*Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing (LCPC'98), Chapel Hill, NC, August 1998.*1

Improving Compiler and Run-Time Support for Irregular Reductions

Hwansoo Han, Chau-Wen Tseng

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Compilers for distributed-memory multiprocessors parallelize irregular reductions either by generating calls to sophisticated run-time systems or relying on the shared-memory interface supported by software DSMs. Run-time systems gather/scatter nonlocal results (e.g., CHAOS, PILAR) while software DSMs apply local reductions to replicated buffers (e.g., CVM, TreadMarks). We introduce LOCALWRITE*, a new technique for parallelizing irregular reductions based on the* owner-computes *rule. It eliminates the need for buffers or synchronized writes, but may replicate computation. We investigate the impact of* connectivity *(node/edge ratio),* locality *(accesses to local data) and* adaptivity *(edge modifications) on their relative performance.* LOCALWRITE *improves performance by 50–150% compared to using replicated buffers. Gather/scatter using CHAOS generally provides the best performance, but* LOCALWRITE *can outperform CHAOS for applications with low locality or high adaptivity. We also discover the* flush-update *coherence protocol can improve performance by 15–25% for software DSMs over an invalidate protocol.*

1 Introduction

Science is being transformed by the emergence of computation as a vital research tool. As scientists and engineers begin to tackle larger and more complex problems, they are increasingly beginning to exploit parallel computation to provide the computing power they need for research and development. Compilers can make the benefits of parallelism easier to obtain by efficiently mapping applications onto the complexities of high-performance architectures. Experiences show compilers for software DSMs running on distributed-memory multiprocessors can be quite efficient for dense-matrix computations with regular access patterns [8, 14]. However, the ability of compilers to efficiently support scientific applications with irregular access patterns is less established.

Irregular data accesses are performed by scientific applications in order to solve increasingly complex problems. The core of these applications is frequently comprised of *reductions*, associative computations (e.g., SUM, MAX) which may be reordered and parallelized [26]. These irregular computations are frequently run on distributed-memory multiprocessors due to their scalability and performance.

Researchers have previously proposed several methods to provide support for irregular reductions on distributed-memory multiprocessors. One approach, taken by distributed-memory compilers which generate explicit interprocessor messages (e.g., Fortran D [13], HPF [21]), is to rely on sophisticated run-time systems (e.g., CHAOS [5], PILAR [22]) which can identify and gather nonlocal data. A second approach is to combine shared-memory compilers (e.g., SUIF [8]) with software distributed-shared-memory (DSM) systems (e.g., TreadMarks [24], CVM [18]), which provide a shared-memory interface. Software DSMs are less efficient than explicit messages, but are much simpler compilation targets [4, 20].

In this paper, we introduce LOCALWRITE, a new compiler and run-time parallelization technique which can improve performance for certain classes of irregular reductions. We evaluate and compare the performance of different approaches to supporting irregular computations as a function of how densely the data in the underlying application is connected, how much nonlocal access is required, as well as how frequently the connection pattern changes during the course of the computation. These parameters are important in identifying areas in which software DSMs can match or even exceed the efficiency of explicit messages. Experiments are conducted in a prototype system [9, 20] using the CVM [17] software distributed-shared-memory (DSM) as a compilation target for the SUIF [8] shared-memory compiler. Our paper makes the following contributions:

This research was supported by NSF CAREER Development Award #ASC9625531 in New Technologies. The IBM SP-2 and DEC Alpha Cluster were provided by NSF CISE Institutional Infrastructure Award #CDA9401151 and grants from IBM and DEC.

- develop and evaluate LocalWrite, a new compiler and run-time technique for parallelizing irregular reductions based on the owner-computes rule
- evaluate the impact of *connectivity*, *locality*, and *adaptivity* on techniques for parallelizing irregular reductions on both shared and distributed-memory multiprocessors
- measure improvements from general-purpose coherence protocols for irregular reductions on software DSMs

The remainder of the paper begins with background material on existing parallelization techniques for reductions, followed by a description of LocalWrite. We outline our prototype compilation and run-time system, then present experimental results obtained by varying parameters for a number of representative irregular applications. We conclude with a discussion of related work.

2 Background

2.1 Irregular Reductions

We begin by examining the example irregular reduction shown in Figure 1. The computation loops over the edges of an irregular graph, computes a value, and applies it to both endpoints of the edge. The process repeats for many time steps. Occasionally the edges in the graph are modified. The computation is irregular because accesses to array `y` are determined by the *index arrays* `idx1` and `idx2`, preventing the compiler from analyzing accesses exactly at compile time.

The example also demonstrates two important features of scientific applications. First, most computations are *iterative*, where the same code is executed many times. The example code is iterative since the main computation is inside a *time-step* loop `t` with many repetitions. The number of time steps executed is a function of the application, but is usually quite large. Iterative computations are a boon to software DSMs, which can take advantage of repeated communication patterns to predict prefetches to nonlocal data [20, 30].

Second, many irregular scientific computations are *adaptive*, where the data access pattern may change over time as the computation adapts to data. The example in Figure 1 is adaptive because condition `change` may be satisfied on some iterations of the time-step loop, modifying elements of the index array `idx1` and changing overall data access patterns as a result.

2.1.1 GatherScatter

Compilers currently parallelize irregular reductions using either of two approaches, GatherScatter or ReplicateBufs, depending on the target architecture. We start by examining GatherScatter. Data-parallel compilers for

```
x[nodes],y[nodes]    // data in nodes
do t =               // time-step loop
  if (change)        // change accesses
    idx1[] =
  do i = 1, edges    // work on edges
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])  // computation
    y[n] += force          // update edge
    y[m] += -force         //   endpoints
```

Figure 1. Irregular Reduction Example

```
x[nodes+buf1],y[nodes+buf2]
inspect(idx1,idx2)       // inspector
localize(idx1,idx2)      // translate addrs
do t =
  if (change)            // adaptive code
    idx1[] =
    inspect(idx1,idx2)   // inspector
    localize(idx1,idx2)  // translate addrs

                         // executor
  gather(x)              // get nonlocal data
  do i = my_edges        // local computation
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])  // computation
    y[n] += force          // update local
    y[m] += -force
  scatter_with_add(y)    // update nonlocal
```

Figure 2. GatherScatter Example

distributed-memory multiprocessors (e.g., IBM SP-2, network of PCs) can generate explicit messages between processors [14]. Reductions with regular accesses can be converted directly to collective communication [23].

Irregular reductions may be parallelized by generating an *inspector* to identify nonlocal data needed by each processor. The inspector also generates a *communication schedule* and performs *address translation*, modifying indices of nonlocal data to use local buffers [5, 23, 28]. Inspectors are expensive, but their cost can be amortized over many time steps. On each time step an executor *gathers* nonlocal data using the communication schedule, performs the computation using local buffers, and *scatters* nonlocal results to the appropriate processors. This approach was pioneered by the CHAOS run-time system [5].

An example of GatherScatter is displayed in Figure 2. The inspector is invoked outside the time-step loop `t`, along with a routine *localize*() to convert nonlocal indices in `idx1` and `idx2` into local buffer indices. Inside the time-step loop, *gather*() is first invoked to actually collect the nonlocal data. The computation then uses local buffer indices stored in idx1 and idx2. Each processor only performs a portion of the reduction. Results of local reductions are then accumulated globally with the *scatter_with_add*() CHAOS library routine.

GATHERSCATTER is very precise, but incurs significant overhead caused by the inspector and mapping nonlocal indices into local buffers (i.e., address translation). The cost of the inspector is proportional to the number of edges and may be amortized for iterative computations [5]. If the computation is adaptive, the inspector must be re-executed each time the access pattern changes, as shown in Figure 2, because communication and mapping may need to change as well. Cost of the inspector is thus more important for adaptive computations because it cannot be amortized over as many iterations. In addition, the executor must perform communication and address translation for nonlocal data; this overhead is incurred each time and is proportional to the number of nonlocal accesses.

2.1.2 ReplicateBufs

Compilers for shared-memory multiprocessors (e.g., DEC Sable, SGI Origin2000) also detect parallelism and partition computation between processors. Shared-memory compilers parallelize irregular reductions by having each processor compute a portion of the reduction, storing results in a local *replicated buffer* the same size as the array holding reduction results. Results from all replicated buffers are then combined with the original global data, using synchronization to ensure mutual exclusion [8, 26]. If large replicated buffers are to be combined, the compiler can avoid serialization by directing the run-time system to perform global accumulations in sections using a pipelined, round-robbin algorithm [8]. REPLICATEBUFS works well when the result of the reduction is to a scalar value, but is less efficient when the reduction is to an array, since the entire array is replicated.

