

# Landing CG on EARTH: A Case Study of Fine-Grained Multithreading on an Evolutionary Path

Kevin B. Theobald,<sup>1</sup> Gagan Agrawal,<sup>2</sup> Rishi Kumar,<sup>1</sup> Gerd Heber,<sup>3</sup>  
Guang R. Gao,<sup>1</sup> Paul Stodghill<sup>4</sup> and Keshav Pingali<sup>4</sup>

<sup>1</sup>Department of Electrical and Computer Engineering, University of Delaware

<sup>2</sup>Department of Computer and Information Sciences, University of Delaware

<sup>3</sup>Cornell Theory Center, Cornell University

<sup>4</sup>Department of Computer Science, Cornell University

{theobald,kumar,ggao}@capsl.udel.edu; agrawal@cis.udel.edu;  
heber@tc.cornell.edu; {stodghil,pingali}@cs.cornell.edu

## Abstract

*We report on our work in developing a fine-grained multithreaded solution for the communication-intensive Conjugate Gradient (CG) problem. In our recent work, we developed a simple yet efficient program for sparse matrix-vector multiply on a multithreaded system. This paper presents an effective mechanism for the reduction-broadcast phase, which is integrated with the sparse MVM, resulting in a scalable implementation of the complete CG application.*

*Three major observations from our experiments on the EARTH multithreaded testbed are: (1) The scalability of our CG implementation is impressive, e.g., absolute speedup is 90 on 120 processors for the NAS CG class B input. (2) Our dataflow-style reduction-broadcast network based on fine-grain multithreading is twice as fast as a serial reduction scheme on the same system. (3) By slowing down the network by a factor of 2, no notable degradation of overall CG performance was observed.*

## 1. Introduction

Many existing or proposed parallel machines are based on conventional architecture models and parallel programming paradigms that are reasonably efficient with regular parallel applications, but may not be so efficient (or even do poorly) on irregular problems with intensive communication and synchronization requirements. In such systems, the tasks into

which an application is divided are relatively “heavy-weight” and expensive to create, delete and move between processors. However, it is difficult to partition irregular problems into such “coarse-grain” tasks. Furthermore, low-cost point-to-point asynchronous data computation and synchronization is also very important for irregular codes, but supporting these efficiently has been a challenge for these systems. As a result, it is very difficult to overlap computation and communication/synchronization effectively and to achieve desirable performance for irregular applications.

One such irregular application which motivated this research is the Fracture Mechanics Simulation Problem (FMSP) which involves understanding and predicting how cracks grow. This has applications in many fields, such as aerospace, defense, and energy production. In this paper, we report our work in developing an efficient solution of the conjugate gradient (CG) problem which forms the key and most time-consuming compute kernel of the FMSP problem.

Under our approach, two main phases of CG computation that are challenging are: (1) MVM (the sparse matrix-vector multiplication), and (2) the inner-product calculation producing a scalar, which is subsequently processed and fed back to a DAXPY calculation. The second phase involves a reduction followed by a broadcast, which we refer to as a *reduction-broadcast* phase.

Our solution exploits the capability of a fine-grain multithreaded execution model that can quickly and efficiently manage a large number of threads. Two main features of such a model are: (1) the efficient

support of a large pool of active threads to achieve an effective overlapping of computation with communication and synchronization; (2) the efficient support of asynchronous, thread-to-thread, fine-grain synchronization and communication.

We have demonstrated that a fine-grained multithreaded solution for sparse MVM can be constructed and achieves excellent speedup [23]. The reduction-broadcast phase can be a performance bottleneck as it involves extensive communication with little computation and its solution is the main focus of this paper.

In our approach, the reduction-broadcast is realized as a bidirectional tree implemented in a dataflow style. Each node is a thread, and each edge can carry both data and an associated synchronization event. In the implementation, this tree is itself written directly as a multithreaded program, called as part of the user application. It takes full advantage of the fine-grain multithreading, using multiple event-driven threads to construct the nodes in the dataflow network and using the asynchronous dataflow-like fine-grain synchronization and communication operations for the actual communications among nodes. There is no need for global communication/synchronization, and no additional costs of interprocess communication.

The proposed scheme has been implemented on a fine-grain multithreaded execution emulation testbed based on the EARTH (*Efficient Architecture for Running TThreads*) execution and architecture model, which in turn is an extension of earlier work on hybrid dataflow models [8]. The EARTH project [9, 10, 11, 12, 16] is intended to demonstrate that a fine-grain programming and execution model can be designed and implemented in an *evolutionary approach* [24]. Instead of making a large quantum leap to a full-featured microprocessor supporting fine-grain multithreading at the instruction-set architecture level, the evolutionary approach begins with an existing parallel system, based on off-the-shelf microprocessors, and efficient support for fine-grain multithreading is introduced gradually. An EARTH emulation/simulation testbed has been developed on which experiments involving real programs have been conducted.

