Efficient Compiler and Run-Time Support for Parallel Irregular Reductions

Hwansoo Han, Chau-Wen Tseng

Dept. of Computer Science, University of Maryland, College Park, MD 20742

Abstract

Many scientific applications are comprised of irregular reductions on large data sets. In shared-memory parallel programs, these irregular reductions are typically computed in parallel using replicated buffers, then combined using synchronization. We develop LocalWrite, a new technique which partitions irregular reductions so that each processor computes values only for locally assigned data, eliminating the need for buffers or synchronized writes. Computation is replicated if its results are needed on multiple processors. We experimentally evaluate its performance for three irregular codes on a software DSM running on a distributed-memory multiprocessor and two shared-memory multiprocessors while varying connectivity, locality, and adaptivity. Results show LocalWrite improves performance significantly compared to using replicated buffers, and can match or exceed explicit message-passing gather/scatter for applications with low locality or high adaptivity.

Keywords: parallelizing compilers, locality optimizations, irregular computations

1 Introduction

Scientists are beginning to exploit parallelism to provide the computing power they need for research and development. As they attempt to model more complex problems, irregular adaptive computations become increasingly important. The core of these applications is frequently comprised of *reductions*, associative computations (e.g., SUM, MAX) which may be reordered and parallelized [35]. Because these codes access data in an unpredictable irregular

[1] 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.

manner, they are difficult to parallelize for distributed-memory multiprocessors which require explicit messages to access nonlocal data.

One approach to parallelizing irregular reductions, taken by distributed-memory compilers which generate explicit interprocessor messages (e.g., Fortran D [20]), is to rely on sophisticated run-time systems (e.g., CHAOS [5], PILAR [26]) which can identify and gather nonlocal data. A second approach is to combine shared-memory compilers (e.g., SUIF [11]) with software DSM systems (e.g., TreadMarks [30], CVM [24]), which provide a shared-memory interface. Software DSMs are less efficient than explicit messages, but are much simpler compilation targets [4,25]. In this paper, we examine existing approaches to parallelizing irregular reductions, and propose a new efficient algorithm.

1.1 Irregular Reductions

We begin by looking at 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 [25,39].

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 true on some iterations of the time-step loop, modifying elements of the index array `idx1` and changing overall data access patterns as a result.

1.2 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

```
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
```

Fig. 1. Irregular Reduction Example

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 [3,4,7]. A second approach exploits customized coherence protocols for reductions and nonlocal updates [8,24,25,33,39]. In addition, compilers can also eliminate unnecessary synchronization based on communication analysis [17,25].

TreadMarks is one of the most efficient software DSMs currently available. It relies on an invalidate coherence protocol, and parallelizes irregular reductions using ReplicateBufs. Performance is improved relying on a combination of compile-time and run-time support [30]. First, the compiler identifies the section of index array used by each processor, then prefetches it to reduce latency. 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 [30].

1.3 Contributions

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 the performance of different parallelization approaches as we vary application characteristics, in order to identify areas in which software DSMs can match or even exceed the efficiency of explicit messages. Experiments are conducted in a prototype system [13,25] using the CVM [24] software distributed-shared-memory (DSM) as a compilation target for the SUIF [11] shared-memory compiler. Our paper makes the following contributions:

– 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

The remainder of the paper begins with background material on existing parallelization techniques for reductions, followed by a description of LocalWrite. We present experimental results and conclude with a discussion of related work.

2 Background

We begin by considering two existing compiler techniques for parallelizing irregular reductions. The choice of either of two approaches, GatherScatter or ReplicateBufs, depends on the target architecture.

2.1 GatherScatter

Data-parallel compilers for distributed-memory multiprocessors (e.g., IBM SP-2, network of PCs) can generate explicit messages between processors [21]. Reductions with regular accesses can be converted directly to collective communication [29]. 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 [5,29,37]. 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.

```
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
```

Fig. 2. GatherScatter Example

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.

