A Comparison of Parallelization Techniques for Irregular Reductions

Hwansoo Han, Chau-Wen Tseng

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

{*hshan, tseng*}@*cs.umd.edu*

Abstract

A large class of scientific applications are comprised of irregular reductions on large data sets. On shared-memory multiprocessors these reductions are typically parallelized by computing partial results into replicated buffers, then combining the values into shared data using synchronization. Recently, a number of alternative techniques have been developed based on selective privatization, local writes, and synchronized writes. In this paper, we present a more efficient version of the local write algorithm which is 56% faster on average. We then experimentally compare the performance of each technique using a number of representative kernels. Results show speedups vary greatly depending on application characteristics such as connectivity, locality, and adaptivity. In general, we find the local write technique provides the best performance, particularly when applications display good locality.

1 Introduction

As scientists attempt to model ever more complex problems, their programs become increasingly reliant on sparse, irregular data structures. One application domain where these *irregular* codes are common is computational fluid dynamics (CFD), where partial differential equations are solved using iterative methods. Unstructured meshes are used to discretize large, complex domains, as well as enable adaptive mesh refinement. Another application domain where irregular codes are common is computation chemistry, where molecular interactions are simulated using n-body algorithms to efficiently calculate Van der Waal's and electrostatic forces between atoms. These molecular dynamics problems use adaptive irregular data structures because the positions of atoms are inherently scattered and dynamic.

This research was supported by NSF CAREER Development Award #ASC9625531 in New Technologies. Time on the Sun HPC10000 was provided by NPACI and SDSC.

```
global x[n_nodes],y[n_nodes]  // data in nodes

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

Figure 1 Irregular Reduction Example

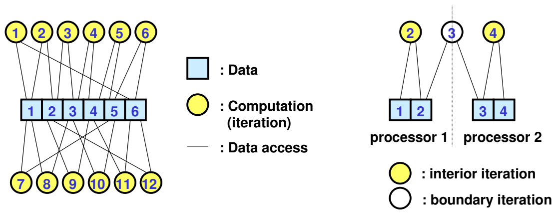

Figure 2 Graphical Representation of Irregular Reduction

The core of these irregular scientific applications are frequently comprised of *reductions*, associative and commutative computations (e.g., SUM, MAX) which may be reordered and parallelized [16]. Figure 1 presents an example of an irregular reduction. The computation loops over the edges of an irregular graph, computes a value, and applies it to both endpoints of the edge. The computation is irregular because accesses to array `y` are determined by the *index arrays* `idx1` and `idx2`. It is a reduction because the addition operations are associative and may be reordered and parallelized. The example computation is also *iterative*, an important characteristic of many scientific applications.

Current commercial compilers typically support irregular reductions poorly or not at all. On shared-memory multiprocessors compilers typically use replicated buffers which can waste memory and incur unnecessary computation overhead. As a result researchers have investigated irregular reductions and proposed several techniques for parallelizing irregular reductions based on selective privatization [18] local writes [9, 6], and synchronized writes [7]. We previously studied irregular reductions on distributed shared-memory systems and compared our local write with various other reduction techniques [9]. In that research, we also showed preliminary result on small scale shared-memory systems(4 processors), but could not fully evaluate the performance of the local write on large shared-memory systems. In this paper, we present a more efficient version of local writes, then experimentally compare these techniques for a variety of conditions on a large scale shared-memory machine(16 processors). We also include two recently proposed irregular reduction techniques, selective privatization and synchronized writes for comparison. Our paper makes the following contributions:

- present a more efficient version of local write technique based on local index arrays (average 56% faster)
- experimentally compare techniques for parallelizing irregular reductions on shared-memory multiprocessors
- evaluate their impact when combined with locality optimizations (local write improves 38% on average)

The remainder of the paper begins with a description of different parallelization techniques for reductions. We then present our experimental comparison and conclude with a discussion of related work.