Significant speedup results have been observed in our experimental study. For example we have achieved an *absolute* speedup of 90 on 120 processors for the NAS CG class B input. This is obtained without graph partitioning or inspect-executor, which means that the same results can be obtained on adaptive problems as well. Our detailed experimental evaluation reveals a number of other aspects of the efficacy of our ap-

proach:

- Our dataflow style reduction-broadcast network based on fine-grain multithreading is indeed very efficient (twice as fast as a serial reduction scheme under the same execution environment). Furthermore, it has been integrated very well with the rest of the CG solution including our sparse MVM solution phase. This is demonstrated by the fact that the overall speedup for the entire CG code is almost as good as the speedup for the sparse MVM phase alone.
- Our results have also demonstrated the performance robustness of the proposed solution scheme. By slowing down the network by a factor of 2, no notable degradation on the overall CG performance was observed.
- Our results have supported the merit of an evolutionary approach for fine-grain multithreading, and provided an interesting performance comparison for different design points in the evolutionary path with various levels of custom hardware support for multithreading.

Overall, we believe that our approach offers a highly efficient, robust, and reasonably easy-to-program mechanism for parallel implementations of communication-intensive irregular applications.

The rest of the paper is organized as follows. The motivation and background for our work is explained in Section 2. Our approach and implementation is described in Section 3 and is experimentally evaluated in Section 4. We compare our work with related work in Section 5 and conclude in Section 6.

## 2. Background and Motivation

The research reported in this paper was motivated from the computational requirements of large-scale scientific applications. One of these application arises from the simulation of fracture at the macroscopic scale in engineering materials. Understanding how fractures develop in materials is critically important to many disciplines including aeronautical engineering and material sciences. A distinct advantage of computer simulation of crack propagation is that it makes it possible for engineers and scientists to run many more experiments than are economical or practical to do in a physical setting.

Fracturing is modeled at the macroscopic level by elastic deformation partial differential equations

(PDE's). In most situations, there are no closed-form solutions to these equations, so numerical methods such as the finite-element method or boundary-element method must be used to solve these equations approximately. In the finite-element method, an approximate solution to the PDE is then expressed as a linear combination of certain basis functions derived from this discretization. The choice of weights that yields the best approximation to the solution to the PDE is determined by using a method like the weighted residual method. This method requires solving a linear system of the form  $Ax = b$  where  $x$  is the vector of unknown weights in the linear combination. The size of  $A$  is proportional to the size of the mesh. In most realistic simulation,  $A$  is of the order of a few million rows and columns, but it is usually very sparse because only a few hundred elements in each row or column may be non-zero. Once the weights have been determined, the physics of the material is used to propagate the crack for a time step. The entire process has to be repeated for a certain number of time-steps.

The key and most time-consuming step in fracture simulation is the solution of this sparse linear system. Therefore, it is natural to consider solving these systems in parallel. In particular, we are interested in studying the parallelization of *iterative* solution methods.

Iterative methods repeatedly refine an initial approximation to the solution of the linear system until the approximation is close enough to the actual solution. The simplest methods, such as Jacobi and Gauss-Seidel iterations, have been known for centuries. However, these methods converge relatively slowly. In the past forty years, faster methods such as conjugate gradient (CG) and GMRES have been developed for solving linear systems iteratively. The conjugate gradient method can be used when the matrix  $A$  in the linear system is symmetric positive-definite. Since the matrices that arise in crack propagation have these properties, we have chosen to use the conjugate gradient method in our test-bed.<sup>1</sup>

Figure 1 shows the pseudo-code for the conjugate gradient algorithm. The notation  $(a, b)$  stands for the inner product of vectors  $a$  and  $b$ . The details of this algorithm are not important; what is important is to recognize that the key computations to be parallelized are the following:

- (i) the sparse matrix-vector (MVM) product  $Ap_j$

<sup>1</sup>The convergence of iterative methods can be improved by proper preconditioning. Although we have developed Element-by-element (EBE) preconditioners and support-tree-based preconditioners for our work, we will not address these in this paper.

```

r0      =  b - Ax0;
p0      =  r0;
for j = 0, 1, ..., till convergence do
    alpha_j = (r_j, r_j) / (Ap_j, p_j)
    x_{j+1} = x_j + alpha_j p_j
    r_{j+1} = r_j - alpha_j A p_j
    beta_j  = (r_{j+1}, r_{j+1}) / (r_j, r_j)
    p_{j+1} = r_{j+1} + beta_j p_j
od

```

**Figure 1. Conjugate Gradient Algorithm**

where  $A$  is the matrix from the linear system to be solved, and

- (ii) the inner product  $(r_j, r_j)$  and subsequent distribution of the single result back to all processors involved.

In our recent work, we have presented a very efficient approach towards executing the sparse matrix-vector multiply on a fine-grained multithreaded parallel system [23]. In this paper, we present a fine-grained multithreaded approach for the step (ii) which we refer to as the *reduction-broadcast* step. This approach is implemented and integrated with our solution to the sparse mvm problem forming the solution of the full CG code.

### 3. Multithreaded Implementation

The bulk of the conjugate gradient calculation alternates among three basic vector operations: matrix-vector multiplication, vector inner product, and DAXPY. Each of these operations presents specific challenges when one tries to parallelize CG. These are compounded by nearly sequential data dependences among the operations at the vector level. Because of these, one cannot tolerate an inefficient, long-latency implementation simply by overlapping it with another computation, and therefore, developing fast, efficient parallel implementations for all operations is the key to getting performance in CG.

Each of the three operations has major portions which can be done in parallel on independent nodes. However, both combining and reducing the partial results, and communicating results to other nodes, require fast implementations in order to avoid becoming "Amdahl's law" bottlenecks.

