Compiler and Runtime Support for Irregular Reductions on a Multithreaded Architecture

Gary M. Zoppetti
Department of Computer and Information Sciences
University of Delaware,
Newark DE 19716
zoppetti@cis.udel.edu

Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University,
Columbus OH 43210
agrawal@cis.osu.edu

Rishi Kumar
Department of Electrical and Computer Engineering
University of Delaware,
Newark DE 19716
kumar@capsl.udel.edu

ABSTRACT
Computation from many scientific and engineering domains use irregular meshes and/or sparse matrices. The codes expressing these computations involve irregular reductions. Irregular reductions pose many challenges to parallel architectures and their compilers in terms of parallelization, locality management, and communication optimization.

Multithreaded architectures offer rich support for local synchronization, overlapping of communication and computation, and low-overhead communication and thread switching. Therefore, they appear to be promising for scalable parallelization of irregular reductions. This paper presents an execution model and a compilation strategy for supporting irregular reductions on a fine-grained multithreaded architecture. The key aspect of this strategy is that the frequency and volume of communication is independent of the contents of the indirection arrays. The performance obtained depends upon the architecture’s ability to overlap communication and computation and is largely independent of the partitioning of the problem.

We present experimental results from compiling three scientific kernels involving irregular reductions (mmw, euler, and moldyn) for execution on the EARTH fine-grained multithreaded architecture. On mmw, which does not involve any left-hand-side (LHS) arrays, we achieve near linear absolute speedups. For euler and moldyn, which do involve left-hand-side (LHS) irregular arrays, our strategy initially incurs some overheads, but the relative speedups are very good. In going from 2 to 32 processors, the relative speedups for euler were 9.28 and 10.36 on its two datasets, while the speedups for moldyn were 9.70 and 10.76 on its two datasets.

1. INTRODUCTION
Computation from many scientific and engineering domains use irregular meshes and/or sparse matrices. The codes expressing these computations involve irregular reductions. The main characteristics of irregular reduction loops are 1) elements of left-hand-side (LHS) arrays may be incremented in multiple iterations of the loop, but only using associative and commutative operations (these arrays are called reduction arrays), 2) there are no loop-carried dependencies, except on elements of reduction arrays, and 3) one or more arrays are accessed using indirection arrays.

It is very challenging to efficiently parallelize codes involving irregular reductions, especially on large parallel machines. Because of accesses through indirection arrays, communication and locality are hard to manage. Not only is the total communication volume large, but the communication requirements typically cannot be determined at compile time. It is also hard to efficiently allocate space for non-local elements.

Irregular reductions have been a major focus of research in the last few years. Considerable effort has been put in compiler and runtime support for parallelization of these codes on distributed-memory machines [2, 16, 17, 25], distributed shared-memory machines [9], shared-memory machines [10, 18] and for cache performance improvement on uniprocessors [7, 11, 19].

Achieving high performance on these codes on large distributed-memory machines, however, continues to be a difficult problem. Moreover, there are no effective solutions for parallelization of adaptive irregular reductions. In an adaptive irregular reduction, the elements of the indirection arrays are modified after every few iterations. This significantly increases the overhead associated with partitioning and runtime preprocessing routines [25], which have been critical for achieving locality, communication efficiency, and effective buffer management.

Recently, there has been much interest in multithreaded architectures. A multiprocessor based upon a multithreaded architecture supports multiple threads of execution on each processor. These architectures also support low-cost thread initiation, low-overhead communication, and efficient communication and synchronization between threads on different processors. Multithreaded architectures are considered a promising medium for scalable parallelization of irregular applications, where the frequent communication and syn-
chronization make parallelization difficult on conventional parallel machines.

In this paper, we present an execution model and compilation strategy for efficient execution of irregular reductions on a multithreaded architecture. The key idea in our execution model is that the frequency and volume of communication is independent of the contents of the indirection arrays. Thus, unlike other approaches to scalable parallelization of irregular reductions, our approach does not require mesh partitioning [3], array remumbering [20], or a high-cost inspector that itself requires communication between processors [25]. The performance depends upon the architecture’s ability to support low-cost communication and overlap communication and computation, and is largely independent of the problem partitioning. Thus, the performance can be obtained on adaptive problems, without paying the high overhead of frequently partitioning.

Code generation and execution using our strategy need both runtime processing and compiler analysis. We have developed a low-cost runtime processing method that we refer to as LightInspector. Unlike the inspector used in the inspector/executor approach [21], this runtime method does not require any communication between processors. Our compiler analysis processes irregular reduction loops to identify array sections accessed through indirection, and array sections used for indirect accesses. In some cases, it performs loop fission. Then, the analysis inserts a call to the runtime routine for each loop created after loop fission.