2 Parallelization Techniques

Irregular reductions were first recognized as an important issue by compilers for distributed-memory multiprocessors. Since these compilers must generate explicit interprocessor communication for nonlocal accesses, the compiler first generates an *inspector* to preprocess the reduction, creating a list of data to be communicated as well as assigning their locations [4]. The reduction is then transformed into an *executor* which gathers nonlocal data, performs the computation using local buffers, and scatters nonlocal results to other processors. Fortunately, this level of precision is not needed for shared-memory multiprocessors. In this section, we review a number of techniques for parallelizing irregular reductions on shared-memory multiprocessors.

We begin by introducing a graphical representation which will be used throughout this paper. The irregular reduction code in Figure 1 can be expressed in a graph shown in Figure 2. Squares represent data arrays (X, Y) and circles represent iterations of the reduction loop. Edges between squares and circles represent data accessed by each iteration. On multiprocessors, iterations (circles) can be further classified as interior iterations (accessing only local data) and boundary iterations (accessing both local and nonlocal data). These figures help illustrate how data and loop iterations are partitioned between processors, and the non-local accesses which result.

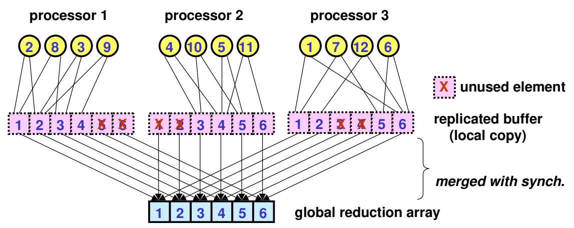

Figure 3 REPLICATEBUFS

2.1 Array Privatization (REPLICATEBUFS)

Compilers for shared-memory multiprocessors typically parallelize irregular reductions by having each processor compute a portion of the reduction, storing results in a *privatized* (locally replicated) buffer [3, 17], as shown in Figure 3. Results from all buffers are then accumulated into the shared array, using synchronization to avoid data races [8]. Large replicated buffers can be combined in parallel by performing global accumulations in sections using a pipelined, round-robbin algorithm [8].

While REPLICATEBUFS is simple and works well for scalar or regular reductions, it can waste processing time on unused elements for sparse reductions. Overhead is high since each processor needs to initialize all elements and merge them into global array after the parallel reduction, even though many of them may be unnecessary (unused). Unused elements increase as reductions get sparse. More importantly, as more processors are used, a larger portion of replicated buffers become unused. This prevents REPLICATEBUFS from scaling to larger numbers of processors.

2.2 Selective Privatization (SELECTPRIV)

To avoid the overhead encountered by REPLICATEBUFS, Yu and Rauchwerger devised an approach based on selective privatization of *conflicting* array elements (those modified by multiple processors) [18]. Figure 4 demonstrates how SELECTPRIV works. Unlike REPLICATEBUFS, SELECTPRIV replicates only conflicting elements that are used in local computation. SELECTPRIV is based on a simplified version of the inspector-executor approach used by distributed-memory compilers.

