The MIT Alewife Machine: Architecture and Performance

Anant Agarwal, Ricardo Bianchini[*], David Chaiken[†], Kirk L. Johnson,
David Kranz, John Kubiatowicz, Beng-Hong Lim[‡], Kenneth Mackenzie, and Donald Yeung
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

Alewife is a multiprocessor architecture that supports up to 512 processing nodes connected over a scalable and cost-effective mesh network at a constant cost per node. The MIT Alewife machine, a prototype implementation of the architecture, demonstrates that a parallel system can be both scalable and programmable. Four mechanisms combine to achieve these goals: software-extended coherent shared memory provides a global, linear address space; integrated message passing allows compiler and operating system designers to provide efficient communication and synchronization; support for fine-grain computation allows many processors to cooperate on small problem sizes; and latency tolerance mechanisms – including block multithreading and prefetching – mask unavoidable delays due to communication.

Microbenchmarks, together with over a dozen complete applications running on the 32-node prototype, help to analyze the behavior of the system. Analysis shows that integrating message passing with shared memory enables a cost-efficient solution to the cache coherence problem and provides a rich set of programming primitives. Block multithreading and prefetching improve performance by up to 25% individually, and 35% together. Finally, language constructs that allow programmers to express fine-grain synchronization can improve performance by over a factor of two.

1 Introduction

The last few years have seen the introduction of a number of parallel-processing systems with truly impressive maximum performance. The amount of raw computation packaged in a single chassis is quickly approaching a trillion operations per second. Unfortunately, end-users rarely benefit from the advertised maximum performance of today's massively parallel systems. Any application that actually exploits the full potential of a machine typically requires months of careful programming, painful debugging, and relentless tuning.

The MIT Alewife machine shows that a parallel architecture can yield a rich shared memory programming environment on a scalable hardware base. The hardware, compiler, and operating system combine to achieve the goal of *programmability* by solving problems that traditionally burden multiprocessor programmers; namely, scheduling computation and moving data between processing elements. Features of this environment include a globally shared address space, a scalable cache coherence mechanism, a compiler that automatically partitions regular programs with loops, a library of efficient synchronization and communication routines, distributed garbage collection, and a parallel debugger. These features allow programmers to write applications quickly. Statistics-gathering tools help to optimize performance.

The goal of *scalability* addresses both the cost of building the machine and its ability to run programs efficiently. The Alewife architecture permits a physically scalable implementation: Alewife machines are built by replicating a single, modular processing node. Passive backplanes provide the wires to connect the nodes in a low-cost, two-dimensional mesh network. In order to provide I/O facilities, VME and SCSI interface boards plug into the edges of the mesh. Whether an Alewife machine has one node or 512 nodes, this physical layout results in a constant cost per node. In the prototype, despite unit quantity purchasing, a single-node costs only about $2,000. With volume fabrication, this cost can be reduced substantially.

This paper shows how the hardware and software components of Alewife provide good performance on parallel applications, without sacrificing physical scalability or programmability. Indeed, most applications were written for other machines and run on Alewife without significant porting effort. The primary challenge to achieving these goals simultaneously is the latency of interprocessor communication, which dominates the time required for intranode memory accesses. Therefore, Alewife provides four classes of architectural mechanisms that implement an *automatic locality management* strategy. This strategy seeks to maximize the amount of local communication by consolidating related blocks of computation and data, and attempts to minimize the effects of non-local communication when it is unavoidable. The four classes of mechanisms are: coherent caches for shared memory, integrated message passing, support for fine-grain computation, and latency tolerance.

Coherent shared memory Although Alewife provides the abstraction of globally shared memory to programmers, the system's physical memory is statically distributed over the nodes in the machine. On each node, a Communications and Memory Manage-

[*]Affiliation: University of Rochester, Rochester, NY 14627

[†]Current affiliation: Digital Equipment Corporation Systems Research Center, Palo Alto, CA 94301

[‡]Current affiliation: IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

This paper to appear in ISCA '95

ment Unit (CMMU)[18] fields memory requests from a Sparcle processor[2] and determines whether requests access local or remote memory. When necessary, the CMMU synthesizes messages that fetch memory from remote nodes.

The memory hardware helps manage locality by caching both private and shared data on each node. A scalable, software-extended scheme called LimitLESS[9] maintains the coherence of cached data. This scheme handles common-case memory accesses in the CMMU hardware, but relies on software traps to enforce coherence for memory blocks that are shared by a large number of processors.

Integrated message passing While the programmer sees a shared memory programming model, for performance reasons much of the underlying software is implemented using message passing. The performance of all of the layers of software that help manage locality (including the compiler, libraries, run-time system, and LimitLESS handlers) depend on an efficient communication mechanism. Features in Sparcle and the CMMU combine to provide a streamlined interface for transmitting and receiving messages: both system and user code can quickly describe and atomically launch a packet directly into the interconnection network; a fast interrupt mechanism speeds message reception; and a direct memory access (DMA) mechanism allows data to flow between the network and memory.

The Alewife hardware supports a seamless interface between the various software layers by integrating the shared memory and message-passing mechanisms. To do so, the system provides forward progress guarantees to shared memory accesses in the face of message reception interrupts. In addition, the DMA engine maintains the coherence between the data in messages and the data in local caches [16].

Fine-grain computation Given a fixed-size data set, the granularity of computation (the time between events that require interprocessor communication) decreases as the number of processors in a system increases. A system that cannot handle small tasks efficiently must attempt to increase synchronization and communication granularity artificially, possibly defeating attempts to manage locality. Alewife's support for fine-grain computation includes fast, user-level messages and support for full/empty bit synchronization.

Alewife's programming languages, parallel C and Mul-T, include constructs for expressing fine-grain synchronization. These constructs allow a thread to synchronize implicitly upon every memory access.

Latency tolerance Block multithreading and prefetching provide the last line of defense in Alewife's locality management strategy. These mechanisms attempt to tolerate the latency of interprocessor communication when it cannot be avoided. Prefetching allows code to anticipate communication by requesting data or locks before they are needed. Block multithreading allows a processor to switch between threads of computation on a cache miss or a failed synchronization attempt.

Latency tolerance requires support from Alewife's hardware and software components. Prefetching and block multithreading both require lockup-free caches[15]. Prefetching requires support in the compiler and special memory operations. Block multithreading requires a fast context switch[3] and a solution to the window of vulnerability problem created by interleaved threads of execution[17].

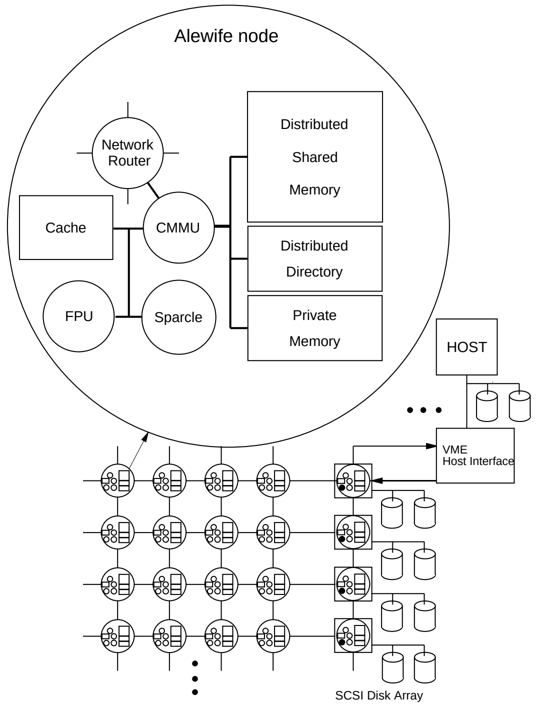

Figure 1: The Alewife architecture.

Although it is helpful to think of Alewife's mechanisms as belonging to four distinct classes, the machine's implementation integrates them tightly. For example, the CMMU's transaction buffer closes the window of vulnerability opened not only by multithreading, but also by fast message handling and software-extended coherence. The transaction buffer also provides storage for prefetching. Similarly, Sparcle's fast interrupt mechanism accelerates LimitLESS event handling, message reception, fine-grain synchronization events, and context switching.

This paper describes the experience gained by designing, fabricating, and running a complete parallel system. Specifically, it evaluates the effectiveness of the Alewife architecture and its locality management strategy. Section 2 describes the machine's implementation and its programming environment to show how the mechanisms combine to produce a coherent system. