Improving Compiler and Run-Time Support for Irregular Reductions Using Local Writes

Hwansoo Han, Chau-Wen Tseng

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

Abstract. Current compilers for distributed-memory multiprocessors parallelize irregular reductions either by generating calls to sophisticated run-time systems (CHAOS) or by relying on replicated buffers and the shared-memory interface supported by software DSMs (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, and can match or exceed gather/scatter for applications with low locality or high adaptivity.

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 [6, 11]. 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. 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

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.

is to rely on sophisticated run-time systems (e.g., CHAOS [4], PILAR [15]) which can identify and gather nonlocal data. A second approach is to combine shared-memory compilers (e.g., SUIF [6]) with software distributed-shared-memory (DSM) systems (e.g., TreadMarks [17], CVM [13]), which provide a shared-memory interface. Software DSMs are less efficient than explicit messages, but are much simpler compilation targets [3, 14].

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 [7, 14] using the CVM [13] software distributed-shared-memory (DSM) as a compilation target for the SUIF [6] 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

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 [14].

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.

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 distributed-memory multiprocessors (e.g., IBM SP-2, network of PCs) can generate explicit messages between processors [11]. Reductions with regular accesses can be converted directly to collective communication [16].

Irregular reductions may be parallelized by generating an *inspector* to identify nonlocal data needed by each processor. 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 [4].

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 [4]. 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.

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 [6]. 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 [6]. Unfortunately, REPLICATEBUFS is inefficient 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.

```
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

```
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

```
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