For each irregular reduction, an inspector analyzes data accesses and inserts references into a hash table to identify conflicting elements. These elements are then privatized, prepared for remote accesses, and stored locally in a compacted array. Inspectors also provide a hashing function for fast access to private copy during parallel reduction. Private copies are allocated in globally accessible area. Inspectors also set up pointers for each processor to enable

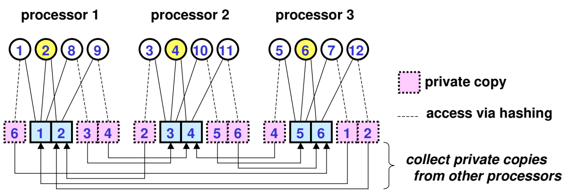

Figure 4 SELECTPRIV

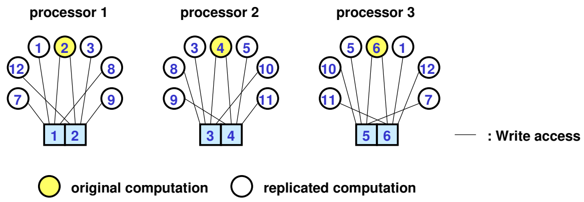

Figure 5 LOCALWRITE

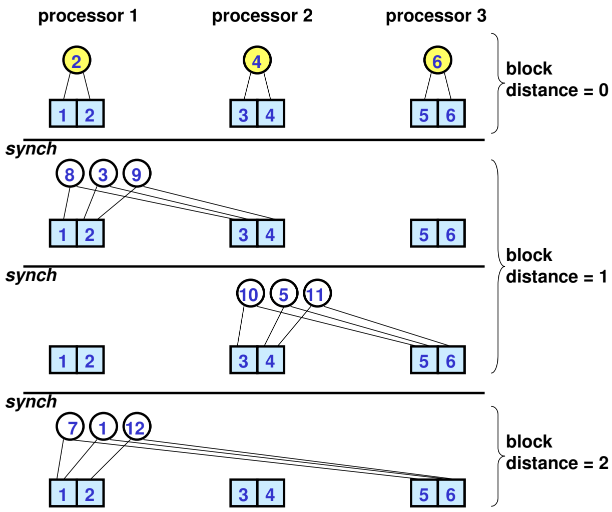

Figure 6 SYNCHWRITE

efficient collection of private copies on remote processors after parallel reductions.

Reductions are transformed into executors which update shared data and local privatized data in parallel without synchronization. As a final step the partial reductions on selectively privatized elements are aggregated and used to update the shared array. SELECTPRIV is a scalable scheme, replicating only elements where necessary, but at the expense of requiring an inspector.

2.3 Local Write (LOCALWRITE)

LOCALWRITE is another technique for parallelizing irregular reductions. We first developed for a compiler targeting software distributed-shared-memory (DSM) systems. It assigns loop iterations so that each processor only performs computations for local data, replicating computation when needed [9]. Figure 5 depicts how LOCALWRITE computes irregular reductions. Array elements are first distributed in blocks to processors. Each processor is then assigned loop iterations that compute results for local data. If a computation updates elements on different processors, it is replicated on both processors and used to modify only local data.

LOCALWRITE is an application of the *owner-computes* rule used in distributed-memory compilers [12]. By assigning only to local data, LOCALWRITE avoids the need for either replicated buffers or synchronization to aggregate values. Data locality is also enhanced since each processor executes iterations which access local data. The tradeoff is replicated computation for boundary iterations which assign values to data on different processors. As processors become faster, performing more computation for better locality and less synchronization is likely to be a win.

LOCALWRITE is implemented by inserting an inspector to preprocess the loop iterations, selecting all iterations which update only local data, as well as boundary iterations which access both local and nonlocal data. The reduction is then transformed into an executor which executes only those iterations. On boundary iterations, the code is transformed so only local data are updated.

2.4 Synchronized Write (SYNCHWRITE)

Whereas previous techniques replicate either data or computation, another technique SYNCHWRITE avoids either by using *synchronization* to ensure mutual exclusion for writes [7]. The basic idea is that iterations are partitioned based on the range of data updated. Iterations can then be executed in parallel without replication, using synchronization to separate groups of conflicting iterations into different computation phases.

The SYNCHWRITE algorithm is displayed in Figure 6. As in LOCALWRITE, array elements are assigned in blocks to each processor. An inspector partitions computations into groups according to the distance (in blocks) between two elements being accessed. If p processors are used, block distances range from 0 to $p-1$. Except for the 0 and $p-1$ block distance groups, computations in each distance group are again grouped into two subgroups that need to be performed in two phases. Synchronizations are inserted between groups and phases to prevent write conflicts.

