Developing a Communication Intensive Application on the EARTH Multithreaded Architecture

Kevin B. Theobald[1], Rishi Kumar[1], Gagan Agrawal[2], Gerd Heber[3], Ruppa K. Thulasiram[1], and Guang R. Gao[1]

[1] Department of Electrical and Computer Engineering, University of Delaware, Newark, DE 19716, USA,
{theobald,kumar,rthulasi,ggao}@capsl.udel.edu,
http://www.capsl.udel.edu
[2] Department of Computer and Information Sciences, University of Delaware,
agrawal@cis.udel.edu,
http://www.cis.udel.edu
[3] Cornell Theory Center, Cornell University, Ithaca, NY 14853, USA,
heber@tc.cornell.edu,
http://www.tc.cornell.edu

Abstract. This paper reports a study of sparse matrix vector multiplication on a parallel distributed memory machine called EARTH, which supports a fine-grain multithreaded program execution model on off-the-shelf processors. Such sparse computations, when parallelized without graph partitioning, have a high communication to computation ratio, and are well known to have limited scalability on traditional distributed memory machines. EARTH offers a number of features which should make it a promising architecture for this class of applications, including local synchronizations, low communication overheads, ability to overlap communication and computation, and low context-switching costs. On the NAS CG benchmark Class A inputs, we achieve linear speedups on the 20-node MANNA platform, and an absolute speedup of 79 on 120 nodes on a simulated extension. The speedup improves to 90 on 120 nodes for Class B. This is achieved without inspector/executor, graph partitioning, or any communication minimization phase, which means that similar results can be expected for adaptive problems.

1 Introduction

One of the most difficult challenges in parallel processing is obtaining high performance for a variety of applications in the presence of high communication and synchronization costs. Multithreaded architectures promise scalable performance for both regular and irregular applications. These systems hide communication and synchronization costs by letting a processor switch to a different thread when a long-latency operation is encountered, and by keeping the cost of this switching low. Multithreaded systems based on dataflow models of computation,

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 625–637, 2000.

such as EARTH [1, 2], offer a further benefit of permitting local control between producers and consumers of data rather than expensive global barriers.

This paper presents an important case study to examine the performance of sparse matrix vector multiply (MVM) on EARTH. Sparse MVM is an important and time-consuming kernel in sparse linear algebra problems, including Conjugate Gradient (CG). We have chosen this because it leads to a very high communication to computation ratio, when parallelized without *graph partitioning* [3] and/or an *inspector/executor* paradigm [4], due to the sparseness of the matrix, and does not perform well on conventional distributed memory machines. We show that even without these techniques, a multithreaded system with local synchronization, overlapping of communication and computation, and low-overhead communication and thread-switching can efficiently parallelize sparse MVM.

The goal is to compute a product $q = Av$, where A is an $n \times n$ matrix and v is a vector of length n. Typically, fewer than 1% of the elements of A are non-zero. A common representation for sparse matrices is Compressed Row Storage (CRS), in which only the non-zeroes of a row are stored, and a separate array (colidx) holds their column positions.

The NAS Parallel Benchmark suite [5] includes a version of CG without graph partitioning. Shared memory machines show reasonably good *relative* speedups (absolute speedups are not reported) [5]. For instance, the cache-coherent SGI Origin-2000 has a speedup of 28 on 64 nodes, while the speed of the non-coherent Cray T3E improves by 25 going from 2 to 64 nodes (single-node performance is not reported). The IBM SP-2 distributed memory machine is the most "off-the-shelf," as it has no hardware support for shared memory; its relative speedup is only 13 on 64 nodes. CG results are generally not reported for PC clusters as their relatively slow networks result in terrible speedups.

We have implemented sparse MVM on the EARTH multithreaded system (described in Sect. 2). In our code, the program on each processor is executed as a sequence of threads where the enabling of a thread is *event driven.* Point-to-point, split-phase style communication is performed between processors, generating events to trigger thread execution in a fully asynchronous manner. This, coupled with the low-cost spawning and termination of threads, contributes to highly scalable performance. Execution on each processor is not broken into separate computation and communication phases and global synchronization is not required. Since graph partitioning and inspector/executor are not used, the same approach can be used for parallelization of linear solvers for adaptive problems, i.e., problems where the matrix A is frequently modified.