2.2 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 [11,35]. 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 [11]. 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.

```
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
```

Fig. 3. ReplicateBufs Example

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).

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 [11]. 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 [25]. 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 [23]. In addition, an *overdrive* protocol which reduces OS overhead for page protections can be employed for iterative codes with stable communication patterns [24].

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 [24], 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 [20]. By assigning only to local data, LocalWrite avoids the need for either buffers or mutual exclusion synchronization. The tradeoff is possibly replicated computation.

Consider how we can apply LocalWrite to the irregular reduction example in Figure 1. 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 [30], as well as which part of each index array is accessed by a processor at each point of the program [17,20,38]. 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

```
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
```

Fig. 4. LocalWrite Inspector Example

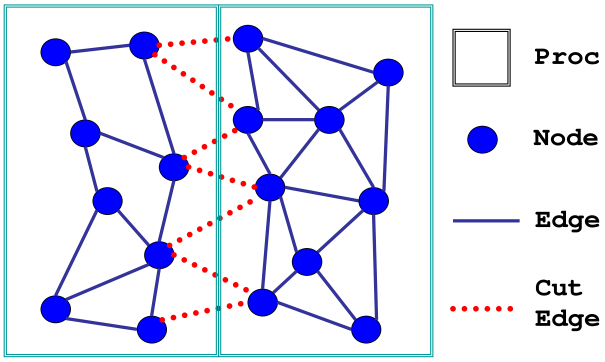

Fig. 5. Cut Edges Example

modify local data. The resulting code is shown in Figure 4.

The code generated to implement LocalWrite is complicated by the presence of *cut edges*, edges whose endpoints are on different processors. Figure 5 shows examples of cut edges for a sparse computation whose nodes are partitioned between two processors. Cut edges are problematic because for many scientific codes the computation needs to update both endpoints of an edge. 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 6. 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 6 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

```
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
```

Fig. 6. LocalWrite Cut Edge Example

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

Fig. 7. LocalWrite Adaptive Code Example

edges.

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.

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 [16]. 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 to detect adaptive computations and generate inspectors.

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 [2]. 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.

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

Once adaptive computations have been identified, we must generate inspectors for LocalWrite. Fortunately, the inspector for LocalWrite simpler than those for GatherScatter, since it 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. An example of LocalWrite for an adaptive code is shown in Figure 7.

Previous research on data-parallel compilers such as the Fortran D compiler have discussed compiler techniques for automatically generating inspectors and executors for CHAOS [18,19,22]. Simplified versions of those techniques should be directly applicable to generating inspectors and executors for LocalWrite. We believe these techniques for automatic generation of inspectors may be easily implemented 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.

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.

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 [22,33] and TreadMarks [30] 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 inspec-

	Ease-of-use	Inspector	Sensitivity to 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

tors 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

5.1 Experimental Platforms

We evaluated our optimizations on both distributed and shared-memory multiprocessors. Our IBM SP-2 has 66MHz RS/6000 Power2 processors operating under AIX 4.1. Nodes are connected by a 120 Mbit/sec bi-directional Omega switch. Our DEC Sable multiprocessor has four 275MHz Alpha 21064 processors and 256 MB of memory running Digital Unix V4.0. Our SUN HPC10000

```
** 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
```

Fig. 8. Key Kernels of Applications

has 64 UltraSparcII processors, 64GB of memory running Solaris 7.0.

Our experiments were conducted by combining the SUIF compiler with the CVM software DSM. We used SUIF to generate pthreads programs (DEC Sable, SUN HPC10000) and CVM programs (IBM SP-2). 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.

CVM is a software DSM that supports coherent shared memory for multiple protocols and consistency models [24]. 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 [25]. To compare results, we also wrote explicitly parallel programs using a version of the CHAOS run-time system implemented on top of MPI (for the IBM SP-2) [5].

