Appears in the *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP),* July 1995.

Efficient Support for Irregular Applications on Distributed-Memory Machines*

Shubhendu S. Mukherjee†, Shamik D. Sharma‡, Mark D. Hill†, James R. Larus†, Anne Rogers⋆, and Joel Saltz‡

†Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706, USA
{shubu,markhill,larus}@cs.wisc.edu

‡Department of Computer Science
University of Maryland
4166 A.V. Williams Building
College Park, MD 20742, USA
{shamik,saltz}@cs.umd.edu

⋆Department of Computer Science
Princeton University
35 Olden Street
Princeton, NJ 08544, USA
amr@cs.princeton.edu

Abstract

Irregular computation problems underlie many important scientific applications. Although these problems are computationally expensive, and so would seem appropriate for parallel machines, their irregular and unpredictable run-time behavior makes this type of parallel program difficult to write and adversely affects run-time performance.

This paper explores three issues—partitioning, mutual exclusion, and data transfer—crucial to the efficient execution of irregular problems on distributed-memory machines. Unlike previous work, we studied the same programs running in three alternative systems on the same hardware base (a Thinking Machines CM-5): the CHAOS irregular application library, Transparent Shared Memory (TSM), and eXtensible Shared Memory (XSM). CHAOS and XSM performed equivalently for all three applications. Both systems were somewhat (13%) to significantly faster (991%) than TSM.

1 Introduction

Irregular computation problems underlie many important scientific applications. These problems arise in computational fluid dynamics, computational molecular dynamics, and particle-in-cell computations. A defining characteristic of these computations is that their data structures are not regular dense matrices and their data-access patterns are unknown until run time. In Fortran 77, irregular programs typically use index arrays (also called indirection arrays) to introduce a level of indirection in array accesses.

Although these problems are computationally expensive, and so would seem appropriate for parallel machines, their irregular and unpredictable run-time behavior makes this type of parallel program difficult to write and adversely affects run-time performance. Message-passing machines are poorly suited to support these programs directly for two reasons. First, most of these machines offer high-bandwidth, high-latency communication, which favors large, infrequent messages, not the short, frequent messages that result from irregular computations. Second, message-passing machines rarely support a shared address space. Irregular applications distribute complex data structures among processors' local address spaces, and hence, must provide mechanisms to name and access remote data. Shared-memory machines alleviate these problems, but introduce new ones. These machines typically use caches to reduce both memory latency and bandwidth requirements. A coherence protocol manages the caches and ensures that all processors see a consistent view of memory. However, when a machine's protocol does not match a program's sharing pattern, the protocol can cause excessive communication and overhead.

The CHAOS system is a well-proven library that supports irregular applications [9] and mitigates the problems of message-passing machines. CHAOS offers parallel data partitioners, a global address space for *distributed arrays*, and operations to move data between processors. It greatly eases the task of programming irregular applications by hiding communication and buffer management and by providing a portable framework for programming these applications. Previous research showed that CHAOS

*This research was 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 PYI/NYI Awards MIPS-8957278, and CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, DOE Grant DE-FG02-93ER25176, ARPA (NAG-1-1485), NSF (ASC 9213821), ONR (SC292-1-22913), EPRI (RP3103-6), ARPA I3 Initiative (N00014-94-10907), NASA (NAG-11560), University of Wisconsin Graduate School Grant, Wisconsin Alumni Research Foundation Fellowship and donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, Sun Microsystems, and Thinking Machines Corporation. The Wisconsin Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

0

achieved good speedups on message-passing machines for irregular applications [9, 16, 24]. In this paper, we use an implementation of CHAOS on a Thinking Machines CM-5.

Recent research in computer architecture has led to another alternative: hybrid shared-memory and message-passing machines that offer programmers the opportunity to select coherence protocols and fall back to message-passing communication [13, 14, 19]. The Wisconsin Wind Tunnel project's approach is a portable, user-level interface called *Tempest* [11, 19, 18], which provides message-passing communication and mechanisms to construct shared-memory protocols. In particular, Tempest provides programs with the novel ability to copy and move data without changing its address (renaming it). In this paper, we use an implementation of Tempest called *Blizzard* [23] that runs on a CM-5.

Tempest is a general-purpose parallel programming interface that is not focused on particular application domains. This research used Tempest in two ways. First, transparent shared memory (TSM) is a Tempest library that provides programs with sequentially-consistent shared memory using a write-invalidate protocol and program-selected block sizes. From the program's perspective, the program appears to be running on a shared-memory machine. Second, extensible shared memory (XSM) improves on TSM by using Tempest features to communicate selected data structures through custom shared-memory or message-passing protocols.

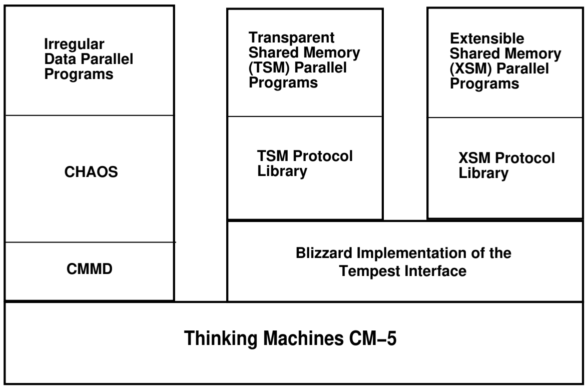

Figure 1: CHAOS, TSM, and XSM Approaches.