We report speedups from different problem sizes and different EARTH configurations in this paper. With the NAS Class A sparse matrix (14,000 rows), we observe linear speedups on our 20-node MANNA distributed memory machine [6]. A simulated expansion of the hardware to more nodes yields an *absolute* speedup of 59 on 120 nodes for purely off-the-shelf systems on Class B (75,000 rows), rising to 90 on 120 nodes when a small chip specifically supporting the EARTH execution model is added. A series of experiments reveal the factors leading to high scalability in our implementation.

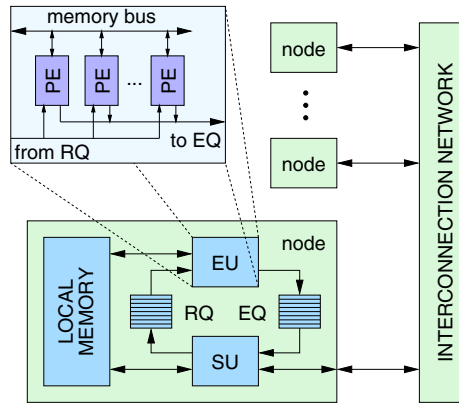


Fig. 1. EARTH Architecture

The rest of the paper is organized as follows. In Sect. 2 we review the details of the EARTH architecture. Our multithreaded approach for sparse MVM is described in Sect. 3. We present our scalability results in Sect. 4, an analysis of how the features of the EARTH system contribute to these results (based on additional experiments) in Sect. 5, and our conclusions in Sect. 6.

2 The EARTH Multithreaded Architecture

EARTH (Efficient Architecture for Running THreads) [1, 2] supports a multithreaded program execution model in which a program is divided into a two-level hierarchy of *fibers* and *threaded procedures*. Fibers are non-preëmptive and are scheduled atomically using dataflow-like synchronization operations initiated by the fibers themselves. These "EARTH operations" make the control and data dependences between fibers explicit, and fibers are scheduled by the rule that one is eligible to begin execution as soon as all relevant dependences have been met. Since fibers can't be interrupted, the producer and consumer of a long-latency operation, such as a data transfer, should be in different fibers.

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

Conceptually, an EARTH node (see Fig. 1) has an *Execution Unit* (EU), which runs the fibers, and a *Synchronization Unit* (SU), which determines when fibers are ready to run, and handles communication between nodes. There is also a Ready Queue (RQ) of fibers waiting to run on the EU, and an Event Queue (EQ) containing requests for EARTH operations, generated by fibers in the EU.

Because EARTH fibers are non-preëmptive, they are ideal for off-the-shelf processors. This is a big advantage, since the costs of developing and introducing a new processor architecture can be prohibitive. Machines can be developed for the EARTH execution model in an *evolutionary* manner [2]. One can begin with an off-the-shelf parallel machine, and gradually replace its stock components with hardware specially designed to support EARTH operations. In this paper (and previous studies) we consider four possible configurations:

Single: Each node has only one processor, which must alternate between the tasks of the EU and the SU.

Dual: Each node has two processors; one performs the EU tasks and the other emulates the behavior of the SU. The Ready and Event Queues are stored in memory shared by the two processors.

External SU: Each node has a regular off-the-shelf processor and a custom hardware SU. The SU can be built fairly cheaply, yet be optimized for performing the EARTH operations [2, 7]. The EU communicates with the SU through special memory addresses.

Internal SU: This is like the External SU, except that the CPU and SU cores are combined in one package. The interface is the same (memory addresses), but communication between them is off the main bus and hence faster.