```
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

```
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. 5. 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. 6. LocalWrite Adaptive Code Example

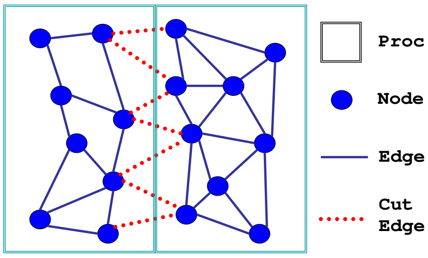

Fig. 7. Cut Edges Example

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 [2, 3, 5]. A second approach exploits customized coherence protocols for reductions and nonlocal updates [13, 14, 18]. In addition, compilers can also eliminate unnecessary synchronization based on communication analysis [7, 14].

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 [17]. 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 [4], which supports irregular reductions using GATHERSCATTER [17].

3 Improving Irregular Reductions

Now that we have seen how irregular reductions are parallelized by existing approaches, we consider LOCALWRITE, a new compiler and run-time technique for parallelizing irregular reductions.

3.1 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 [10]. 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 [17], as well as which part of each index array is accessed by a processor at each point of the program [7, 10]. 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. 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 7 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 5. 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 5 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.

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. 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.

3.2 Compiler Analysis

We have now presented three techniques (ReplicateBufs, LocalWrite, and GatherScatter) for parallelizing irregular reductions. 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.

To use LocalWrite, the compiler must insert inspectors both to calculate computation partitions based on local iterations, and compute updates when access patterns change. 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 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 it is adaptive.

We must then generate inspectors for LocalWrite. Fortunately, they are 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 6.

Previous research on data-parallel compilers such as the Fortran D compiler have discussed compiler techniques for automatically generating inspectors and executors for CHAOS [8, 9, 12]. 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 Experimental Evaluation

4.1 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. 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.

Our experiments were conducted by combining the SUIF compiler with the CVM software DSM. We used SUIF to generate pthreads programs (DEC Sable) 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.

	Description	Problem Size	Time Steps	Connectivity	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 1. Characteristics of Scientific Applications

CVM is a software DSM that supports coherent shared memory for multiple protocols and consistency models [13]. 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 [14]. 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) [4].

4.2 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 1. 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 [8]. 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.

4.3 Application Characteristics

Because of the complexity of the parallelization process, we believe performance will be significantly affected by the application. We describe three important application characteristics, connectivity, locality, and adaptivity. If we view irregular accesses as edges in a graph, we can classify the irregular accesses according its *connectivity*, the edges/nodes ratio. If the ratio of edges to nodes is high, we consider the graph *densely* connected. A low ratio indicates a *sparsely* connected graph. *Locality* is the proportion of data accesses which are to data located on the processor. Another way of measuring locality is by the percentage of edges which are cut edges. Finally, *adaptivity* is the rate at which data access patterns change during program execution. *Static* applications have fixed data access and communication patterns. *Adaptive* applications change dynamically and are less well understood.

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 [17]. 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 [18]. 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.

4.4 Shared-Memory Speedups

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 8 presents 4 processor speedups 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.

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

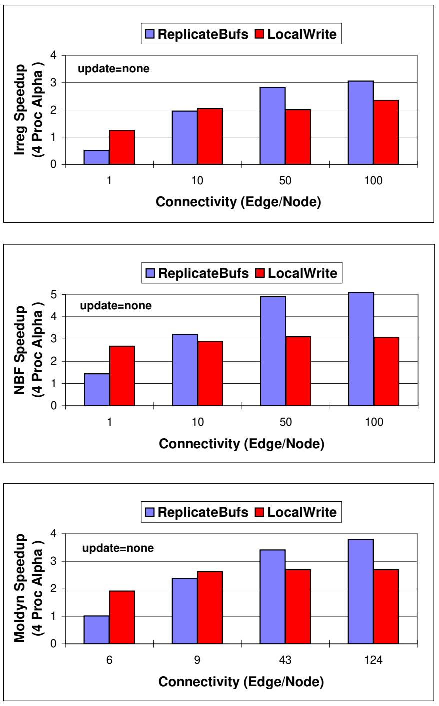

Fig. 8. Speedup vs Connectivity (Alpha)

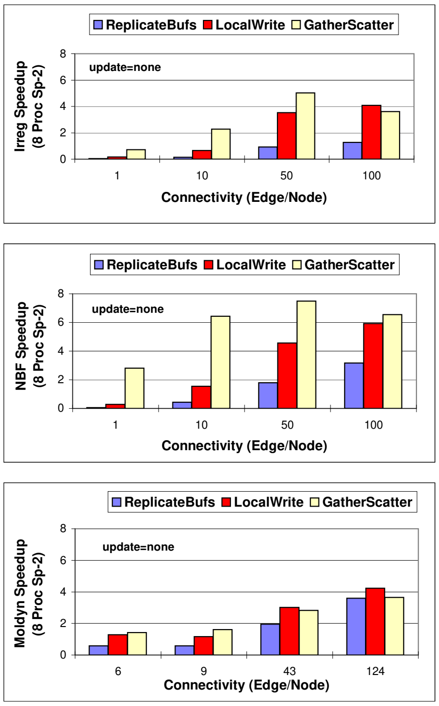

Fig. 9. Speedup vs Connectivity (SP-2)

computation is not profitable unless the graph is sufficiently sparse that ReplicateBufs performs significant amounts of wasted work for replicated buffers.

4.5 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.

Impact of Connectivity Figure 9 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.

Impact of Locality Figure 10 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 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

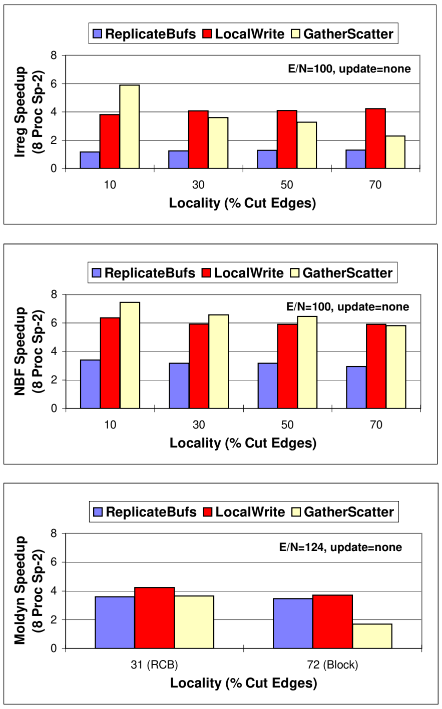

Fig. 10. Speedup vs Locality (SP-2)

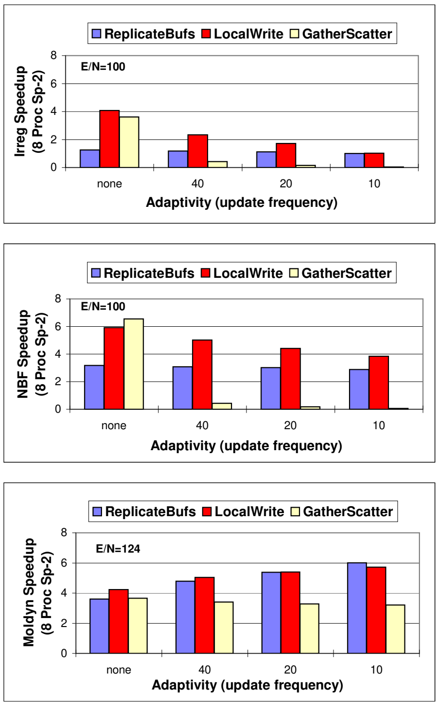

Fig. 11. Speedup vs Adaptivity (SP-2)

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 nonlocal nodes accessed. Experimentally we find it to be more sensitive to locality than either REPLICATEBUFS or LOCALWRITE.

Impact of Adaptivity Figure 11 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.

4.6 Discussion

Our experimental results show LocalWrite is definitely useful as a new technique for parallelizing irregular reductions. On a 4-processor shared-memory multiprocessor LocalWrite outperforms ReplicateBufs only for very sparse graphs. However, we expect it to be more profitable for larger numbers of processors, as it requires less communication and synchronization. On distributed-memory multiprocessors LocalWrite 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.

5 Related Work

The importance of identifying and parallelizing reductions in scientific applications is well established [11, 16]. Irregular reductions have been recognized as being particularly vital. Researchers have investigated both efficient run-time [4, 12] and compiler [2, 8, 17] 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 [12]. Compilers can also automatically generate calls to CHAOS run-time routines [8, 9]. PILAR improves performance by tracking nonlocal access as *intervals* instead of individual elements [15].

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 [2, 5, 3]. 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 [14]. Special coherence protocols to aid reductions also help [14, 18]. Reducing synchronization in software DSMs can improve performance, including nearest-neighbor synchronization [7].

Lu *et al.* found that software DSMs can efficiently support irregular applications when using compile-time analysis to prefetch index arrays at run time [17]. In comparison, we achieve comparable performance based on LocalWrite.

6 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.

7 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. 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.

3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.
12. 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.
13. P. Keleher. Update protocols and iterative scientific applications. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.