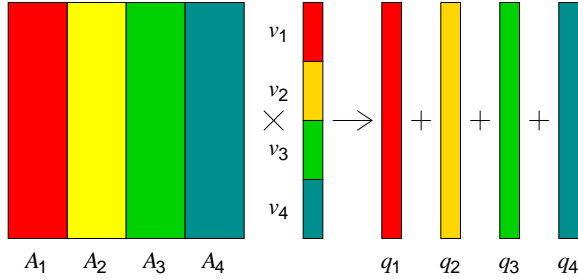


Figure 2. Parallel Partitioning of MVM

### 3.1. Matrix-Vector Multiplication

The dominant part of CG is the multiplication of the  $n \times n$  matrix  $A$  by an  $n$ -element vector.  $A$  is assumed to be too large to keep a whole copy on every node, and will need to be divided among all  $p$  nodes. Unless one uses techniques to reduce communication *algorithmically*, such as graph partitioning, the simplest way to divide the MVM problem is to divide  $A$  into regular strips or blocks. Figure 2 shows the method used in our algorithm (for  $p = 4$ ), which divides  $A$  into vertical sections  $A_1, \dots, A_p$ .

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  $q_1, \dots, q_p$  must be added to produce the final answer.

Some implementations of MVM produce  $q_1, \dots, q_p$  independently, then add them using a binary addition tree. The critical path length of this addition is  $O(n \log p)$ , which can far exceed the time to compute  $q_1, \dots, q_p$  if  $A$  is sparse and  $p$  is large. Instead, we *pipeline* the reduction in a linear chain, applying Cannon’s algorithm [5] to the special case of one dimension. This algorithm reduces the critical path to  $O(n)$ .

As described in an earlier paper [23], the computation is divided into  $p$  phases. The first two are illustrated in Figure 3. 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 (with node 1 sending to node  $p$ ). The starting positions on the nodes are staggered so that that piece can be added to what the left neighbor produces in the next iteration, as shown in Figure 3(b). Staggering the starting position has the additional side benefit of having all processors work in the vicinity of the main diagonal at the same time; this region tends

to be more dense in real problems.

As discussed in our previous paper, the high degree of pipelining can hurt performance on conventional coarse-grained parallel machines. High message overheads, large context switching costs, global barriers, and an inability to overlap communication and computation were all identified as obstacles to an efficient parallel implementations. We implemented this algorithm on EARTH (Efficient Architecture for Running Threads) [9, 10, 11, 12, 16], a platform specifically designed for fine-grain synchronization, low-overhead communications, and asynchronous local control, and showed experimentally how limitations common to most parallel machines based on off-the-shelf processors would cut performance by as much as half [23].

EARTH supports a fine-grained 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” make explicit the control and data dependences between different fibers, and fibers are scheduled according to the rule that a fiber is eligible to begin execution as soon as all relevant dependence conditions have been met. 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 grab any fiber whose data is ready when an existing fiber terminates after initiating a data transfer.

Figure 4 shows how the MVM algorithm is transformed to an EARTH program written in the Threaded-C language [22].<sup>2</sup> The computation is broken into a sequence of fibers.

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 executions of the same fiber, computing pieces of  $q_i$  on one node. Arcs represent data and control dependences. The solid arcs represent the data (in this case, pieces of  $q_i$  between nodes), while the dashed arcs represent synchronization signals only. Dashed arcs pointing down represent one fiber telling itself to perform the next iteration. Dashed arcs pointing down and to the right represent one fiber telling its right neighbor that it has finished using the block that the right neighbor pre-

<sup>2</sup>The development of the algorithm is explained in greater detail in our previous paper [23].

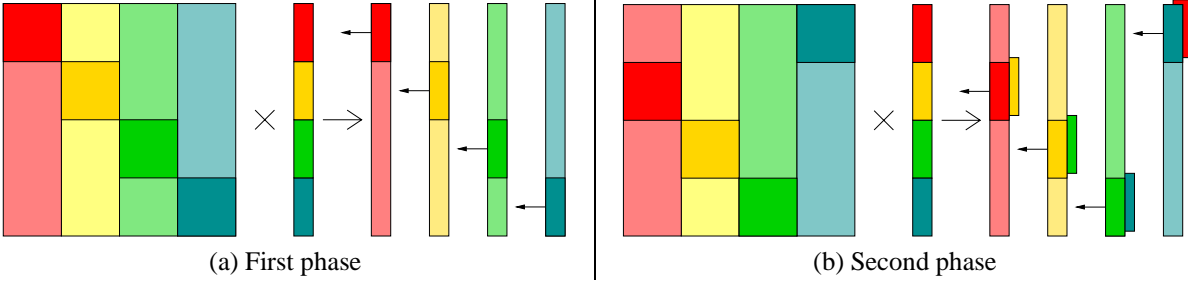


Figure 3. Pipelining of MVM (First 2 Phases)

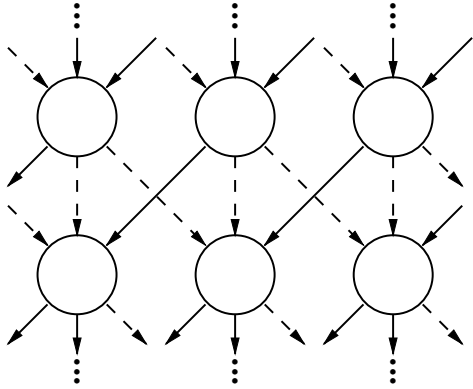


Figure 4. EARTH Implementation of MVM

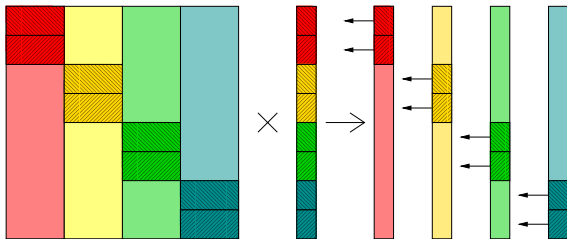


Figure 5. Multithreading MVM

viously sent, and the neighbor is therefore free to send another block. This last synchronization is needed because the arrays used for transferring blocks of data between nodes are reused.

