Compiling Several Classes of Communication Patterns on a Multithreaded Architecture

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

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

Guang Gao
Department of Electrical and Computer Engineering
University of Delaware, Newark DE 19716
ggao@capsl.udel.edu

ABSTRACT
Communication optimizations play a crucial role in performance of parallel applications which are compiled and executed on distributed memory machines. Multithreaded architectures can support multiple threads of execution on each processor, with low-cost thread initiation, low-overhead communication, and efficient data transfer and synchronization between threads on different processors. These mechanisms can be used for achieving an effective overlap between communication and computation, and therefore, good performance on communication intensive parallel applications.

We focus on generating correct and efficient multithreaded code for array based programs that involve different classes of communication patterns. We consider producer-consumer, scalar reductions, and near-neighbor communication patterns. We describe multithreaded programming methodologies suitable for handling loops with each of these patterns. We further show how a compiler can generate threaded code for loops with such patterns.

We present experimental results from two benchmark programs, CG, and Tomcat. Our results show that: 1) the compiler generated multithreaded code achieves high performance, not previously seen from distributed memory compilers, and 2) the performance of compiler generated code is comparable to the performance of hand-written multithreaded codes.

1. INTRODUCTION

Communication volume and efficiency are major issues in extracting high performance from codes compiled for execution on distributed memory multiprocessors. With the increasing popularity of cluster of workstations, where communication latency can be relatively high, communication becomes an even more important issue for the compiler. A number of techniques have been used in the past for improving communication performance, including compiler and runtime analysis for communication aggregation, and static analysis for overlapping communication and computation.

One promising approach for improving communication performance is based upon multithreading. A multiprocessor based upon multithreading supports multiple threads of execution on each node. It hides communication and synchronization costs by letting the processor switch to other thread when a long latency operation is encountered. Multithreaded systems based on dataflow model such as EARTH [11, 15], offer a further benefit of permitting local control between producers and consumers of data rather than expensive global barriers.

While multithreaded systems like EARTH can deliver high performance on threaded codes, automatically generating such multithreaded code remains an open problem. In our research, we are focusing on generating efficient multithreaded code for array-based programs with several classes of communication patterns. Our compiler processes a shared memory code written in a dialect of C called Earth-C [9], and generates Threaded-C code. Threaded-C [14] is an explicitly multithreaded dialect of C developed specifically for programming EARTH multithreaded system.

We focus on loops with three types of communication patterns: 1) near-neighbor communication, 2) global reductions, and 3) producer-consumer based patterns. In applications with one or more of the above patterns, the communication to computation ratio can be high and can degrade performance. Thus, we believe that multithreading can be exploited for achieving communication and computation overlap in applications with such patterns. For each of these three patterns, we first describe the style of multithreaded code that can effectively hide communication latencies. Then, we describe how the compiler can automatically generate such code.

We present experimental results from two benchmarks that involve one or more of the above three patterns. Our results show the following: 1) the compiler generated multithreaded code achieves high performance, not previously seen from distributed memory compilers, and 2) the performance of compiler generated code is comparable to the
Real A(n, n), B(n, n); ! data arrays
Real u, v, w, z;

for (i = 1; i < n-1; i++)
for (j = 1; j < n-1; j++)
A(i, j) = u * B(i, j-1) + v * B(i-1, j) + w * B(i+1, j) + x * B(i, j+1);

for (i = 1; i < n-1; i++)
for (j = 1; j < n-1; j++)
B(i , j) = A(i , j);  

Figure 1: Jacobi Kernel

performance of hand-written multithreaded codes.

The rest of the paper is organized as follows. Section 2 describes the three communication patterns that we are focusing on. Experimental results are presented in Section 3. We compare our work with related research efforts in Section 4 and conclude in Section 5.

2. COMMUNICATION PATTERNS

In this paper we are focusing on three types of communication patterns. Previous efforts on developing multithreaded codes for applications with such communication patterns have shown that certain programming methodologies help exploit multithreading and achieve high performance [11, 15]. In this section, we describe multithreaded programming methodologies suitable for each of the three communication patterns.

2.1 Near Neighbor Communication Pattern

Near neighbor communication patterns arise in regular problems or stencil computations. Figure 1 shows the kernel for near neighbor communication as found in the Jacobi benchmark.

Code generation for near-neighbor communication has been addressed in a number of distributed memory compilation projects. Depending upon the ratio of communication and computation, communication can inhibit scalable performance for codes with such patterns. Our goal is to examine if the compiler generated multithreaded code can help overlap communication and computation for such codes and improve performance.