5.2 Applications

We evaluated the performance of our compiler/software DSM interface with three sample applications, Irreg, Nbf, and Moldyn. Application param-

Name	Description	Problem Size	Steps	Edge/Node	Cut Edges
Irreg	Irregular CFD Mesh	10000 Nodes	50	1–100	10–70%
Nbf	Non-Bonded Force (GROMOS)	32000 Nodes	50	1–100	10–70%
Moldyn	Molecular Dynamics (CHARMM)	16384 Molecules	50	6–124	31–72%

Table 2
Scientific Applications

eters are presented in Table 2. 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.

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.

Nbf is a kernel abstracted from the GROMOS molecular dynamics code [18]. Instead of an edge list as in Irreg, it maintains a list of partners for each molecule. Partner lists are more compact than edge lists, but are 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.

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.

5.3 Application Characteristics

Our experiments evaluated the impact of connectivity, locality, and adaptivity on program performance for each reduction implementation technique. For our experiments, we used a standard baseline version of each program to perform

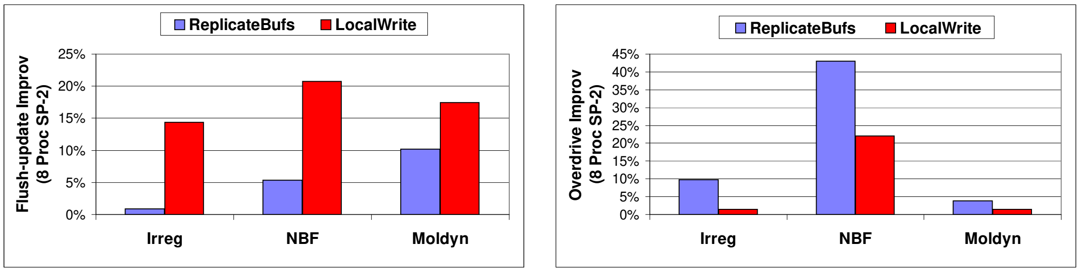

Fig. 9. Improvements vs Coherence Protocols

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 for molecular dynamics codes [30]. Lower connectivities around 10 are more representative of sparse mesh iterative PDE solvers.

Locality of 30% cut edges represents codes 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 [33]. In our experiments the locality is directly specified in the meshes generated. Only for MOLDYN do we apply RCB to decrease the percentage of cut edges from 72% to 31%. 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.

5.4 Impact of Coherence Protocols

We begin by examining the impact of enhanced coherence protocols in improving performance. Figure 9 displays improvement in performance in the baseline version of each program due to using the flush-update and overdrive protocols in CVM [25]. Improvements, displayed along the y-axis, are measured as the percentage change in performance using the flush-update protocol compared to the default invalidate protocol. Results show that the flush-update protocol is beneficial, improving the performance of all three applications by 15–20% for LOCALWRITE and 0–10% for REPLICATEBUFS. Overall, improvements are comparable to that achieved by TreadMarks applying explicit optimizations for irregular reductions [30].

When the compiler determines the application is not adaptive, we can also employ the overdrive protocol to eliminate OS page protection overheads [24]. We found 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. Due to the higher performance of the flush-update protocol, we include it as part of the baseline performance for each program. Unfortunately we were unable to use overdrive as our baseline protocol, since the current implementation does not support adaptive computations.

5.5 Distributed-Memory Speedups

Next, we examine the performance of reduction parallelization techniques for message-passing multiprocessors.