We have implemented our execution and compilation strategy using the EARTH-C compiler infrastructure [12]. Our strategy was evaluated on the EARTH multithreaded architecture developed by Guang Gao’s group [13]. We used three scientific kernels involving irregular reductions (wmm, euler, and meldyn) for our experiments.

The rest of the paper is organized as follows. Section 2 presents our execution strategy along with an example. We describe the runtime processing in Section 3 and the compiler algorithm in Section 4. The experimental evaluation of our work is presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2. EXECUTION STRATEGY

In this section, we give an example of a code that performs an irregular reduction. We then describe the execution strategy we propose for this class of applications. Details of the runtime processing and compiler analysis we use for supporting this execution strategy are described in the next two sections.

2.1 Irregular Reduction Loops

As stated earlier, an irregular reduction loop has the following three characteristics:

- Elements of lhs arrays may be incremented in multiple iterations of the loop, but only using associative and commutative operations (these arrays are called reduction arrays);
- There are no loop-carried dependencies, except on elements of the reduction arrays; and
- One or more arrays are accessed using indirection arrays.

Figure 1: A Simple Loop involving Indirection

Figure 1 shows a simple loop that performs an irregular reduction. In iteration i of the loop, the code makes two indirect references to the array X using IA(i, 1) and IA(i, 2). We say that the array X is accessed using two indirection array sections, denoted by IA(1, num_edges, 1, 1) and IA(1, num_edges, 1, 2). We use the standard triplet notation for describing array sections, where the first element inside the square brackets denotes the start point, the second element denotes the end point, and the last element denotes the stride.

Codes from many important scientific and engineering domains contain loops with such indirection array sections. The loop shown in Figure 1 is based upon a Computational Fluid Dynamics (CFD) code that uses unstructured meshes [5]. An unstructured mesh contains nodes and edges, with each edge designating two nodes as end points. In the figure, IA(num_edges, 2) lists the two nodes associated with each edge. The loop shown in Figure 1 iterates over the edges in the mesh. In each iteration of the loop, certain values (like velocity, momentum, etc.) associated with the two nodes are updated.

Molecular Dynamics codes contain loops of a similar structure [14]. Edges denote interaction between the molecules, and the nodes denote the molecules themselves. Loops iterate over the interactions, and update the values associated with the molecules.

Parallelizing irregular reductions on distributed-memory machines is a particularly difficult problem because significant attention needs to be paid to data partitioning, work distribution, and communication analysis and optimization. The commonly used technique is based upon the inspector/executor paradigm [21]. A runtime analysis phase (called the inspector) is used to determine the data partitioning, work distribution, and communication required between any two processors before and after the irregular reduction loop. The inspector is also responsible for managing the buffer for off-processor references on each processor.

While this method generally works well for static irregular reductions, there are no effective solutions for parallelization of adaptive irregular reductions. In an adaptive irregular reduction, the elements in the indirection arrays are modified after every iteration or every few iterations. This significantly increases the overhead associated with the use of inspectors.

2.2 Execution Strategy for Multithreaded Architectures

Recently, there has been much interest in multithreaded architectures. A multiprocessor based upon a multithreaded architecture supports multiple threads of execution on each processor. These architectures also support low-cost thread initiation, low-overhead communication, and efficient com-
communication and synchronization between threads on different processors.

Multithreaded architectures are considered very effective for irregular and communication intensive applications. Therefore, it is natural to consider multithreaded architectures for the efficient execution of irregular reductions. If the architecture can mask communication costs, much of the overhead associated with partitioning and runtime preprocessing can be reduced, resulting in scalable parallelization of irregular reductions.

We now present our proposed execution strategy for irregular reductions on a multithreaded architecture. The execution strategy for the loop shown in Figure 1 is given in Figure 2. The key idea in this strategy is to make the communication independent of the data and work distribution and the contents of the indirection arrays. Though the total volume and frequency of communication is high, our premise is that sophisticated support for overlapping communication and computation will enable high performance.

Initially, iterations and the data are divided between the processors in a block (or cyclic) fashion. Each processor is responsible for executing num_edges/num_procs iterations. When the code is run on num_procs processors, each processor executes the loop in k * num_procs phases. k is a small constant, and the most obvious choices of k are 2 and 4. For ease of presentation, we assume that the number of iterations assigned on each processor is a multiple of k * num_procs. Our actual implementation is general and is not based upon this assumption. In each phase, only a portion of the reduction array X is available locally on each processor.

On processor 0, the first num_nodes/(k * num_procs) elements are available locally during the 0th phase, the next num_nodes/(k * num_procs) elements are available locally during the 1st phase, and so on.

Conceptually, the reduction array X is divided into k * num_procs portions. Processor 0 owns the i th portion during phase i. Other processors start with different portions of the array in their 0th phase. On processor proc.id during phase phaseno, the portion of the reduction array owned is (k * proc.id + phaseno) mod (k * num_procs) and is denoted as reduc portion.

Consider the i th portion of the reduction array and let k = 2. If i is an even number, this portion of the reduction array is owned by processor (i div 2) in the 0th phase, processor ((i div 2) + 1) during the 1st phase, and finally by processor (((i div 2) + num_procs - 1) mod(num_procs)) during the (2 * (num_procs - 1))th phase. If i is an odd number, the i th portion of the reduction array owned by processor (i div 2) in the 0th phase, processor (i div 2 + 1) during the 1st phase, and finally by processor (((i div 2) + num_procs - 1) mod(num_procs)) during the (2 * num_procs - 1)th phase. Thus, for values of k greater than 1, any portion of the reduction array is owned by one of the processors during every k th phase only. This allows for more communication and computation overlap, but also increases the threading overhead since k threads per processor are used.

The runtime routine LIGHTINSPECTOR is used to divide the iterations assigned to a processor into different phases.
The details of the algorithm used in this routine are presented in the next section. We briefly explain the function-
ality of this routine here.

The phase to which an iteration is assigned depends upon the elements of the reduction array X updated in that iter-
ation, which in turn depends upon the contents of the indirect arrays. Consider iteration i and the values of IA(i, 1) and IA(i, 2). Let X(IA(i, 1)) belong to the reduction portion for phase r1, and let X(IA(i, 2)) belong to the reduction portion for phase r2. The following three possibilities exist:

- **r1 = r2.** In this case, iteration i is obviously assigned to phase r1. Since both X(IA(i, 1)) and X(IA(i, 2)) are within the range of X owned by this processor during this phase, both of them are updated during the execution of the main loop.
- **r1 < r2.** In this case, iteration i is assigned to phase r1. The element X(IA(i, 1)) is owned by the processor during phase r1 and is updated. However, the element X(IA(i, 2)) is not owned by the processor during this phase. The length of the array X is extended to create a remote buffer location. The value of IA(i, 2) is reassigned to point to this buffer location. During phase r2, an iteration of the second inner loop (shown in Figure 2) increments the value of X(IA(i, 2)) using the value in the buffer location. The arrays loop2_out, IB1_out, and IB2_out are used for managing the second loop.
- **r1 > r2.** In this case, iteration i is assigned to phase r2. The rest of the processing is analogous to the previous case.

3. **Runtime Processing**

As stated in the previous section, the runtime processing is responsible for 1) dividing the iterations on each processor into separate phases, 2) managing the buffer space for the reduction arrays, and 3) setting up the second loop which updates the reduction arrays using the values stored in the remote buffer.

If the reduction arrays are accessed using only a single distinct indirect array reference, runtime analysis can be used to examine the contents of this indirect array section, and partition the iterations into phases. All updates to the elements of the reduction arrays can be made when the element is owned, and therefore the buffer for off-processor references and the second loop are not required.

The more challenging case arises when the two or more indirect array sections are used to access and update the reduction arrays. For example, in the loop shown in Figure 1, two indirect array references, IA(1,1) and IA(1,2), are used for accessing reduction array X.

The routine is referred to as LightInspector because it is considerably lighter-weight than the inspector used in the inspector/executor paradigm [21]. Specifically, the LightInspector does not require any communication between processors. For ease of presentation, we have assumed that only two distinct indirect array references are used. The algorithm can be trivially extended to handle the case when more than two distinct indirect array references are used.

There are three main steps in the algorithm. During the first step, the reduction array references during each iteration are analyzed and the phases during which they are owned are determined. The smaller of these two phases is the phase to which this iteration is assigned.

The second step involves analyzing the iterations assigned to each phase. If both reduction array elements updated by an iteration are owned during this phase, the processing is relatively simple. If one of the reduction array elements updated by this iteration is owned during a future phase, more processing needs to be performed. A buffer element is allocated. Moreover, if this future phase is r, Iter2[r] is used to create an iteration of the second loop during the rth phase. The third step involves analyzing the iterations assigned to the second loop of each phase.

3.1 Example

Figure 3 illustrates the input and output of the LightInspector routine running on 2 processors (P0, P1) with k = 2. There are k * num_procs = 4 phases per processor. The input and output for only P0 is shown. The sample mesh contains 8 nodes and 20 edges. Each processor therefore owns 10 edges, numbered 0 through 9. Since there are 4 phases per processor, each processor owns 2 nodes per phase. For each reduction array, each processor must maintain 2 * k = 4 local buffers and 1 remote buffer. Each local buffer has 2 elements, and therefore, the remote buffer starts at location 8. The two indirect arrays that are inputed to the LightInspector are indir1_in and indir2_in. The LightInspector runs independently on each processor, creating the corresponding indirect array references indir1_out and indir2_out.