SYNCHWRITE is a scalable scheme, but limited due to the underutilization of processing power. When we use more processors, we produce longer block distance group. However, the longer block distance group requires fewer processors. When distance is d on p processors, only $p-d$ processors are needed for the group. In addition to that, the groups computed in two phases utilize only half the processors in one phase. Thus, SYNCHWRITE utilizes less than half the processors at best when it computes for groups with more than 0 block distance. Another drawback is the increased cost of synchronization. Since

Technique	Replicated Data	Replicated Computation	Synchronization	Inspector	Scalable
REPLICATEBUFS	all		$\surd$		
SELECTPRIV	some		$\surd$	$\surd$	$\surd$
LOCALWRITE		$\surd$		$\surd$	$\surd$
SYNCHWRITE			$\surd$	$\surd$	

Table 1 Characteristics of Techniques

larger number of processors produce more block distance groups, the number of synchronizations between groups increases. Consequently, time spent on synchronization increases.

Table 1 summarizes some of the differences between the four techniques for parallelizing irregular reductions.

2.5 Local Write Implementations

In addition to the different parallelization approaches, we found the implementation chosen for LOCALWRITE can also affect performance. Two possible implementations are shown in Figure 7.

One implementation of LOCALWRITE is proposed by Gutierrez *et al.* [6]. They called their algorithm *data write affinity* with *loop index prefetching* (DWA-LIP). DWA uses an inspector to build a linked list (stored in an array `next`) to indicate to each processor which iteration it should execute next. Each processor then traces through the list during the computation. This implementation can be extended to support the SYNCHWRITE algorithm.

In our implementation of LOCALWRITE the inspector for each processor instead generates a local index array, where the index array is used to access shared data. In essence the inspector gathers the indices used in the original index array, and stores them in the local index array in compacted form. This implementation uses more memory than DWA, but improves locality because it does not need to skip around in memory. It also avoids the overhead of fetching the next loop index on each loop iteration.

```
// Original Code

do i = 1, edges
  y[idx[i]] += ...


//---------------------------------------------
//Data Write Affinity  // Local Write
/* inspector */        /* inspector */
count = 0              count = 0
do i = 1, edges        do i = 1, edges
  if (own(idx[i]))       if (own(idx[i]))
     next[count] = i        idx2[count]= idx[i]
     count += 1             count += 1

/* reduction */
ii = next[0]           /* reduction */
do i = 1, count        do i = 1, count
  y[idx[ii]] += ...      y[idx2[i]] += ...
  ii = next[i]
```

Figure 7 LOCALWRITE Implementation Examples

3 Experimental Evaluation

We evaluated irregular reduction techniques on a SUN HPC 10000 with 64 400MHz UltraSparc II processors and 64 gigabytes of memory, using Solaris 7.0. For our experiments, we currently generate inspectors by modifying SUIF output by hand.

3.1 Applications

We evaluated the performance using three sample applications, IRREG, NBF, and MOLDYN. These applications contain an initialization section followed by the main computation enclosed in a 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 iteration 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. NBF is a kernel abstracted from the GROMOS molecular dynamics code. 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. 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.

3.2 Input Characteristics

Because of the complexity of the parallelization process, we believe performance will be significantly affected by the input. We describe three important input characteristics, connectivity, locality, and adaptivity. If we view irregular accesses as edges in a graph, we can classify inputs according to their *connectivity*, the edges/nodes ratio (E/N). 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 proces-

Connectivity			Locality (% interior iterations)						Adaptivity
E/N	#Nodes	#Edges	4 processors		8 processors		16 processors		adaptation interval
			base	RCB	base	RCB	base	RCB	
6	70304	421824	0	95	0	93	0	90	10 – infinity
9	70304	632736	33	95	33	92	32	90	10 – infinity
43	70304	3023072	30	91	29	86	27	82	10 – infinity
124	70304	8717696	32	86	30	80	27	74	10 – infinity

Table 2 Input Mesh Characteristics

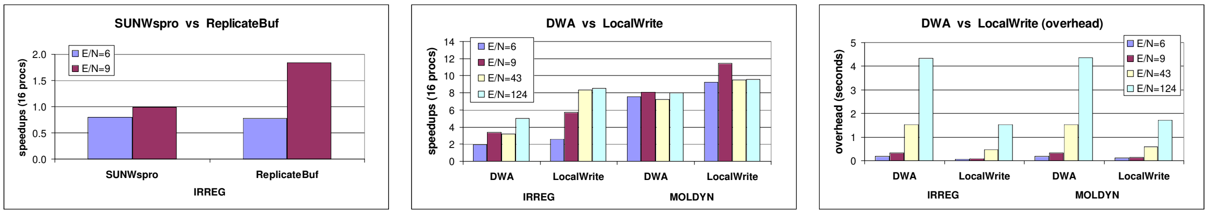

Figure 8 Comparisons with SUNWspro and DWA

sor. To measure locality we use the percentage of interior iterations (those accessing only local data) among all iterations. Finally, *adaptivity* is the rate at which data access patterns change during program execution. One way to present adaptivity is to specify adaptation interval. *Static* applications have infinite adaptation intervals, resulting fixed data access and communication patterns.

