tree to represent bodies in 3-dimensional space. Each node in the tree represents a region in space, with a child representing one octant of its parent's space. The tree is unbalanced and deeper in regions of high body density. To calculate the force on a body, the algorithm performs a depth-first traversal of the tree. If an interior node is sufficiently far away from the body, the bodies in that region are approximated by a point mass at the tree node. Otherwise, the algorithm "opens" up the interior node and traverses its subtrees. If the force computation encounters a body at the leaf of the tree, it computes interactions with that body.

The Barnes algorithm in C** includes unstructured accesses to tree nodes during two phases, the force computation phase, and the tree-build phase, with the center-of-mass calculation in between. The compiler inserts directives for 4 parallel phases in the program where transitions between non-home and home accesses occur (Figure 4).

Figure 6 shows that communication optimization reduces shared-memory wait time significantly for 32-byte cache blocks. However, Barnes shows good spatial locality and the unoptimized version benefits significantly from 1024-byte blocks making it marginally faster than the optimized version. Both 1024-byte versions are slightly faster than a hand-optimized SPMD version of Barnes [5] that uses a write-update protocol for efficient shared-memory communication on the CM-5.

5.3 Water

Water [17] evaluates forces and potentials in a system of water molecules over a number of time steps. The potential of the system includes inter-molecular potentials arising from interactions between molecules. The program computes interactions between all pairs of molecules that lie within a spherical cutoff range equal to half the length of the box enclosing all molecules. In the data-parallel implementation of Water, each molecule potentially computes interactions with half

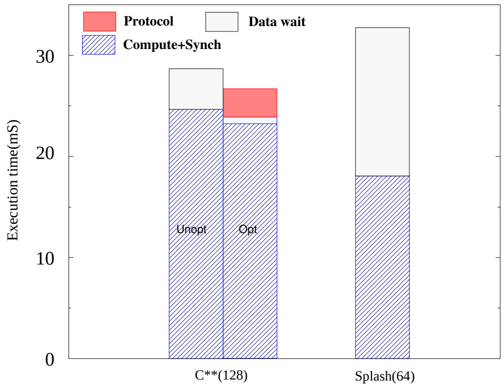

Figure 7: Execution time for 3 versions of **Water** — C** with and without optimized communication, and shared-memory Splash. Numbers in parentheses indicate cache block sizes.

the remaining molecules following it in the ordered data set.

Compiler-directed communication optimizations target the interaction computation phase which uses a static repetitive producer-consumer sharing pattern — a molecule's position updated in one iteration is read by $\frac{n}{2}$ other molecules in the following iteration.

Communication optimization reduces shared-memory wait time for Water (Figure 7), but results in small execution time improvements overall (1.05x). The optimized version is 1.2x faster than the Splash version [18], which is optimized for transparent shared memory, and does not utilize custom protocols or message-passing primitives for communication. The cache block sizes were chosen to show the best case for each version.

5.4 Discussion

The predictive protocol decreases remote memory access latency at the cost of an extra pre-send phase and the cost of building communication schedules in augmented protocol handlers. This technique is beneficial on multiprocessor machines with significant remote memory access latency, such as Blizzard on the CM-5 (200 microseconds average remote access latency), or networks of workstations without hardware support for shared memory. The tradeoff is likely to be different for shared-memory multiprocessors or hardware-assisted DSMs, which have smaller remote access latencies.

The predictive protocol also coalesces neighboring cache blocks in the pre-send phase to amortize message startup costs over large messages. The benefits of this optimization should extend uniformly to all classes of distributed-memory multiprocessor machines, possibly with better results than on the CM-5 network which is optimized for small messages.

6 Conclusion

Many scientific applications simulate physical systems using iterative parallel computations. Typically, these applications involve communication patterns that are also repetitive. This paper demonstrates that cooperation between a data-parallel language compiler and a predictive protocol in a cache-coherent DSM can automatically improve shared-memory communication for repetitive producer-consumer or migratory communication patterns. The compiler uses simple static analysis to identify points in the program where potentially repetitive communication patterns exist. The predictive protocol augments the default shared-memory protocol to use communication schedules generated in one iteration to pre-send data in subsequent iterations.