Finally, to exploit EARTH's multithreading ability, our code splits each block multiplication into two halves, each of which produces half of the result vector. This is shown in Figure 5. 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 Figure 4 is replicated for each half.

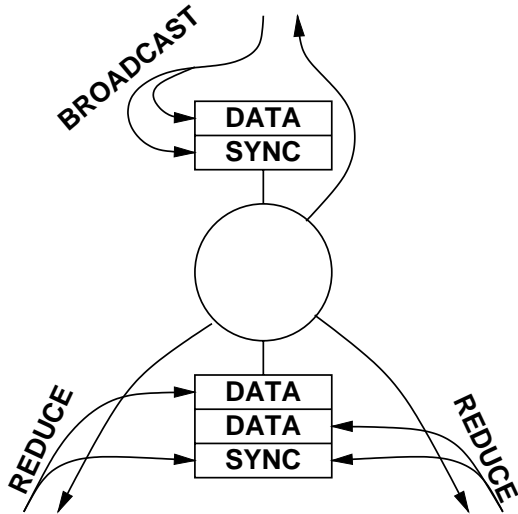
### 3.2. Reduction and Broadcasting

We have previously shown that this pipelining approach can achieve excellent speedups on a machine with good support for fine-grain multithreading, asynchronous control and efficient communication [23]. The pipelining approach makes the program inherently parallel. Furthermore, we can maintain this parallelism throughout most of the inner product and DAXPY computations as well. Performing the local portion of either is easy if both vectors are partitioned in exactly the same way. A nice property of the MVM code is that the last iteration sends the  $i^{th}$  piece of  $q$  back to node  $i$ . Therefore, if all vectors used in CG are partitioned in the same way, the result of an MVM will be aligned in *exactly* the same manner, ready for use in dot product or DAXPY calculations.

However, the inner product and DAXPY operations threaten to hurt the performance achieved by the MVM code, because they require introducing sequential sections of code. At some point, the local values of an inner product computation must be combined, and the result (or some derivative value) must be re-broadcast to all the nodes. Unfortunately, the technique used with MVM to overlap matrix multiplication and reduction is not appropriate to a simple dot product, where each node will generate only a single value. The critical paths of both reduction and broadcasting can be minimized through the use of trees, but these operations will still be bottlenecks if the latencies between different levels of the tree are too high.

Fortunately, EARTH provides a very efficient fine-grain communication mechanism which serves as the basis for a fast general reduction and broadcast tree. Figure 6 shows the main building block of this tree.

This building block has input buffers for incoming data and *synchronization slots* which count how many



**Figure 6. Building Block for Reduction-Broadcast Tree**

input values have arrived and trigger the execution of a fiber when all required inputs are ready. It is in the style of a classical static dataflow “actor” [6].

The core of the building block is a pair of fibers, one for reduction and one for broadcast. When a child produces a value, it uses the EARTH `DATA_SYNC` operator to pass that value to one of the input locations and synchronize the sync slot. Note that both children in Figure 6 use the same synchronization slot. That slot waits for two incoming data before starting the local reduction fiber; this guarantees that the reduction will not occur until both values are there, even if one comes in long after the other. When the reduction fiber runs, it sends the reduced value to that node’s parent, and synchronizes with a sync slot in that parent. The parent is either another node in the tree or the code which uses the final reduced value. A second fiber, with an independent sync slot and data input, is used when the parent wants to broadcast to the children.

Each node of the tree is a separate instance of a threaded procedure. This threaded procedure is a generic reduction-broadcast function in the Threaded-C library. The procedure includes initialization fibers for connecting each node to its parents and children. An important feature of this tree is that the initialization only needs to be done once. Once the tree has been constructed, it can be used repeatedly by simply using the `DATA_SYNC` operators in EARTH to inject values into the tree, which automatically begins reduction or broadcasting. This saves the initialization over-

heads if the same tree is used many times, as in CG.

Thus, the program does not “call” a reduction “function.” Instead, the distributed portions simply send data to the reduction tree when they have data. The top-level function arranges its sync slots so that a fiber that uses the reduced value is triggered when the root node of the tree produces a value. The tree itself is distributed among the processors for greater parallelism. Furthermore, if different input values are produced at greatly differing times, the locality of synchronization allows some parts of the reduction to be done provided *their* inputs have arrived.

One final useful property of this approach to reduction is that the threaded procedure is entirely written in the Threaded-C language. Some parallel systems may provide built-in primitives for parallel reduction or broadcast. But the programmer can modify our Threaded-C reduction library procedure as needed (such as changing the topology); this flexibility is not generally available when built-in functions are used.