This figure illustrates the three software approaches studied in this paper. All run on a Thinking Machines CM-5. CHAOS uses the CMMD message-passing library for communication. TSM provides shared memory using the Blizzard implementation of the Tempest interface. XSM uses customized protocols on the same Blizzard substrate.

Figure 1 illustrates the structure of the three systems: CHAOS, TSM, and XSM. The systems provide a unique opportunity to understand the essential characteristics of irregular problems by comparing three different approaches to programming them on the same hardware platform. We used three *complete* irregular applications—*unstructured*, *moldyn*, and *DSMC*. CHAOS has speedups of 20.4, 23.1, and 19.2, respectively (all numbers on 32 processors). TSM achieves speedups of 1.9, 20.3, and 17.2. XSM has speedups of 20.9, 23.1, and 20.9. The bottom line is that CHAOS and XSM are roughly equivalent and both are better than TSM.

In general, we found a lot of similarity between the techniques necessary to achieve good performance for these three approaches. This result should not be surprising since all three ran the same application code on the same hardware and, consequently, mapped the same program abstractions to the same hardware constraints. We identified three crucial issues for achieving high performance on distributed memory machines.

- **Partitioning**. Irregular programs typically operate on fine-grain data (4–100 bytes). The structure of a problem defines interactions and data dependences among operations on the data. For example, a computation on a node in an unstructured mesh may need values from nodes connected by mesh edges. To run this problem on a parallel machine, a programmer must partition both the program that iterates over data and the data itself. A simple block or cyclic partition may be inefficient because problem irregularities may lead to load imbalance or excessive communication among processors. In such cases, partitioners must be aware of the problem's structure.

- **Mutual Exclusion**. Irregular programs require efficient mutual exclusion to support remote atomic updates and global reductions. Locks perform poorly in both roles. A better approach is for each processor to accumulate local contributions until it can update a shared resource efficiently.

- **Data Movement**. Fine-grained data distribution and irregular data-access patterns complicate efficient communication. Vectorizing or combining messages to be transferred between two processors amortizes message-sending overhead and reduces the overhead of communication. Careful partitioning of data and loop iterations and caching data can reduce the volume of communication.

The rest of this paper examines these three issues for three irregular applications (*unstructured*, *moldyn*, and *DSMC*) running on three systems (CHAOS,

```
for (i = 0; i < number_timesteps; i++)
{
   ...
   for (j = 0; j < number_edges; j++)
   {
      n1 = edge[j].left_node;
      n2 = edge[j].right_node;
      w = f(n1, n2, j);
      y[n1] += g1(x[n1], x[n2], w);
      y[n2] += g2(x[n1], x[n2], w);
   }
}
```

Figure 2: Sequential irregular loop

TSM, and XSM). Section 2 discusses related work. Section 3 and Section 4 discuss CHAOS and Tempest, respectively, and show how to parallelize an irregular loop with these systems. Section 5 describes the three applications in detail and focuses on how we handle partitioning, mutual exclusion, and data transfer. Section 6 gives a general discussion and discusses two additional issues—address space management and buffer management. Section 7 presents our conclusions.

2 Related Work

Most work on irregular applications has focused on message-passing distributed-memory machines. Koelbel and Mehrotra built a system, Kali [12], that is similar to PARTI [20], CHAOS' predecessor. Culler et al. [7] discuss many of the same issues as this paper in describing their improvements to EM3D running under Split-C. Chakrabarti and Yelick [4] describe parallelizing the Gröbner basis problem, which is an irregular application with a finer granularity of sharing than those in this paper. Unlike this paper, previous work did not compare alternative implementations.

Two recent papers address some issues on shared-memory machines. Tomko and Abraham [25] show that careful data reordering techniques for an irregular application improved performance between 8%–16% on a Kendall Square KSR-1. Mirchandaney et al. [15] suggest protocol enhancements to TreadMarks [6]—a distributed shared-memory system—to improve mutual exclusion and communication for irregular scientific problems.

Falsafi et al. [10] describe how custom Tempest protocols improved the performance of two irregular applications (Barnes [1] and EM3D). They do not compare Tempest against alternative implementations, such as CHAOS.

```
Partition x and y
Partition iterations of inner loop
Build translation tables for x and y
Inspector

for (i = 0; i < number_timesteps; i++)
{
   ...
   Gather x
   for (j = 0; j < local_number_edges; j++)
   {
      n1 = local_edge[j].left_node;
      n2 = local_edge[j].right_node;
      w = f(n1, n2, j);
      y[n1] += g1(x[n1], x[n2], w);
      y[n2] += g2(x[n1], x[n2], w);
   }
   Scatter y
}
```

Figure 3: Irregular loop parallelized with CHAOS

3 CHAOS

CHAOS [21]—the successor to the PARTI library [20]—is a library that supports parallel execution of irregular applications.

3.1 CHAOS Overview

CHAOS provides four types of support for irregular applications:

Data and Iteration Partitioners. CHAOS supports data partitioners, such as recursive coordinate bisection (RCB) [2], recursive spectral bisection [17], and others. It also provides loop iteration partitioners for rules like owner-computes and almost-owner-computes.

Support for Global Address Space. CHAOS implements a global address space for *irregularly distributed* arrays. It maintains a translation table that maps global indices to local indices for each such array on every processor. The application sees only one address space because CHAOS copies remote data into a processor's local address space and then hides the renaming by changing the indirection array to point to the local copy.

Inspector/Executor Primitives. At the heart of CHAOS is the inspector/executor model [22]. Before a computation, a preprocessing step, called an *inspector*, identifies the communication in subsequent loops. When the loops execute, CHAOS primitives, such as `gather` and `scatter`, use the information collected by the inspector to communicate values in distributed arrays. The information permits several optimizations, such as vectorizing communication and retrieving a single copy of multiply-referenced off-processor data. Section 3.2 illustrates the use of these primitives with an example.

Buffer Management. CHAOS uses software caching—implemented with hash tables—to store remote data in a processor's local memory.

3.2 Programming Example

Figure 2 shows an irregular loop from *unstructured*. Figure 3 shows this loop expressed with CHAOS primitives. CHAOS executes the same code on all N processors, using the *single-program-multiple-data* (SPMD) computation model. Processors perform different work, because each proccessor's data and metadata (e.g., indirection arrays) are different. A programmer must insert calls to CHAOS primitives before the outer loop to: partition arrays `x` and `y`, partition inner loop iterations, build translation tables for `x` and `y`, and inspect the communication pattern of the arrays. Within the inner loop, the programmer inserts the CHAOS primitive `gather` to collect the most recent values of `x` from other processors. Similarly, after the inner loop, the programmer inserts the CHAOS primitive `scatter` to send updated values of `y` to other processors.

4 Tempest

This section provides an overview of the Tempest interface, on which we implemented both transparent shared memory (TSM) and extensible shared memory (XSM).

4.1 Tempest Overview

Tempest in an interface to a portable substrate for parallel program communication. Tempest provides mechanisms that allow programmers, compilers, and program libraries to implement and use message passing, shared memory, and other hybrid models. Tempest is designed so that it can be supported on many platforms, providing portability across these systems. The Blizzard system implements the Tempest substrate on a Thinking Machines CM-5 and is being ported to the Wisconsin COW (Cluster of Workstations) [11].

```
for (i = 0; i < number_timesteps; i++)
 {
   ...
   for (j = start_edge; j <= end_edge; j++)
   {
      n1 = edge[j].left_node;
      n2 = edge[j].right_node;
      w = f(n1, n2, j);
      lock(y_lock[n1]);
      y[n1] += g1(x[n1], x[n2], w);
      unlock(y_lock[n1]);
      lock(y_lock[n2]);
      y[n2] += g2(x[n1], x[n2], w);
      unlock(y_lock[n2]);
   }
 }
```

Figure 4: Irregular loop parallelized in TSM

The Tempest mechanisms are low-overhead "active messages," bulk data transfer, virtual memory management, and fine-grained memory access control. Memory access control allows a programmer to prevent access to an aligned memory block (e.g., 32 or more bytes). Inappropriate accesses (e.g., a store into a *ReadOnly* block) generate faults that are vectored to a user-level handler. The Tempest Interface Specification [18] and several other papers discuss Tempest in more detail [10, 11, 19].

TSM uses a COMA-like transparent shared-memory cache-coherence protocol called *stache* [19]. The *stache* protocol uses a fraction of the local memory as a large, fully-associative cache to hold data evicted from the hardware cache. A *stache miss* occurs when a processor references shared data that is not currently cached in its local memory. This paper uses two *stache block* sizes—32 bytes and 1024 bytes. Tempest also provides a synchronization library (implemented with Tempest messages) for mutual exclusion. XSM adds custom protocols to the TSM base protocol.

4.2 Programming Example

This section shows how to parallelize the irregular loop in Figure 2 for transparent shared memory (TSM) and how to improve its performance with custom user-level protocols.

The programming model for TSM is SPMD with a single global address space. Arrays `x` and `y` are globally shared arrays. The easiest way to parallelize this loop is to block partition edges among processors and protect updates to `y[n1]` and `y[n2]` with locks.

```
for (i = 0; i < number_timesteps; i++)
{
   ...
   for (j = start_edge; j <= end_edge; j++)
   {
      n1 = edge[j].left_node;
      n2 = edge[j].right_node;
      w = f(n1, n2, j);
      y[n1] += g1(x[n1], x[n2], w);
      y[n2] += g2(x[n1], x[n2], w);
   }

   reduce y
}
```

Figure 5: Irregular loop parallelized with XSM

Figure 4 contains the code executed by all processors. The variables `start_edge` and `end_edge` define the range of global edges for each processor. Array `y_lock` contains the locks for nodes of array `y`.

To improve the performance of the TSM code, we can implement a custom protocol to manage communication through arrays `x` and `y`. In addition, we can eliminate locks on array `y` with a reduction protocol (Figure 5) that accumulates values for `y` locally, and reduces the entire array after execution of the inner loop. With this change, however, processors will still need to fault in array `x`. We can improve this implementation further by writing an update protocol that captures the sharing of blocks in array `x` during the first iteration and directly sends updates before the inner loop in subsequent iterations. Falsafi et al. [10] discuss several flavors of custom update protocols.

5 Results

This section describes how we ran three irregular applications—*unstructured*, *moldyn*, and *DSMC*—using three alternative systems: CHAOS, transparent shared memory (TSM) on Tempest, and extensible shared memory (XSM) on Tempest. For each application, it describes partitioning, mutual exclusion, and data transfer in the three systems. In Section 6, we discuss two other issues—address space and buffer management. Table 1 describes the input sets used for our three applications.

5.1 Unstructured

Unstructured is abstracted from a computational fluid dynamics application that uses an unstructured mesh to model a physical structure, such as an airplane wing or body. The mesh is represented by *nodes*, *edges* that connect two nodes, and *faces* that connect three or four nodes. The mesh is static, so its connectivity does not change. The computation contains a series of loops that iterate over nodes, edges, and faces.