Figure 2: Communication Setup in Near Neighbor Pattern

We exploit multithreading for such codes in the following way. Assume, for our discussion, that the problem is distributed along a single dimension only. For a problem size of NXN elements and p processors, each processor is assigned N/p rows. We further divide the rows assigned to a processor between a number of threads. The number of rows assigned to each thread is called thread granularity. Each thread is an independent unit of execution which can be started once all the data required for the execution is available.

Suppose we choose a thread granularity of 1 row per thread. In Figure 2 we present an illustration of threads and their dependencies for the Jacobi kernel on k processors. In this figure, normal arrows represent data transfer and synchronization between threads, and the bold arrows represent threaded function invocation. The thread corresponding to each row in this kernel needs to send its own data to its neighbors on top and bottom. For storing the data owned by other threads, we construct two temporary buffers on each thread to hold the updated values of the array B (Figure 1) received from the neighbors on top and bottom.

The threads corresponding to the first and the last row on each processor need data from other processors. Therefore, they have to wait for off-processor communication to be finished before their computation can be enabled. However, all other threads only need data available locally. Thus, they will typically be enabled sooner than the first and last threads. Computations on such threads can be performed while off-processor communication is carried out and the first and the last threads are enabled.

2.2 Global Reduction

A reduction is a commutative and associative operation that produces an array of n dimensions from an original array of m dimensions, where 0 ≤ n < m. Elements of the lhs may be incremented in multiple iterations of the loop, but only using associative and commutative operations. Reduction operations occur in many scientific kernels such as conjugate gradient (CG), tomat, among others. The commutative and associative operations include sum, product, max/min etc. Figure 3 shows example of a sum reduction.

A reduction like this involves computing a partial result at each node, followed by a global reduction where partial results from all processors are combined. On conventional parallel systems, the global reduction is implemented in one of the two ways. In the first approach, one processor is responsible for receiving partial results from all other processors, and then carries out the computation of the final result. The disadvantage with this approach is that one processor can become a bottleneck. In the second approach, a series of l synchronous phases are used. In each phase, a processor collects and combines partial results from N/2l other processors. Thus, a single processor is no longer a bottleneck. However, having l clearly separated phases of communication and computation can hinder performance.

We exploit multithreading to achieve overlap of communication and computation in global reduction by using a reduction-tree. The width and height of the tree can be dynamically chosen. The main difference from existing approaches is that no synchronization is done across nodes that are at the same level of the tree, i.e. the global reduction is not divided into marked phases of computation and communication. The multithreaded system’s ability to perform local synchronizations is used for this purpose.
Real A(n);
Real sum = 0;
for (i = 0; i < n; i++)
{
  S1: sum = sum + A(i);
}

Figure 3: SUM Reduction

Figure 4 shows the building block of reduction tree. This block has input buffers for incoming data and synchronization slots which count how many input values have arrived and trigger the computation when all required inputs are ready. Each node performs it synchronization locally, independent of other nodes at the same level of the tree. When the reduction thread runs, it sends the reduced value to that node's parent, and synchronizes with a sync slot in that parent. A second thread, with an independent sync slot and data input, is used when the parent wants to broadcast to the children. An important feature of this tree is that for a given height and width of the tree, the initialization needs to be done only once. Once the tree has been setup it can be used repeatedly by simply injecting new values into the tree.

In Figure 5 we present an example of sum reduction on an array A consisting of 16 elements. Assuming that the partial reductions are to be carried out on 4 nodes, each node is assigned 16/4 = 4 elements. Every node has two threads, the producer thread and the reducer thread. Producer threads compute the partial sum of 4 elements, and then pass the result to the reducer thread on the same node. These reducer threads are part of the reduction tree, and they pass the results to their parents, which continue the reduction operation till a single reduced value is obtained.

2.3 Producer Consumer Communication pattern

In this subsection, we focus on a generalization of the reduction tree discussed in the previous subsection. We refer to this pattern as a producer-consumer pattern. In such a pattern a node, referred to as the consumer, requires the data generated by some other node, which is referred to as the producer. The consumer typically acts as a producer for some other node in the system. A familiar example of producer-consumer relationship arises in the multithreaded solution to the sparse matrix-vector multiply problem.

Many scientific applications involve sparse matrices due to the inherent nature of the physical and chemical systems that constitute these problems. A common kernel in these applications is the product of a sparse matrix with a column vector to produce another column result vector. This kernel is the time and computation dominating portion for many applications, including the NAS benchmark conjugate gradient (CG).