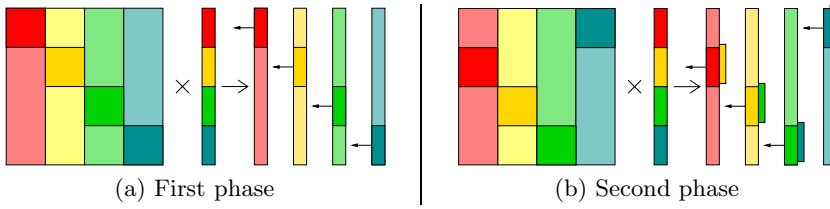
3 Multithreaded Implementation

We assume that A is too large to have a complete copy on every node, and needs to be divided among all p nodes. Unless one uses *algorithmic* techniques to reduce communication (see Sect. 1), the simplest way to divide MVM is to split A into p regular strips or blocks. Our algorithm divides A into vertical sections $A_1, \ldots, A_p$. The vector v is also partitioned into sections corresponding to the strips of A. During one multiplication, each node i multiplies its own A_i and v_i, producing a partial result q_i of size n. Neither A nor v have to move, but the vectors $q_1, \ldots, q_p$ must be added to produce the final answer.

The only communication required is the reduction of the components of q. The reference MPI implementation of the NAS CG code adds the q_i vectors using a binary tree. Therefore, the running time of the reduction is $O(n \log p)$.

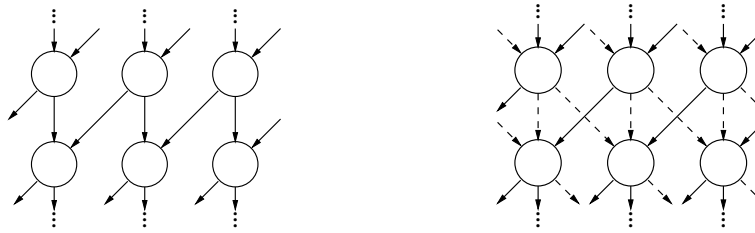
An alternative is to *pipeline* the reduction in a linear chain. The computation is divided into p phases; the first two are illustrated in Fig. 2 (where $p = 4$). During each phase, node i multiplies one part of its A_i with v_i, producing a part of q_i with only n/p elements. This piece is then sent to the left neighbor (mod p). The starting positions are staggered so that that piece can be added to what the left neighbor produces in the next iteration, as shown in Fig. 2(b).

The total communication burden is the same as for the binary tree. However, here it is evenly balanced among all nodes, so the reduction takes only $O(n)$. Furthermore, by pipelining the reduction, it can be effectively overlapped with the computation (if the architecture allows this).



(a) First phase (b) Second phase

Fig. 2. Pipelining of 4-Node MVM Reduction (First 2 Phases)



(a) Data flow (b) Data flow with syncs

Fig. 3. Implementation of MVM on EARTH

This pipelined reduction is a straightforward specialization of Cannon's algorithm [8] to one dimension. Its implementation in a conventional parallel machine, however, can be challenging because of the frequent communication steps. The high degree of pipelining can hurt performance on conventional coarse-grain parallel machines in at least some of the following ways:

1. If the overheads of sending a message are too high, the overheads become collectively much more significant with p phases than with $\log p$ phases.
2. If a global barrier is required to synchronize the communication of the pieces of q from one phase to the next, then any temporary load imbalance from one phase to the next will force all nodes to wait for the slowest node. (Such variances are likely in a sparse matrix.)
3. If separate communication and computation phases are required, the opportunity to overlap these is lost.

EARTH, on the other hand, is specifically designed for fine-grain synchronization, low-overhead communications, and asynchronous local control. It is therefore an ideal platform for this algorithm. In Sect. 5, we show quantitatively how these properties of EARTH contribute to the performance of MVM.

Fig. 3 shows how the MVM algorithm is transformed to an EARTH program. The computation is broken into a sequence of fibers (a). Each circle represents one fiber, which performs the multiplication of one $n/p \times n/p$ section of A with some v_i. A column of fibers (circles) runs on one node, and represents successive iterations on the same node. Arcs represent data and control dependences. The

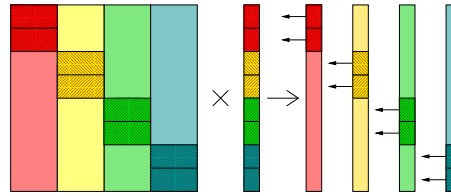


Fig. 4. Multithreading MVM

solid arcs represent the data (in this case, pieces of q_i between nodes), while the dashed arcs represent synchronization signals only. (In this program, all iterations are executed by one program fiber, which is repeatedly instantiated and doesn't need to "send" data to itself.)