Table 2 shows the CHAOS, TSM, and XSM execution times and speedups of the parallel phase of *unstructured* running on 32 processors. The first row contains the CHAOS timing and speedup. The rest of the table details improvements achieved with TSM and XSM. The first column lists the optimizations, which are described and discussed in the body of the paper. The second column lists the stache block size, the third column reports the execution time on 32 processors, and the fourth column gives the speedup. Note, the *unstructured* times do not include preprocessing (inspector and partitioning), since some of these steps are not completely parallelized for the TSM and XSM versions. However, in CHAOS, they constitute less than 6% of the total time and can be amortized over the large number of iterations typical of production runs of this code.

5.1.1 Partitioning

The structure of the mesh, which is static but unknown until run time, determines interactions among processors. The mesh is described by associating names with the nodes, edges, and faces. Unfortunately, these names usually do not reflect the mesh's structure. As a result, block or cyclic partitions can lead to excessive communication. To rectify this, all three implementations (CHAOS, TSM, and XSM) partition the nodes using recursive coordinate bisection (RCB) [2], which groups related nodes. Once the nodes have been grouped, a simple partitioning scheme suffices for the edges. An edge that connects two nodes in the same partition is assigned to that partition and an edge that crosses between partitions, known as a *cut edge*, is assigned to the partition with fewer edges. Faces are partitioned in a similar fashion.

The three implementations partition the data in the same way, but differ in how they use these partitions. The CHAOS implementation changes the indirection arrays to reflect a node's new location and then assigns data to processors based on the partitioning. Data in a single partition is assigned to one processor's local memory. The TSM and XSM versions also change the indirection arrays, but instead of assigning a partition to a processor, these versions reorder the array so that data in the same partition is placed in contiguous addresses in shared memory. Once the data is partitioned, loops that iterate over

Table 1: Input data sets for the three applications.

Application	**Scientific Domain**	**Input Data Set**	**Mesh Statistics for 32 Processors**
unstructured	Computational Fluid Dynamics (CFD)	9428 nodes, 59863 edges, 5864 faces, 50 iterations	32% cut edges with recursive coordinate bisection partitioner
moldyn	Molecular Dynamics	8788 nodes, 1 million interactions, interaction list rebuilt twice, 30 iterations	58% cut interactions with recursive coordinate bisection partitioner
DSMC	Particle in cell	initially 48600 particles, eventually 72693 particles, 9720 cells, 400 iterations	Average outflux per processor = 176 particles per iteration Average influx per processor = 174 particles per iteration

the nodes, edges, and faces are partitioned in the obvious way.

Partitioning reduces communication and improves performance dramatically. The number of cut edges is a good metric for communication cost. For the input mesh, the RCB partitioner reduces the number of cut edges from 99% (block partitioner) to 32%. This reduction in cut edges reduces the number of stache misses from 84.6 million in *TSM-initial* to 25.9 million in *TSM-partition (level 1)* and accounts for the substantial performance improvement (156%) in *TSM-partition (level 1)*.

CHAOS's gather primitive explicitly packs nodes for another processor (those part of cut-edges) before sending them. The TSM implementation acquires data through stache misses. To reduce the stache misses caused by cut edges, we reorder data within each partition a second time using RCB. This reordering called *TSM-partition (level 2)* groups related nodes within a partition and increases spatial locality, which reduces the number of stache misses caused by cut-edges by 2.6% and improves performance over *TSM-partition (level 1)* by 13%.

Stache misses in the TSM implementation arise from both true sharing through cut edges and false sharing of different nodes residing in the same stache block. In *TSM-partition (level 2)*, we reduced the stache misses caused by true sharing by rearranging each processor's local portion of the mesh. Padding the partitions to block boundaries (*TSM-padding*) reduces false sharing, decreases the number of stache misses by 15%, and improves performance by 23%.

5.1.2 Mutual Exclusion

The computation in *unstructured* consists of a series of loops over nodes, edges, and faces. For each iteration of a typical edge loop, the loop updates variables associated with nodes $n1$ and $n2$ to include the contribution represented by the edge $\langle n1, n2\rangle$. The updates for a node form a reduction that can be performed in any order, so long as each update occurs atomically.

All three implementations perform reductions directly. In CHAOS, a reduction requires three steps. First, the inspector discovers the connectivity of the mesh, which determines the sharing patterns. Second, each processor computes its local contribution to each node. And third, `scatter`, a library primitive, uses the connectivity information to send local contributions to nodes' owners.

The TSM implementations perform a preprocessing step to split nodes in the partitioned graph into two groups: internal nodes, which have no incident cut edges, and external nodes. For our input matrix, approximately 28% of the nodes are internal after the partitioning. Internal nodes are only locally updated. Updates to external nodes require mutual exclusion to guarantee atomicity. The TSM implementations use locks for this purpose. Locks, however, perform poorly for two reasons. First, locks in Blizzard are expensive. Second, sharing, both true and false, forces some stache blocks to ping-pong between processors, which causes a large number of misses.

The final TSM version (*TSM-reduction*) implements the reduction directly. First, each processor computes its local contributions to the nodes and then participates in a global reduction. The global reduction operates in a pipelined fashion by dividing the global array into N pieces and having each of N processors update a different piece in $N-1$ steps. *TSM-reduction* also removes the padding to make the data structures compact. Since the shared arrays are updated only during the reduction phase, severe ping-ponging due to false sharing no longer occurs. These optimizations reduce the number of stache misses from 21.4 million to 2.3 million and improve performance by 823% over *TSM-padding*.

The XSM version implements the reduction with a custom protocol called *direct-reduction*, which also optimizes the data transfer. We discuss this in the

Table 2: Unstructured results.

