Shared Memory versus Message Passing for Iterative Solution of Sparse, Irregular Problems

Frederic T. Chong and Anant Agarwal
{ftchong,agarwal}@lcs.mit.edu

Laboratory for Computer Science
Massachusetts Institute of Technology

October 27, 1996

Abstract

The benefits of hardware support for shared memory versus those for message passing are difficult to evaluate without an in-depth study of real applications on a common platform. We evaluate the communication mechanisms of the MIT Alewife machine, a multiprocessor which provides integrated cache-coherent shared memory, message passing, and DMA. We perform this evaluation with "best-effort" implementations which solve several sparse, irregular benchmark problems with a preconditioned conjugate gradient sparse matrix solver (ICCG).

We find that machines with fast global memory operations do not need message passing or bulk transfer to support our irregular problems. This is primarily due to three reasons. First, a 5-to-1 ratio between global and local cache misses makes memory copies in bulk communication expensive relative to communication via shared memory. Second, although message passing has synchronization semantics superior to shared memory for data-driven computation, efficient shared memory can overcome this handicap by using global read-modify-writes to change from the traditional owner-computes model to a **producer-computes** model. Third, bulk transfers can result in high processor idle times in irregular applications.

Keywords: multiprocessors, shared memory, message passing, bulk transfer, iterative solution, irregular, sparse matrix

1 Introduction

Distributed shared memory and message passing are two dominant communication mechanisms in parallel systems. Most machines incorporate either message passing [Thi93a] [Int91] [Che93] or shared memory [LT88] [LLG+92] [BFKR92] [SGI94]. A few machines support both mechanisms [ABC+95] [HKO+94]. The precise benefit of shared memory versus message passing has remained an open question, largely due to the difficulty of finding experimental platforms supporting an "apples to apples" comparison and the difficulty of writing applications in both styles without bias.

This paper addresses this open question using the MIT Alewife machine [ABC+95] and sparse, irregular applications. Alewife integrates distributed, cache-coherent shared memory and message passing, providing an ideal platform for our experiments. Although this study evolved from research [CSBS95] [BCL+95] [CS95] and researchers[1] with a clear message-passing bias, our results point to shared memory as the most useful communication mechanism for a multiprocessor. Our kernels were originally optimized for message passing and were rewritten for shared memory. We chose especially challenging fine-grain, irregular benchmarks based upon the solution of sparse linear systems via ICCG (conjugate gradient preconditioned with incomplete Cholesky factors) [GvL83]. We expected ICCG to favor message passing because it is an iterative algorithm with little data reuse. However, we found shared memory to be an extremely efficient communication mechanism even without the benefits of caching.

Although we expected clear gains from the superior synchronization semantics of message-passing with Active Messages [E+92], we were able to overcome the synchronization difficulties of shared memory by shifting from an owner-computes model of computation to a producer-computes model (see Section 4.1). We also expected to see performance gains when using long messages containing aggregated data as opposed to short messages or cache-line transactions. Aggregation has been a common approach on traditional multiprocessors with high-overhead communication. A small number of long messages incurs less message send and receive overhead than a large number of small messages. However, on modern systems with fast active messages or hardware-supported shared memory, we found that data copying to aggregate messages significantly reduces the performance advantages of long messages.

While we found shared memory and message passing to be comparable in performance for the critical kernels of our applications, we found shared memory to be more convenient for initialization and preprocessing phases. Consequently, we used shared memory for these phases even when the kernels used message passing. This was a convenient option which was available because the Alewife system allows the use of both message passing and shared memory on the same data structures in the same program.

Previous work by Chandra, Larus, and Rogers [CLR94] studied four benchmarks running on simulations of separate message-passing and shared-memory machines based upon the CM5. They also argue that shared memory performance is comparable to message passing, but shared memory was nearly a factor

[1] For example, one of the original developers of the CMMD message-passing library [Thi93b] on the CM5.

matrix	Description	Order	Nonzeros	Seq MFLOPS	Values	Avg Deg	Colors
BUS1138	Power system	1138	4054	0.7	Y	1.3	5
CAN1072	Aircraft model	1072	12444	0.7	N	5.3	7
BCSPWR10	Eastern US power system	5300	21842	0.7	N	1.6	4
BCSSTK18	Nuclear power plant model	11948	149090	*0.7	Y	5.7	14
OCEAN	World Ocean Model	143437	962623	*0.7	N	2.9	4
BCSSTK32	Automobile Chassis	44609	2014701	*0.7	N	22.1	37

Table 1: Benchmark matrices.

of two slower than message passing on EM3D, their sparse, irregular graph problem. They also speculate that integrating bulk transfer mechanisms with shared memory would be useful for that EM3D.

We also examine sparse, irregular graph problems, but we use ICCG instead of their naive red-black relaxation as a solution method. ICCG converges much more quickly than relaxation methods and is a much more powerful tool for solving realistic problems. As we shall see, however, the irregular nature of the ICCG computation causes bulk transfer techniques to incur substantial copying costs and idle times. Furthermore, our actual Alewife hardware has relative costs which differ from the assumptions made by Chandra et al. This difference, coupled with the fine granularity of our problems, results in **higher** performance for shared memory than message passing.

In the next two sections, we describe our experimental platform, our benchmarks, and the motivation behind our choice of applications. In Section 4, we describe our computation kernel and how each communication mechanism fits in. We present our performance results in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 Motivation and Benchmark Matrices

We concentrate upon ICCG not only because it is challenging, but also because it is a powerful method for solving a wide range of sparse linear systems. Our benchmarks, shown in Table 1, represent problems that arise in power system optimization and reconfiguration, aircraft design, building design, automotive design, and oceanography. ICCG is an important method because problems in the real world are sparse and can be quite large. As we shall see in Section 4, ICCG is an iterative method which dramatically saves on factorization time and data size. Data size is often critical. Even using ICCG, OCEAN will only run in the memory of 16 or more Alewife processors and BCSSTK32 on 32 or more.

The benchmarks of Table 1 are represented by symmetric positive definite sparse matrices, where **order** refers to the number of rows or columns (they are equal) and **nonzeros** refers to the number of nonzero elements in the matrix. **Seq MFLOPS** refers to the computation rate of an optimized sequential version of our kernel on a single Alewife SPARC-based processor[2]. We will use this rate to compute speedups. The (*) denotes problems which are too large to fit into the memory of one processor, but we assume a 1 MFLOP rate (corresponding to the smaller problems) as if they did. This assumption avoids super-linear speedups which can result from sequential times based upon a workstation paging to disk or a single multiprocessor node using the memory from multiple nodes.

Most of our benchmarks are from the Harwell-Boeing benchmark suite [DGL92]. Most of the matrices, especially the large ones, only come with the pattern of the nonzero entries, not the actual floating point values. BUS1138 and BCCSTK18 are the two largest power system and structures matrices in the HB suite with included values. We will be using these two matrices to measure convergence and end-to-end run times in Section 4. **Values** indicates which matrices have values. **Avg Deg**, roughly proportional to **Nonzeroes** divided by **Order**, indicates the average number of incoming arcs for a node of the kernel computation, described in Section 4.1. Each incoming arc results in 2 floating point operations in the kernel. **Colors** indicates the number of colors required in the multicolor ordering described in Section 4.4. The larger the ratio between **Order** and **Colors**, the better the parallelism and message aggregation.

3 Experimental Platform

We conduct our experiments on the MIT Alewife multiprocessor [ABC+95], shown in Figure 1. Not only is the Alewife machine is ideal for our experiments because it efficiently supports distributed shared memory, message passing, and DMA (Direct Memory Access), but it also provides a good indication of the relative performance and hardware cost of each mechanism for typical multiprocessors of the present and of the future. These mechanisms are integrated so that a single execution of a program can use all of these mechanisms on the same data structures. The heart of each processing node is the CMMU (Communications and Memory Management Unit), a custom VLSI component which serves as global and local memory controller, as well as network interface.

The efficiency of Alewife's communication mechanisms is essential to exposing the tradeoffs and results of this study. A shared-memory write-miss only takes 66 cycles plus 1.6 cycles per hop in the network to the processor where the data resides. Alewife supports active messages of the form:

send_am(proc, handler, args...)

which causes a message to be sent to processor **proc**, interrupt the processor, and invoke **handler** with **args**. An active message with a null handler, no body and no arguments, only takes 102 cycles plus .8 cycles per hop. The Alewife network interface (within the CMMU memory controller) can hold up to fourteen

[2]The current Alewife runs at 20 MHz, but a bug fix in the CMMU memory controller will increase performance by 50 percent to 33 MHz.

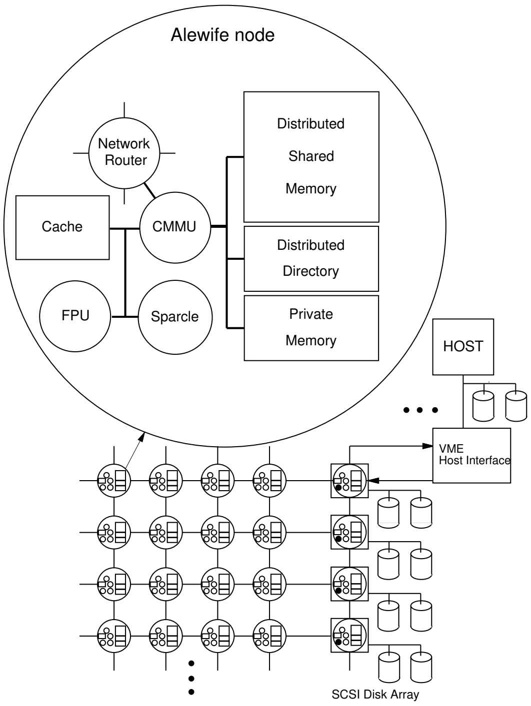


Figure 1: The MIT Alewife multiprocessor.

32-bit arguments for an active message. Longer messages are supported via DMA by adding **(base, length)** pairs to the end of an active message. Alewife messages are not limited in length and the CMMU automatically sends the DMA data after the handler arguments in an active message. On the receive side, the handler is invoked with its arguments and may either tell the CMMU to store the DMA data to memory or receive the data directly from the network interface.

The hardware cost and performance of communication mechanisms in the CMMU will generalize well to multiprocessors for some time into the future. In order to leverage commodity microprocessors, multiprocessors need to support communication through a separate chip on a bus. The CMMU demonstrates that shared memory, message passing, and DMA can be efficiently supported by such a communication chip. The natural integration of these mechanisms into a single chip also serves to hold their relative hardware cost and performance somewhat invariant. Shared memory will always be somewhat lower overhead than message passing. To send a message, the processor needs write a header and data to memory-mapped locations of the communications chip. To write to shared memory, the processor only needs to write the data onto the bus with the shared address.

A fundamental difficulty in experimenting with an academic machine such as Alewife is generalizing to future technologies. In particular, can communication mechanisms keep up with increasing processor speeds? Fortunately, our results do not depend upon this question. Instead, since our applications are memory bound, our results depend upon whether communication mechanisms can keep pace with local memory systems.

Many applications, especially irregular applications such as our larger benchmarks, are larger than caches and are limited by main memory speeds. While a quad-issue, 300 MHz Alpha is more than 30 times faster than Alewife's 20 MHz Sparcle processor, many current memory systems are no faster than Alewife's 500 ns miss time. Even the fastest current prototypes, such as Digital's cache-less workstation [CB96], still require 90 ns to reference main memory. Memory latency limitations will remain severe. Synchronous DRAMs and wide datapaths will not help with irregular accesses.

The key is that network technology scales at least as well as DRAM speeds. In fact, there is no reason why a remote miss should take any more than five times a local miss on any machine. It takes one local-miss-time to service a remote miss at the receiver. It takes much less than one local-miss-time to get the request off the sender. That leaves more than three local-miss-times for the network transit time. With current VLSI switch technology [DCB$^+$94] keeping pace with processor cycle times, future networks should have no trouble keeping up with local memory speeds.

Not only is a fast remote miss feasible, it is necessary to make shared-memory machines easy to use and viable for a wider range of applications. Given that future fine-grain mechanisms will perform comparably to local memory systems, results on Alewife generalize to future systems.

In the next section, we will describe how Alewife's mechanisms affect the implementation of our sparse, irregular benchmarks.

4 Communication Mechanisms for Parallel ICCG

This section starts out describing the dominant kernel of ICCG and explaining how our communication mechanisms affect its implementation. We then give an overview of the entire ICCG algorithm and provide measurements that show that the kernel is important. Finally, we describe data mapping to enhance locality and reordering to increase parallelism and facilitate data aggregation.

4.1 Communication in the Kernel

The core of our study is a directed acyclic graph (DAG) computation which arises from sparse triangular solution. As shown in Figure 2, the DAG arises from the solution of $Lx = b$, where L is a lower triangular sparse matrix, x is the vector of unknowns, and b is a right-hand-side vector of values. In our example, the computation is broken into five DAG nodes, each representing a row of L, an element of x, and an element of b. Each DAG node computes the element of x associated with it by substituting incoming values into the equation represented by the node's row. DAG node i depends upon node j if row i of L contains a non-zero in column j. Each edge in the DAG represents 2 FLOPS of computation and the transfer of one double-precision value. For example, the edge from node 0 to node 4 indicates that we must compute $v = L_{04} \cdot x_0$, communicate v to node 4,

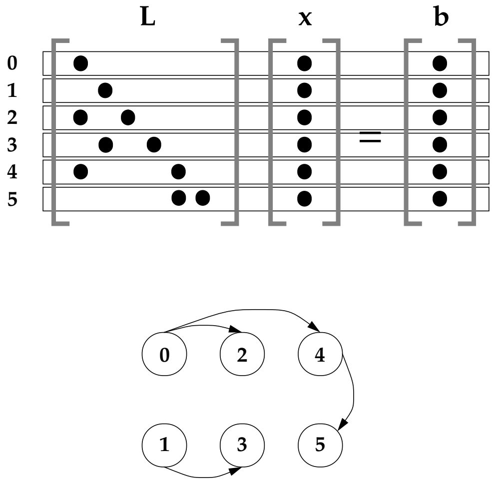


Figure 2: A DAG representing solution by substitution of $Lx = b$. Non-zeros are represented by filled circles.

and subtract v from b_4. The next two sections describe our message-passing and shared-memory implementations of this computation.