On a machine with global barriers, this diagram would adequately describe the simple control structure of the program. However, we want to exploit the features of the EARTH program execution model, namely, multithreading, local synchronization between fibers, and overlapping of communication and computation. We use EARTH's *sync slots* to permit a fiber to start as soon as all required control and data dependences are met, which in this case means 1) the previous iteration on the same node has finished, and 2) the q_i block from the previous iteration has been received from the right neighbor.

However, there is a catch here. If a node always sends its q_i output to the *same* buffer on its left neighbor, the fiber running iteration j must wait until the fiber running iteration j on the left neighbor has finished reading the data from the buffer, or else data could be overwritten. Yet there is no signal path to inform the right neighbor when it is clear to send. Therefore, we add such paths, as shown in part (b). Furthermore, this synchronization can't occur within one iteration, since fibers in EARTH are atomic and non-preëmptive (one can't synchronize "part" of a fiber). There must be a downward movement of sync signals, as shown in (b). Therefore, we allocate *two* buffers in each node, and use one buffer on odd-numbered iterations and the other buffer on even iterations.

This implementation now allows local synchronization between nodes, but doesn't allow overlapping of communication and computation. The fibers in iteration j must wait for the fibers in iteration $j-1$ to finish and send their results. If the architecture has separate hardware for communications, then processors could be sitting idle waiting for the communication to complete. Since EARTH assumes separate communication hardware (a separate CPU or specialized Synchronization Unit), we want to take advantage of this feature.

To do this, we exploit the other major feature of EARTH: multithreading. We split each block multiplication into two halves, each of which produces half of the result vector. This is shown in Fig. 4. Each half is computed by a separate fiber. Now the top halves and the bottom halves of the block multiplications can occur concurrently, as long as each has its own buffers. Essentially, the program in Fig. 3(b) is replicated for each half.

The code studied here uses the CRS format described in Sect. 1, with a separate structure for each A_i. The algorithm was adapted to handle strips of different sizes, so the number of nodes doesn't need to divide n. The code was written in Threaded-C [1, 2], an explicitly threaded programming language, which extends ANSI-C with EARTH operations. Ordinary sequential code was used for the core block multiplication, which dominates the execution time. This routine is a 2-D loop, and our C compiler optimizes the inner loop at the expense of the outer. For the partitioned version, the overhead of the outer loop is much more critical, since the inner loop has far fewer iterations (as explained in Sect. 5). As our compiler lacks any flags or pragmas to favor the outer loop, the core was rewritten in assembly language (43 instructions).[1]

4 Scalability Results

The experiments in this study are based on the EARTH implementations for the MANNA [6]. This machine has 20 dual-processor nodes and can run the Single and Dual configurations listed in Sect. 2. Our experiments were run using SEMi, an accurate, cycle-by-cycle simulator of the MANNA's processors, system bus, memory and interconnection network [1, 2]. 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 have extended the simulator to model faster processors based on the CPU, bus speed and cache parameters of the PowerPC 620-based PowerMANNA, the MANNA's successor. Here we assume 200MHz processors, with each node having a 200MByte/sec connection to the network. The specialized SU hardware is also simulated, and its speed and interface characteristics are based on the existing MANNA hardware. This gives us confidence that the results obtained from simulating the specialized hardware are reasonably close to what could be achieved with real hardware. As a further check, the MVM code was run on the real MANNA up to 20 nodes for the Single and Dual configurations; the speedup results are nearly identical.

The matrices used come from the NAS Conjugate Gradient (CG) benchmark [5]. We used the Class A (n = 14,000) and B (75,000) problem sizes. These matrices contain 1.9 and 13.7 million non-zeroes, respectively.

Results for the two inputs on the 4 different EARTH configurations are shown in Fig. 5 and 6. The speedups shown are *absolute*, i.e., the parallel performance is compared against the *sequential* code rather than the single-node *parallelized* code. On 120 nodes, the speedups achieved for Class A on the Single, Dual, External SU, and Internal SU versions are 28, 44, 63, and 79, respectively. In the case of Single, the one-processor threaded version is slower than the sequential version by a factor of 68%. For the other three configurations, the degradation of the threaded version on one processor is less than 7%. The Single version has a high overhead of supporting threads, because no extra hardware is available for

[1] An improved native C compiler should make this unnecessary. This kernel is not used for the sequential version as it actually runs slower than the compiled version on large blocks.

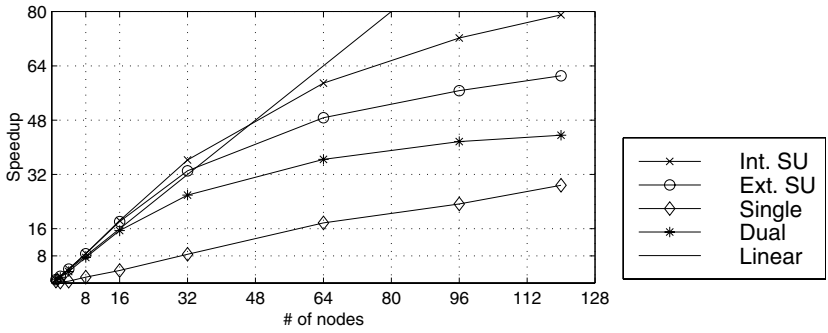


Fig. 5. Speedup on Class A (14,000 Rows)

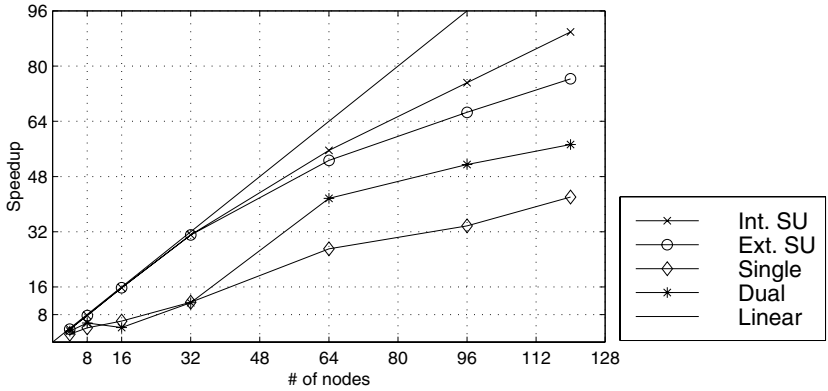


Fig. 6. Speedup on Class B (75,000 Rows)

performing the actions of the SU. We believe that the results for the External and Internal versions on 120 nodes are very encouraging, considering that the problem is not very large for that many nodes. The speedups of the Single, Dual, External SU, and Internal SU versions for Class B on 120 nodes are 44, 59, 78, and 90, respectively.

We have also written a threaded version of the full NAS CG benchmark, which has a number of reduction operations besides the MVM. Although we have not yet conducted a full set of experiments, our initial results show that the CG code has the same scalability as the MVM code. This is mainly because the MVM loop takes more than 95% of the total execution time of CG.

5 Performance Analysis

In this section, we examine the MVM performance in greater detail. We wish to answer two questions:

1. What limits the performance of our implementation of MVM on EARTH?
2. How important are the defining characteristics of the EARTH program execution model to the performance results?

To answer these questions, we ran a series of experiments with Class A input on the same platforms in Sect. 4.

A major loss of efficiency comes from the partitioning itself. The core which multiplies A_i (in whole or in part) with v_i is a 2-D loop, but the sparseness of A limits the iterations of the inner loop when p is large. For instance, the Class A input averages fewer than 140 non-zeroes per row, which means that on 120 nodes, each row of A_i has usually only one or two non-zeroes. The overheads due to partitioning must be significant irrespective of how it is parallelized.

To estimate the upper bound of the performance of our algorithm, we ran a special version of the sequential code, in which we partitioned A and v into p parts and multiplied them part by part, rather than in a single pass. The multiplication of each part was further broken into p stages. This mimics the kind of partitioning seen in the parallel version. On the other hand, to model the beneficial cache effects that often accompany parallelization, this code reuses the same (n/p)-element array for holding partial sums. While this produces incorrect results, it does not affect the *control* structure of the code, and so tells us how much benefit we are likely to get from cache effects.