Version	Block Size (bytes)	Time (seconds)	Speedup
CHAOS		33	20.38
TSM			
initial partition	1024	11628	0.06
level 1	1024	4550	0.15
level 2	1024	4011	0.17
padding	1024	3257	0.21
reduction	1024	353	1.91
XSM			
direct-reduction	1024	128	5.26
block-update	32	54	12.49
node-update	1024	32	20.86

next subsection.

5.1.3 Improving Data Transfer

As mentioned in Section 1, the irregular and fine-grain data dependences in irregular problems complicate efficient message-passing communication. The CHAOS implementation of *unstructured* used highly optimized `gather` and `scatter` routines that exploit information collected by the inspector to collect multiple messages to a processor into a single transfer and to eliminate redundant communication.

TSM-reduction reduced communication overhead with a larger block size, which reduces the number of stache misses (from 21.4 million with 32-byte blocks to 2.3 million with 1024-byte blocks). The partitioning and reduction optimizations discussed previously make large block sizes practical by increasing spatial locality and reducing false sharing.

XSM versions replace the "all-purpose" protocol of TSM with three custom protocols that both reduce communication and execution time. The first, *direct-reduction*, uses information about the mesh's connectivity and Tempest's virtual channel mechanism to improve reductions. The other two, *block-update* and *node-update*, use information about the mesh's connectivity to reduce the cost of acquiring data.

Direct-reduction determines the mesh's connectivity through a preprocessing step similar to CHAOS's inspector phase and then uses this information to establish virtual channels between processors that share cut-edges. *Direct-reduction* then sends local contributions directly to the processor that owns the data. This reduces the message traffic for reductions substantially (the traffic volume decreases by 21% and the number of messages decreases by 50%) and improves performance over *TSM-reduction* by 175%.

Although partitioning and renaming nodes improves spatial locality and reduces communication, references in the edge- and face-loops to nodes updated in earlier loops still cause stache misses. An update protocol can avoid these misses by sending updated nodes directly to consumers. Our first update protocol (*block-update*) captures nodes' sharing lists during the program's first iteration by running a modified version of the TSM protocol that records sharing. In subsequent iterations, the protocol sends shared stache blocks directly between processors using virtual channels. Note that since sharing is recorded in the protocol and a node has the same address on all processors, this improvement does not require inspector code and a loop's computation portion need not change. This update protocol reduces communication volume by 74% and improves performance by 137% over *direct-reduction*.

Block-update captures the sharers at the granularity of a block, which is too coarse and results in some nodes being sent unnecessarily. Using small blocks (32 bytes versus 1024 bytes) reduces this effect, but does not eliminate it entirely. Our second update protocol (*node-update*) eliminates this effect by recording sharing information on a per node rather than a per block basis. *Node-update* examines the partitioned mesh to determine the sharing patterns and establishes virtual channels at the start of the program. This reduces communication volume by 49% and improves performance by 67% over *block-update*.

In summary, properly partitioning data and efficiently performing reductions improves transparent shared memory's performance by a factor of 33. At this point, it becomes necessary to manage *unstructured*'s primary data structures with a custom protocol, which further improves the reductions and distributes data through an update, rather than an invalidation, protocol. These changes produce another factor of 11 improvement, which brings the performance to the level of CHAOS.

5.2 Moldyn

Moldyn is a molecular dynamics application. Its computational structure resembles the non-bonded force calculation in CHARMM [3]—a well-known molecular dynamics code used at NIH to model macromolecular systems. Molecules in *moldyn* are uniformly distributed over a cuboidal region with a Maxwellian distribution of initial velocities. A molecule's velocity and the force exerted by other molecules determine the molecule's position. The force computation limits interactions to molecules within a cut-off radius. An interaction list—rebuilt every 20 iterations—records pairs of interacting molecules. Table 3 shows the results for our implementations of *moldyn*.

Table 3: Moldyn results.

Version	Block Size (bytes)	Time (seconds)	Speedup
CHAOS		38	23.13
TSM			
late-commit	1024	474	1.85
reduction	1024	43	20.32
XSM			
bulk-reduction	1024	38	23.11

5.2.1 Partitioning

Moldyn has two main data structures, a molecule list and an interaction list, and two main loops, the interaction list computation and the force computation. The CHAOS implementation uses the RCB partitioner to assign molecules to processors. The partition for the molecules also partitions the interaction list, the interaction list computation loop, and the force computation loop. The processor with the lower-numbered molecule handles interactions between pairs of molecules. The processor that holds an interaction computes iterations of the force computation loop, which walks over the interaction list. This partitioning eliminates the need to communicate data between the interaction list computation and force computation, but may lead to an imbalance in number of interactions assigned to each processor in the force computation phase.

Both the TSM and XSM versions rename and reorder molecules in shared memory using the partitioning generated by RCB. These implementations do not use the same assignment algorithm as the CHAOS implementation for the interaction list, the interaction list computation loop, and the force computation loop. Instead, they use an assignment that tries to equalize the computation to generate the interactions and distribute remote interactions evenly across the interaction list. The sequential code to compute the interaction list is a triangular loop over pairs of molecules:

```
for (i = 0; i < num_molecules; i++)
   for (j = i + 1; j < num_molecules; j++)
      { ... }
```

Because iterations of the outer loop have unequal numbers of inner loop iterations, block partitioning the outer loop would cause a load imbalance. To rectify this, we rewrite the loop in the following way:

```
for each processor
    Compute interactions among local molecules

for (p = 0; p < num_procs; p++)
   for (q = p + 1; q < num_procs; q++)
       Compute interaction between p's molecules
       and q's molecules
```

We divide up the iterations of the p and q loops to distribute the $\langle p, q\rangle$ pairs equally and then assign interactions to the processor that generates them. This yields an irregular block distribution, which is also used for the force computation loop.