The first phase in runtime processing divides the iterations (edges) between the 4 phases. Phases 0, 1, and 2 are assigned 3 edges while phase 3 is assigned 1 edge. The inspector’s second pass places the edges in the new indirect arrays and modifies their values to point to local or remote buffer locations. Consider the 7th edge in the example. Since node 7 is owned during phase 0, this edge is assigned to the 0th phase. Node 4 is owned in a future phase, namely phase 2. This value must be changed to point to a buffer location, which in this case is 9.

The final phase of the runtime processing creates the arrays copy1_out and copy2_out. These two arrays are used during the second loop in each phase. They allow values associated with nodes owned in that phase to be updated from values that were previously buffered. For example, during phase 2, processor P0 needs to update the value associated with node 4 using the value buffered in position 9. Nodes 4 and 5 are owned during phase 2. In this example, the number of phases on each processor (4) is the same as the number of local buffers. Therefore, no renumbering of local nodes is required.

4. **Compiler Analysis**

We now describe the compiler analysis we perform to process the irregular reduction loops and generate code for our proposed execution strategy.

Our analysis assumes that no array is accessed through more than one level of indirection. Source to source trans-
formation techniques are available for replacing a loop with multiple levels of indirection with a sequence of loops such that each of them has only a single level of indirection [6]. We also cannot process loops in which an array is accessed through indirection in multiple dimensions.

The first phase of our compiler analysis extracts reduction
array sections and indirection array sections. A reduction array section is a regular section of an array that is accessed through an indirection array and updated using associative and commutative operations in the loop. An indirection array section is a regular section of an array that is used to access one or more reduction array sections.

After this analysis, the reduction array sections are divided into one or more reference groups.

Definition 1. Consider any irregular reduction loop. A set of reduction array sections belongs to the same reference group if they are all accessed using the same set of indirection array sections.

The significance of a reference group is that if all of the reduction array sections updated in a loop belong to the same reference group, the same LIGHTINSPECTOR can be used for partitioning the iterations of the loop into phases.

If all reduction array sections updated in a given irregular reduction loop do not belong to the same reference group, we apply loop fission [24] to break the original loop into a sequence of loops such that each of them only updates array sections belonging to the same reference group. Note that the irregular reduction loop does not involve any loop-carried dependencies, except for elements of the reduction arrays being updated in multiple iterations, so performing such loop fission is relatively straightforward. Some of the scalar values computed in the original loop may now be required in multiple loops, so temporary arrays may need to be introduced.

After loop fission, each loop can be easily processed to generate code for the execution strategy presented in Section 2. The indirection array sections are used to form the parameters to the LIGHTINSPECTOR. The reduction array sections are used to establish the communication.

5. EXPERIMENTAL RESULTS

In this section, we describe the compilation infrastructure we have used for our implementation, the multithreaded architecture we have used for evaluating our techniques, and the experimental results we have obtained. We present results from compiling and executing three scientific kernels involving irregular reductions. The three kernels are mvm, euler, and moldyn. mvm multiplies a sparse matrix and a vector. For this kernel, the reduction array is not accessed using indirection. While the execution strategy, memory management, and synchronization method presented in Section 2 are applicable to this code, the LIGHTINSPECTOR described in Section 3 is not required. Results from mvm are presented in subsection 5.3, and the results from euler and moldyn are presented in subsection 5.4.

5.1 EARTH-C Language and Compilation Infrastructure

EARTH-C is a dialect of C which allows programmers to specify parallelism and locality [12, 22]. Though the EARTH-C compiler targets a multithreaded environment, the programmer is not required to specify thread boundaries, thread synchronization, or communication as the compiler performs these tasks. The EARTH-C compiler is responsible for compiling any C code for correct execution; however, better performance can usually be achieved if the programmer uses additional directives for specifying parallelism and locality.

The work on the EARTH-C compiler was initiated by Laurie Hendren’s group at McGill University [12, 22, 26]. This compiler is based upon the McCAT compiler [8], which is a general framework for analyzing and transforming C programs.

The target language of the EARTH-C compiler is Threaded-C. Threaded-C is a low-level language targeting
any multithreaded system which supports a non-preemptive model of thread execution. Threaded-C extends C with directives to specify thread boundaries, communication, and synchronization between threads.

5.2 Multithreaded Architecture

EARTH (Efficient Architecture for Running Threads) [13] supports a multithreaded program execution model in which a program is divided into a two-level thread hierarchy of fibers and threaded procedures. Fibers are non-preemptive and are scheduled atomically using dataflow-like synchronization operations. These “EARTH operations” comprise a rich set of primitives and are initiated by the fibers themselves. They make explicit the control and data dependencies between different fibers, and fibers are scheduled according to the rule that a fiber is eligible to begin execution as soon as all dependence conditions have been met. Since fibers cannot be interrupted, the producer and consumer of a long-latency operation, such as a data transfer, should be in different fibers.

This synchronization model allows the use of local synchronizations between fibers using only those dependences that are actually relevant, rather than the use of global barriers. It also enables an effective overlapping of communication and computation by allowing a processor to execute any fiber whose data is ready when an existing fiber terminates after initiating a data transfer.

Conceptually, an EARTH node consists of an Execution Unit (EU), which executes the fibers, and a Synchronization Unit (SU), which determines when fibers are ready to run. The SU also handles communication between nodes. On each node the EU and SU communicate via two queues: a Ready Queue of fibers that the SU determines can execute and an Event Queue of synchronization and communication requests generated by the EU and handled by the SU.

Because EARTH fibers are non-preemptive, they are well-suited for execution on off-the-shelf processors. This is a big advantage, since the cost of developing and introducing a new processor architecture can be prohibitive. Machines can be designed for the EARTH execution model and developed in an evolutionary manner. One can begin with an off-the-shelf parallel machine, and gradually replace its stock components with specially designed hardware to support the EARTH operations.

Though the EARTH model has been emulated on many multiprocessors and clusters of workstations, we used the emulation based on the MANNA multiprocessor from GMD. The experiments in this study were performed using an accurate, complete cycle-by-cycle simulator of the MANNA’s i860XP processors, system bus, memory system and the interconnection network. The difference in the clock cycle counts between the simulator and the real MANNA have been measured and are typically less than 2% on real benchmarks. We obtained our results using the simulator’s manna-dual mode, where the 2 i860XP processors per node function as the EU and SU, respectively.

5.3 Matrix Vector Multiply Results

The mvm kernel we used was extracted from the NAS Conjugate Gradient (CG) benchmark. We experimented with three different matrices based upon the datasets distributed with the NAS CG benchmark: class W, class A, and class B. The number of rows for each of these matrices is 7,000; 14,000; and 75,000; respectively. The number of non-zero elements are 508,402; 1,853,104; and 13,708,072; respectively.

An important decision in using our strategy is the choice of the parameter k. The number of phases executed on each processor is \( k \times \text{num nodes} \). A larger value of k allows better overlap of communication and computation and increases the ability to tolerate load imbalance. However, it also results in higher threading overhead and loss of locality. We experimented with \( k = 1, 2, \) and 4.

The sequential versions were timed on one i860XP processor, and then the multithreaded version produced by the compiler was executed on 2, 4, 8, 16, 32, and in a few cases, 64 nodes.

Figure 4 shows the execution times for mvm on the class W and class A matrices. The sequential execution time on the class W matrix was 41.38 seconds. The absolute speedups on 2 processors were 1.97 with the \( k = 1 \) strategy and 1.98 with each of the \( k = 2 \) and \( k = 4 \) strategies. On 4 and 8 processors, the speedups from all three versions were slightly better than linear. Better cache utilization is the main reason for better than linear performance. On 16 processors, the speedups from the \( k = 2 \) and \( k = 4 \) versions were 15.23 and 15.02, respectively. The speedup from the \( k = 1 \) version was a bit lower at 14.26. On 32 processors, the speedups were 21.61 with the \( k = 1 \) strategy, 24.55 with the \( k = 2 \) strategy, and 23.42 with the \( k = 4 \) strategy.

The \( k = 2 \) version performed the best across all processors, with the \( k = 4 \) version coming in second. The \( k = 1 \) version provides no opportunity for overlapping computation and communication, and thus performed the worst. The difference between the performance of the \( k = 2 \) and \( k = 1 \) versions was 13.99% on 32 processors. The \( k = 4 \) version did not produce further gains from the ability to overlap communication and computation, and also incurred some additional threading overhead. It performed slightly worse (at most 4.84%) than the \( k = 2 \) version.

The performance on the class A matrix followed the same trend. The sequential execution time was 154.55 seconds. The absolute speedups on 2 processors were 1.94 with the \( k = 1 \) strategy, and 1.95 with each of the \( k = 2 \) and \( k = 4 \) strategies. Again, on 4, 8, and 16 processors the speedups were slightly better than linear for all versions. On 32 processors, the speedups were 28.41 with the \( k = 1 \) strategy, 30.65 with the \( k = 2 \) strategy, and 30.21 with the \( k = 4 \) strategy.

As with the class W results, the \( k = 2 \) version performed slightly better than the \( k = 4 \) version, and they both performed significantly better than the \( k = 1 \) version. On 32 processors, the performance difference between the \( k = 2 \) and \( k = 1 \) versions was 7.90%. To further examine the performance trend, we also executed the code on 64 processors. Here a difference of 15.31% was noted. The maximum difference between the \( k = 2 \) and \( k = 4 \) versions was 3.48% on 64 processors.

Figure 5 shows the execution times for mvm on the class B matrix. Because of memory constraints, we were unable to execute both the sequential version and the parallel version on 2 processors. Since a class B matrix is quite large (75,000 rows), we also obtained timings for 64 processors.
5.4 Euler and Moldyn Results

5.4.1 Datasets and Experiment Design

euler and moldyn contain irregular reductions in which the lhs reduction arrays are accessed using indirect. Unlike mwm, these two codes require the runtime analysis we presented in Section 3. euler is derived from a Computational Fluid Dynamics (CFD) code [5] and moldyn is derived from a molecular dynamics code [14].

We used two datasets for each of these codes. The two meshes used for euler had 2,800 nodes and 17,377 edges, and 9,428 nodes and 59,863 edges, respectively. moldyn was evaluated using problems with 2,916 molecules and 26,244 interactions, and 10,976 molecules and 65,856 interactions, respectively.

All sequential versions were timed on one i860XP processor, and then the multithreaded version produced by the compiler was executed on 2, 4, 8, 16, and 32 nodes. All measurements were performed using 100 iterations of the time-step loop. For parallel versions, the inspector was executed once.

For codes with irregular reductions that involve lhs irregular accesses, one of the important decisions in using our strategy is the distribution of loop iterations, and the arrays associated with the iterations (like the arrays IA and Y in Figure 1). The most obvious choice is a block distribution, where num_edges/num_procs consecutive iterations are assigned to each processor. Data parallel languages like HPF [15] also allow a cyclic distribution in which the arrays or iterations are distributed among the processors in a round-robin fashion. We report performance from both these distributions.

As discussed for mwm, another decision in using our strategy is the choice of the parameter k. We experimented with k = 1, 2, and 4. Thus, we initially created six different variations of our basic strategy for euler and moldyn. The cyclic distributions generally outperformed the block distributions, and k = 2 performed better than k = 1 and k = 4. Therefore, we present detailed results from only 4 variations of the strategy, and omit the times for k = 1 and k = 4 using a block distribution. These four strategies will be referred to as 1c, 2c, 4c, and 2b, respectively.

5.4.2 Experimental Data

Figure 6 shows the execution times for euler on its 2K and 40K meshes. The sequential execution time on the 2K mesh was 7.84 seconds. The speedups of the 2 processor versions were 1.10 with the 1c strategy, 1.20 with the 2c strategy, 1.17 with the 4c strategy, and finally, 1.24 with the 2b strategy. The relative speedups in going from 2 to 32 processors were 7.12 with the 1c strategy, 9.28 with the 2c strategy, 8.49 with the 4c strategy, and finally, 6.78 with

Relative speedups for class B mwm were computed against the best 4 processor version, which was the k = 2 version.

Figure 4: Parallel Performance of mwm: Class W (left) and Class A (right)

Figure 5: Parallel Performance of mwm: Class B
the 2b strategy.

Two main observations from these results follow. First, the 2c version consistently did better than the 1c and 4c versions. The performance difference between the 2c and 1c versions was between 15% and 30%. This shows that the ability to overlap communication and computation is extremely important. The performance difference between the 2c and 4c versions was between 3% and 6%. This shows that the threading overhead for the 4c version was not offset by the potential for more overlap of computation and communication. In other words, \( k = 2 \) provided the right balance for masking latencies arising from communication, synchronization, and uneven workloads, while keeping the threading overhead reasonably low.

Second, in comparing the block and cyclic distributions (with \( k = 2 \)), the block distribution performed better by 5% on 2 processors and by 1% on 4 processors, while the cyclic distribution performed better on 8 or more processors. The cyclic version was faster by 16% on 8 processors, 23% on 16 processors, and 33% on 32 processors. We carefully analyzed the number of iterations assigned to each phase on all processors. A block distribution resulted in a significant load imbalance, whereas a cyclic distribution did not. We believe this is the main reason the cyclic distributions performed significantly better. On smaller configurations, a reasonably good load balance was possible with either distribution. Though we could not measure cache miss rates, we believe that the block distribution performs marginally better on small configurations because of better locality.