Thus, for this algorithm, if the modified sequential code runs k times slower than the normal sequential code for a partition factor of p, that suggests the partitioning overheads will limit the speedup on an ideal parallel machine with p nodes to p/k. If cache benefits dominate, then $k < 1$, and thus a superlinear speedup may be attained. In the graphs that follow, we include the curve calculated in this manner as an "Upper bound" speedup.

In Sect. 3, we argued that the multithreading and local synchronization provided by EARTH were essential to getting the most performance from the MVM algorithm. To test this argument, we ran experiments on two modified versions of the Threaded-C MVM code, in which features of EARTH are removed. This way we can measure the benefits quantitatively.

The first experiment ("No multithreading") removes the overlapping of communication and computation by having a single fiber per iteration on each node, as in Fig. 3(b). This code has the local synchronization feature of EARTH but does not take advantage of the ability to switch to another thread of execution during a long-latency operation such as transferring a block of data.

The second experiment ("Global barrier") removes the benefits of local synchronization from the preceding experiment by simulating the effect of a global barrier in the no-multithreading code. In this experiment, SEMi halts each node when it is about to begin or end a communication phase, and when all nodes

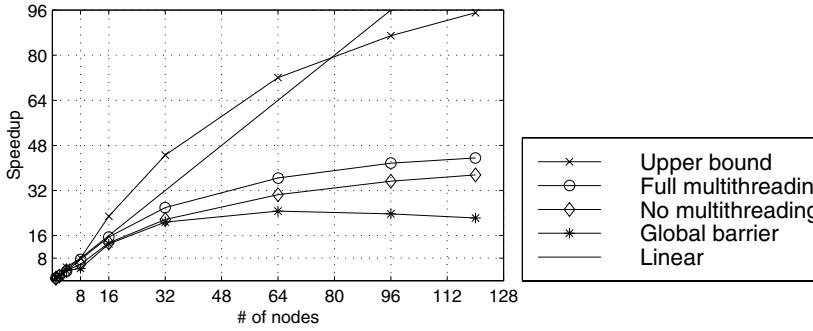


Fig. 7. Comparison of Parallel Versions on Dual

have halted, SEMi causes all nodes to continue where they halted. (The synchronizations are still local, but the earlier nodes are forced to wait for the last node.) The "global barrier" thus simulated is instantaneous, and all we are measuring is the cost of having some nodes wait for others, not the cost of the synchronization itself.

Results for the Dual and Internal SU configurations are shown in Fig. 7–8.[2] Each curve shows the original speedup curve from Sect. 4 ("Full multithreading"), the theoretical upper bound, and the speedups for the two simulations described above. We can make three main observations:

1. When we have efficient hardware support for EARTH's communication and multithreading operations, the speedup curves of our fully multithreaded implementation are reasonably close to the "upper limit" according to the limitations of our partitioning strategy. This tells us that our Threaded-C implementation is very effective at exploiting the parallelism which is inherent in the algorithm.
2. For this application, local synchronization gives a great improvement in performance. Global synchronization eliminates the ability to tolerate temporary imbalances in the load among the nodes. We observed that our uniform partitioning balanced the static work per node to within 10% of average,[3] and that over time, the load on one node stays *roughly* in sync with the other nodes [9].[4] However, there can be slight variations from iteration to iteration. While these variations average out in the long run, they can cause a slowdown if a global barrier always forces the node with less work to wait

[2] Other data can be found in our technical report [9].

[3] If p is not a divisor of n, then the last node will have fewer columns than the others. Uniform balancing may not occur with other types of inputs. However, since the current program is already able to handle different numbers of columns on each node, it would be easy to adjust the sizes of the A_i strips by counting non-zeroes.

[4] The staggered starting position is important, since most sparse matrices are far denser near the main diagonal.