An example of REPLICATEBUFS is displayed in Figure 3. The buffer ybuf is used to store partial reductions results for y. Each processor only performs a portion of the overall computation, but since the compiler does not know which elements of y will be accessed, it must make the buffer as large as the original array. Not only does this waste memory, it increases overhead because the entire buffer must be first initialized, then used to update the actual array at the end with *reduce_sum*(), even if not all elements of the buffer are used during the reduction. As a result, the computation is on the order of MAX((O(nodes),O(edges/procs)) instead of O(edges/procs).

```
x[nodes],y[nodes],ybuf[nodes]
do t =
  if (change)
    idx1[] =
  ybuf[] = 0            // initalize buffer
  do i = 1, my_edges    // local computation
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])
    ybuf[n] += force    // updates stored in
    ybuf[m] += -force   //   replicated ybuf
  reduce_sum(y, ybuf)   // combine buffers
```

Figure 3. REPLICATEBUFS Example

2.2 Compiling for Software DSMs

One approach for eliminating the overhead of mapping nonlocal data on distributed-memory multiprocessors is to combine shared-memory parallelizing compilers with software distributed-shared-memory (DSM) systems which emulate global address space in software. Relying on software DSMs simplifies compilation, but is inherently less efficient than sending explicit messages.

Compilers for software DSMs have adopted a number of techniques for improving efficiency. One approach relies on precise communication analysis to insert explicit messages when analysis exactly identifies interprocessor communication [3, 4, 6]. For irregular computations, the compiler can prefetch index array sections to reduce latency [24].

A second approach exploits customized coherence protocols for reductions and nonlocal updates [7, 20, 25, 30]. Less precise communication analysis is required, since the compiler does not need to identify the sending and receiving processors. Instead, the compiler relies on the observation that most scientific applications perform iterative computations. Compilers can thus utilize access histories from previous time steps to guide communication and aggregate messages to the same processor, reducing latency and overhead [18, 20, 30]. In addition, compilers can also eliminate unnecessary synchronization based on communication analysis [10, 20].

For both approaches, experiments show performance is comparable to message-passing codes for regular applications, and better than current commercial distributed-memory compilers for irregular applications [4, 20, 24].

2.2.1 TreadMarks Improvements

TreadMarks is one of the most efficient software DSMs currently available. It relies on an invalidate coherence protocol based on lazy release consistency (LRC). Treadmarks parallelizes irregular reductions using REPLICATEBUFS. Performance is improved relying on a combination of compile-time and run-time support [24].

First, the compiler identifies the section of index array used by each processor, then prefetches it to reduce latency by inserting a call to the run-time function *validate*(). At run time, the contents of the index array are analyzed and nonlocal data requests are aggregated. Reduction results are stored in a local buffer, then globally updated in parallel in a pipelined manner. Experiments show performance on an

SP-2 is comparable to that achieved by CHAOS [5], which supports irregular reductions using GATHERSCATTER [24].

3 Improving Irregular Reductions

Now that we have seen how irregular reductions are parallelized by existing approaches, we consider two methods for improving compiler and run-time support in our combined SUIF/CVM system: special coherence protocols, and LOCALWRITE, a new compiler and run-time technique for parallelizing irregular reductions.

3.1 Protocol Improvements

Irregular reductions are identified by the SUIF compiler and parallelized using algorithm REPLICATEBUFS [8]. Instead of compiler analysis to insert prefetches to index arrays as in TreadMarks, CVM can use a *flush-update* protocol to automatically update any nonlocal data accessed, without explicitly examining the index array [20]. Under this protocol, CVM keeps track of shared data identified by the compiler. Access histories keep track of data requested by other processors on the first time step, then automatically sends updates to those processors whenever the data is modified. Accesses to updated nonlocal data thus do not pay the round-trip fetch latency experienced in invalidate-based protocols. Updates are not sent immediately, but are flushed at barriers, allowing updates to the same processor to be automatically aggregated and communicated in a single message to reduce overhead. The flush-update protocol was first developed for scientific computations with regular access patterns, but also works for irregular reductions.

More recently, CVM has been improved in two additional areas. First, the underlying coherence protocol was modified to use an *automatic update release consistency* (AURC) model, since it has been shown to perform well when ownership of data between processors is static and stable [16]. In addition, an *overdrive* protocol which reduces OS overhead for page protections can be employed for iterative codes with stable communication patterns [18].

The basic CVM invalidate protocol deals easily with adaptive codes, since no assumptions are made about previous data access patterns. Complications may ensue for adaptive codes when using the flush-update or overdrive protocols [18], since these protocols rely on information from previous iterations of the time-step loop. We describe in Section 3.3 how compiler analysis is needed to decide when to use the new protocols for irregular codes.

3.2 LocalWrite

Two sources of inefficiency in REPLICATEBUFS are large local buffers and mutual exclusion during global accumulation of buffers. Eliminating these overhead provides the motivation for LOCALWRITE, a new technique for parallelizing irregular reductions. LOCALWRITE attempts to improve locality and reduce overhead for reductions by partitioning computation so that each processor only computes new values for locally-owned data. LOCALWRITE is an application of the *owner-computes* rule used in distributed-memory compilers [13]. By assigning only to local data, LOCALWRITE avoids the need for either buffers or mutual exclusion synchronization. The tradeoff is possibly replicated computation.

```
do t =
  do i = edges
    n = idx1[i]
    m = idx2[i]
    if (own(y[n],y[m]))       // check owner
      force = f(x[m], x[n])
    if (own(y[n]))            // check owner
      y[n] += force
    if (own(y[m]))            // check owner
      y[m] += -force
```

Figure 4. LOCALWRITE On-the-fly Example

```
inspect(idx1, idx2)  // calc my_edges
do t =
  do i = my_edges    // only local LHS
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])
    y[n] += force
    y[m] += -force
```

Figure 5. LOCALWRITE Inspector Example

3.2.1 Example

Consider how to apply LOCALWRITE to the irregular reduction example in Figure 1. There are two possible ways of implementing LOCALWRITE. One way is figuring out local ownership on the fly by testing the ownership of left-hand side (LHS) variables, as shown in Figure 4. It is a simple and easy way to generate code, but every processor would go through the whole iteration space testing ownership on every iteration. On-the-fly checking thus performs poorly as the iteration space gets larger. A better approach is to create an inspector version of LOCALWRITE which investigates ownership of reduction variables and stores the information once outside the time-step loop, as shown in Figure 5. We believe the inspector version of LOCALWRITE can thus be potentially applicable to wide range of applications. In the rest of the paper we use LOCALWRITE to indicate the inspector implementation.

LOCALWRITE consists of two parts: selection of local iterations and execution of local iterations. Compiler analysis can recognize irregular reductions and identify which array variables (`idx1`, `idx2`) are used as index arrays [24],

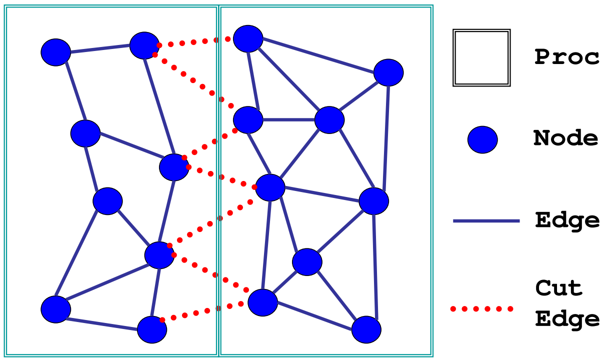

Figure 6. Cut Edges Example

```
inspect(idx1, idx2)  // calc my_edges
do t =
  do i = my_edges    // if some local LHS
    n = idx1[i]
    m = idx2[i]
    force = f(x[m], x[n])  // replicated
    if (own(y[n])
      y[n] += force
    if (own(y[m])
      y[m] += -force
```

Figure 7. LOCALWRITE Cut Edge Example

as well as which part of each index array is accessed by a processor at each point of the program [10, 13, 29]. Based on this information, we insert calls to inspectors to select a partial iteration space where each iteration only writes to the local portion of each variable. The inspector then examines the values of index arrays at run time to build a list of loop iterations which modify local data.

3.2.2 Cut Edges

The code generated to implement LOCALWRITE is complicated by the presence of *cut edges*, edges whose endpoints are on different processors. Figure 6 shows examples of cut edges for a sparse computation whose nodes are partitioned between two processors. Given a good partition, most edges are between nodes assigned to the same processor. However, some edges cross processor partitions. We call these cut edges.

For many scientific computations, such as the example code in Figure 1, the computation needs to update both endpoints of an edge. For edges where both endpoints are on the same processor, the iteration is simply assigned to that processor. However, iterations computing values for cut edges would then need to update data on two processors. To ensure each processor only updates the local portions of an array, LOCALWRITE will need to replicate the computation for the iteration on both processors. In comparison, GATHERSCATTER and REPLICATEBUFS would perform the iteration on one processor, and combine results with updates at the end of the reduction.