5.2.2 Mutual Exclusion

When a processor generates an interaction, it must append it to the interaction list, which is a reduction. In the CHAOS implementation, mutual exclusion is unnecessary, because a processor only appends entries to its local interaction list. With shared memory, the list is shared and mutual exclusion is required while building the list and computing forces. Our initial TSM implementation, which is not shown in the table, used a global counter, protected by a lock, to indicate the next free entry. This implementation had two problems: the counter was a bottleneck and locks were expensive. We eliminated both problems with a *late-commit* reduction, in which each processor collects its interactions into a local buffer. At the end of the loop, each processor knows how many interactions it generated and joins a partial sum computation to find a starting index in the global list. Each processor then copies its local list to the global interaction list.

The *late-commit* version also optimizes the mutual exclusion in the force computation loop by splitting the loop into two parts. The first part computes the forces for local interactions (roughly 42% of total interactions) and does not need locks. The second part computes the forces for cut interactions and still needs locks to protect updates to remote molecules.

The force computation in *moldyn*, however, is also a reduction like the interaction list computation. The CHAOS and TSM implementations use the same mechanisms as the reductions in *unstructured*. Our XSM implementation uses *bulk-reduction*, an optimized version of the shared memory reduction. *Bulk-reduction* mimics the data movement in the shared memory reduction, but uses Tempest's virtual channel mechanism to reduce communication overhead. Using *bulk-reduction* improves performance by 13% over *TSM-reduction*. This performance gain comes largely from reducing the number of messages by 60%. (The communication volume actually increases by 25% over *TSM-reduction*.) We do not use the

Table 4: DSMC results.

Version	Block Size (bytes)	Time (seconds)	Speedup
CHAOS		86	19.16
TSM	1024	97	17.17
XSM	1024	79	20.89

more optimized direct reduction protocol from *unstructured* for *moldyn*, because the force computation is very computation intensive and, although 58% of the interactions involve molecules on different processors, communication overhead is only a small fraction of the total time.

5.2.3 Improving Data Transfer

As in *unstructured*, our CHAOS implementation uses the highly optimized `gather` and `scatter` primitives from the CHAOS library. However, we must run the inspector every time the interaction list is rebuilt to determine the sharing pattern of the molecules. Like *unstructured*, our TSM implementation uses a large block size to reduce the number of stache misses. The interaction list is generated carefully to provide the necessary spatial locality. However, unlike *unstructured*, the XSM implementation uses a simpler reduction protocol (*bulk-reduction*) that reduces the number of messages communicated in *TSM-reduction* substantially. We do not use an update protocol for the molecules, because the number of misses from molecule reads is very small relative to the size of the interaction list. The simple reduction protocol and absence of an update protocol in XSM avoids any of the preprocessing that is necessary in the CHAOS implementation.

In summary, as in *unstructured*, properly partitioning the data and efficiently implementing reductions were crucial to achieving good performance. In addition, replacing the global lock protecting the interaction list with a more distributed computation was also important.

5.3 DSMC

DSMC studies properties of a gas by simulating the movement and collision of a large number of particles in a three-dimensional domain with a direct simulation Monte Carlo method [26]. *DSMC* divides the domain into *cells* in a static Cartesian grid. Each cell contains particles, which collide only with other particles in the cell. Particles enter the domain through either a jet-stream or the sides of the domain and leave through the sides of the domain. Each particle has associated physical quantities, such as velocity, rotational energy, and position, that change over time. The list of particles is stored in compressed sparse row format. Each cell contains its starting index in the particle list and the number of particles assigned to it.

The *DSMC* computation is divided into three distinct phases: collision, move, and index. The collision phase performs collisions between pairs of randomly-chosen particles in each cell. The move phase assigns particles to cells based on their coordinates and brings in new particles through the jet stream and sides of the domain. And the index phase reconstructs the cell-to-particle mapping based on the cell assignment computed for the particles in the move phase. Table 4 shows the results for our implementations of *DSMC*.

5.3.1 Partitioning

All three of our implementations (CHAOS, TSM, and XSM) block partition the cells among the processors. We partition along the x-dimension first, because more than 65% of the moves from a cell are along the x-direction (parallel to the jet-stream flow). This partition induces an initial partition for the molecules: a molecule is assigned to the processor that owns the cell that holds it. As a particle moves between cells, it is reassigned as necessary.

5.3.2 Mutual Exclusion

Since particles can move across processors, some form of mutual exclusion is needed to synchronize updates to the cell data structures (for example, the number of particles in a cell). We avoid locks because they are expensive and cause false sharing. Instead, all three of our implementations use a late-commit model.

CHAOS uses a primitive called *scatter_append* that appends to a cell the particles that move into it. During the move phase, CHAOS accumulates changes to the state of the particles locally. At the end of the move phase, CHAOS determines which particles move to other processors and sends them with the *scatter_append* primitive.

Our TSM and XSM implementations exploit the observation that most particles move to one of a processor's four neighbors. Therefore, we first record particles that move from a processor in a local stack—one for each of four neighboring processors. Then, in four phases each processor writes to dedicated receive buffers on the neighboring processors. In the TSM version, the four phases are demarcated by barriers. To handle the infrequent case of a particle moving to some other processor, we lock that processor's shared receive buffer and perform a write. In the XSM version we replaced the remote writes with active messages, one message per particle, that performed the write directly on the receiver. This allowed us to remove the barriers and perform all writes simultaneously, since writes to the same receive buffer by different processors are synchronized through the active message handlers, which execute atomically with respect to other handlers.