5.5.1 Impact of Connectivity

Figure 10 presents 8 processor speedups for ReplicateBufs, LocalWrite, and GatherScatter as connectivity varies for the static version of each application (no connectivity updates). 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

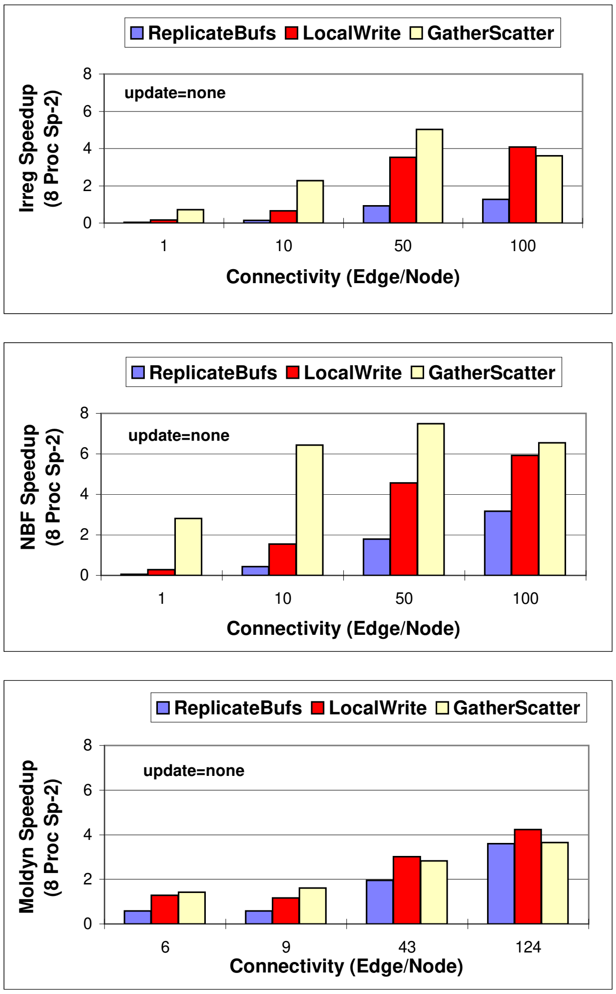

Fig. 10. Speedups vs Connectivity (IBM SP-2)

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.

5.5.2 Impact of Locality

Figure 11 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. We vary

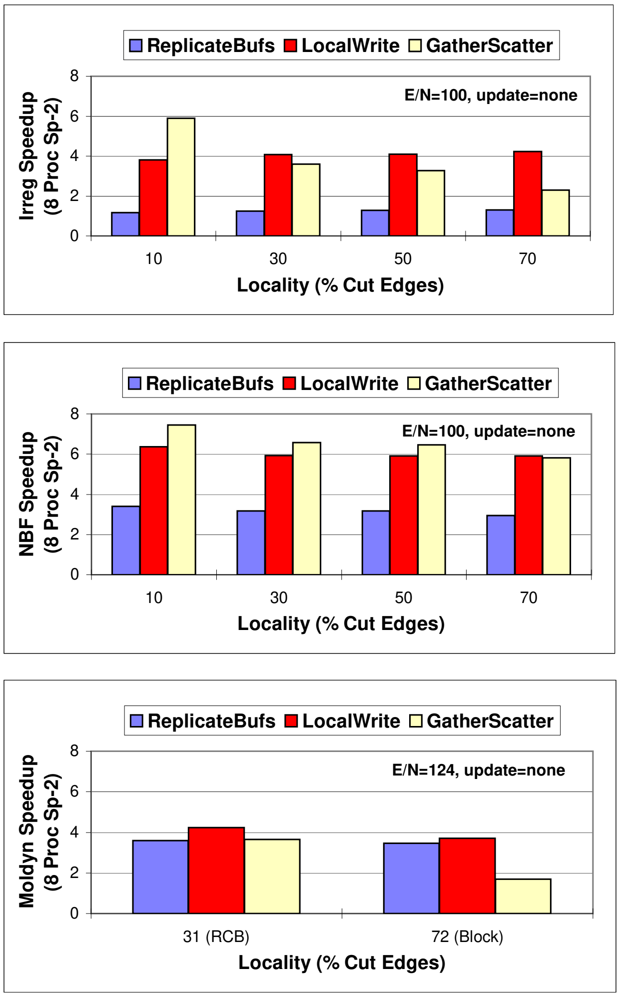

Fig. 11. Speedups vs Locality (IBM SP-2)

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 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. Work performed by GatherScatter increases in proportion to the number of non-local nodes accessed. Experimentally we find it to be more sensitive to locality than either ReplicateBufs or LocalWrite.