An example of computation replication in LOCALWRITE is shown in Figure 7. The inspector must assign to a processor any iteration which writes to data owned by a processor. Iterations representing cut edges are then assigned to two processors. When the iteration for the cut edge is executed, both processors perform the force computation, then use the result to update local data. In Figure 7 we use guarded assignment statements for simplicity. In our experiments we actually create a separate copy of the loop for cut edges, in order to avoid computing guards at run time. The inspector for LOCALWRITE must then generate two computation partitions, one for local edges and one for cut edges.

3.2.3 Comparisons

When should LOCALWRITE be applied? Previous reduction techniques require communication between processors at the end of a reduction to produce a global result. REPLICATEBUFS must combine values from a local buffer as large as the global array using mutual exclusion in *reduce_sum*(). GATHERSCATTER must combine values of nonlocal elements buffered locally with collective communication in *scatter_with_add*(). In contrast, LOCALWRITE avoids communication by directly updating the elements of global reduction variables owned by each processor.

LOCALWRITE has two sources of overhead, the inspector and replicated computation. During each iteration of the time-step loop, each processor executes only the necessary iteration space using the stored information calculated by the inspector. As with GATHERSCATTER, the cost of the inspector can be amortized over many iterations of the time-step loop. However, the inspector needed for LOCALWRITE is less expensive than the inspector needed for GATHERSCATTER. The difference is because LOCALWRITE only partitions computation, whereas the GATHERSCATTER inspector must also build a communication schedule and perform address translation.

As we have seen, depending on the computation performed LOCALWRITE may need to replicate computation for cute edges. Performance may degrade if the cost of replicated computation is so expensive as to overcome the benefit of reduced buffer and synchronization overhead. The performance of LOCALWRITE is thus sensitive to the ratio of replicated computation to saved communication.

LOCALWRITE faces one additional issue: load balancing. Both REPLICATEBUFS and GATHERSCATTER can partition computation evenly among processors by assigning equal numbers of edges to each processor. In contrast, LOCALWRITE assigns equal numbers of nodes to each processor. If connections between edges are not uniformly distributed, load imbalance may result if the numbers of edges assigned to each processor vary greatly. In particular, scientific applications may have access patterns which are clustered, as in astrophysical or molecular dynamics n-body computations. In such cases, LOCALWRITE will need to assign nodes to processors in a way which preserves load balance. An algorithm such as recursive coordinate bisection (RCB) may need to be applied, rather than simply using a simple block partition. Such an approach would reduce the efficiency of LOCALWRITE for adaptive computations. However, the overhead for LOCALWRITE should still be less than for GATHERSCATTER, because the LOCALWRITE inspector does not build communication schedules or perform address translation.

3.3 Compiler Analysis

Compiler analysis is required to exploit both improved CVM protocols and LOCALWRITE. In this section, we examine the motivation and analyses required.

3.3.1 Detecting Adaptive Computations

First, we discuss why detecting adaptive computations is important for both improved protocols and LOCALWRITE. When access patterns change, using a flush-update protocol may cause update messages to be continuously sent to processors which no longer require the data. Consider the example in Figure 1. When condition `change` is true, modifications to index arrays can affect both the pattern of nonlocal reads and writes. In the worst case, processors may send updated data to all other processors even though the data is not needed. This problem only affects performance, not legality, since unnecessary data is simply ignored. Realistically, access patterns do not tend to change so dramatically, so performance degradation may be minimal or even nonexistent.

The problem is more acute when employing the overdrive protocol. If processors begin accessing nonlocal data unexpectedly, out-of-date values may be retrieved or segmentation faults may occur, affecting program legality. Similarly, the need for change detection for LOCALWRITE is obvious. The compiler needs to detect changes in nonlocal access patterns so that it can insert inspectors to repartition loop iterations to ensure only local data is written. Otherwise the application will no longer produce correct results, because multiple processors may update the same array elements in an unsynchronized manner.

To avoid these problems, we implemented in SUIF compiler analysis to detect sparse adaptive computations. Analysis is fairly straightforward, and similar to a simplified form of *glacial* variable analysis [1]. We examine global arrays accessed in parallelized loops. Array subscripts containing index arrays are found and the index arrays are marked. We also record whether the index array is read or written. The compiler then examines the program to determine whether index arrays are modified within the time-step loop. If no index arrays are modified, the computation is not adaptive. Otherwise the computation is adaptive.

```
inspect(idx1, idx2)      // calc my_edges
do t =
  if (change)
    idx1[] =             // access changed
    inspect(idx1, idx2)  // recalc my_edges
  do i = my_edges
    ...
```

Figure 8. LOCALWRITE Adaptive Code Example

When the compiler identifies adaptive computations, the compiler can insert a warning to the software DSM (in the form of a library call) prior to code modifying index array. The software DSM can then exit overdrive mode, or decide to modify update sets for the flush-update protocol.

3.3.2 Generating Inspectors

For LOCALWRITE, compilers must insert inspectors to partition computation so that each processor only writes to local data. In addition, the compiler must determine when the pattern of writes to global arrays can change, inserting additional inspectors to recompute the computation partition. The location where index arrays are modified is also where new inspectors must be inserted, as shown in Figure 8.

Note that unlike for GATHERSCATTER, changes in the pattern of reads to nonlocal data do not require executing an inspector, since the computation partition is determined entirely by the pattern of writes. Information on when access patterns change is also useful for GATHERSCATTER, since an inspector must be executed to build a new communication schedule whenever the pattern of either writes or reads are changed [15].

Inspectors for LOCALWRITE are simpler than those for GATHERSCATTER, since the inspector only needs to partition computation. The inspector thus only needs to examine the LHS of global assignments in parallel computations, not all global references. Communication schedules and address translation tables are not needed.

Previous research on data-parallel compilers such as the Fortran D compiler have discussed compiler techniques for automatically generating inspectors and executors for CHAOS [11, 12, 15]. Simplified versions of those techniques should be directly applicable to generating inspectors and executors for LOCALWRITE. We are in the process of implementing automatic generation of inspectors in SUIF.

4 Analytic Comparison

We have now presented three techniques (REPLICATEBUFS, LOCALWRITE, and GATHERSCATTER) for parallelizing irregular reductions. We compare them analytically in this section before quantifying their performance experimentally in Section 5.

			Sensitivity to		
	Ease-of-use	Inspector	Connectivity	Locality	Adaptive
REPLICATEBUFS	High	None	High	Low	Low
LOCALWRITE	Med	Partial	Low	Med	Med
GATHERSCATTER	Low	Full	Med	High	High

Table 1. Algorithm Comparison

We believe all three techniques have advantages. REPLICATEBUFS and LOCALWRITE can be used by shared-memory compilers, whereas distributed-memory compilers will need to apply GATHERSCATTER. REPLICATEBUFS is the simplest to implement, while GATHERSCATTER requires both complex compiler and run-time support. The implementation effort required for LOCALWRITE is in between; we believe it is closer to REPLICATEBUFS than GATHERSCATTER.

4.1 Application Characteristics

In addition to implementation effort, deciding which technique to apply will also depend on expected performance. Because of the complexity of the parallelization process, we believe performance will be significantly affected by the application. In particular, application characteristics of connectivity, locality, and adaptivity are likely to affect performance.

We first consider *connectivity*. If we view irregular accesses as edges in a graph, we can classify the irregular accesses according to the ratio of number of edges to number of nodes. If the ratio of edges to nodes is high, we consider the graph *densely* connected. A low ratio indicates a *sparsely* connected graph. An alternative way of viewing connectivity is by the amount of reuse of data. Densely connected graphs will access each node more times than sparsely connected graphs. Since computation is proportional to the number of edges while communication is proportional to the number of nodes, densely connected graphs will also have a high computation to communication ratio.

A second factor impacting irregular reductions is *locality*, the proportion of data accesses which are to data located on the processor. Applications with high locality will generally possess fewer cut edges. In comparison, low locality will cause a large fraction of edges to access nonlocal data. Locality can be measured both by the percentage of nonlocal data accesses and the percentage of edges which are cut edges. In general, the two characteristics are correlated.

Another factor impacting techniques for supporting irregular reductions is *adaptivity*, the rate at which data access patterns change during program execution. *Static* applications which have fixed data access and communication patterns are fairly well understood. Many scientific computations, however, require the connections in the underlying sparse graph to be updated during the course of the computation. Previous studies involving CHAOS [15, 25] and TreadMarks [24] show irregular applications can achieve speedup for adaptive computations, but have not compared the performance of different approaches as a function of adaptivity.