5.3.3 Improving Data Transfer

Our CHAOS implementation uses the highly optimized *scatter_append* primitive from the CHAOS library. Our TSM version uses large stache blocks (1024 bytes), which reduces the number of stache misses by 94% over 32-byte blocks. Large stache blocks improve performance because they allow the dedicated receive buffer to stay with the writing processor for the entire duration of its batched write. Finally, our XSM implementation replaces writes to the dedicated receive buffer with direct sends using Tempest active messages (one per particle), which improves performance over the TSM version by 22%. This optimization reduces the volume of message traffic dramatically (by 88%) at the cost of an increase in messages (roughly 73%).

In summary, carefully partitioning the data and using a late-commit model achieved good performance for DSMC. In addition, using Tempest's mechanisms to replace a request-reply protocol with direct messages helped to improve performance over our TSM implementation.[1]

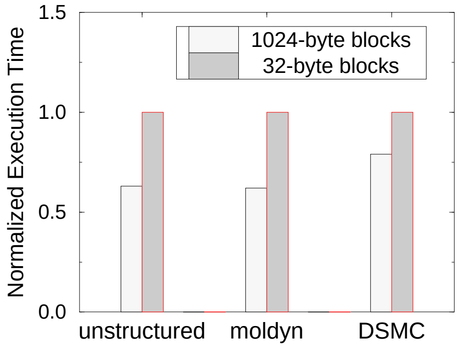

Figure 6: Effect of block size for the best TSM versions. The vertical axis denotes execution time divided by the execution time for 32-byte blocks.

6 Discussion

This paper examines three approaches to programming irregular applications on a message-passing computer (a CM-5). CHAOS is a library designed to support this type of application. Transparent shared memory (TSM), running on Tempest, uses a fixed coherence protocol to provide an application with a shared address space and cache coherence. Finally, extensible shared memory (XSM) uses Tempest mechanisms to improve the communication of important data structures.

The paper focuses on three issues—partitioning, mutual exclusion, and data transfer—that are crucial to achieving good performance for irregular applications. Not surprisingly all three approaches use similar techniques to improve performance. Most important is to partition data and loop iterations carefully to balance computational load and decrease communication. Because of processors' distinct address spaces, CHAOS must divide, in advance, all data and loop iterations. With TSM and XSM, complete data partitioning is not always necessary. Instead, a programmer's effort can be focused on the computationally-important data structures, while everything else is left in transparent shared memory.

Both CHAOS and the tuned TSM and XSM programs avoid protecting updates of global data with mutual exclusion by accumulating changes locally and globally reducing the partial values in a separate phase. This is particularly important for TSM and XSM, in which locks are expensive and often introduce problems with false sharing.

The TSM programs improve communication by using a large block size (1024 bytes versus 32 bytes) to reduce the number of stache misses. This optimization is only practical if false sharing does not increase, which requires carefully partitioning of data and loop iterations to increase spatial locality and requires replacing locks with reductions. Figure 6 shows that the 1024-byte block implementations are 27% – 61% faster than the corresponding 32-byte block ones.

The CHAOS and XSM programs improve data transfer through preprocessing, software caching, communication vectorization, and bulk transfer. The CHAOS programs rely on an inspector to determine a program's sharing patterns, which are used by the library primitives (`gather`, `scatter`, and `scatter-append`) to combine messages and eliminate unnecessary communication. In XSM, communication patterns can be found, if necessary, either by an inspector or a modified version of the TSM protocol.

In addition to these three issues, two other issues—address space management and buffer management—are important for supporting these applications, but are less specific to an application than to a system.

[1] Moon and Saltz's dynamic load balancing technique [16] improved upon the performance of the CHAOS implementation reported in Table 4 by 16%. We did not use that load balancing technique in any of our *DSMC* implementations due to insufficient time.

CHAOS implements a global address space by distributing arrays and modifying translations of array elements to support irregular data distributions. The translation table is the same size as a distributed array. Tempest, on the other hand, provides a shared address space by using virtual memory hardware and maintains translations at page granularity (typically, 4KB). Thus, both TSM and XSM require less space for translations and, in most cases, do not need explicit code in the program to establish or use translations.

Page-granularity translations, however, increase memory overhead if some storage on a page is not in active use. CHAOS does not introduce this concern, at it stores remote data compactly in a processor's memory by renaming (changing its virtual address) data from other processors. Remote array elements are stored after the local part of a distributed array. TSM and XSM programs use Tempest to allocate data in shared memory at the same virtual address on all processors. Tempest allocates an entire page if any block on that page is accessed. Hence, some blocks on an allocated page may be unused. In the three applications, the percentage of allocated, but not accessed, blocks is low to moderate—5% for *moldyn*, 40% for *DSMC*, and 51% for *unstructured* for their best XSM versions with 32-byte blocks.[2]

7 Conclusions

This paper examines three irregular applications—*unstructured*, *moldyn*, and *DSMC*—run using three software systems—CHAOS, TSM, and XSM—on a common hardware base—a 32-processor Thinking Machines CM-5. *CHAOS* is a library designed to support irregular applications on message-passing machines. *Transparent shared memory (TSM)* is fine-grain distributed shared memory using a fixed cache-coherence protocol implemented with Tempest. *Extensible shared memory (XSM)* extends TSM by allowing a program to use message-passing and custom protocols to improve the communication of crucial data structures.