4.1.1 Message Passing

Our DAG is essentially a dataflow computation [ACM88] and is easily implemented via active messages. In our example, let us assume that DAG node 0 is on processor 0 and node 4 is on processor 4. The edge from node 0 to node 4 is a **non-local edge**, an edge between nodes on different processors. Each processor keeps a presence counter per local node to keep track of how many incoming edges have been satisfied for each node in its local memory. Once all incoming edges have been satisfied, the outgoing edges can be processed. Since node 0 has no incoming edges, $x_0 = b_0$ and processor 0 can send $v = L_{04} \cdot x_0$ to node 4 on processor 4 via an active message of the form:

send_am(in_edge_handler, 4, 4, v).

This causes an active message to be sent from processor 0 to processor 4. When the message arrives, processor 4 is interrupted and the handler procedure in Figure 3 is executed with the arguments **in_edge_handler(4,v)**. This handler causes the presence counter on node 4 to be decremented and v to subtracted from the solution for x_4.

Longer messages are also straightforward. We use longer messages in an attempt to reduce send and receive overhead. We buffer up multiple non-local edges, represented by **(node_num, value)** pairs, into buffers in memory, one buffer each processor that edges are destined for. We will see later that this buffering incurs significant cost in memory operations (end of Section 4.1) and idle time (in Section 5). Once a buffer contains the desired number of edges for a long message, we send its contents as an active message with many arguments:

```
void in_edge_handler(int node_num, double value)
{
  NODE_TYPE *node = node_array[node_num];

  node→counter−−;
  node→x −= value;
}

/* main loop */
for (i=0; i<num_local_nodes; i++) {

  /* get next node in schedule */
  node = prescheduled_nodes[i];

  /* wait until incoming edges are done */
  while (node→counter > 0);

  process_outgoing_edges(node);
}
```

Figure 3: Message-passing code for DAG execution.

send_am(proc, many_in_edge_handler, node_num_1, value_1, . . . , node_num_n, value_n)

The handler then performs the appropriate decrement and subtraction for each edge. Note that this approach is generally limited to the size of the output queue in the network interface. On Alewife, we send up to 4 node-value pairs in a message. However, the integrated DMA mechanism allows us to send a buffer containing many more node-value pairs and receive them directly from the network interface as if they were arguments.

We find that using a prescheduled computation schedule is more efficient than the dynamic, data-driven schedule often associated with dataflow computations. Consequently, the main loop (also shown in Figure 3) uses a prescheduled order, described later in Section 4.4, which allows each processor to process nodes in an efficient order. In our example, node 4 only has one incoming edge and its outgoing edges can be processed as soon as processor 4 arrives at node 4 in its prescheduled ordering.

4.1.2 Shared Memory

The use of shared memory is somewhat less obvious for this computation. The traditional method would be to schedule the nodes into phases (levels of a topological sort) and use barrier synchronization between levels. However, this has been found to be less efficient than finer-grain scheduling and synchronization [CSBS95]. The problem with shared memory is that processor 0 can perform a remote write to

```
void process_outgoing_edges(NODE_TYPE *node)
{
  int i;
  NODE_TYPE *dest_node;

  /* loop over all outgoing edges */
  for (i=0; i < node→num_out_edges; i++) {

     /* this is the destination of the current edge */
     dest_node = node→out_edge[i]→dest_node;

     /* spin until we get the lock on the cache line */
     lock_node(dest_node);

     /* update counter and data */
     dest_node→x += node→x *
                node→out_edge[i]→L;
     dest_node→counter−−;

     /* unlock the cache line */
     unlock_node(dest_node);
  }
}
```

Figure 4: Shared-memory code for DAG execution.

shared memory on processor 4 to communicate value v, but there is no synchronization event that tells processor 4 to subtract v from x_4.

Our problem arises from the **owner-computes** model, which specifies that the owner of the left-hand-side of the computation $(x_4 = x_4 - v)$, processor 4, must perform the subtraction. We can avoid this problem by adopting a **producer-computes** model, which specifies that the producer of the value v, processor 0, compute the subtraction via a remote read-modify-write using shared memory. In our implementations, processor 0 performs both the update to x_4 and the decrement of the presence counter of node 4 together, using a shared-memory synchronization primitives to keep the read-modify-writes atomic. Figure 4 outlines the code involved. The presence counter and the variable x are kept in the same 16-byte cache line. Generally, up to four messages are required for every non-local edge: a write ownership request to the home node, an invalidate to the previous writer, a cache-line transfer from the previous writer to the home node, and a cache-line transfer from the home node to the current writer. In our shared memory experiments, only two to three messages are required per non-local edge, compared to one for message passing, due to some caching when the previous writer is the same processor as the current writer. The main loop of the computation is the same as the message-passing version.

Operation	Cycles
Shared Memory	
Acquire+Release Remote Lock	unshared 58 + 1.5/hop
Acquire+Release Remote Lock	shared 144 + 1.5/hop
Read+Write Remote Data (Cached)	8
Compute Edge	40
Increment Presence Counter	2
Message Passing	
Message Send	31
Null Handler Interrupt	95
Compute Edge	40
Increment Presence Counter	2
Message Aggregation	
Buffer Edge	28

Table 2: Costs of relevant Alewife operations. An **unshared** remote lock means that messages only need to travel from the locking processor and the home processor of the lock. A **shared** remote lock must wait for an additional round-trip to invalidate other processors with cached copies of the lock. **hops** refers to network distance between the locking processor and the home processor in the unshared case; the shared case must also include network distance between the home node and the farthest sharer. Once the remote lock is acquired, the rest of the shared-memory operations are local and costs are similar to message-passing operations. **Compute Edge** refers to the memory and multiply-add operations associated with each edge of the computation.

Table 2 shows the costs of significant operations in using different communication mechanisms. We can see that the cost of buffering is significant. Sending long messages reduces message overhead by up to 126 cycles per every non-local edge, beyond the first, in a message. However, buffering cost reduces these savings by 22 percent. More importantly, buffering an edge for later transmission can cost nearly half as much as directly communicating the data via shared memory.

4.2 Overview of ICCG

Unlike previous studies [CSBS95], we have extensively studied the entire ICCG algorithm to ensure that our results are consistent with a complete, best-effort implementation. We obtain supporting data by instrumenting two state-of-the-art scientific packages: Chaco [HL95], a graph partitioner and mapper from Sandia National Laboratories; and BlockSolve [JP94], a conjugate gradient linear systems solver from Argonne National Laboratory.