4.1.1 Effect of Characteristics

Before we attempt to evaluate each parallelization technique for irregular reductions, we analyze the likely effect of application characteristics. We begin with REPLICATEBUFS. Its overheads are replicating buffers, which requires more memory, as well as initialization and accumulation. Buffer elements not modified still need to be processed when accumulating to global value. We thus expect REPLICATEBUFS to perform best for densely connected graphs, since the overhead of replicating buffers is amortized over many computations. For sparsely connected graphs unnecessary computation is performed. In the worst case, where the edge/node ratio is 1, no parallelism is exploited.

REPLICATEBUFS should perform well even for applications with low locality, since it must buffer the entire reduction array in any case. REPLICATEBUFS should perform well for applications with high adaptivity, because no inspectors need to be executed after changes. Immediately after a change in the connection, software DSMs will not be able to prefetch new data not previously accessed, paying a latency and overhead penalty. However, the adaptivity penalty is only incurred for the new data accessed, most likely a small percentage of the overall data set. In addition, the overhead is paid only on the first time step immediately following the change in connection pattern.

In comparison, LOCALWRITE should perform best for sparsely connected graphs with low connectivity, since it does not suffer either the wasted buffers of REPLICATEBUFS or the high inspector overhead for GATHERSCATTER. LOCALWRITE is sensitive to locality, since cut edges cause computation replication. It is also sensitive to adaptive computations, since inspectors need to be rerun. However, inspectors are only needed when the pattern of writes change, not reads. Inspectors are also inexpensive relative to those needed for GATHERSCATTER.

GATHERSCATTER should be only moderately sensitive to connectivity. High connectivity is needed only to provide computation to amortize inspector costs. GATHERSCATTER is sensitive to locality, because the cost of building communication schedules and performing address translation is proportional to the number of nonlocal accesses. Because inspectors are expensive and need to be updated even for changes in read accesses, GATHERSCATTER should be slowed the most by adaptive computations. These analyses are summarized in Table 1. We will see whether these intuitions hold in our experiments.

5 Experimental Evaluation

This section presents our experimental results. We discuss our experimental environment, present our overall results, and discuss the effect of our compiler optimizations.

5.1 Experimental Platform

We evaluated our optimizations on an IBM SP-2 with 66MHz RS/6000 Power2 processors operating AIX 4.1. Nodes are connected by a 120 Mbit/sec bi-directional Omega switch capable of a sustained bandwidth of approximately 40 Mbytes per second. We also evaluated the performance of irregular reductions on a DEC Sable multiprocessor with four 275MHz Alpha 21064 processors and 256 megabytes of memory, using Digital Unix V4.0. Codes were run in parallel using the *pthreads* shared-memory run-time library.

Our experiments were conducted using the SUIF compiler, CVM software DSM system, and CHAOS run-time library. We used SUIF to generate parallel pthreads programs (for a DEC Alpha multiprocessor) and parallel CVM programs (for the IBM SP-2). We also wrote explicitly parallel CHAOS programs (for the IBM SP-2). We used a version of CHAOS implemented on top of MPI.

SUIF is a optimizing and parallelizing compiler developed at Stanford [8]. It has been successful in finding parallelism in many standard scientific applications. It parallelizes reductions using REPLICATEBUFS. We have implemented automatic change detection in SUIF, but are still in the process of implementing automatic inspector generation. For our experiments, we currently generate inspectors for LOCALWRITE by modifying SUIF output by hand. CHAOS is a run-time library for supporting irregular applications [5]. It provides run-time support for parallelizing reductions using GATHERSCATTER.

CVM is a software DSM that supports coherent shared memory for multiple protocols and consistency models [17]. Its primary coherence protocol implements a multiple-writer version of *lazy release consistency*, which allows processors to delay making modifications to shared data visible to other processors until special synchronization [19]. SUIF was retargeted to generate code for CVM by providing a run-time interface based on CVM thread creation and synchronization primitives. Performance was improved by adding customized protocol support for reductions, as well as a *flush update* protocol that at barriers automatically sends updates to processors possessing copies of recently modified shared data [20].

5.2 Applications

We evaluated the performance of our compiler/software DSM interface with three sample applications, IRREG, NBF, and MOLDYN. Application parameters are presented in Table 2. The key computation in each application is shown in Figure 9. These applications contain an initialization section followed by the main computation enclosed in an sequential time-step loop. The main computation is thus repeated on each iteration of the time-step loop. Statistics and timings are collected after the initialization section and the first few iterations of the time-step loop, in order to more closely match steady-state execution.

5.2.1 Irreg

IRREG is representative of iterative partial differential equation (PDE) solvers found in computational fluid dynamics (CFD) applications. In such codes, unstructured meshes are used to model physical structures. The mesh is represented by nodes and edges. The main computation kernel iterates over the edges of the mesh, computing modifications to its end points. IRREG computes a force which is applied to both endpoints of an edge. Modifications to the value of all nodes is in the form an irregular reduction.

We tested versions of IRREG with 10000 nodes for 50 time steps. We generated uniformly distributed meshes with edge/node ratios of 1, 10, 50, and 100 to measure the impact of connectivity. Meshes were generated with different amounts of locality, with the percentage of cut edges varying from 10%, 30%, 50%, and 70%. We also tested adaptive versions of IRREG where where the edges were modified every 10, 20, and 40 time steps, as well a static version. Using LOCALWRITE will cause computation to be replicated for cut edges, introducing a source of inefficiency.

5.2.2 NBF

NBF is a kernel abstracted from the GROMOS molecular dynamics benchmark [11]. Instead of a list of edges as in IRREG, it maintains a list of partners for each molecule. Partner lists are more compact than edge lists, but are more fragmented and difficult to manage. NBF computes a force which is applied to both a molecule and its partner. Compared to IRREG, the force computation is more expensive. Modifications to every molecule is in the form of an irregular reduction.

We tested versions of NBF with 32000 molecules for 50 time steps. We set the number of partners for each molecule at at 1, 10, 50, and 100 to measure the impact of connectivity. Meshes were generated with different amounts of locality, with the percentage of cut edges varying from 10%, 30%, 50%, and 70%. Since molecules change position, the interaction list must be occasionally updated. We evaluated a static version of NBF as well as adaptive versions where the interaction list is recalculated every 10, 20,

	Description	Problem Size	Time Steps	Connectivity
IRREG	Irregular CFD Mesh	10000 Nodes	50	1–100
NBF	Non-Bonded Force (GROMOS)	32000 Nodes	50	1–100
MOLDYN	Molecular Dynamics (CHARMM)	16384 Molecules	50	6–124

Table 2. Sparse Scientific Applications

```
** Irreg
do t =
  do i = num_edges
    n1 = left[i]
    n2 = right[i]
    force = (x[n1]-x[n2])/4
    y[n1] += force
    y[n2] += -force
```

```
** NBF
do t =
  do i = num_atoms
    do p = num_partners
      j = partners[i,p]
      d = x[i]-x[j]
      force = d**(-6)/1000
      y[i] += force
      y[j] += -force
```

```
** Moldyn
do t =
  do i = num_interactions
    n1 = left[i]
    n2 = right[i]
    d = distance(x[n1],x[n2])
    if (d < cutoff)
      force = d**(-7) - d**(-4)/2
      y[n1] += force
      y[n2] += -force
```

Figure 9. Key Kernels of Applications

and 40 time steps. Using LOCALWRITE will cause computation to be replicated for cut edges, introducing a source of inefficiency.

5.2.3 Moldyn

MOLDYN is abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular systems. A list of interactions (edges) is maintained between pairs of molecules. Since the strength of interactions between molecules drops with increasing distance, only molecules within a cutoff distance of each other are assumed to interact. The main computation kernel iterates over all interactions between molecules, computing a single force which is applied to both interacting molecules. Modifications to every molecule is in the form of an irregular reduction.

We tested versions of MOLDYN with 16384 molecules for 50 time steps. Molecules are generated and distributed uniformly. We set cutoff distance for interactions at 1, 1.5, 2.5, and 3.5 Angstroms to measure the impact of connectivity. Lower cutoff distances produce fewer interactions between molecules, since molecules more distant than the cutoff are assumed not to interact. The cutoff distances resulted in edge/node ratios of 6, 9, 43, and 124, respectively. Unlike IRREG and NBF, we were unable to generate versions of MOLDYN with different locality directly. Instead, we generated versions based on either a block or recursive coordinate bisection (RCB) partition strategy. Block partition resulted in 72% cut edges, while RCB had 31% cut edges.