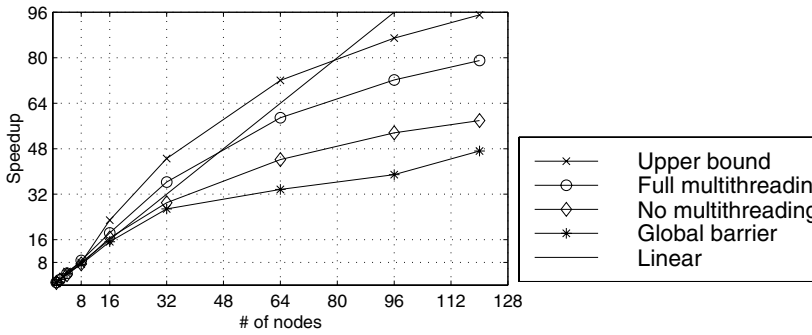


Fig. 8. Comparison of Parallel Versions on Internal SU

for other nodes to catch up. EARTH's local dataflow synchronization mechanism allows for looser coupling between nodes.

3. Finally, the ability to divide code into multiple threads of control is helpful in overlapping computation and communication. When we exploit this ability in our Threaded-C code, the performance improves roughly 15%.

Additional statistics collected by SEMi showed that the network is almost 40% saturated with the multithreaded code on Class A, showing that we make effective use of the communication hardware.

6 Conclusion

The aim of this study was to implement the sparse MVM core computation part of the CG linear systems solver on the EARTH multithreaded system, and to evaluate and analyze its performance. Sparse MVM has typically not performed well on conventional distributed memory machines, due to the high communication costs. As mentioned in Sect. 3, a straightforward multiplication program needs to communicate $O(n)$ data between each pair of neighboring nodes. Therefore, it is important to overlap this communication with computation and to minimize all other overheads wherever possible.

We chose a straightforward partitioning strategy for our implementation, and optimized it to take advantage of the special features of the EARTH multithreading model. Fine-grain threads ("fibers") in our code are synchronized strictly according to which data they need rather than through global barriers. Partitioning the program into multiple fibers permits overlapping of computation with communications. Low overheads in performing the multithreading and communication operations reduce the costs of frequent data transfers.

The current implementation of MVM on EARTH was shown to hide the communication latency effectively, thereby increasing the performance to a very high level. For example, a speedup of 59 is attained with 120 processors (on Class

B) when strictly off-the-shelf processors are used. Specialized hardware support for the EARTH model can increase the speedup to 90 on 120 nodes, without compromising the use of off-the-shelf processors for the main CPU.

The program was written in Threaded-C, an explicitly threaded variant of C which makes the features of EARTH visible to the programmer. While converting sequential code to *any* parallel language requires some effort, the main effort was in conceiving the high-level details of the parallel implementation. Once this was done, the conversion to Threaded-C was relatively straightforward. We believe that once programmers have gained sufficient experience in using Threaded-C, the programming effort is no higher than for MPI.

In conclusion, we have shown that a multithreaded system with local synchronization, overlapping of communication and computation, and low-overhead communication and thread-switching can efficiently parallelize the sparse MVM application.

Acknowledgements

We thank GMD First (Berlin) for research collaboration and for providing us with the MANNA machine used in this study. The authors also acknowledge partial support from DARPA, NSA, and NASA through the HTMT project; NSF (grants CISE-9726388, MIPS-9707125, EIA-9972853, and CCR-9808522); and DARPA through the DIVA project. Agrawal was also supported by NSF CAREER award ACR-9733520.

The authors would like to thank the current and former members of the ACAPS group at McGill University, and the CAPSL group at the University of Delaware, for their insights, ideas, encouragement and help.

References

[1] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[2] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill University, Montréal, Québec, May 1999.

[3] S.T. Barnard and H. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NAS Systems Division, NASA Ames Research Center, November 1992.

[4] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[5] Numerical Aerospace Simulation Facility. NAS parallel benchmarks, 1997. http://www.nas.nasa.gov/Software/NPB/.

[6] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 704–709, Cancún, Mexico, April 1994. IEEE Computer Society.

[7] Ian Stuart MacKenzie Walker. Towards a custom EARTH synchronization unit. Master's thesis, University of Delaware, Newark, Delaware, July 1999.
[8] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.
[9] Kevin B. Theobald, Rishi Kumar, Gagan Agrawal, Gerd Heber, Ruppa K. Thulasiram, and Guang R. Gao. Developing a communication intensive application on the EARTH multithreaded architecture. CAPSL Technical Memo 38, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 2000. In ftp://ftp.capsl.udel.edu/pub/doc/memos.