5.5.3 Impact of Adaptivity

Figure 12 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 actual 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 13 displays the elapsed time for each iteration of the time-step loop. Time in seconds is plotted according to a logarithmic scale on

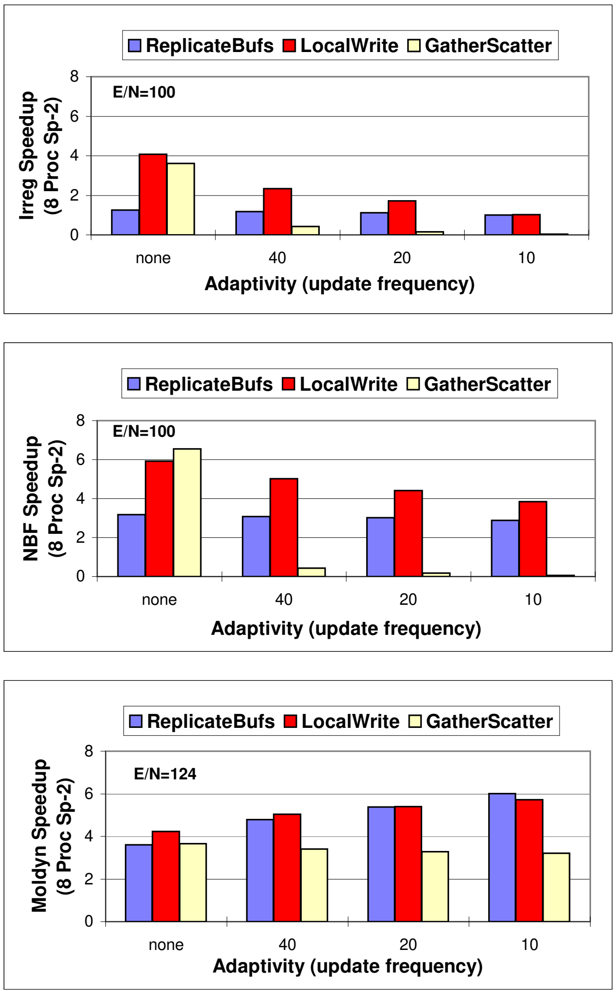

Fig. 12. Speedups vs Adaptivity (IBM SP-2)

---

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.