Since molecules change position, the interaction list must be occasionally updated. We evaluated a static version of MOLDYN as well as adaptive versions where the interaction list is recalculated every 10, 20, and 40 time steps. MOLDYN computes the same value for both endpoints of an edge. Using LOCALWRITE will thus cause computation to be replicated for cut edges, introducing a source of inefficiency.

MOLDYN is a larger and more complicated code than either IRREG or NBF. Its force calculation is more expensive. Additionally, the work required to update the interaction list is significantly more expensive than individual iterations of the time step loop. Sequential execution times for MOLDYN can double when updates are performed every ten time steps.

5.2.4 Baseline Version

For our experiments, we need a standard baseline version of each program to perform comparisons. We chose for the base version of each application connectivity of 100, locality of 30%, and adaptivity set to none. Connectivity of 100 represents a moderately dense graph, and is similar to the connectivity reported by other researchers [24]. Locality of 30% cut edges represents problems with good partitions, as calculated by algorithms such as RCB. For instance, both applications in Mukherjee *et al* achieved locality of around 30% with RCB [25]. We chose to use the static, non-adaptive version of each program as the baseline in order to separate issues of adaptivity, particularly the high cost of inspectors in GATHERSCATTER.

5.3 Shared-Memory Speedups

5.3.1 Impact of Connectivity

We begin by examining performance on a four-processor DEC Alpha multiprocessor. Interprocessor communication is relatively inexpensive due to shared memory, so we expect overhead to be the main factor impacting performance. Figure 10 presents 4 processor speedups for REPLICATEBUFS and LOCALWRITE as the connectivity (edges/node ra-

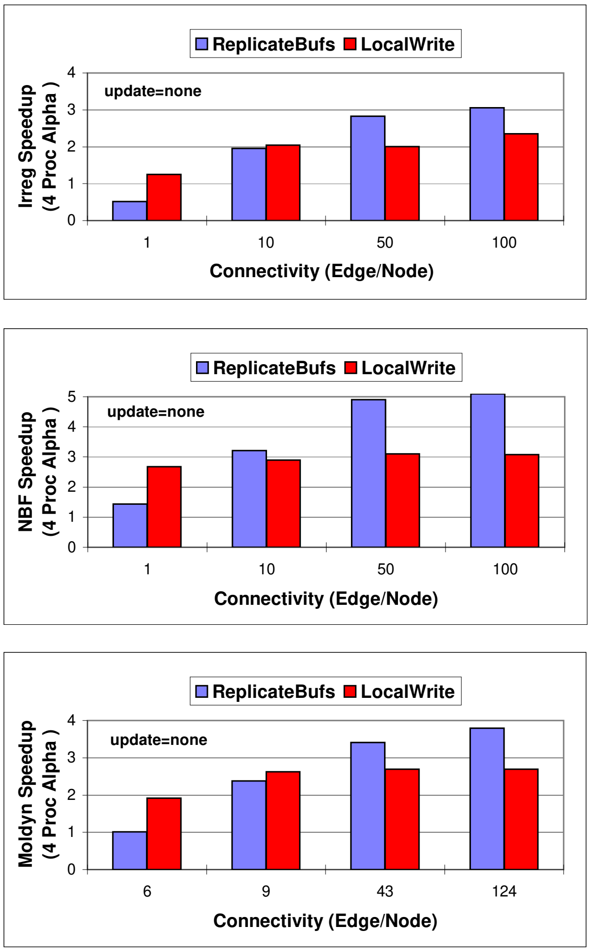

Figure 10. Speedups vs Connectivity (DEC Alpha)

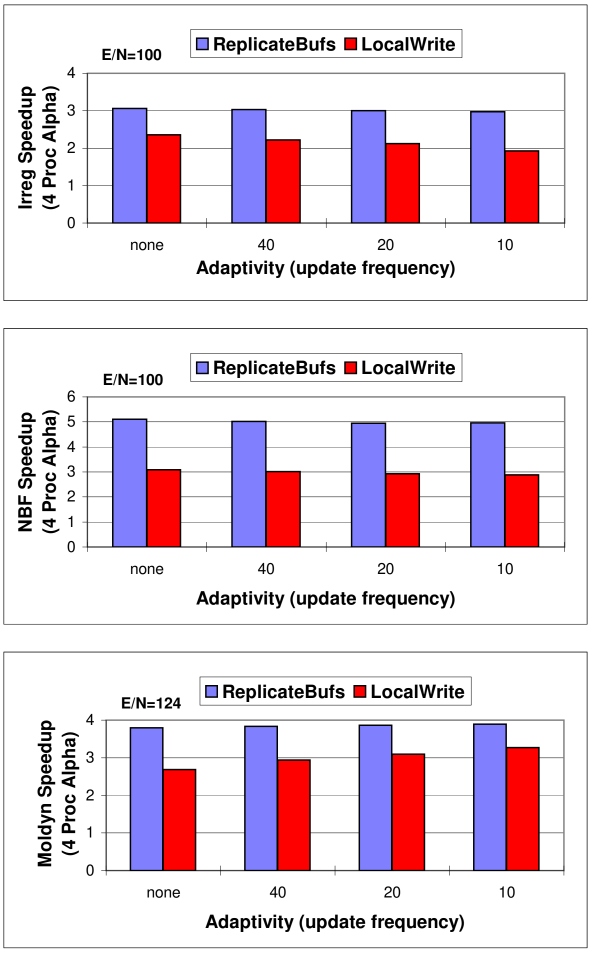

Figure 11. Speedups vs Adaptivity (DEC Alpha)

tio) varies while the adaptivity is held static (no updates). Speedups is measured along the y-axis. The x-axis displays connectivity measured as the edge/node ratio. Speedups for REPLICATEBUFS and LOCALWRITE, the two parallelization techniques, are presented as bars of different shades.

Since the amount of work is proportional to the number of edges, speedups generally increase as graphs become denser (i.e., the edge/node ratio increases), since we hold the number of nodes fixed. We see that LOCALWRITE enjoys an advantage for sparsely connected meshes. However, REPLICATEBUFS speedups improve more quickly, since more edges also increases the proportion of useful elements in replicated buffers. In comparison, LOCALWRITE pays a penalty for extra replicated computation for cut edges.

After the edge/node ratio passes 10 on a 4 processor Alpha multiprocessor, REPLICATEBUFS outperforms LOCALWRITE. It thus appears communication costs are low enough on shared-memory multiprocessors that replicating computation is not profitable unless the graph is sufficiently sparse that REPLICATEBUFS performs significant amounts of wasted work for replicated buffers.

5.3.2 Impact of Adaptivity

Figure 11 presents speedups for densely connected graphs as the adaptivity varys from no updates to updating the edges every 10 time steps. The graphs are structured as before, except the x-axis now represents adaptivity instead of connectivity. Results show REPLICATEBUFS speedups remain stable while LOCALWRITE degrades very slightly since only LOCALWRITE has to pay the overhead of reinspecting the connection pattern after changes. Stable speedups indicate adaptivity does not significantly affect performance on shared-memory multiprocessors.

5.4 Distributed-Memory Speedups

Performance results are quite different on the IBM SP-2, since communication is expensive for distributed-memory multiprocessors. We examine the performance of each techniques under different conditions. We use the flush-update protocol with both REPLICATEBUFS and LOCALWRITE by default.

5.4.1 Impact of Connectivity

Figure 12 presents 8 processor speedups for REPLICATEBUFS, LOCALWRITE, and GATHERSCATTER as connectivity varies for the static version of each application (no connectivity updates). Unlike on the Alpha multiprocessor, results are different for each application. The difference is due to the fact communication costs are high relative to a a shared-memory multiprocessor, so performance is more dependent on the balance between communication and computation.

We begin with IRREG. The cost of the force computation for IRREG is inexpensive, so replicating computation to reduce communication is a major win. We see LOCALWRITE achieves significantly better speedups than REPLICATEBUFS, since the cost of interprocessor communication to accumulate local results into global data is expensive even with customized reduction protocols. The performance of REPLICATEBUFS improves for denser graphs since unused buffer elements are reduced, but LOCALWRITE improves even more quickly.

In IRREG, GATHERSCATTER achieves better speedups than LOCALWRITE for less connected meshes due to its more efficient communication, but the gap narrows as connectivity increases since the computation/communication ratio increases and communication becomes less important. Performance improves for GATHERSCATTER as connectivity increases, probably because the communication/computation ratio is reduced. Speedups drop for GATHERSCATTER going from connectivity of 50 to 100, possibly because of large buffer requirements. In comparison, LOCALWRITE continues to improve with higher connectivity. As a result, when the edge/node ratio reaches 100 LOCALWRITE actually outperforms GATHERSCATTER for IRREG.