Where does the triangular solve of our kernel come from? Our benchmarks are linear systems represented as $Ax = b$, where A is a sparse matrix, x is a vector of unknowns, and b is a right-hand side (RHS) vector of values. ICCG only works on A-matrices which are symmetric, positive, and definite.

1. Data Mapping (Spectral Bisection) 2. Matrix Reordering (Multicolor) 3. Incomplete Cholesky Factorization 4. Iterative Solution Triangular Solves (2) Inner Products (2) Vector Updates (3) Matrix-Vector Products (2)

Table 3: Computations for Parallel ICCG

$Ax = b$ can be solved directly by factoring A into $A = LU$, via Cholesky factorization, and performing two triangular solves: $Ly = b$ (forward substitution) and $Ux = y$ (backward substitution). Coarse-grain blocking approaches [RG93] perform well for this approach. Unfortunately, L and U can be expensive to compute and contain significantly more nonzeros than the original A. For this reason, an approximate factorization is often used. We compute an incomplete Cholesky factorization, where $L'_{ij} = L_{ij}$ if $A_{ij} \neq 0$, otherwise $L_{ij} = 0$. U' is similarly defined. L' and U' are exactly as sparse as A and we use them to iteratively arrive at a solution. Note that the sparsity of L' and U' make coarse-grain approaches impractical. The resulting iterative computation is inherently fine-grain and irregular.

Parallel implementation of ICCG involves four computation phases, shown in Table 3. First, an input matrix is mapped onto a target architecture, usually via a graph partitioner. Second, the matrix is renumbered via a reordering algorithm to provide better convergence and parallelism properties during factorization and solution. Third, a preconditioner is computed by calculating the incomplete Cholesky factors. Fourth, using an initial guess, we iteratively arrive at a solution.

For this study, we will assume that graph partitioning and matrix reordering can be done offline. This is because the applications we examine are irregular but not dynamic. They involve sparse matrices which are extremely long-lived and are solved repeatedly for multiple right-hand sides. For example, the finite-element discretization of an airplane is generally entered by hand. The structure of matrix A, representing the airplane, changes little over time. However, the RHS representing flying conditions changes continually.

Table 4 summarizes several attributes of our benchmark datasets. In particular, it shows that many iterations are required to solve these linear systems and that triangular solve takes at least 40 percent of runtime out of a total runtime which includes incomplete Cholesky and iterative solution. Iteration counts and timings are from BlockSolve running on top of MPI [Mes93] on a collection of Sun workstations connected by Ethernet. This system runs the same computation as would a full implementation of ICCG on Alewife. The data, partitions, and convergence would be the same. However, the overheads and latencies are substantially higher in the cluster than on Alewife. We are only interested in the iteration counts and a rough idea of what percentage of end-to-end application time is spent in triangular solve.

Matrix	Property	Number of Processors				
		1	4	8	16	32
BUS1138	percent nonlocal edges	0	6	15	19	27
	average iterations	255	303	314	133	114
	percent app iter	93	99	99	99	99
	percent app tri-solve	38	41	41	56	59
BCSSTK18	percent nonlocal edges	0	8	13	19	37
	average iterations	230	180	219	209	210
	percent app iter	88	91	97	98	99
	percent app tri-solve	40	46	55	54	65
CAN1072	percent nonlocal edges	0	9	21	32	40
BCSPWR10	percent nonlocal edges	0	10	20	45	54
OCEAN	percent nonlocal edges	-	-	-	2	4
BCSSTK32	percent nonlocal edges	-	-	-	-	24

Table 4: Locality, convergence, and rough ICCG timing attributes of some benchmarks. **nonlocal edges** shows that partitioning quality is generally high. OCEAN and BCSSTK32 do not fit on small numbers of processors. BUS1138 and BCSSTK18 have full numerical data and have the following data from BlockSolve: **percent app tri-solve** shows that we can expect triangular solve to be greater than approximately 40 percent of end-to-end time. **average iterations** shows the number of iterations for convergence. **percent app iter** shows the percentage of end-to-end time spent in iterative solution as opposed to factorization.

Because triangular solve has the finest grain communication of all the phases of ICCG, the percentage of triangular solve time will be at least as high on Alewife as in the single processor BlockSolve case. The single processor BlockSolve case has no communication cost. Parallel ICCG on Alewife can only increase the percentage of end-to-end time spent in triangular solve.

4.3 Data Mapping

Although our kernel treats our computation as a directed graph, we will use a data mapping algorithm for undirected graphs and choose directions for the edges later (using multicolor reordering). This approach produces better mappings and application performance than previous DAG-oriented approaches [CSBS95].

For data placement, we use Chaco, a well-established sequential code from Sandia National Laboratories which provides a range of mapping algorithms. We use a multilevel algorithm which uses a coarsening heuristic to condense the graph, and then partitions the condensed graph with recursive spectral bisection. Chaco assigns a mapping from partitions to processors on the parallel system and then refines it with the Kernighan-Lin algorithm [KL70].

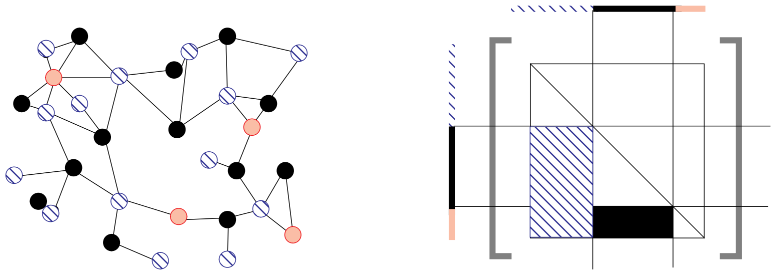


Figure 5: Graph coloring and resulting matrix structure using multicolor reordering and incomplete Cholesky factorization.

4.4 Multicolor Reordering and Message Aggregation

After the data is mapped to processors, we hold the mapping fixed but we can renumber the nodes. The mapping works on the undirected graph, which corresponds to treating the A-matrix as an adjacency matrix. We can renumber the nodes of the graph, effectively permuting the rows and columns of A (and the rows of x and b), without altering the linear system. The renumbered graph is isomorphic to the original graph, but the ordering is important because incomplete Cholesky factorization will produce triangular matrices with the same structure as whatever ordering A is in. The triangular matrices have the same structure as the original undirected graph (which makes the mapping useful), but the triangular structure determines the directions on all the edges. In both factorization and triangular solution, data will travel along each edge from node i to node j iff $i < j$.

Renumbering can enhance parallelism and convergence in the incomplete factorization and iterative solution phases. The edge directions of the DAG can be chosen to avoid long critical paths, yet still propagate information quickly for convergence. Renumbering can also result in greater **slackness**, amount of computation able to proceed between communications, which creates more data available to be aggregated into long messages. Long messages are often used to try to amortize communication overhead. We use the **multicolor reordering** algorithm [ST82], described later in this section, which is one of the best known algorithms for enhancing parallelism and message aggregation.