Table 2 summarizes input characteristics. These inputs are generated using MOLDYN with various *cutoff* radii. If two nodes are close enough to fall within *cutoff* radius, an edge is connected between the two nodes. For our experiments, we need a standard baseline input to perform comparisons. We chose for the base input of connectivity of 124, locality of 30%, and adaptivity set to none.

3.3 Comparison with Commercial Compilers

We begin by examining how successful commercial compilers are in automatically parallelizing irregular reductions. The leftmost graph in Figure 8 presents speedups obtained with the SUNWspro Fortran compiler, compared to REPLICATEBUFS. For denser inputs, code generated by SUNWspro would crash, so we compare only sparse inputs. We found neither approach achieves speedups when connectivity is six. but REPLICATEBUFS is able to achieve some speedups for when connectivity reaches nine. Results show commercial compilers are not yet very effective in parallelizing irregular reductions.

3.4 Comparing LocalWrite Implementations

Next, we compare the performance of the two implementations of LOCALWRITE described in Section 2.5. The middle graph in Figure 8 presents speedups of DWA and LOCALWRITE (with local index arrays) for two applications as connectivity varies. The rightmost graph in Figure 8 displays the cost of inspectors for both DWA and LOCALWRITE. Results indicate LOCALWRITE has both less expensive inspectors and achieves better speedups. Performance is improved from 20–160%, on average speedups are raised 56%. For the remainder of this section, we will thus use LOCALWRITE to refer to the more efficient implementation.

3.5 Shared-Memory Speedups

We now compare the effectiveness of all four parallelization techniques, beginning with speedups for the baseline input. Figure 9 shows speedups for each reduction technique as the number of processors increases. Speedup is measured along the y-axis and x-axis represents the number of processors. We see that overall, the performance of LOCALWRITE improves the most as the number of processors increases. Even with replicated computations, it scales best since it improves locality and has no conflict writes, avoiding any synchronization. In modern architectures interprocessor communication may cost more than replicating computations.

REPLICATEBUFS displays good performance since the connectivity (E/N ratio) is relatively high ($E/N = 124$), but the overhead of unused elements on 16 processors becomes larger and begins to degrade performance. The performance of SELECTPRIV is comparable to that of LOCALWRITE up to 8 processors, but its performance on 16 processors also begins to slow down. Overhead for increased number of boundary iterations and communication cost over more processors prevent SELECTPRIV technique from scaling well. The performance of SYNCHWRITE shows poor scalability. As we previously discussed, underutilizing processors and increased synchronization cost are major factors prohibiting its scalability. All speedup

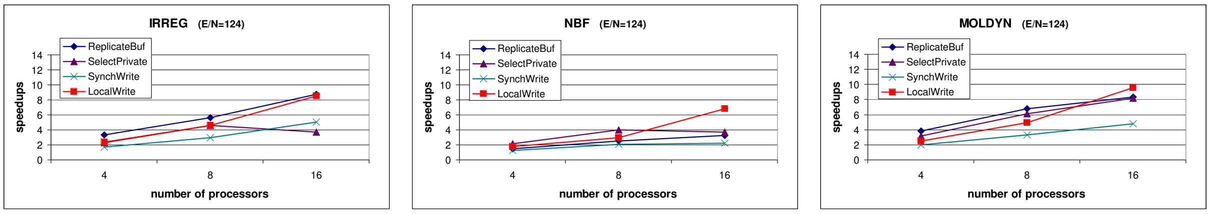

Figure 9 Speedups for Baseline Input

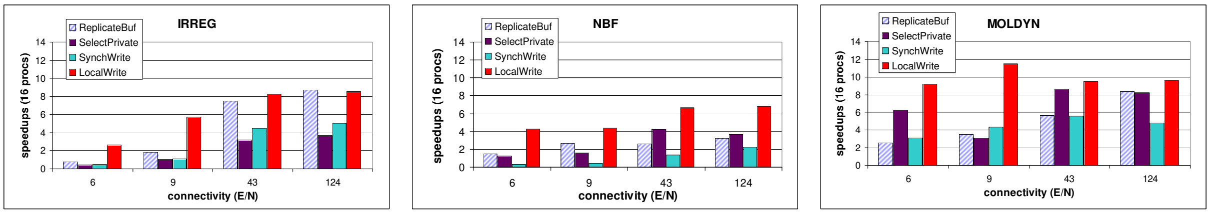

Figure 10 Impact of Connectivity

numbers in the later experimental section are calculated comparing against the performance of baseline inputs on uni-processor.

3.5.1 Impact of Connectivity

Figure 10 presents 16 processor speedups for REPLICATEBUFS, SELECTPRIV, SYNCHWRITE, and LOCALWRITE as connectivity varies. Speedup is measured along the y-axis. The x-axis displays connectivity measured as the edge/node ratio. Since the amount of computation is proportional to the number of edges, speedups generally increase as inputs become denser (i.e., the edge/node ratio increases) since we hold the number of nodes fixed. The amount of communication is proportional to the number of nodes. Since the number of nodes is fixed, computation/communication ratio increases as inputs becomes denser. LOCALWRITE clearly outperforms other techniques for sparsely connected meshes. Since LOCALWRITE avoids communication by replicating a certain portion of computation, its speedups degrade the least for sparsely connected meshes. In contrast, REPLICATEBUFS speedups drop very quickly for lower connectivity, since the number of unused elements in its buffers will rise.

3.5.2 Impact of Locality

For our baseline input, the percentage of interior iterations is 27–32%, which causes high overhead for SELECTPRIV, SYNCHWRITE, and LOCALWRITE. To improve the locality of our applications, we applied recursive coordinate bisection (RCB) to recursively divides data nodes into partitions based on their geometric coordinates [2]. RCB, space-filling curves, and graph-partitioning algorithms have all been shown to improve locality in irregular scientific codes [1, 15, 10]. For our input data sets RCB improves the percentage of interior iterations to 74–90%. Figure 11 compares 16 processor speedups for baseline input and preprocessed input with RCB. Y-axis displays speedups and x-axis shows locality measured in the percentage of interior iterations. Each graph shows two sets of bars for each sparse and dense input.

LOCALWRITE, SELECTPRIV, and SYNCHWRITE are more affected by locality improvement than REPLICATEBUFS. By increasing interior iterations, LOCALWRITE needs to replicate fewer computations, improving speedups by 38% on average. SELECTPRIV also uses less hashing in the reduction loop and less communication when it collects private copies. SYNCHWRITE also benefits from improved locality, since expensive synchronized updates reduces. However, total number of synchronizations remains same as baseline input. Meanwhile, REPLICATEBUFS is less sensitive to locality. Due to increased locality it may access local buffer with more locality, but the amount of work in initialization and accumulation of the whole local buffer remains same. Results show locality optimizations can greatly improve the performance of reductions parallelized using LOCALWRITE.