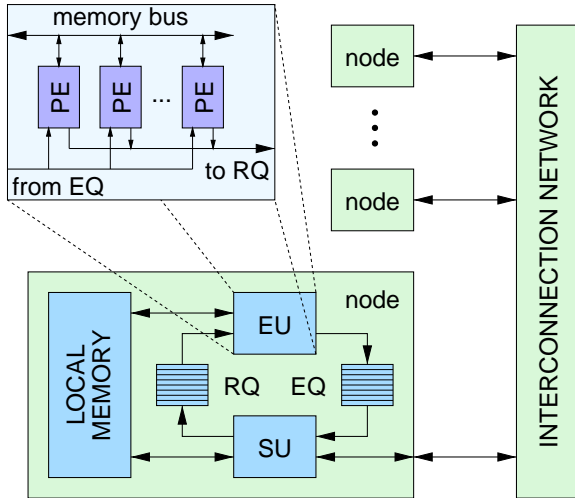
## 4. Experimental Results

This section describes the scalability of our implementation of CG on different versions of the fine-grain multithreaded EARTH system mentioned in Section 3.1, with an emphasis on the fine-grain reduction-broadcast code covered in Section 3.2. The first subsection briefly summarizes the key features of the EARTH architecture and describes the experimental testbed. This is followed by sections on the performance of the reduction-broadcast program alone and of the entire CG application.

### 4.1. The EARTH Multithreaded Architecture

The EARTH two-level thread hierarchy and synchronization mechanisms were mentioned in Section 3. An architecture suitable for such a model is shown in Figure 7. 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 and handles communication between nodes. There is also a Ready Queue of fibers waiting to be executed by the EU, and an Event Queue containing requests for EARTH operations, generated by the EU and read by the SU.

All implementations of EARTH have used off-the-shelf processors. In this paper (and previous studies) we consider four possible configurations. These represent various stages along the evolutionary path



**Figure 7. EARTH Architecture**

(described in Section 1) from a purely off-the-shelf system toward a system based on full-custom multi-threaded hardware:

**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 hardware SU performs specialized EARTH operations, and can be built fairly cheaply, yet be optimized for executing these operations [24]. The EU communicates with the SU through special memory addresses.

**Internal SU:** This is like the External SU, except that the CPU core and SU core are combined into one package. The interface is the same (memory addresses), but communication between them is off the main bus and hence faster. The off-the-shelf CPU core is otherwise unchanged.

The experiments in this study are based on the EARTH implementation for the MANNA parallel machine [4]. Our experiments were run using SEMi, an accurate (to within 2%), complete cycle-by-cycle simulator of the MANNA's processors, system bus, memory system and interconnection network [9, 24]. Data

for the runs on the Dual and Single configurations, up to 20 nodes, have been confirmed on the real machine. The system is simulated for a 200MHz processor and a 100MB/s network.

The relevance of these results to another machine is highly dependent on the architectural parameters of that machine. A system based purely on off-the-shelf processors, without any specialized support for fine-grain multithreading, is likely to have performance results similar to the Single or Dual platform. But network bandwidth is also a crucial factor. As explained in Section 3.1, the total communication per node during one MVM is  $O(n)$ . This stays constant even as the number of nodes increases and the computation per node correspondingly decreases [23]. Thus, there is an upper bound on speedup, which is reached when the network becomes saturated. For instance, scalability will be poor on a PC cluster with 100Mb/s Ethernet, even if the processors are assisted by custom hardware Synchronization Units. Excess communication overheads (e.g., the SU requires the intervention of the OS to talk to the network) can also hurt performance. However, our simulations made conservative assumptions about hardware capabilities, and similar results should be obtained on any system with adequate network bandwidth and low overheads.

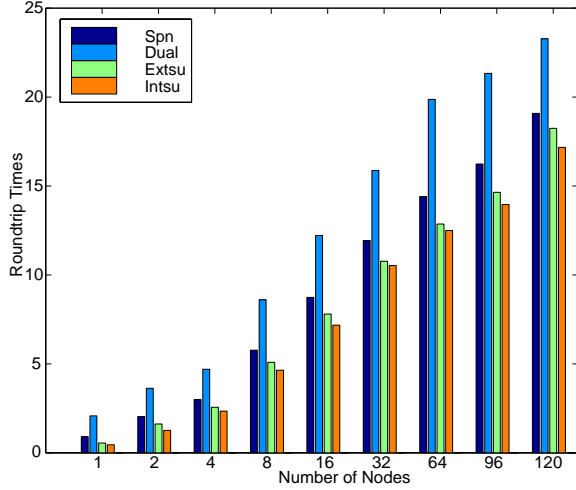
## 4.2. Performance of Reduction and Broadcast

This section describes the performance of our EARTH systems on several "microbenchmarks" for testing the reduction-broadcast code.