The performance on the 10K mesh followed the same trend. The sequential execution time was 29.07 seconds. The speedups of the 2 processor versions were 1.11 with the 1c strategy, 1.12 with the 2c strategy, 0.95 with the 4c strategy, and finally, 1.16 with the 2b strategy. The relative speedups in going from 2 to 32 processors were 7.62 with the 1c strategy, 10.36 with the 2c strategy, 9.95 with the 4c strategy, and finally, 6.94 with the 2b strategy. The 2c version consistently outperformed the 1c and 4c versions. The 2b version again performed marginally better on 2 and 4 processors, and the 2c version performed significantly better on 8, 16, and 32 processors.

Figure 7 shows the execution times for \texttt{moldyn} on its 2K and 10K datasets. The sequential execution time on the 2K dataset was 10.80 seconds. The speedups of the 2 processor versions were 1.30 with the 1c strategy, 1.19 with the 2c strategy, 1.15 with the 4c strategy, and finally, 1.11 with the 2b strategy. The relative speedups in going from 2 to 32 processors were 7.50 with the 1c strategy, 9.70 with the 2c strategy, 8.70 with the 4c strategy, and finally, 6.50 with the 2b strategy. The 2c version had the best performance on 4, 8, 16, and 32 processors. The 2b version was consistently slower by 20% to 30%. This is because the 2K dataset resulted in a severe load imbalance using a block distribution, even on small configurations. On 2 processors, the 1c version performed the best. This is because, on 2 processors, \( k = 1 \) results in the iterations being partitioned between 2 phases, and \( k = 2 \) results in the iterations being partitioned between 4 phases. Fewer phases can mean less copying from buffers and good locality, depending upon the dataset. On larger configurations, the \( k = 1 \) version's inability to overlap communication and computation resulted in lower performance than the \( k = 2 \) version.

The sequential execution time on the 10K dataset was 28.98 seconds. For this dataset, the two processor versions showed significant slowdowns compared to the sequential version. The “speedups” of the 2 processor versions were 0.82 with the 1c strategy, 0.57 with the 2c and 4c strategies, and finally, 0.56 with the 2b strategy. We believe that the loss of locality is the main reason for the performance on the 2 processor versions. The level of performance degradation is dataset dependent, and turns out to be much more significant on this dataset, as compared to the two meshes used for \texttt{euler} and the 2K dataset used for \texttt{moldyn}. The relative speedups in going from 2 to 32 processors were quite good. They were 8.42 with the 1c strategy, 10.76 with the 2c strategy, 10.51 with the 4c strategy, and finally, 9.15 with the 2b strategy. Like the 2K results for \texttt{moldyn}, the 1c version gave the best performance on 2 processors, but was significantly slower than the 2c version on larger configurations. Two interesting observations on this dataset are 1) the 4c version is marginally faster than the 2c version on 8 and 16 processors, and 2) the 2b version is the fastest version (by a small fraction) on 4 and 8 processors. The load balance achieved with the cyclic distribution is not very good for this dataset. As a result, the 4c version performed better on some configurations because of its ability to tolerate load imbalance.

5.4.3 Discussion

We now make several observations from the experimental
results obtained for euler and moldyn. A cyclic distribution generally performed better on the datasets we used, except on a small number of processors. This is because the number of iterations assigned to different processors for a given phase was roughly the same with the cyclic distributions, but quite uneven with the block distributions. Also, the $k = 2$ versions performed better than the $k = 4$ and $k = 1$ versions. This shows that the ability to overlap communication and computation is extremely important, but the overlap can be achieved by having only $2 \times \text{num\_procs}$ phases on each processor.

Our execution strategy incurs substantial initial overheads. The best absolute speedup on 2 processors was 1.30, and the worst was 0.56. Though we could not measure the cache miss rates, it is clear that the partitioning of iterations into phases results in loss of locality, which leads to the overhead. However, the relative speedups in going from 2 to 32 processors were very good. This establishes that our strategy successfully exploits a multithreaded system’s ability to overlap communication and computation and hide communication latencies.

We now briefly compare our results with the results obtained on compiling irregular reductions on conventional distributed-memory machines. Though it is always hard to make any comparisons across hardware platforms, we believe that comparing the speedups can point to relative benefits of the strategies. In the earlier work of Agrawal and Saltz [1, 2], euler was used as a benchmark with the same 2K and 10K meshes. On an Intel Paragon, after using partitioning and communication optimizations, almost no speedups were obtained on the 2K mesh, and a relative speedup of 8 was obtained in going from 2 to 32 processors on the 10K mesh. On a 32K mesh, nearly perfect speedups were obtained. Without the use of partitioning, our results are significantly better on the 2K mesh and comparable on the 10K mesh. We did not experiment with the 32K mesh because we did not observe significantly better speedups in going from the 2K to 10K mesh. In summary, without the use of partitioning, our scheme is significantly better on small datasets, and comparable on medium-sized datasets. This is because the ability to overlap communication and computation is critical when a relatively small dataset is parallelized on a large configuration. The cache performance becomes a more significant issue with larger datasets, and therefore, our strategy does not result in better speedups on larger datasets.