Consider the operation \( A \cdot v = q \), where \( A \) is a sparse matrix and \( v \) and \( q \) are vectors of length \( n \). Without using any algorithmic techniques for partitioning of the problem set, the matrix \( A \) is divided into \( p \) regular strips with each strip assigned to one of the available \( p \) processors. The vector \( v \) is also divided into sections corresponding to strips of \( A \). During each phase of multiplication, each node \( i \) multiplies its own \( A_i \) and \( v_i \) producing a partial result \( q_i \) of size \( n \). To produce the final result vectors \( q_1, \ldots, q_p \) have to be added. If a binary tree is used for reduction, the cost will be \( O(n \log p) \).

Alternatively, the computation and communication can be divided into \( p \) phases. The first two phases of this scheme are illustrated in Figure 6 (where \( p = 4 \)). During each phase, node \( i \) multiplies one part of its \( A \) with \( v_i \), producing a part of \( q_i \) with only \( n/p \) elements. This piece is then sent to the left neighbor. The starting position of nodes are staggered so that piece can be added to what the left neighbor produces in the next iteration as shown in Figure 6(b).

Since the communication and computation have been evenly divided between the processors, the cost of the global reduction is \( O(n) \). However, this scheme cannot typically be implemented efficiently in conventional parallel systems. Having \( p \) distinct phases of communication and computation can severely hurt performance. With the use of multithreading, we implement this basic scheme as follows. Instead of \( p \)
phases, we use 2p phases. Phase i on a processor is dependent upon the data that is used by the phase i – 2 on the adjacent processor. Two threads are used on each processor, for odd and even numbered phases respectively. Once all the data required for execution of a thread is available, it is enabled. No global synchronization operations are performed.

2.4 Compiler Code Generation

The first goal of our compiler was to automatically detect the classes of communication patterns described in the previous section. Recognizing reductions and near-neighbor communication patterns is a well studied problem in compiler literature [3, 4, 7, 10, 16]. The techniques implemented in our compiler are very similar to the techniques used by previous projects and are not described here. The recognition and processing of codes that can use the producer-consumer pattern is a topic for a separate paper [17]. More details of our compiler implementation are also available from Kumar’s masters thesis [12].

3. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we evaluate the performance obtained from our compiler. We have experimented with two programs, NAS conjugate gradient C6 and Tomcatv. Conjugate Gradient has a combination of producer-consumer and global reduction patterns. Tomcatv comprises of both near-neighbor and global reduction patterns. We have chosen C6 and Tomcatv because they have been hard for compilers to parallelize on distributed memory machines.

We have used the emulation of EARTH model based on MANNA in our experimental study. MANNA 2.0 (Massively parallel Architecture for Numerical and Non-Numerical Applications) was developed at GMD-FIRST in Germany [6]. The configuration available at University of Delaware is a 20-node MANNA consisting of two crossbars with 10 nodes apiece and two bidirectional links between them. To obtain experimental results for more than 20 nodes, Kevin Theobald has developed an accurate simulator for the MANNA hardware called SEMI (Simulator for EARTH, MANNA and the i860) [15, 13]. SEMI is a complete cycle-by-cycle simulator of the MANNA’s i860XP processors, system bus, memory system and the interconnection network, and it can simulate up to 120 nodes. 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 [15].

3.1 Conjugate Gradient Results

Conjugate gradient (CG) comprises a sparse matrix-vector multiply (MVM), which requires the producer-consumer pattern, and several loops requiring scalar global reductions.

The matrix sizes used in our experiment are based upon the problem sizes associated with the distribution of the NAS conjugate gradient (CG) benchmark. Here we report results for the class A (14,000 rows) problem size. We timed the sequential versions on one i860XP processor of the MANNA multiprocessor, and then the multithreaded versions were executed on 1, 2, 4, 8 and 16 nodes under MANNA-Dual configuration. SEMI was used to extend the timing results to 32 nodes. The sequential execution time for class A was 425.651 seconds. The absolute speedup was 1.64 on 2 nodes for hand written version and 1.69 for compiler generated version. Hand written code gave an absolute speedup of 3.26 on 4 nodes, 7.31 on 8 nodes, 14.49 on 16 nodes, and 24.77 on 32 nodes, while the compiler generated code gave absolute speedups of 3.19, 7.07, 12.74 and 20.65 for 4 nodes, 8 nodes, 16 nodes, and 32 nodes, respectively. Both the versions showed good scalability with hand written code showing a relative speedup of 31.01 on 32 nodes and the compiler generated version having a relative speedup of 27.64. The absolute and relative speedups for class A are shown in Figure 7.
The difference in performance of hand written and compiler generated code occurs due to two reasons. First, the PGI (Portland Group, Inc.) compiler used for compiling applications on MANNA does not do a good job of optimizing outer loop of the 2 dimensional matrix vector multiplication kernel. Hence the hand written code uses a core multiplication loop written in assembly language for better performance. Our compiler does not perform such low-level optimizations, and simply uses the existing low-level compiler for code generation. Second, the EARTH-C compiler breaks structure references, and multi-dimensional array references in simpler forms which introduces temporary variables in the compiler generated code.