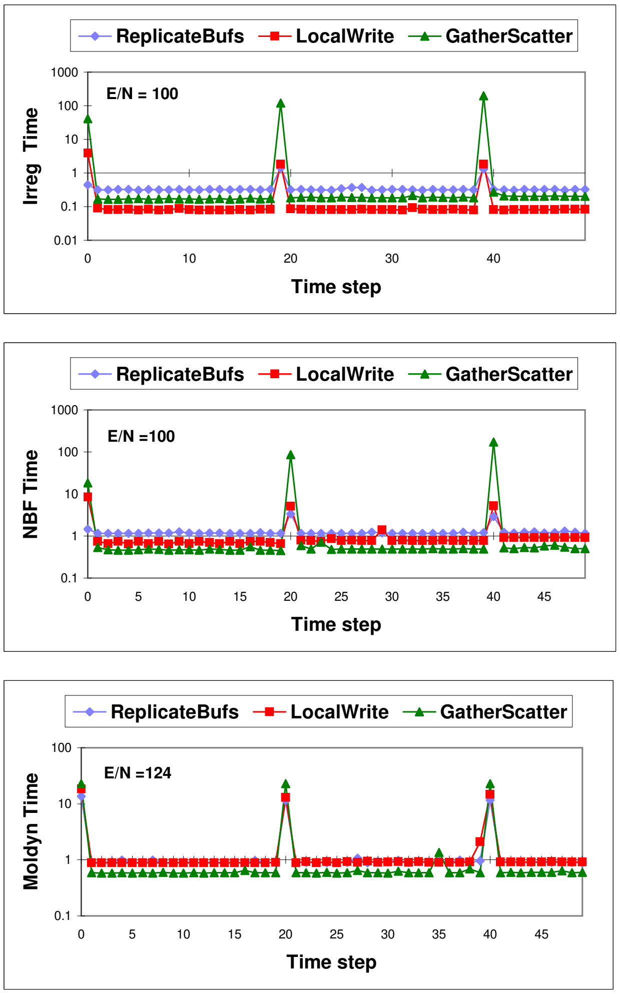

Fig. 13. Time vs Time-Step Iteration

5.6 Shared-Memory Speedups

Even though LocalWrite was designed for software DSMs running on distributed-memory machines where communication is expensive, we believe it can also be useful on shared-memory multiprocessors because of its better scalability. To this end we examine its performance on two shared-memory multiprocessors, a 4-processor DEC Sable Alpha and a 8-processor SUN HPC10000.

Figures 14 and 15 present speedups on the DEC Alpha and SUN HPC10000 for ReplicateBufs and LocalWrite as the connectivity (edges/node ratio) 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.

On the 4-processor DEC Alpha, the crossover point is reached when the edge/node ratio passes ten. For more densely connected graphs, ReplicateBufs outperforms LocalWrite because it avoids the extra preprocessing overhead required by LocalWrite. However, LocalWrite performance is higher than ReplicateBufsfor the 8-processor SUN HPC10000, even for very densely connected graphs. These results suggest that LocalWrite is preferable for larger multiprocessors (less synchronization, redundant computation) and faster processors (locality more important, processing less important).

5.7 Discussion

Our experimental results show LocalWrite is definitely useful as a new technique for parallelizing irregular reductions. On distributed-memory multiprocessors it achieves higher speedups in nearly every case, achieving performance approximately 50–150% better than ReplicateBufs 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.

On shared-memory multiprocessors the tradeoffs between LocalWrite and ReplicateBufs are less clear. LocalWrite is obviously best for very sparse graphs, but whether it outperforms ReplicateBufs depends on the number and speed of processors on the system. However, architectural trends favor LocalWrite.

6 Related Work

The importance of identifying and parallelizing reductions in scientific applications is well established [21,29,35,37]. Researchers have investigated compiler analyses [27,28] and ways to provide efficient run-time and compiler