The combination of compiler-analysis and memory system support gives this approach two advantages. First, dynamic run-time support from the memory system allows our approach to optimize adaptive problems whose reference patterns cannot be analyzed by a compiler and which incur large overheads in compiler-implemented shared-memory approaches. Second, communication pattern analysis in the compiler enables automatic custom protocol usage. This approach inherits some of the advantages of application-specific protocols, but is far simpler for a programmer.

Experiments with three applications show that pre-sending data with this approach effectively reduces the amount of time spent waiting for shared data when compared to the request-response model of a write-invalidate coherence protocol. In two cases, the optimized program was 1.05x and 1.50x faster than the unoptimized version. In the third case, the unoptimized program was able to exploit a larger cache block size to run slightly faster than the optimized program.

Acknowledgements We are indebted to Chuck Koelbel and the other anonymous referees for many comments that helped improve the presentation significantly.

References

[1] Gagan Agrawal and Joel Saltz. **Interprocedural Compilation of Irregular Applications for Distributed Memory Machines**. In Proceedings of Supercomputing '95, San Jose, CA, November 1995.

[2] Satish Chandra, James R. Larus, and Anne Rogers. **Where is Time Spent in Message-Passing and Shared-Memory Programs?** In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), pages 61–75, October 1994.

[3] Satish Chandra, Brad Richards, and James R. Larus. **Teapot: Language Support for Writing Memory Coherence Protocols**. In Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI), May 1996.

[4] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. **Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures**. Journal of Parallel and Distributed Computing, 22(3):462–479, September 1994.

[5] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. **Application-Specific Protocols for User-Level Shared Memory**. In Proceedings of Supercomputing '94, pages 380–389, November 1994.

[6] High Performance Fortran Forum. **High Performance Fortran Language Specification**. Version 1.0, May 1993.

[7] W. Daniel Hillis and Guy L. Steele, Jr. **Data Parallel Algorithms**. Communications of the ACM, 29(12):1170–1183, December 1986.

[8] Scott R. Kohn and Scott B. Baden. **Irregular coarse-grain data parallelism under LPARX**. Journal of Scientific Programming. To appear.

[9] Scott R. Kohn and Scott B. Baden. **A Parallel Software Infrastructure for Structured Adaptive Mesh Methods**. In Proceedings of Supercomputing '95, San Jose, CA, November 1995.

[10] Jeffrey Kuskin et al. **The Stanford FLASH Multiprocessor**. In Proceedings of the 21st Annual International Symposium on Computer Architecture, pages 302–313, April 1994.

[11] James R. Larus, Brad Richards, and Guhan Viswanathan. **Parallel Programming in C**: A Large-Grain Data-Parallel Programming Language**. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[12] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. **Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse**. In Proceedings of Supercomputing '93, pages 361–370, Portland, Oregon, November 1993.

[13] Umakishore Ramachandran, Gautam Shah, Anand Sivasubramaniam, Aman Singla, and Ivan Yanasak. **Architectural Mechanisms for Explicit Communication in Shared-Memory Multiprocessors**. In Proceedings of Supercomputing '95, San Jose, CA, November 1995.

[14] Steven K. Reinhardt, James R. Larus, and David A. Wood. **Tempest and Typhoon: User-Level Shared Memory**. In Proceedings of the 21st Annual International Symposium on Computer Architecture, pages 325–337, April 1994.

[15] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. 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), pages 297–307, October 1994.

[16] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. **Run-time and Compile-time Support for Adaptive Irregular Problems**. In Proceedings of Supercomputing '94, pages 97–106, November 1994.

[17] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. **SPLASH: Stanford Parallel Applications for Shared Memory**. Computer Architecture News, 20(1):5–44, March 1992.

[18] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. **The SPLASH-2 Programs: Characterization and Methodological Considerations**. In Proceedings of the 22nd International Symposium on Computer Architecture, pages 24–36, Santa Margherita Ligure, Italy, June 1995.