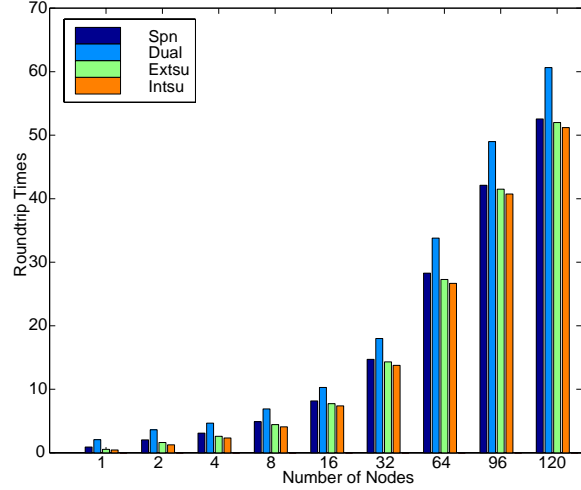
To test the total latency of reduction and broadcast, we invoke the reduction-broadcast threaded procedure with the appropriate node count; the instance we invoke creates the underlying tree and becomes the root of the tree. The fanout of the tree is 3 (the empirically optimal value). If there are  $p$  nodes, then the broadcast tree has  $p$  outputs. Each output is connected to a threaded procedure which simply forwards the broadcast value to the corresponding input to the reduction tree. Floating point addition is used as the reduction function. We run multiple broadcast-reduction phases, and measure the average round-trip time.

Figure 8 shows the average round-trip times for broadcast+reduction on the four versions of EARTH. This latency is the time taken in the following steps:

1. A fiber in the main procedure sends data to the broadcast root, using DATA\_SYNC;
2. The broadcast propagates through the tree;



**Figure 8. Total Round Trip Latency for Reduction-Broadcast Tree (in Microseconds) with Fanout of 3**



**Figure 9. Total Round Trip Latency for Sequential Reduction-Broadcast (in Microseconds)**

3. Each function at the leaf forwards the broadcast value to the corresponding reduction input;
4. The reduction propagates through the tree;
5. The root of the tree sends the reduced value back to the main function, triggering for re-execution the fiber at the start of this list.

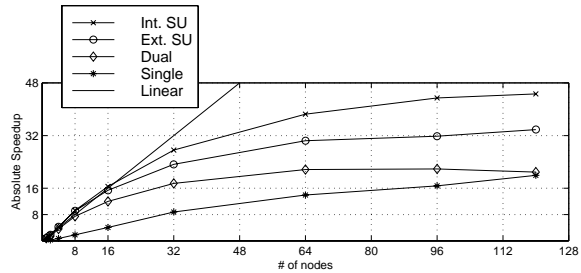
A recent survey [7] showed that many commercial processors take longer than our total round-trip time to perform a single data transfer!

Figure 9 shows the corresponding times for a sequential reduction, in which all reduction values are sent to a single node, which performs the reduction sequentially. These results clearly show that sequential reduction can lower the overheads of parallelism for small reductions, but the cost of the reduction itself becomes overwhelming for a large number of nodes.

### 4.3. Scalability of Conjugate Gradient

The matrix sizes we have used are based upon the problem sizes associated with the distribution of the NAS Conjugate Gradient (CG) benchmark [19]. We have used the Class W ( $n = 7,000$ ), A (14,000) and B (75,000) problem sizes for our experiments. The total numbers of non-zeroes in the matrices are 508,402, 1,853,104, and 13,708,072, respectively.

The scalability of the class W problem on the 4 different EARTH configurations is shown in Figure 10.



**Figure 10. Speedup on Class W (7,000 Rows)**

The speedups shown are absolute, i.e., the parallel performance is compared against the sequential version. On 120 processors, the speedups achieved on the Single, Dual, External SU, and Internal SU versions are 18, 20, 34, and 44, respectively. The threaded version on 1 processor is slower than the sequential version by 74% for the Single configuration, 22% for the Dual configuration, 4% for the External SU version and 5% for the Internal SU version. The performance of Single and Dual versions are relatively low because the amount of work per communication operation is very small for Class W, and there is no specialized hardware support to lower the communication overheads. The speedups of External SU and Internal SU are close to linear up to 32 processors, but degrades later. This is clearly because the problem size is very small for

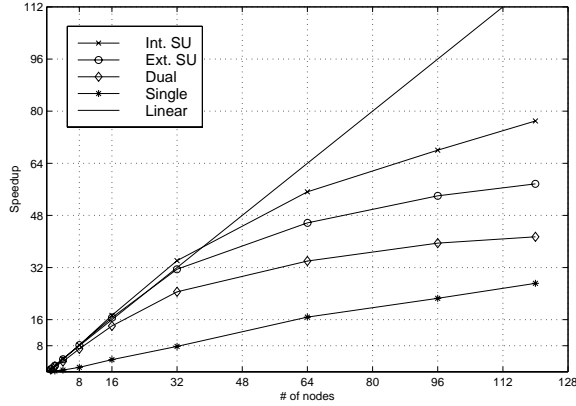


Figure 11. Speedup on Class A (14,000 Rows)

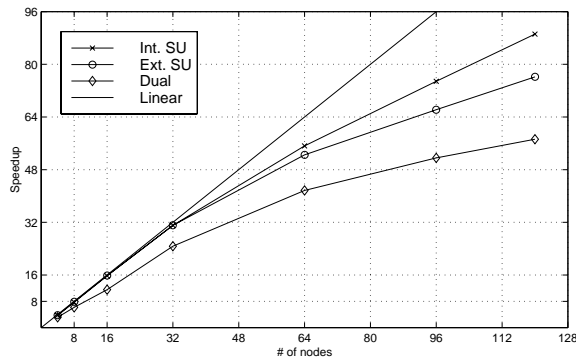


Figure 12. Speedup on Class B (75,000 Rows)

execution on large configurations.

The scalability of the Class A problem is shown in Figure 11. On 120 processors, the speedups achieved on the Single, Dual, External SU, and Internal SU versions are 28, 44, 63, and 79, respectively. In the case of Single, the 1-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 performing the actions of the SU. We believe that the performance of the External and Internal versions on 120 processors are very encouraging, considering that the problem is not too large for execution on 120 processors.