Multicolor reorderings are similar to red-black relaxation. Nodes are colored to ensure independence and then the computation works on the nodes one color at a time. Specifically, the nodes of the graph are colored such that no edge connects nodes of the same color, as shown on the left side of Figure 5. Then the matrix A is renumbered such that the row and column numbers are sorted by color. After incomplete factorization, this results in triangular matrices with nonzero structure as on the right side of Figure 5. Each block of nonzeros represents a portion of computation which can proceed in parallel, dependent only upon blocks earlier in the sorted ordering of colors (blocks to the left in the lower triangular matrix).

The sorting of the colors effectively provides us with a topological sort of our

Matrix	Buffer length	Number Processors 4	8	16	32
BUS1138	2	1.8	1.7	1.5	1.3
	3	2.5	2.1	1.7	1.4
	4	3.1	2.4	1.8	1.5
BCSSTK18	2	2.0	2.0	1.9	1.8
	3	3.0	2.9	2.6	2.3
	4	3.9	3.8	3.3	2.6
CAN1072	2	2.0	1.9	1.7	1.6
	3	2.8	2.6	2.1	1.9
	4	3.6	3.2	2.4	2.0
BCSPWR10	2	2.0	2.0	1.9	1.7
	3	3.0	2.8	2.6	2.2
	4	3.8	3.6	3.3	2.5
OCEAN	2	-	-	2.0	2.0
	3	-	-	2.9	2.9
	4	-	-	3.8	3.8
BCSSTK32	2	-	-	-	1.9
	3	-	-	-	2.7
	4	-	-	-	3.4

Table 5: Average message length for a given buffer size and number of processors. Buffer size and message length in units of number of non-local DAG edges (four 32-bit words).

computation DAG. If each processor completes[3] its nodes in order sorted by the multicolor numbering, the computation is deadlock-free. In fact, this ordering turns out to be an extremely efficient computation schedule which outperforms data-driven schedules generated at runtime.

The sorted dependencies also allow the results of each block to be buffered till the end of the computation of the entire block without deadlock. This maximizes opportunities for message aggregation within each block. In our implementations, a processor will communicate buffered results whenever it is idle, rather than waiting till the end of each block. Table 5 shows the average length of messages for each benchmark at given buffer sizes and number of processors.

If we can color large datasets with a relatively small number of colors, blocks will be large, resulting in high parallelism and aggregation. Recall that Table 1 gave the number of colors for each benchmark. Our results will show high parallelism on our benchmarks. Unfortunately, even if a block is large, it is distributed among p processors and its results are sent to p processors. Consequently, the amount of data available for long messages is proportional to the results of the block divided by p^2. This division causes even our largest benchmarks to benefit little from message aggregation.

[3]Recall that, depending upon our mechanism-dependent implementations, a processor may perform varying amounts of the computation of a node that it “owns”, but it will always “complete” every owned node by checking input dependencies and passing results along outgoing edges.

How do multicolor orderings affect convergence and what are the benefits of increased parallelism? Studies [DM89], on similar data-sets to ours, indicate that the convergence rate of multicolor orderings is within roughly a factor of two of the best reorderings optimized for convergence. The increased parallelism allows our triangular solve speedups scale to much higher number of processors than with other orderings in previous studies [CSBS95].

5 Results

In this section, we present performance results from our experiments on Alewife. Our experiments focus on three implementations. Each implementation uses a different mechanism to communicate data along non-local edges of our computation DAG, edges between DAG nodes on different processors. For intuition behind each implementation, refer back to Section 4.1.

First, **shared memory** uses a global read-modify-writes for each non-local edge. Second, **short message** uses an active message for each non-local edge. Third, **buffer n** buffers up to n non-local edges (each 4 32-bit words) in local memory before sending a message. Messages containing four or less non-local edges are sent via an active message. Messages containing more than four are sent via an active message with integrated DMA. We divide our results into two sections. First, we show that there is no benefit to message aggregation in our smaller benchmarks. Second, we show that the benefits of aggregation are extremely limited in our large benchmarks. In both sections, we show that shared memory performs well relative to message passing, especially in the large benchmarks when receive occupancy is critical to network congestion.

5.1 Smaller Benchmarks

Figure 6 shows Alewife speedups for our four smaller benchmarks with each of our communication mechanisms. Recall speedups are relative to optimized sequential code running with the assumption that datasets fit in memory (see Table 1). We see that shared memory and short active messages perform equally well. Note that buffered messages of length 1 are essentially short active messages with the overhead of buffering. We see that increasing the number of edges buffered into longer messages **decreases** performance for our smaller benchmarks.

The primary reason performance decreases with longer messages is idle time. This idle time results from waiting for enough data to aggregate into a long message. Additionally, savings in communication is reduced by the buffering costs discussed in Section 4.1. We can see these effects in Figure 7, which illustrates non-local computation and idle time for BCSSTK18, which is representative of all four small benchmarks. Non-local computation represents the average time spent on computation and communication overhead on each non-local edge. Idle time represents average time spent in spin wait for each node to have all its incoming edges satisfied. While the non-local computation time increases more slowly