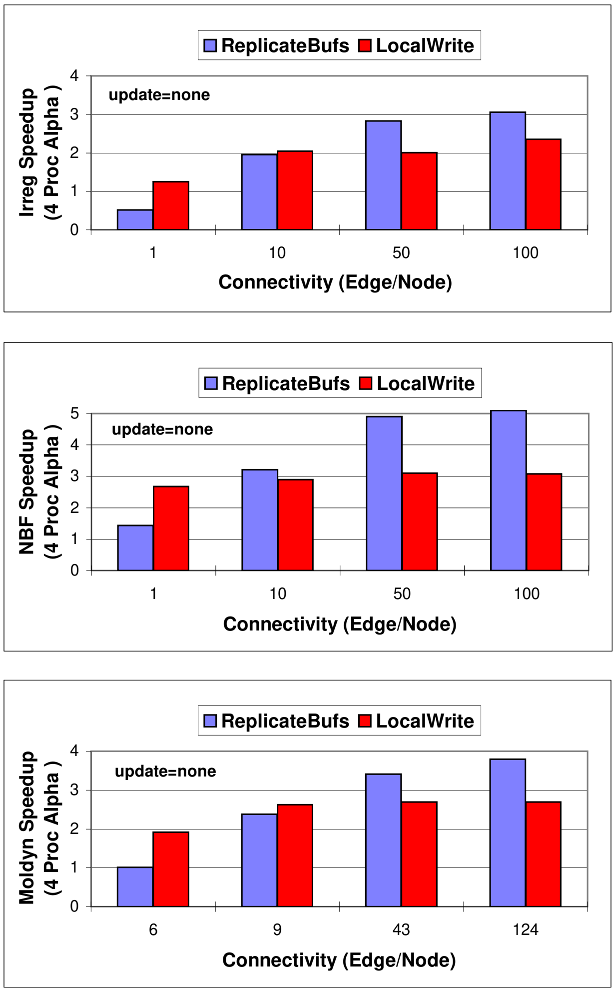

Fig. 14. Speedups vs Connectivity (DEC Alpha)

support [3,36]. Irregular reductions have been recognized as being particularly vital. Researchers have investigated both efficient run-time [5,22] and compiler [3,18,30,36] support. 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 [22]. Compilers can also automatically generate calls to CHAOS run-time routines [18,19]. PILAR improves performance by tracking nonlocal access as *intervals* instead of individual elements [26].

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 [3,7,4]. 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

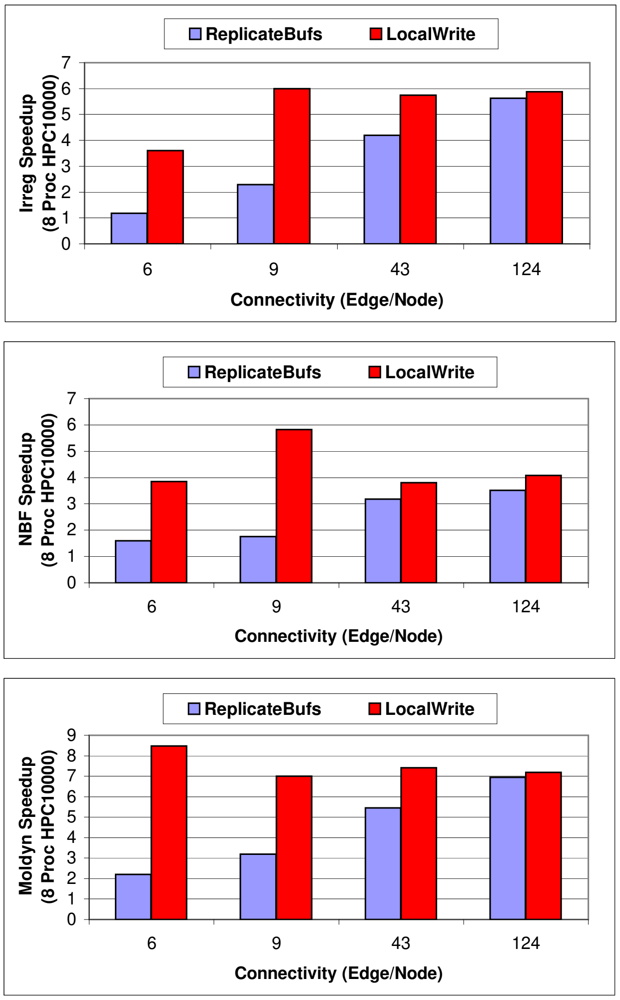

Fig. 15. Speedups vs Connectivity (Sun HPC10000)

to eliminate misses [25,39]. Special coherence protocols to aid reductions also help [25,33]. Reducing synchronization in software DSMs can improve performance, including nearest-neighbor synchronization [17]. Lu *et al.* found that software DSMs can efficiently support irregular applications when using compile-time analysis to prefetch index arrays at run time [30]. In comparison, we present a machine-independent approach based on compiler and run-time transformations which is applicable even on shared-memory multiprocessors.