3.5.3 Impact of Adaptivity

Many irregular scientific programs are adaptive, meaning their data access patterns change over time [14]. Parallelization techniques which use inspectors are affected because the inspectors must be rerun. Overall performance then becomes dependent on the adaptation rate (interval between changes). Techniques which have less expensive inspectors achieve better performance for adaptive codes.

Figure 12 presents the cost of inspector for each reduction technique relative to the cost of a single iterations of sequential execution. It helps explain how inspector cost affect the speedups for adaptive codes. REPLICATEBUFS does not use inspectors, so it does not have any overhead. Measurements show the inspector for LOCALWRITE is less expensive than those needed for SELECTPRIV and SYNCHWRITE.

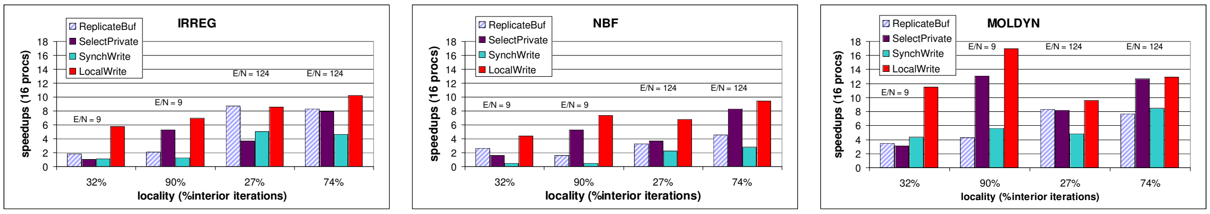

Figure 11 Impact of Locality

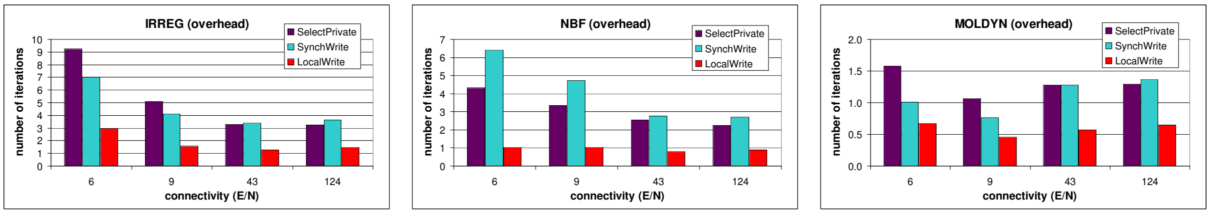

Figure 12 Overhead of Inspectors

Figure 13 displays speedups when adaptation interval changes. Once applications adapt, inspectors need to be performed again. If adaptation occurs too often, the cost of inspectors can not be amortized fully. Infinite interval represents static computation in the graph. Our experiment results show LOCALWRITE is fairly effective even for adaptive computations. For many applications it outperforms REPLICATEBUFS even for programs where data accesses change after only tens of iterations.

4 Related Work

The importance of identifying and parallelizing reductions in scientific applications is well established [13, 16]. Irregular reductions have been recognized as being particularly vital. The inspector-executor paradigm was first developed as a general scheme to optimize irregular applications for message-passing multiprocessors and used in the CHAOS run-time system [4, 14]. Compiler techniques were developed to automatically generate calls to the run-time routines [11]. Researchers have also developed techniques to improve the data locality of irregular computations, using dynamic copying of data elements [5] or partitioning computation and data [1, 15].

We earlier developed LOCALWRITE for use on a software DSM running on a message-passing multiprocessor [9]. We compared it to REPLICATEBUFS and explicit message-passing using in CHAOS. Results showed LOCALWRITE consistently outperformed REPLICATEBUFS, but was less efficient than explicit messages used by CHAOS, except for problems with low connectivity or high adaptivity. We also compared LOCALWRITE to REPLICATEBUFS on a 4-processor multiprocessor workstation, but found REPLICATEBUFS to be generally better.