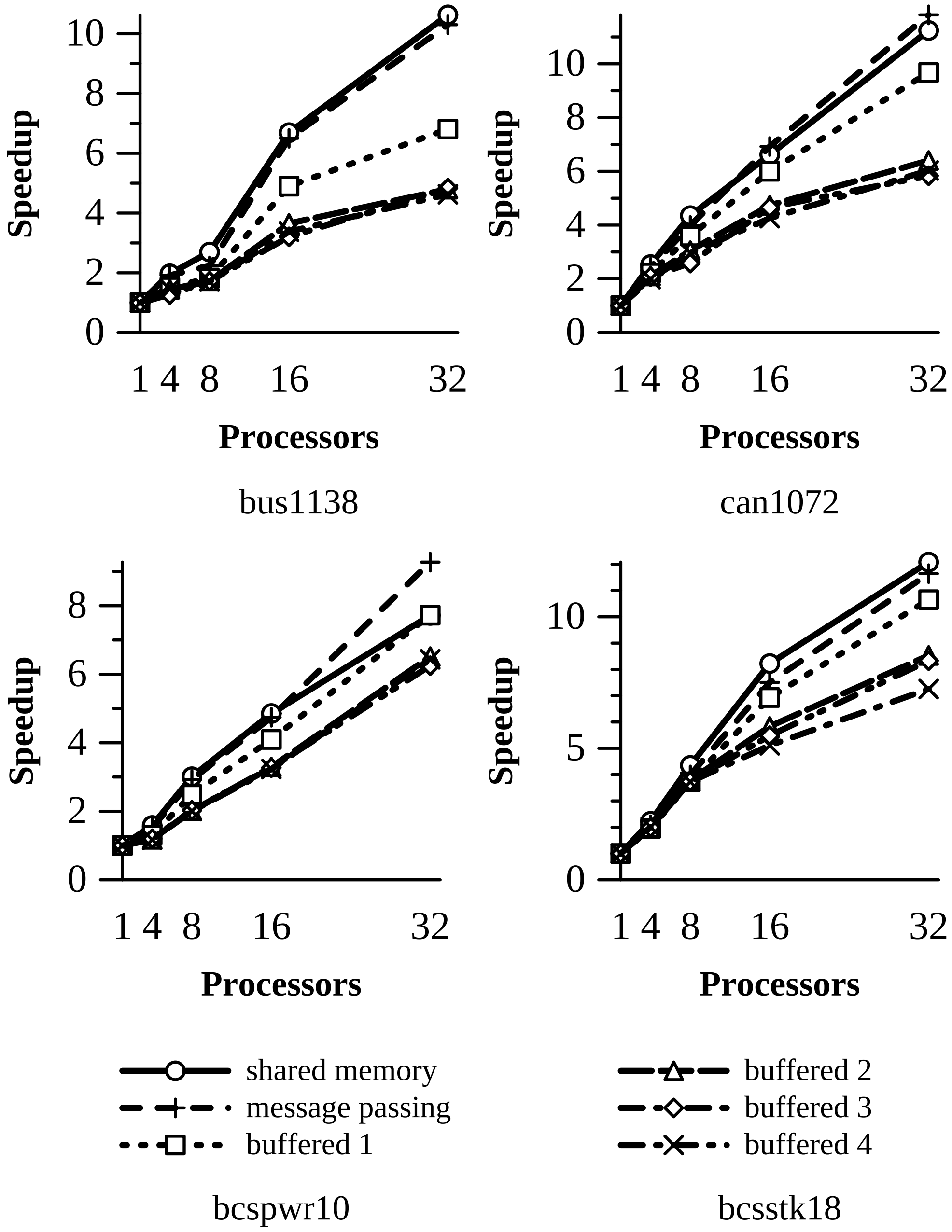
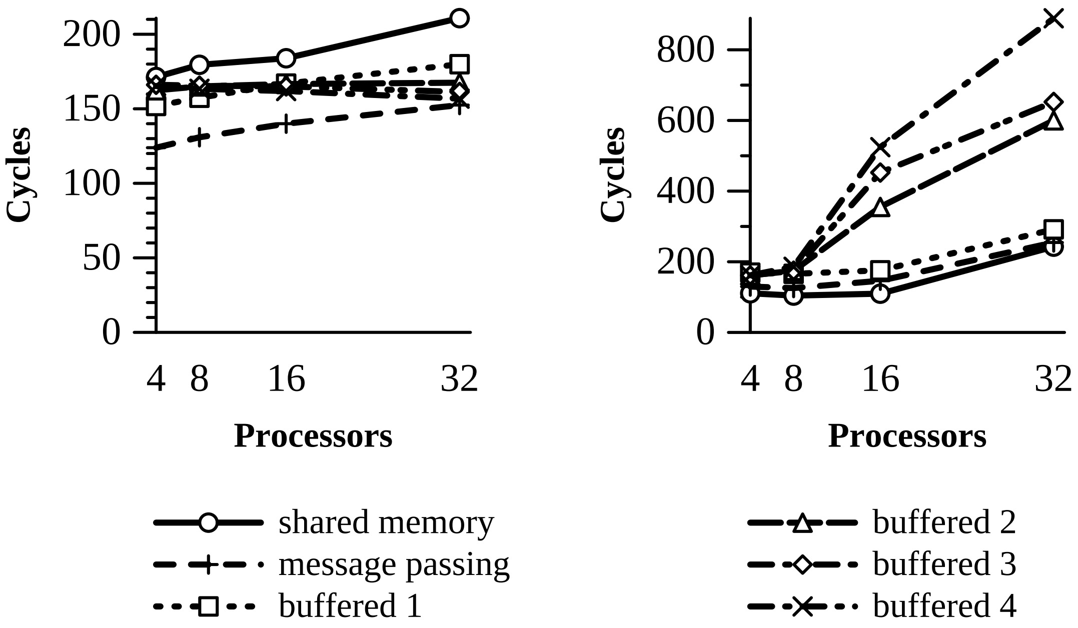


Figure 6: Alewife Speedups



bcsstk18 per edge nonlocal times   bcsstk18 per node idle time

Figure 7: BCSSTK18 breakdowns.

as we use longer messages, this benefit is made insignificant by the rapidly increasing idle time which results from aggregation. This rapid increase occurs even though we have specifically used a multicolor ordering to maximize available data for aggregation and allow overlap with computation.

5.2 Larger Benchmarks and DMA

We expect longer messages to perform better on our two larger benchmarks. Table 6 gives speedups and breakdowns for our two larger benchmarks. However, BCSSTK32 is the only benchmark that shows any improvement as messages become larger. This is because idle time actually **decreases** somewhat as we buffer more messages. This is counter-intuitive, because idle time increased on all our other benchmarks because of time spent waiting for edges to buffer up. It turns out that network congestion is a significant factor in BCSSTK32. The larger dataset size and higher degree of the nodes (lower sparsity of the matrix) result in higher communication. The higher communication volume of shorter messages causes increased network congestion, resulting in higher latencies and idle time[4]. The **short message** and **buffered 1** cases even result in network overflow traps, a software trap which inhibits message sends and empties the network of congestion. Longer messages decrease communication volume by decreasing the number of message headers.

[4] Note that idle times seem extremely high because they are per node and each node of BCSSTK32 averages about 22 incoming edges.

	ocean		bcsstk32
	16 proc	32 proc	32 proc
Speedup			
shared memory	8.0	15.6	12.9
short message	7.8	15.3	11.3
buffered 4	6.8	12.0	11.3
buffered 3	6.8	11.8	11.2
buffered 2	6.9	11.8	11.1
buffered 1	7.0	13.7	10.8
Non-local edge cycles			
shared memory	202	206	189
short message	141	139	130
buffered 4	202	194	140
buffered 3	199	195	145
buffered 2	196	190	149
buffered 1	178	174	151
Idle cycles per node			
shared memory	52	57	1444
short message	55	61	1821
buffered 4	107	138	1631
buffered 3	101	132	1647
buffered 2	100	125	1671
buffered 1	90	96	1868

Table 6: Large benchmark speedups and breakdowns.

Why does shared memory do so well when it has double the communication volume of short messages? The answer is that shared memory messages are handled with very low occupancy by the CMMU rather than via a processor interrupt. This results in a higher receive rate which keeps the network clear of congestion.

Will the performance of **bcsstk32** continue to improve as we increase message size? To send messages which contain more than 4 edges, we need to use DMA. The left side of Figure 8 shows speedup for 32 processors as message buffering increases. For reference, speedups for shared memory and short message implementations are also plotted as horizontal lines.

The right side of Figure 8 shows the key limitation to message buffering. As buffering increases, the average length of messages quickly reaches a limit caused by dependencies in the program. Recall that, in order to avoid deadlock and satisfy data dependencies, a message may need to be sent before filling a buffer. As mentioned in Section 4.4, the amount of independent data available for aggregation is divided by p^2, where p is the number of processors. BCSSTK32 is limited to an average message length of about 8 edges (128 bytes), and consequently receives the most benefit from buffer size 8. OCEAN is similarly limited. Our smaller benchmarks have message lengths which are too small receive no benefit from aggregation.