3.2 Tomcatv Results

Tomcatv contains near neighbor communication and a global reduction. We used a dataset with 256 rows for our experiments.

We created versions of this benchmark with three different thread granularities. The first version of hand written and compiler generated codes had a thread granularity of 1, i.e. each row was assigned to a separate thread. The second version had a thread granularity of 2. Due to decrease in the number of threads, opportunities for overlapping communication and computation are reduced. In the 3rd version each processor executes only one thread. All the n/p rows on a particular processor are assigned to a single thread. We report the performance of all 3 compiler generated versions and the best hand written version, which in all cases turned out to be the version with thread granularity of 1.

Figure 8 shows the timing results for the problem size 64. The speedups for best hand written and compiler generated versions are shown in Figure 9. For all 3 hand written versions the corresponding compiler generated versions performed within 30%. The sequential execution time for problem size 64 was 2857 ms. The absolute speedup of hand written code with thread granularity of 1 was 1.71 on 2 nodes, while speedup of compiler generated version was 1.33 on 2 nodes. Absolute speedup of hand written code was 3.54 on 4 nodes, 7.16 on 8 nodes, 13.91 on 16 nodes and 24.50 on 32 nodes, while that of compiler generated version was 2.73 on 4 nodes, 5.51 on 8 nodes, 10.94 on 16 nodes, and 20.77 on 32 nodes. Both the versions show good scalability with a relative speedup of 27.51 for hand written code and 30.67 for compiler generated code.

With the compiler generated code of thread granularity of 2, degradation in performance was observed. The absolute speedup was 1.13 on 2 nodes, 2.46 on 4 nodes, 4.88 on 8 nodes, 9.78 on 16 nodes, and only 18.14 on and on 32 nodes. With the increase in thread granularity there is an imbalance between a thread finishing its work, and getting new buffer values from its neighbors to start the next round of computations. As a result the amount of time spent waiting is increased which increases the total execution time.

With the compiler generated code in which all the rows on a particular processor are in same thread, each thread must finish all the rows assigned to it before it can enable rows on its neighbors. Thus there is no overlap of computation and communication. The absolute speedup on 2 nodes was 1.14, on 8 nodes it was 2.23, while on 32 nodes it was only 16.47. But all the versions showed good relative speedup, with this version having a relative speedup of 28.55 on 32 nodes.

4. RELATED WORK

Over the last decade, a number of projects have focussed on compiling array based codes on distributed memory machines [1, 3, 10, 16]. These projects have developed, implemented, and evaluated a number of techniques for an-
alyzing and optimizing communication. In comparison to these efforts, our work is unique in using multithreading for optimizing communication performance.

One approach to improving performance of long latency operations is to explicitly overlap them with computation. For overlapping communication with computation, Hanxleden and Kennedy have developed a general framework, which includes performing early placement of sends and late placement of receives [8]. Chakrabarti et al. have developed dataflow techniques for achieving overlap of communication and computation [5]. Agrawal has developed an IBPCP (Interprocedural Balanced Code Placement) framework for achieving overlap across procedure boundaries [2]. Again, our work is different in using multithreading to achieve overlap between computation and communication. As compared to the explicit placement of split-phase operations required in these approaches, multithreading allows runtime scheduling of threads and is successful in achieving overlap even for applications where compiler analysis cannot explicitly place a split-phase operation.

5. SUMMARY

This paper reports our research in automatically generating multithreaded code for loop-based programs. We have shown how our compiler generates multithreaded code for loops with three different pattern types: global reductions, near-neighbor communication, and producer-consumer pattern.

The major observations from our experimental results are as follows. First, the compiler generated performance for C and Tomcat is very impressive. We are not aware of any comparable results from previous distributed memory compilers. Particularly noteworthy are the very close to linear relative speedups for C, and high speedups for Tomcat for small problem sizes. The difference between compiler-generated and hand-generated multithreaded codes was nearly 20% for C and 30% for Tomcat. Our analysis has shown that by implementing some standard scalar compiler optimizations in the low-level compiler, the difference between these versions can be bridged.

Overall, we believe that we have demonstrated multithreading to be an effective mechanism for achieving overlap of communication and computation that can be exploited well by compilers. In the future, we will like to extend our work to generate code with other communication patterns, and to evaluate them on other communication-intensive applications.

6. REFERENCES