Data locality has been recognized as a significant performance issue for modern processor architectures. Most researchers have focused on loop transformations for regular computations [31,40]. More recently a few groups have examined improving locality for irregular codes on uniprocessors using data and computation partitioning techniques [16,12]. Al-Furaih and Ranka investigated using graph partitioning to improve locality [1], while Ou *et al.* examined spectral partitioning and space-filling curves [34]. Ding and Kennedy explored applying dynamic copying (packing) of data elements based on loop traversal order, and show major improvements in performance [6]. They were able to automate most of their transformations in a compiler. Mellor-Crummey *et al.* use a geometric partitioning algorithm based on space-filling curves to map multidimensional data to memory [32]. In comparison, we improve locality characteristics of parallel reductions, a key component of irregular computations.

We previously introduced the LocalWrite algorithm in the context of compiling for software DSMs [15,14]. This paper extends and refines our algorithm in greater detail. Two other research groups have investigated optimization techniques for parallel irregular reductions. Gutierrez *et al.* developed *data-write affinity*, a technique similar to LocalWrite which uses an adjunct prefetch array to select which loop iterations which should be executed locally [9]. In comparison, we reduce overhead by avoiding the prefetch array, reordering the index arrays directly. They used experiments on an SGI Origin2000 to demonstrate the usefulness of LocalWrite for parallelizing irregular reductions on large-scale shared-memory multiprocessors. Gutierrez *et al.* later developed SynchWrite, a scheme to avoid computation replication using additional synchronization to separate computation into non-conflicting phases [10]. Experiments on an SGI Origin2000 showed performance improved over ReplicateBufs, but did not compare against LocalWrite.

Yu and Rauchwerger introduced a framework for dynamically selecting different parallelization alternatives for irregular reductions, based on run-time information and a cost model [41], They also developed a technique based on selective privatization of conflicting elements, and compared its performance against ReplicateBufs and the version of LocalWrite described by Gutierrez *et al.* Their experiments on a HP-V shared-memory multiprocessor showed different algorithms were better as connectivity and adaptivity varied. Their cost model was able to predict which technique to apply to achieve the best performance.

7 Conclusions

In this paper we investigate improving compiler and run-time support for irregular reductions, the key computation for many sparse scientific computations. 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] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

[2] T. Autrey and M. J. 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.

[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] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

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

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

[9] E. Gutierrez, O. Plata, and E. Zapata. On automatic parallelization of irregular reductions on scalable shared memory multiprocessors. In *Proceedings of the Fifth International Euro-Par Conference (EUROPAR'99)*, Toulouse, France, August 1999.

[10] E. Gutierrez, O. Plata, and E. Zapata. A compiler method for the parallel execution of of irregular reductions on scalable shared memory multiprocessors. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[11] 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.

[12] H. Han, G. Rivera, and C.-W. Tseng. Software support for improving locality in scientific codes. In *Proceedings of the Eighth International Workshop on Compilers for Parallel Computers (CPC'2000)*, Aussois, France, January 2000.

[13] 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.

[14] H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, October 1998.

[15] H. Han and C.-W. Tseng. Improving compiler and run-time support for irregular reductions using local writes. In *Languages and Compilers for Parallel Computing, Eleventh International Workshop, LCPC'98*, volume 1656 of *Lecture Notes in Computer Science*. Springer-Verlag, Chapel Hill, NC, August 1998.

[16] H. Han and C.-W. Tseng. A comparison of locality transformations for irregular codes. In *Proceedings of the 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 2000.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

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

[25] 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.

[26] 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.

[27] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular Fortran programs. In *Proceedings of the 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Pittsburgh, PA, May 1998.

[28] Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.

[29] 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.

[30] 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.

[31] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[32] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[33] 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.

[34] C. Ou, M. Gunwani, and S. Ranka. Architecture independent locality-improving transformations of computational graphs embedded in k-dimensions. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[41] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, NM, May 2000.