Finally, the scalability of the class B problem on three different EARTH configurations is shown in Fig-

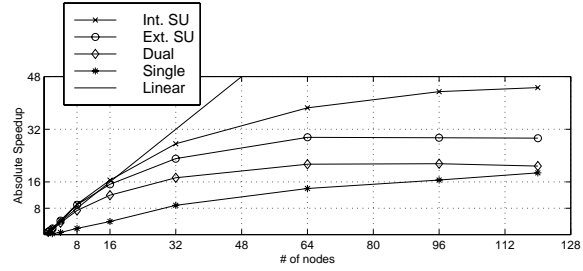


Figure 13. Performance of Class W (7,000 Rows) with Slower Network

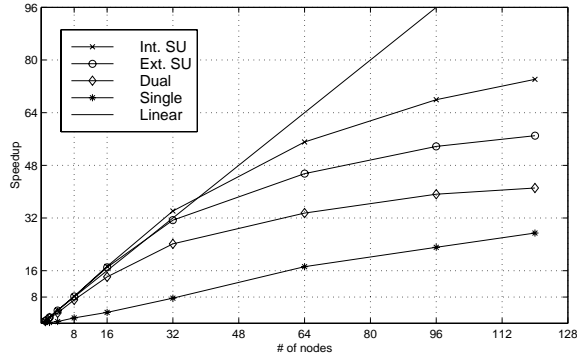
ure 12. The performance on the Single version is not reported, because the large problem sizes and poor speedups made the simulations extremely time-consuming. The speedups of the Dual, External SU, and Internal SU versions on 120 processors are 55, 78, and 90 respectively.

Our earlier paper reported similarly good speedups for MVM alone [23]. Comparing the new results with those shows that the CG speedups are lower than the pure-MVM speedups, but only by less than 1%. What this means is that, while the inherent sequential data dependences of reduction and broadcasting do degrade parallel performance, the efficiency of our reduction implementation renders the degradation insignificant. Overall, we believe that the speedup of 90 on 120 processors achieved with extra hardware support on the same chip is extremely good, and demonstrates the suitability of the EARTH model for sparse problems.

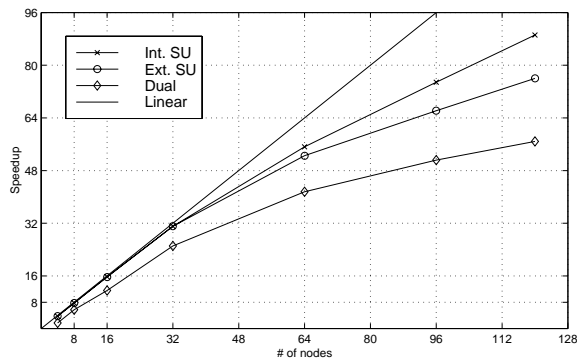
To evaluate the robustness of the architecture's ability to overlap communication and computation, we simulated the system with a network offering half the original bandwidth, and keeping all other performance factors the same. Speedups on Class W, A, and B with the slower network offering only half the bandwidth are shown in Figures 13, 14, and 15, respectively. The speedups are almost identical to the speedups with the original network. On Class B problem with External SU, the difference in performance ranged from 1.7% on the 8 processors case to only 0.04% on the 96 processor case. On Class A problem with External SU, the difference in performance ranged from 2.3% on 8 processors to 0.54% on 96 processors.

## 5. Related Work

A number of other projects have focused on developing sparse matrix computations on scalable paral-



**Figure 14. Performance of Class A (14,000 Rows) with Slower Network**



**Figure 15. Performance of Class B (75,000 Rows) with Slower Network**

lel machines. Most of these efforts have used either a message passing library such as the Message Passing Interface (MPI) [18], a runtime library built on top of the message passing layer, or have used a research distributed-shared-memory machine where the cache coherence protocol can be modified to suit the requirements of the application.

Saltz *et al.* developed the notion of *inspector/executor* for efficiently parallelizing irregular applications on message passing machines [21]. The key idea is to perform runtime analysis for predetermining the communication between processors and aggregating the messages. The notion of *inspector/executor* was implemented in the PARTI/CHAOS library [3, 14] which was used to implement efficient and scalable message passing versions of several irregular applications like the EULER solver [17] and the molecular dynamics code CHARMM [13].

In some other efforts, compiler techniques were de-

veloped for parallelizing irregular applications written in High Performance Fortran (HPF) extensions [1, 25, 15, 20, 26]. However, these techniques have not yet been fully implemented in a production level compiler.

Pingali and co-workers have used restructuring compiler technology to synthesize sparse matrix code from (i) dense matrix programs, and (ii) descriptions of sparse formats. Their Bernoulli system is a generic programming system in which an algorithm needs to be written just once, and can be combined with descriptions of sparse formats to generate efficient sparse code [2].

## 6. Conclusions

We have reported our work in developing a fine-grained multithreaded solution for the Conjugate Gradient (CG) problem.

There are two challenging and communication intensive steps in forming an efficient solution of CG: performing the sparse matrix-vector multiply (MVM), and performing reductions followed by a broadcast. In our recent work, we have developed a very efficient solution to executing MVM on a multithreaded system. This paper present an efficient and effective mechanism for the reduction-broadcast, which is implemented and integrated with the sparse MVM, resulting in a very efficient implementation of the full CG code.