For NBF, results are similar. LOCALWRITE again achieves better speedups than REPLICATEBUFS. Overall REPLICATEBUFS speedups are higher, since the force computation is more expensive than in IRREG. GATHERSCATTER outperforms LOCALWRITE, though the disparity again narrows for denser graphs.

For MOLDYN, LOCALWRITE still achieves better speedups than REPLICATEBUFS, especially for sparse graphs. However, the force computation in MOLDYN is the most expensive of all, so the difference between LOCALWRITE and REPLICATEBUFS is reduced. Once again, GATHERSCATTER achieves the best performance for sparse graphs due to its precise messages, though the difference decreases as connectivity increases.

Overall, we see that LOCALWRITE always outperforms REPLICATEBUFS for distributed-memory multiprocessors. GATHERSCATTER generally achieves the highest speedups because of its more efficient communication. However, as connectivity rises, increasing inefficiencies in GATHERSCATTER allows LOCALWRITE to achieve comparable or even better performance.

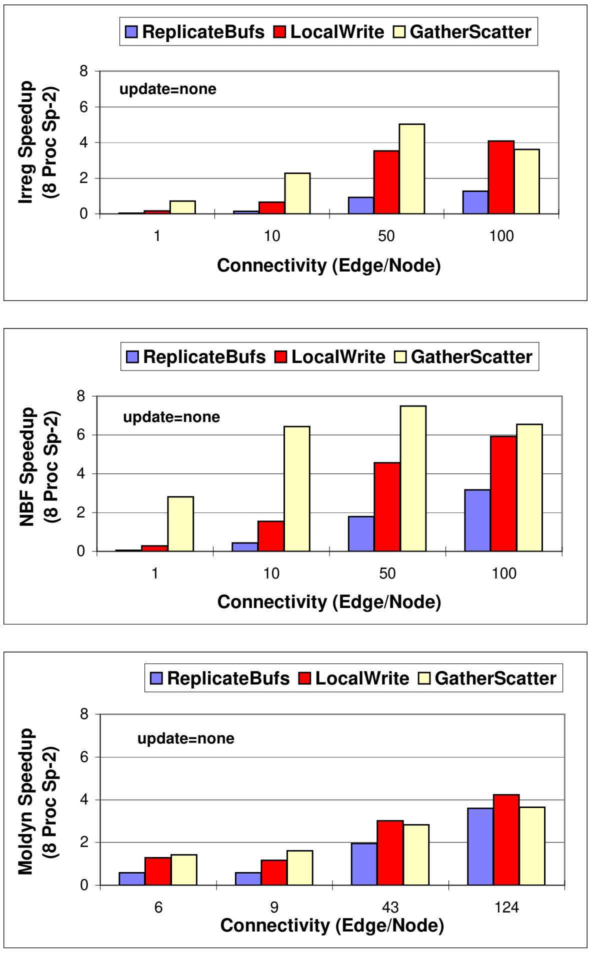

Figure 12. Speedups vs Connectivity (IBM SP-2)

5.4.2 Impact of Locality

Figure 13 presents 8 processor speedups for REPLICATEBUFS, LOCALWRITE, and GATHERSCATTER as locality varies for each application. Locality, measured as the percentage of cut edges, is measured along the y-axis. Recall we vary locality in IRREG and NBF by changing the initial mesh, whereas for MOLDYN we change the molecule partition from block-wise to recursive coordinate bisection (RCB).

We see that the performance of REPLICATEBUFS is not significantly affected by locality in any application, and performance for LOCALWRITE only declines slightly. We believe speedup for LOCALWRITE turns out to be relatively insensitive to locality because coherence in CVM is maintained at page level (8K), and communication is aggregated and sent as individual messages by the flush-update protocol. As a result, the total amount of data communicated is less important unless it varies significantly. For IRREG and NBF the total amount of data communicated does not appear to vary sufficiently to cause a significant change in performance. The impact of computation replication for cut

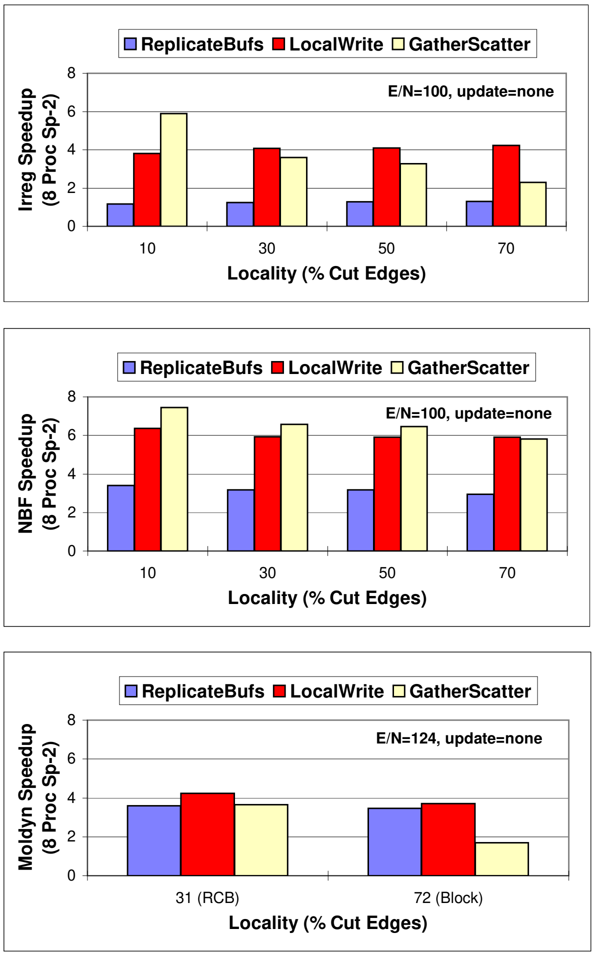

Figure 13. Speedups vs Locality (IBM SP-2)

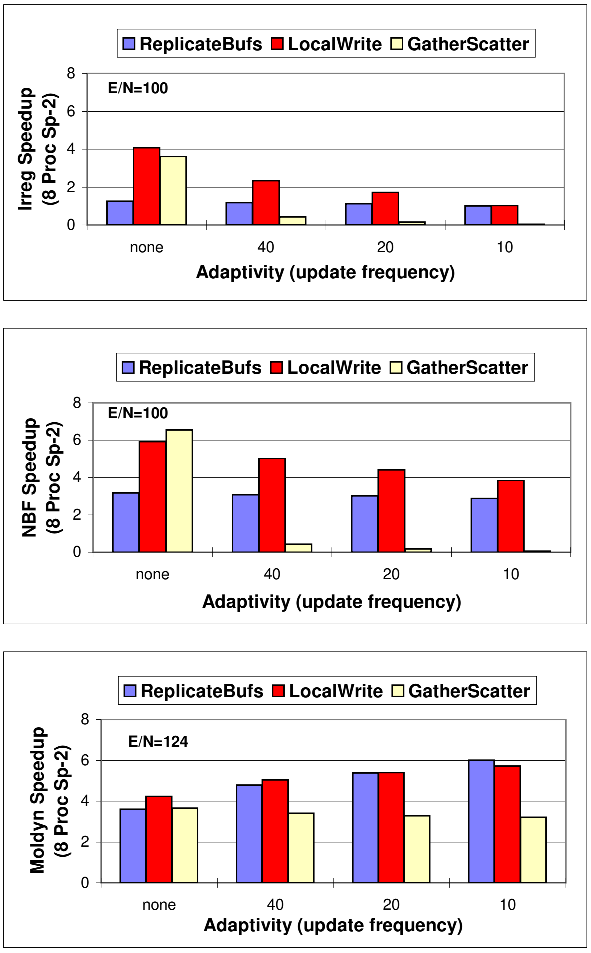

Figure 14. Speedups vs Adaptivity (IBM SP-2)

edges on LOCALWRITE also appears to be less than feared, and does not appear to be a significant source of overhead in IRREG or NBF.

In comparison, we see that the speedups of GATHERSCATTER drop significantly as locality decreases and more cut edges appear. When cut edges are around 30%, GATHERSCATTER speedups are comparable to LOCALWRITE. For IRREG and MOLDYN, GATHERSCATTER performance degrades significantly when cut edges reach 70%. The decrease is less for NBF, possibly because GATHERSCATTER is more efficient for partner lists than edge lists. As discussed previously, work performed by GATHERSCATTER increases in proportion to the number of nonlocal nodes accessed. Experimentally we find it to be more sensitive to locality than either REPLICATEBUFS or LOCALWRITE.

5.4.3 Impact of Adaptivity

Figure 14 presents 8 processor SP-2 speedups for REPLICATEBUFS, LOCALWRITE, and GATHERSCATTER as adaptivity varies for each application. Adaptivity, measured as the number of time-step iterations between changes in the graph, is displayed along the x-axis. Versions range from none (no change) to 10 (graph is modified every ten time steps).