How do these results generalize to systems with higher communication over-

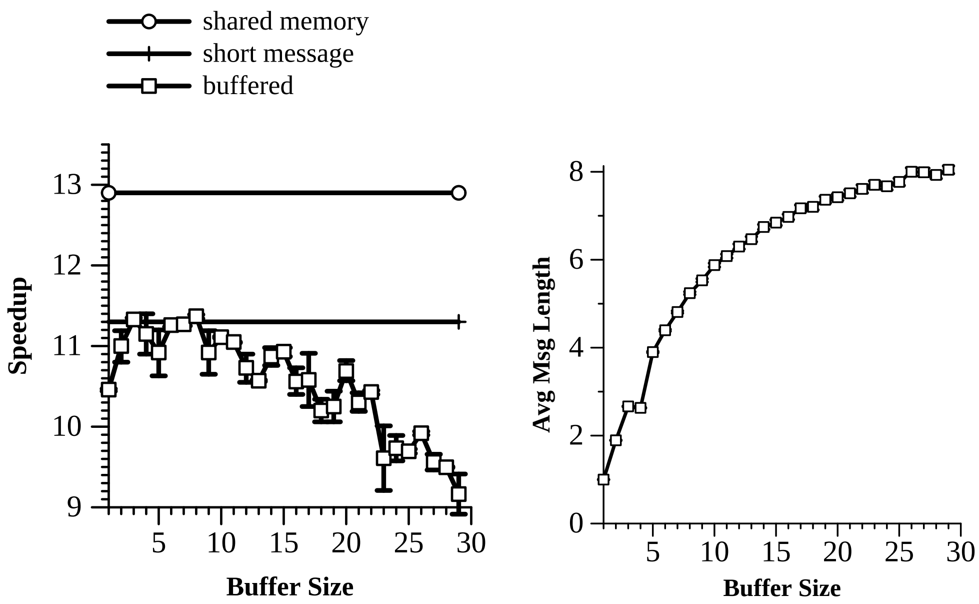


Figure 8: **(Left)** BCSSTK32 with larger buffers on 32 processors. Buffering is in terms of non-local edges of the computation. Buffering of 10 indicates that up to 160 bytes of data are sent in a message. **(Right)** Average number of edges sent per message at each buffering level for this benchmark on 32 processors.

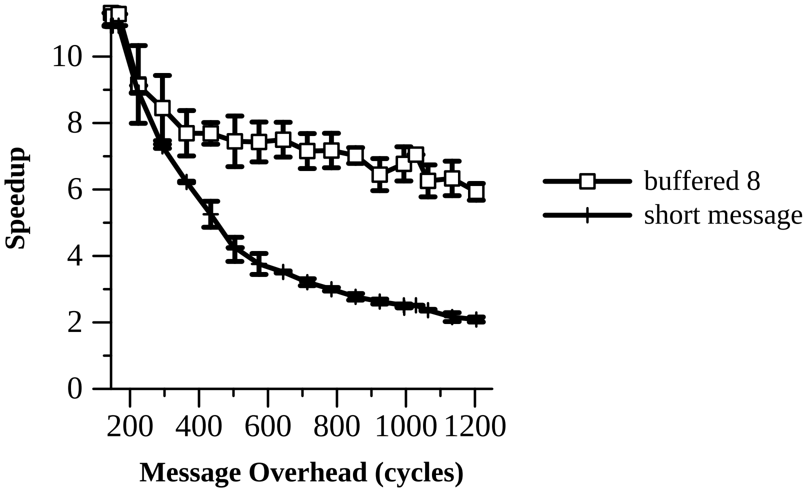


Figure 9: BCSSTK32 with speedups on systems with increasing communication overhead. Short messages (16 bytes) and buffer-size-8 (128 bytes) messages are shown.

heads? Figure 9 compares short messages to buffer-size-8 messages for send-receive overheads ranging from Alewife's 126 cycles to a hefty 1200 cycles. Overheads were simulated on Alewife by inserting delays in message send primitives and in active message handlers. As we expect, aggregated messaging is more tolerant of increased overheads. However, aggregation is need not be very large, less than 128 bytes for our benchmarks. Such limited aggregation is more in the class of short messaging mechanisms than bulk transfer. In fact, most DMA engines are only just breaking even with direct messaging mechanisms at this point. Consequently, our applications do not justify the inclusion of DMA hardware in a system, even when communication overheads are high.

Larger benchmarks will not be possible until Alewife is expanded to a 128-node system. However, our largest benchmark, with 2 million elements, is a respectable size and comparable to the largest experiments reported in the literature.

In summary, long messages only show some benefits in a small regime for our benchmarks and machine sizes. As we move to larger numbers of processors, congestion may become more of a factor, even on our smaller and sparser benchmarks then BCCSTK32. However, data dependencies limit the benefits of message aggregation. Shared memory performs well, with its low occupancy becoming an important factor when network congestion is an issue.

6 Related Work

Numerous studies have argued for hardware support for efficient cache-coherent shared memory, primarily to increase the ease of use of multiprocessors. Our study agrees that ease of use is important, but we also find that shared memory is an extremely efficient communication mechanism which outperforms message passing and DMA in our applications.

Many studies have examined shared memory without thorough comparison with message passing and DMA. Yeung and Agarwal [YA93] explored fine-grain synchronization and language support for preconditioned conjugate gradient on regular problems on Alewife. Singh, Holt, and Hennessy [SHH95] studied hierarchical N-body methods on distributed shared memory machines. They found significant benefits from caching due to re-use in their applications.

Woo, Sing, and Hennessy [WSH94] also found the advantages of bulk transfer over efficient shared memory to be limited. Their applications were not as irregular and aggregation was not as expensive. However, they were able to gain many of the benefits of bulk transfer by using large cache lines and prefetching.

Our use of a producer-computes model to cope with shared memory synchronization is similar to the Remote Queues concept presented in [BCL+95].

Mukherjee et al [MSH+95] recently studied fine-grain, irregular applications on the Chaos system [SMC91] for message passing and software distributed shared memory on the CM5. Lu et al [LDCZ95] looked at different applications coded for PVM message passing and Treadmarks software DSM.

These studies deal with an entirely different regime of overheads associated with software DSM. These overheads make communication aggregation and relaxed consistency models crucial to achieving acceptable application performance. Unfortunately, aggregation techniques are not applicable to our applications and only hardware-supported shared memory can achieve acceptable performance. On the plus side, efficient hardware support can provide good performance without relaxing memory models. While relaxed models such as Wisconsin's **delayed-update** model [FLR$^+$94] are complementary to this study and could improve our application performance, our results show that we can still achieve acceptable performance on an architecture which chooses not to support application-specific models.