After extensive performance tuning of programs on all three systems, experiments showed that CHAOS and XSM performed best, with TSM trailing respectably on two of the three applications. CHAOS achieved 32-processor speedups of 20.4, 23.1, and 19.2 for *unstructured*, *moldyn*, and *DSMC*, respectively. TSM achieved speedups of 1.9, 20.3, and 17.2, thus performing poorly on one application and competitively for the other two. XSM achieved speedups of 20.9, 23.1, and 20.9, which are similar to CHAOS's.

By fixing the applications and hardware, our experimental setup allowed us to evaluate the techniques each software system used to map program abstractions to the restricted world of message passing. The best performing programs shared similar techniques in three key respects. First, computation and data must be partitioned to ensure that data is local to the processor that uses it. Irregular applications required sophisticated partitioning since simple block or cyclic partitions failed to distribute the load adequately. Second, the common operation of reducing shared data is best done by exploiting an operator's associativity to perform reductions locally and then reduce the partial sums with a coordinated global update. Standard techniques for mutual exclusion, such as locks, did not achieve acceptable performance. Third, using message vectorization to reduce message overheads was critical. Programs that ignore communication considerations—such as naive TSM—perform poorly.

The three systems offer a disparate set of advantages that make direct comparison difficult. However, based on our experience, we believe that:

- CHAOS is an efficient and portable run-time library for irregular applications. Its main advantage is that it requires only clearly-defined modifications to source code. The CHAOS project is exploring ways to extend the library to a wider range of irregular applications [5].

- TSM performs well for applications whose natural partitions result in acceptable communication overhead (e.g., have good spatial and temporal shared data locality). TSM also supports any application in a straight-forward manner. However, achieving good performance with TSM can require significant programming effort to restructure a computation to improve data locality. TSM performance, moreover, is not robust and the performance bottlenecks can be obscure.

- XSM offers an attractive alternative to TSM. It offers the possibility of robust performance optimization that requires modest changes to TSM programs. These changes can often be encapsulated in libraries. Nevertheless, developing a new protocol can require considerable effort to understand a program's communication bottlenecks.

Although this paper considered hand-written applications, a more important use of these systems may be as a compiler run-time system. CHAOS is already being used in this role [8].

[2] We use 32-byte blocks to collect these statistics instead of the expected 1024-byte blocks, because larger blocks sizes tend to hide unused portions of a page.

Acknowledgements

We are indebted to the members of the Wisconsin Wind Tunnel[3] and CHAOS[4] projects for help with Blizzard, CHAOS, and applications. In particular, we would like to thank Steve Reinhardt for help with the Tempest interface, Bongki Moon for help with DSMC, and Madhusudan Talluri for suggesting a way to remove some locks in *unstructured*. We would also like to thank Satish Chandra, Sashikanth Chandrashekaran, Douglas Clark, and Mukund Raghavachari for helpful comments on this paper.

References

[1] J.E. Barnes and P. Hut. A Hierarchical O(N log N) Force Calculation Algorithm. *Nature*, 324(4):446–449, December 1986.

[2] M. J. Berger and S. H. Bokhari. A Partitioning Strategy for PDEs across Multiprocessors. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[3] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D . J. States, S. Swamintathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculation. *Journal of Computational Chemistry*, 4(187), 1983.

[4] Soumen Chakrabarti and Katherine Yelick. Implementing an Irregular Application on a Distributed Memory Multiprocessor. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 169–178, May 1993.

[5] Chialin Chang, Alan Sussman, and Joel Saltz. Support for Distributed Dynamic Data Structures in C++. Technical Report CS-TR-3416 and UMIACS-TR-95-19, University of Maryland, College Park, January 1995.

[6] Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[7] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[8] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing Analysis and Indirect Accesses to Distributed Arrays. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, chapter e. To appear, August 1993.

[9] 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.

[10] 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.

[11] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[12] Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[13] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

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

[15] Ravi Mirchandaney, Seema Hiranandani, and Ajay Sethi. Improving the Performance of DSM Systems via Compiler Involvement. In *Proceedings of Supercomputing '94*, pages 763–772, 1994.

[16] Bongki Moon and Joel Saltz. Adaptive Runtime Support for Direct Simulation Monte Carlo Methods on Distributed Memory Architectures. In *Scalable High Performance Computing Conference (SHPCC '94)*, pages 176–183, May 1994.

[17] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Mat. Anal. Appl.*, 11:430–452, June 1990.

[18] Steven K. Reinhardt. Tempest Interface Specification (Revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.

[19] 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.

[20] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and Run-time Compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

[21] Joel Saltz, Ravi Ponnusamy, Shamik D. Sharma, Bongki Moon, Yuan-Shin Hwang, Mustafa Uysal, and Raja Das. A Manual for the CHAOS Runtime Library. Technical Report 3437, Computer Science Department, University of Maryland, College Park, March 1995.

[22] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[23] 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.

[24] 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.

[25] Karen A. Tomko and Santosh G. Abraham. Data and Program Restructuring of Irregular Applications for Cache-Coherent Multiprocessors. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 214–225, 1994.

[26] Richard G. Wilmoth. Direct Simulation Monte Carlo Analysis of Rarefied Flows on Parallel Processors. *AIAA Journal of Thermophysics and Heat Transfer*, 5(3):292–300, July-Sept 1991.

[3] URL `http://www.cs.wisc.edu/~wwt`

[4] URL `http://www.cs.umd.edu/projects/hpsl.html`