Gutierrez *et al.* were the first to demonstrate the usefulness of LOCALWRITE on large-scale shared-memory multiprocessors. Their implementation of LOCALWRITE use an adjunct prefetch array to select which loop iterations which should be executed locally [6]. In comparison, we reordering the index arrays directly. They later developed SYNCHWRITE, a scheme to avoid computation replication using additional synchronization to separate computation into non-conflicting phases [7], but did not compare against recent techniques such as LOCALWRITE and SELECTPRIV.

Yu and Rauchwerger introduced a framework for dynamically selecting different parallelization alternatives, based on run-time information and a cost model [18]. They also developed SELECTPRIV based on selective privatization of conflicting elements, and compared its performance against REPLICATEBUFS and the version of LOCALWRITE described by Gutierrez *et al.* Their cost model was able to predict which technique to apply to achieve the best performance.

In comparison, our experimental study does not attempt to dynamically choose between techniques, but expands their study by including SYNCHWRITE, a more efficient implementation of LOCALWRITE, and varying number of processors. Our results indicate LOCALWRITE is more scalable and tends to achieve the best performance under most conditions; our version of LOCALWRITE can easily be integrated into their decision tree for dynamically selecting parallelization algorithms for irregular reductions.

5 Conclusions

In this paper we investigate compiler and run-time support for irregular reductions, the key computation for many scientific computations. We compared four techniques based on array privatization, selective privatization, local writes, and synchronized writes. We vary the connectivity, locality, and adaptivity of the applications to gain a better understanding of the advantages of different reduc-

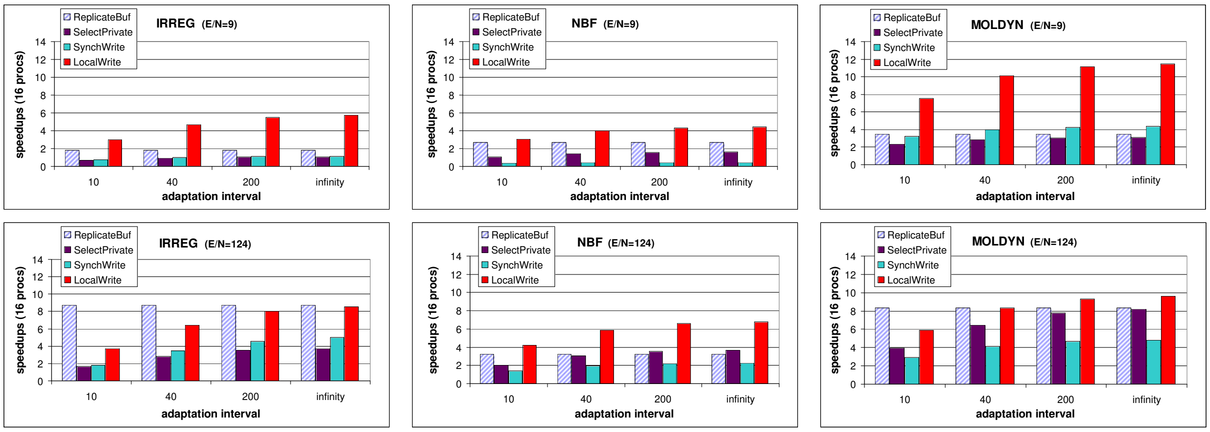

Figure 13 Impact of Adaptivity

tion parallelization techniques. We found LOCALWRITE, a technique which improves locality and eliminates synchronization at the expense of replicating computation, to achieve the best overall performance. As processors become faster relative to memory and network speeds, we expect the advantages of LOCALWRITE to increase. 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.

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] M. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Transactions on Computers*, 37(12):570–580, 1987.

[3] W. Blume et al. Polaris: The next generation in parallelizing compilers,. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[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] 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.

[6] 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 (Euro-Par'99)*, Toulouse, France, August 1999.

[7] 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.

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

[9] H. Han and C.-W. Tseng. 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

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