Overall, we believe that our experimental results validate our main hypothesis, which is that good speedups can be obtained on a multithreaded system without the use of expensive partitioning schemes. This shows that multithreaded systems are very well-suited for adaptive problems. We plan to experiment with adaptive problems in our future work.

6. RELATED WORK

This work has been strongly influenced by previous joint work between two of the authors of this paper, Agrawal and Kumar, and Theobald et al. [23]. The previous work focused on hand-parallelizing a sparse matrix-vector multiply (MVM) and the NAS Conjugate Gradient (CG) code on the EARTH multithreaded architecture. The work presented here is different in two important ways. First, we have looked at irregular reductions that involve reduction arrays accessed through indirection arrays. Sparse MVM and CG do not have this characteristic, and therefore do not require the runtime analysis we have presented in Section 3. Second, we have augmented the compiler to generate code for the multithreaded architecture. The memory management and synchronization schemes used in our strategy are very similar to those used for sparse MVM and CG in the previous work.

Bokhari and Mavris have ported an unstructured mesh code on the Tera corporation’s MTA machine [4]. They also observe that multithreaded architectures are an excellent fit for unstructured meshes. In the Tera multithreaded architecture, thread execution is preemptive. Such a design significantly eases the compilation task, but such machines cannot be built from off-the-shelf uniprocessors.

Irregular reductions have been a major focus of research in the last few years. Considerable effort has been put in compiler and runtime support for parallelization of these codes on distributed-memory machines [2, 16, 17, 25], distributed shared-memory machines [9], shared-memory machines [10, 18] and for cache performance improvement on uniprocessor machines [7, 11, 19]. The execution strategy we use is different from all of the above approaches in that the performance does not depend on partitioning or array renumbering. As described in Section 3, the runtime processing we use is very different and much lighter-weight than the runtime processing used in the inspector/executor approach [25]. Our com-
piler analysis is also different because of the need to form groups of reduction array sections on the basis of the indirect array sections used to access them, and the need to perform loop fission.

The EARTH-C compiler project was initiated by Laurie Hendren at McGill University and has been continued at the University of Delaware. Previous work as part of this project includes a thread generation algorithm [12], a set of optimizations like loop invariant code motion and common subexpression elimination that use the results of heap and connection analysis [22], aggressive communication optimizations like message aggregation and eliminating redundant communication [26], and a number of static analysis techniques to improve thread granularity [27]. These efforts targeted pointer-based recursive programs, and none of the work applies to parallelization of irregular reduction codes, which use arrays and iterative constructs.

7. CONCLUSIONS AND FUTURE WORK

We have presented an execution strategy, with associated runtime and compilation techniques, for supporting irregular reductions on a fine-grained multithreaded architecture. The key aspect of this strategy is that the frequency and volume of communication is independent of the contents of the reduction arrays. This means that expensive mesh partitioning or other communication optimization techniques are not required, making our strategy well-suited for adaptive irregular problems.

We have presented experimental results from compiling three scientific kernels involving irregular reductions. On the warp kernel, we have achieved near linear speedups. For the other two kernels, easier and moldyn, our scheme involved initial parallelization overheads but the relative speedups in going from 2 to 32 processors were very good. This establishes that our strategy successfully exploits a multithreaded system’s ability to overlap communication and computation and hide communication latencies. The results on small datasets are particularly impressive, because the high communication to computation ratio on these datasets (even after performing mesh partitioning) has resulted in poor speedups on conventional distributed-memory machines.

In our future work, we plan to develop an incremental version of the LIGHTINSPECTOR and experiment with adaptive irregular problems.

Acknowledgments

We would like to thank the ACAPS group, led by Laurie Hendren at McGill University, for providing us with a copy of the EARTH-C compiler. This work would not have been possible without the support of the CAPS group, led by Guang Gao and Kevin Theobald in the Electrical and Computer Engineering Department at the University of Delaware. The CAPS group developed the EARTH system and frequently helped us during our experiments. The authors also thank Chau-Wen Tseng and Hwaanoo Han at the University of Maryland for providing us with a copy of the moldyn code and datasets.

8. REFERENCES