Our main observations include:

- The speedup results of our CG implementation are very impressive: for example we have achieved an *absolute* speedup of 90 on 120 processors on the NAS CG class B input.
- Our dataflow style reduction-broadcast network based on fine-grain multithreading is indeed very efficient (twice as fast as a serial reduction scheme under the same execution environment). Furthermore, it has been integrated very well with the rest of the CG solution including our sparse MVM solution phase. This is demonstrated by the fact that the overall speedup for the entire CG code is almost as good as the speedup for the sparse MVM phase alone.
- Our results have also demonstrated the performance robustness of the proposed solution scheme. By slowing down the network by a factor of 2, no notable degradation on the overall CG performance was observed.

- Our results have supported the merit of an evolutionary approach for fine-grain multithreading, and provided an interesting performance comparison for different design points in the evolutionary path with different hardware support.

Overall, we believe that our approach offers a highly efficient, robust, and reasonably easy-to-program mechanism for parallel implementations of communication-intensive irregular applications.

## Acknowledgements

We acknowledge NSF for the partial support of the current research through grants NSF-CCR-9808522, NSF-CISE-9726388, NSF-CDA-9703088, NSF-EIA-9726388, NSF-ACI-9870687 and NSF-EIA-9972853. We also acknowledge the support from the Defense Advanced Research Projects Agency (DARPA), the National Security Agency (NSA) and the National Aeronautics and Space Administration (NASA) through a subcontract with Jet Propulsion Laboratory (JPL) and the California Institute of Technology (CalTech) for the Hybrid Technology Multithreaded Architecture Project at the University of Delaware. We thank GMD First (Berlin) for research collaboration and for providing us with the MANNA machine used in this study. The authors also acknowledge support from the Delaware Biotechnology Institute (DBI). Author Agrawal was also supported by an NSF CAREER award, ACR-9733520.

## References

- [1] Gagan Agrawal and Joel Saltz. Interprocedural compilation of irregular applications for distributed memory machines. In *Proceedings of Supercomputing '95*, San Diego, California, December 1995. In <http://www.supercomp.org/sc95/proceedings/>.
- [2] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill. A framework for sparse matrix code synthesis from high-level specifications. In *Proceedings of SC2000: High Performance Networking and Computing*, Dallas, Texas, November 2000. In <http://www.sc2000.org/>.
- [3] Harry Berryman, Joel Saltz, and Jeffrey Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.
- [4] 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.
- [5] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.
- [6] Jack B. Dennis, Guang-Rong Gao, and Kenneth W. Todd. Modeling the weather with a data flow supercomputer. *IEEE Transactions on Computers*, 33(7):592–603, July 1984.
- [7] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1998.
- [8] G. R. Gao. An efficient hybrid dataflow architecture model. *Journal of Parallel and Distributed Computing*, 19(4):293–307, December 1993.
- [9] 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.
- [10] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, Limassol, Cyprus, June 1995. ACM Press.
- [11] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. ACAPS Technical Memo 68, School of Computer Science, McGill University, Montréal,

- Québec, October 1993. In <ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos>.
- [12] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 288–294, Cancún, Mexico, April 1994. IEEE Computer Society.
- [13] Yuan-Shin Hwang, Raja Das, Joel H. Saltz, Milan Hodoscek, and Bernard R. Brooks. Parallelizing molecular dynamics programs for distributed memory machines. *IEEE Computational Science & Engineering*, 2(2):18–29, Summer 1995. Also available as University of Maryland Technical Report CS-TR-3374 and UMIACS-TR-94-125.
- [14] Yuan-Shin Hwang, Bongki Moon, Shamik D. Sharma, Ravi Ponnusamy, Raja Das, and Joel H. Saltz. Runtime and language support for compiling adaptive irregular programs. *Software — Practice and Experience*, 25(6):597–621, June 1995.
- [15] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.
- [16] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 178–188, Philadelphia, Pennsylvania, May 1996.
- [17] D. J. Mavriplis, R. Das, R. E. Vermeland, and J. Saltz. Implementation of a parallel unstructured Euler solver on shared and distributed memory architectures. In *Proceedings of Supercomputing '92*, pages 132–141, Minneapolis, Minnesota, November 1992.
- [18] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3-4), 1994.
- [19] Numerical Aerospace Simulation Facility. NAS parallel benchmarks, 1997. In <http://www.nas.nasa.gov/Software/NPB/>.
- [20] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of Supercomputing '93*, pages 361–370, Portland, Oregon, November 1993.
- [21] 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.
- [22] Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 1998. In <ftp://ftp.capsl.udel.edu/pub/doc/memos>.
- [23] Kevin B. Theobald, Rishi Kumar, Gagan Agrawal, Gerd Heber, Ruppia K. Thulasiram, and Guang R. Gao. Developing a communication intensive application on the EARTH multithreaded architecture. In *Proceedings of the European Conference on Parallel Computing (Euro-Par 2000)*, Munich, Germany, August–September 2000. Springer-Verlag. To appear as a Distinguished Paper.
- [24] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill University, Montréal, Québec, May 1999.
- [25] Reinhard von Hanxleden. Handling irregular problems with Fortran D - a preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. Also available as CRPC Technical Report CRPC-TR93339-S.
- [26] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6):737–753, June 1995.