For both IRREG and NBF, we see REPLICATEBUFS is unaffected, LOCALWRITE gradually degrades, but still outperforms REPLICATEBUFS at each point. GATHERSCATTER degrades so quickly speedups become negligible. The results are mostly expected. Possible excess communication due to the flush-update protocol does not appear to be significant. LOCALWRITE incurs moderate overhead, because it must reanalyze the graph to partition computation between processors. In comparison, GATHERSCATTER faces the most overhead because it has to rebuild (either incrementally or completely) its local address buffers and communication schedule.

In comparison, for MOLDYN speedups of REPLICATEBUFS and LOCALWRITE actually increase as the underlying graph is changed. This is because recalculating the molecular interaction list is computationally intensive and provides additional opportunities to exploit parallelism. Speedups increase because of longer sequential times, even though ac-

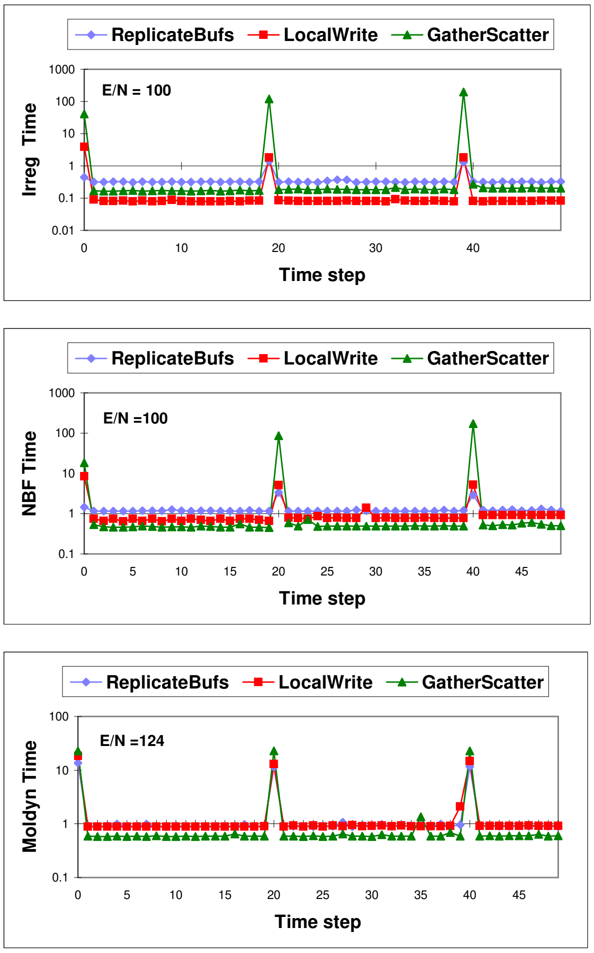

Figure 15. Time vs Time-Step Iteration

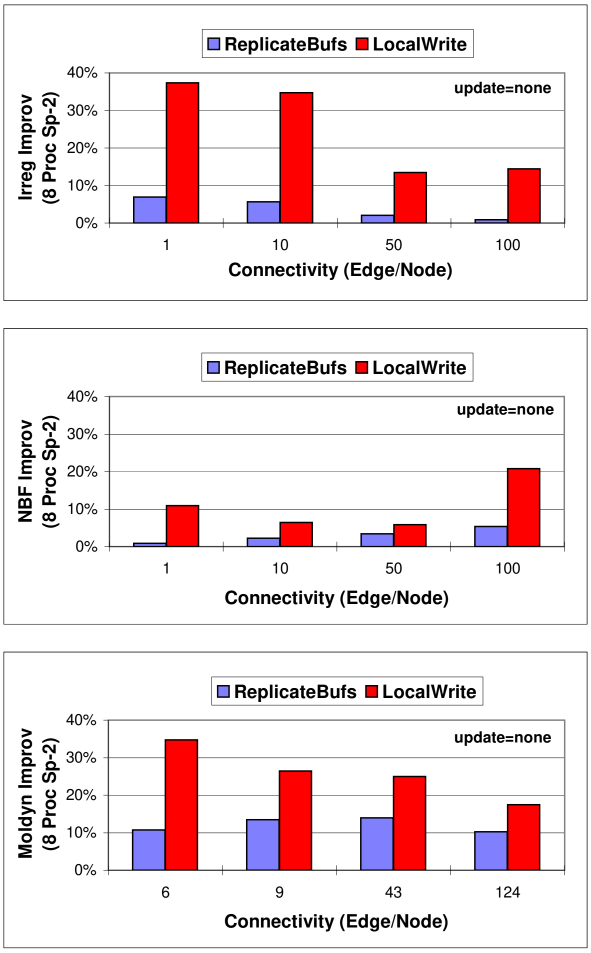

Figure 16. Protocol Improvements vs Connectivity

tual parallel execution times lengthen as well. Similarly, the speedups for GATHERSCATTER do not drop as precipitously because longer sequential times partially compensate for the higher inspector overhead for adaptive codes. Speedups for LOCALWRITE grow more slowly due to inspector overhead. When access patterns change most frequently (every ten iterations), REPLICATEBUFS finally outperforms LOCALWRITE.

Another way of viewing the impact of adaptivity on the performance of different techniques is to examine the execution time for each iteration of the time-step loop. Figure 15 displays the elapsed time for each iteration of the time-step loop. Time in seconds is plotted according to a logarithmic scale on the y-axis, while the number of the time-step loop is shown in the x-axis.

Results show on most time steps, GATHERSCATTER executes the most quickly, followed by LOCALWRITE, then REPLICATEBUFS. IRREG is an exception because LOCALWRITE actually runs quicker than GATHERSCATTER on every time step. On time steps where the underlying graph is changed, GATHERSCATTER requires a significantly amount of additional time to rebuild the communication schedule and address translation table. In comparison, LOCALWRITE has to do less work, and REPLICATEBUFS is not affected beyond the actual computation to change edges in the graph. From the REPLICATEBUFS times for MOLDYN, it is obvious that rebuilding the molecule interaction list is very expensive, regardless of the parallelization technique applied.

5.4.4 Impact of Protocol Improvements

Figure 16 displays improvement in performance due to using the flush-update protocol in CVM [20] as connectivity varies. Improvements are displayed along the y-axis. Improvements are measured as the percentage change in performance using the flush-update protocol compared to the default invalidate protocol. We see that REPLICATEBUFS is not much affected (5–10%), but LOCALWRITE benefits markedly (15–35%) from the flush-update protocol, decreasing as connectivity rises. Except for NBF, performance impact is greatest for sparsely connected arrays, where the communication/computation ratio is higher.

We also measure the improvements relative to adaptivity.

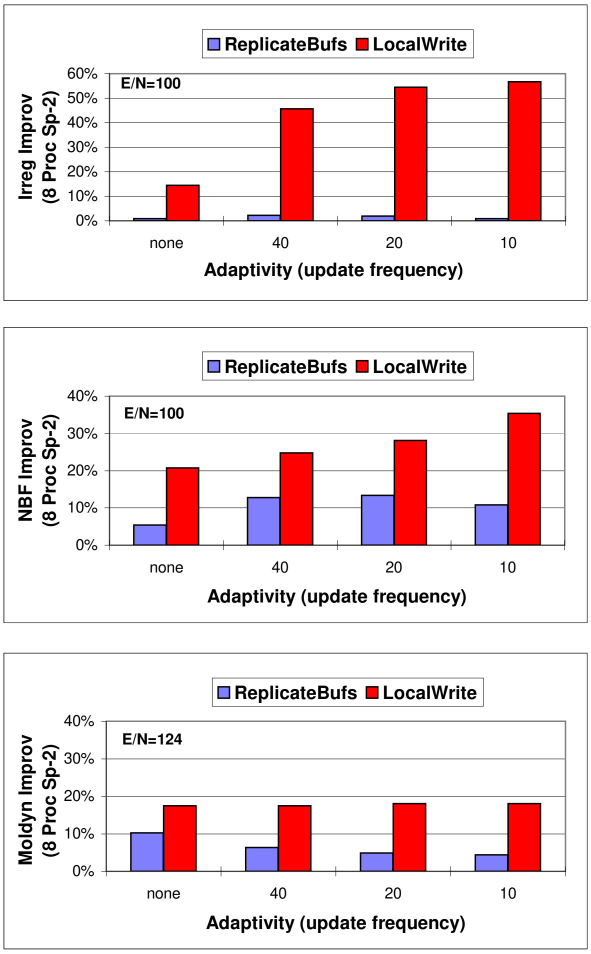

Figure 17. Protocol Improvements vs Adaptivity

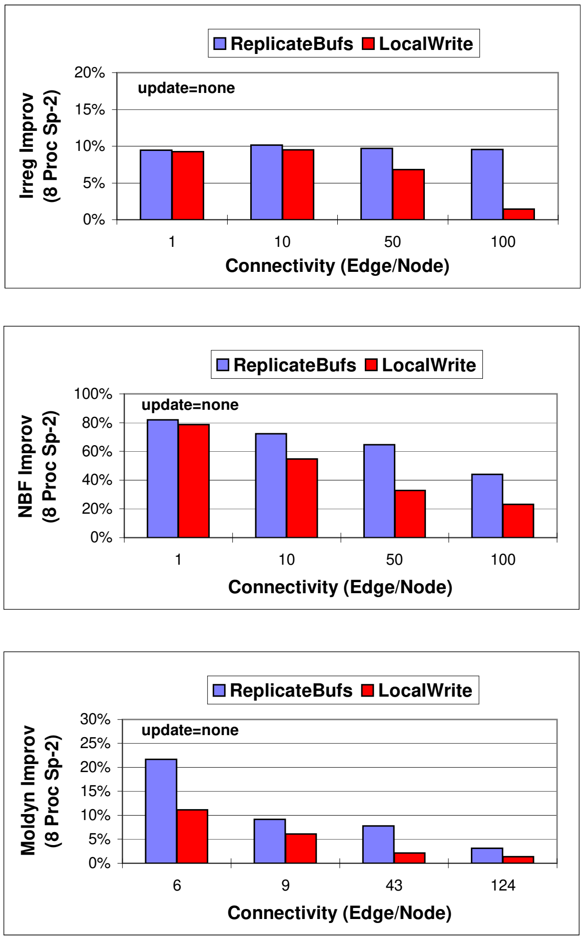

Figure 18. Overdrive Improvements vs Connectivity

Figure 17 displays improvements along the y-axis as adaptivity is varied along the x-axis. We see that REPLICATEBUFS improves slightly, but LOCALWRITE benefits more (20–50%), particularly for highly adaptive computations where communication costs increase. Overall, improvements due to the flush-update protocol are comparable to that achieved by TreadMarks applying optimizations for irregular reductions [24].

When the compiler determines the application is not adaptive, we can employ the overdrive protocol to eliminate OS page protection overheads [18]. Figure 18 displays improvements from overdrive relative to the flush-update protocol. Improvements are displayed along the y-axis as connectivity is varied along the x-axis. We see significant improvements (20-80%) are achieved in NBF, with lesser improvements (10%) for IRREG and MOLDYN. In contrast to the flush-update protocol, the main beneficiary of the overdrive protocol is REPLICATEBUFS instead of LOCALWRITE. These results show detecting adaptivity in scientific codes can be important for achieving good performance on the SP-2. Unfortunately we were unable to use overdrive as our baseline protocol, since the current implementation does not support adaptive computations.

5.5 Discussion

Our experimental results show LOCALWRITE is definitely useful as a new technique for parallelizing irregular reductions. On shared-memory multiprocessors LOCALWRITE outperforms REPLICATEBUFS only for very sparse graphs, but on distributed-memory multiprocessors it achieves higher speedups in nearly all conditions, improving performance roughly from 50–150% under a variety of conditions.

When compared to GATHERSCATTER, experimental results show LOCALWRITE can be the better choice under certain conditions. GATHERSCATTER generally provides better performance, but the gap can be small. There are conditions under which LOCALWRITE is the appropriate technique for parallelizing irregular reductions. LOCALWRITE outperforms GATHERSCATTER under conditions of low locality or high adaptivity.

6 Related Work

The importance of identifying and parallelizing reductions in scientific applications is well established [14, 23, 26, 28]. Irregular reductions have been recognized as being particularly vital. Researchers have investigated both efficient run-time [5, 15] and compiler [3, 11, 24] support. Rinard and Diniz use generalized reductions as the basis of their parallelizing compiler [27].

Most research has been based on the CHAOS run-time system, a library written to support the needs of parallel irregular applications which incorporates the GATHERSCATTER algorithm. CHAOS has been show to scale well for adaptive computations [15], but our results appear to indicate CHAOS is less efficient for modern multiprocessors where communication costs are higher relative to computation costs. Hanxleden and Kennedy present compiler analysis needed to automatically generate calls to CHAOS runtime routines in the Fortran D compiler [11, 12].

Lain and Banerjee show performance can be improved when irregular accesses possess spatial locality by tracking nonlocal access as *intervals* instead of individual elements. The interval approach is used in the PILAR run-time system of the PARADIGM compiler [22]. Chakrabarti *et al.* demonstrate interval representation can be used to combine communication optimizations for both regular and irregular accesses [2].

Several research groups have examined combining compilers and software DSMs, usually with enhancements based on combining explicit messages with underlying mechanisms for supporting nonlocal accesses [3, 6, 4].

Analysis of nonlocal accesses and communication is used to help the runtime system aggregate communication and synchronization. In some cases the iterative nature of an application is used to predict nonlocal accesses and presend data to eliminate misses [20, 30]. Special coherence protocols to aid reductions also help [20, 25]. Reducing synchronization in software DSMs can improve performance, including nearest-neighbor synchronization [10].

Lu *et al.* found that software DSMs can efficiently support irregular applications when using compile-time analysis to prefetch index arrays at run time [24]. In comparison, we attempt to achieve the same performance with less compiler analysis, based on reduction and flush-update coherence protocols.

7 Conclusions

In this paper we investigate improving compiler and runtime support for irregular reductions, the key computation for many sparse scientific computations. Compilers currently parallelize irregular reductions through either REPLICATEBUFS (local reduction to buffer, then global accumulation) or GATHERSCATTER (gather nonlocal data, then scatter nonlocal results). We present LOCALWRITE, a new technique for parallelizing sparse reductions. *Inspectors* partition computation according to ownership, potentially replicating computation but eliminating the need for synchronized writes. Experiments on both shared and distributed-memory multiprocessors indicate LOCALWRITE can significantly improves performance for some irregular applications. We also vary the connectivity, locality, and adaptivity of the applications to gain a better understanding of the advantages of different reduction parallelization techniques. By improving compiler support for irregular reductions, we are contributing to our long-term goal: making it easier for scientists and engineers to take advantage of the benefits of parallel computing.

8 Acknowledgements

The authors are grateful to Pete Keleher for providing CVM and valuable assistance on this paper. We wish to thank Honghui Lu for providing several benchmark codes, and our referees for their helpful comments.

References

[1] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. In D. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Ninth International Workshop (LCPC'96)*, volume 1239 of *Lecture Notes in Computer Science*. Springer-Verlag, Santa Clara, CA, 1996.

[2] D. Chakrabarti, N. Shenoy, A. Choudhary, and P. Banerjee. An efficient uniform run-time scheme for mixed regular-irregular applications. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[3] S. Chandra and J.R. Larus. Optimizing communication in HPF programs for fine-grain distributed shared memory. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[4] A. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.

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

[6] S. Dwarkadas, A. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Boston, MA, October 1996.

[7] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific pro-

tocols for user-level shared memory. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

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

[9] H. Han and C.-W. Tseng. Compile-time synchronization optimizations for software DSMs. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

[10] H. Han, C.-W. Tseng, and P. Keleher. Reducing synchronization overhead for compiler-parallelized codes on software DSMs. In Z. Li, P.-C. Yew, S. Chatterjee, C.-H. Huang, P. Sadayappan, and D. Sehr, editors, *Languages and Compilers for Parallel Computing, Tenth International Workshop, LCPC'97*, volume 1366 of *Lecture Notes in Computer Science*. Springer-Verlag, Minneapolis, MN, 1997.

[11] R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[12] R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

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

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

[15] Y.-S. Hwang, B. Moon, S. Sharma, R. Ponnusamy, R. Das, and J. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software—Practice and Experience*, 25(6):597–621, June 1995.

[16] L. Iftode, C. Dubnicki, E. Felten, and K. Li. Improveing release-consistent shared virtual memory using automatic update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[17] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[18] P. Keleher. Update protocols and iterative scientific applications. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

[19] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[20] P. Keleher and C.-W. Tseng. Enhancing software DSM for compiler-parallelized applications. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.

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

[22] A. Lain and P. Banerjee. Exploiting spatial regularity in irregular iterative applications. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[23] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

[24] H. Lu, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

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

[26] W. Pottenger. The role of associativity and commutativity in the detection and transformation of loop level parallelism. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[27] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–992, November 1997.

[28] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reductions operations for distributed parallel machines. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996.

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

[30] G. Viswanathan and J.R. Larus. Compiler-directed shared-memory communication for iterative parallel computations. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, November 1996.