The goal of our study is to discover architectural implications of sparse, irregular applications through thorough study of a specific class of such applications. However, our results should generalize well to runtime systems such as Chaos and to parallelizing compilers as they develop towards handling sparse and irregular codes.

7 Conclusion

From in-depth study of a set of practical, sparse, irregular problems implemented with shared memory, message passing, and DMA on the MIT Alewife multiprocessor, we conclude that cache-coherent shared memory is the most general and effective multiprocessor communication mechanism for such applications.

We discover that a producer-computes model of computation can avoid awkward synchronization problems that arise between owner-computes and shared memory. We find that fine-grain communication may often out-perform bulk transfers. Message aggregation can be expensive relative to fast fine-grain mechanisms such as shared memory or short active messages. Buffering data to local memory for later DMA can cost nearly half as much as directly communicating the data via shared memory. Moreover, processor idle time and network congestion make bulk transfer unattractive for all but a small regime of message sizes (about 128 bytes) on large datasets (matrices with over 2 million elements). In fact, the low receive overhead of shared memory message traffic avoids network congestion when message passing or DMA can not.

8 Acknowlegements

Thanks to Fredrik Dahlgren, Matt Frank, Steve Keckler, Kathy Knobe, John Kubiatowicz, Joel Saltz, Rob Schreiber, Shamik Sharma and Deborah Wallach.

The Alewife project is funded in part by ARPA contract N00014-94-1-0985 and in part by NSF grant MIP-9504399. This and related documents are available from http://www.ai.mit.edu/people/ftchong/

References

[ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In **Proc. 22nd Annual International Symposium on Computer Architecture**, June 1995.

[ACM88] Arvind, David E. Culler, and Gino K. Maa. Assessing the benefits of fine-grained parallelism in dataflow programs. In **Supercomputing '88**. IEEE, 1988.

[BCL+95] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz. Remote queues: Exposing message queues for optimization and atomicity. In **1995 Symposium on Parallel Architectures and Algorithms**, Santa Barbara, California, July 1995.

[BFKR92] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.

[CB96] David Conroy and Lance Berc. Personal communication, Digitial Systems Research Center, March 1996.

[Che93] Chesney. The Meiko CS-2 system architecture. In **Annual ACM Symposium on Parallel Algorithms and Architectures**, 1993.

[CLR94] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs. In **ASPLOS VI**, pages 61–73, San Jose, California, 1994.

[CS95] Frederic T. Chong and Robert Schreiber. Parallel sparse triangular solution with partitioned inverses and prescheduled DAGs. In **1995 Workshop on Solving Irregular Problems on Distributed Memory Machines**, Santa Barbara, California, April 1995.

[CSBS95] Frederic T. Chong, Shamik D. Sharma, Eric A. Brewer, and Joel Saltz. Multiprocessor runtime support for irregular DAGs. **Parallel Processing Letters: Special Issue on Partitioning and Scheduling for Parallel and Distributed Systems**, pages 671–683, December 1995.

[DCB+94] André DeHon, Frederic Chong, Matthew Becker, Eran Egozy, Henry Minsky, Samuel Peretz, and Thomas F. Knight, Jr. METRO: A router architecture for high-performance, short-haul routing networks. In **Proceedings of the International Symposium on Computer Architecture**, pages 266–277, May 1994.

[DGL92] Ian S. Duff, Roger G. Grimes, and John G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection. Technical Report TR/PA/92/86, CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France, October 1992.

[DM89] Iain S. Duff and Geŕard A. Meurant. The effect of ordering on preconditioned conjugate gradients. **BIT**, 29:635–657, 1989.

[E+92] Thorsten von Eicken et al. Active messages: a mechanism for integrated communication and computation. In **Proceedings of the 19th Annual Symposium on Computer Architecture**, Queensland, Australia, May 1992.

[FLR+94] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill James R. Larus, Anne Rogers, and David A. Wood. Application-specific protocols for user-level shared memory. In **Supercomputing 94**, 1994.

[GvL83] G. Golub and C. F. van Loan. **Matrix Computations**. John Hopkins University Press, Baltimore, 1983.

[HKO+94] Mark Heinrish, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The performance impact of flexibility in the Stanford FLASH multiprocessor. In **ASPLOS VI**, pages 274–285, San Jose, California, 1994.

[HL95] Bruce Hendrickson and Robert Leland. The Chaco user's guide. Technical Report SAND94-2692, Sandia National Laboratories, July 1995.

[Int91] Paragon XP/S product overview. Intel Corporation, 1991.

[JP94] Mark T. Jones and Paul E. Plassman. BlockSolve v2.0: Scalable library software for the parallel solution of sparse linear systems. ANL Report (updated draft) 92-46, Argonne National Laboratory, October 1994.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. **The Bell System Technical Journal**, pages 291–307, February 1970.

[LDCZ95] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In **Supercomputing 95**, December 1995.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford Dash multiprocessor. **Computer**, 25(3):63–80, March 1992.

[LT88] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In **Proceedings of the 1988 International Conference on Parallel Processing, Vol. I Architecture**, pages 303–310. , University Park, Pennsylvania, [8] 1988.

[Mes93] Message Passing Interface Forum. MPI: A message passing interface. In **Supercomputing '93**, pages 878–883. IEEE, 1993.

[MSH+95] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. In **Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95**, pages 68–79, Santa Barbara, California, July 1995.

[RG93] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In **Supercomputing '93**, pages 503–512. IEEE, 1993.

[SGI94] Power Challenge technical report. Technical report, Silicon Graphics Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043, 1994.

[SHH95] Jaswinder Pal Singh, Chris Holt, and John Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-hut, fast multipole, and radiosity. **Journal of Parallel and Distributed Computing**, 27(2), June 1995.

[SMC91] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. **IEEE Transactions on Computers**, pages 603–611, 1991.

[ST82] R. Schreiber and W. Tang. Vectorizing the conjugate gradient method. In **Proceedings Symposium CYBER 205 Applications**, Ft. Collins, CO, 1982.

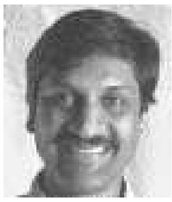
[Thi93a] Thinking Machines Corporation, Cambridge, MA. **CM-5 Technical Summary**, November 1993.

[Thi93b] Thinking Machines Corporation, Cambridge, MA. **CMMD Reference Manual (Version 3.0)**, May 1993.

[WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In **Asplos VI**, pages 219–229, San Jose, California, 1994.

[YA93] Donald Yeung and Anant Agarwal. Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient. In **Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming**, pages 187–197, San Diego, California, May 1993.

Fred Chong is a Ph.D. student advised by Anant Agarwal at MIT. He received his S.B. in 1990 and S.M. in 1992 from MIT. His research interests include communication, applications, theory, and VLSI for parallel systems.



Anant Agarwal received his B.Tech at the Indian Institute of Technology in Madras, India, in 1982, and his M.S. and Ph.D. at Stanford University in 1987. Currently, he is an Assistant Professor of Computer Science and Electrical Engineering at MIT, where he leads the Alewife Project.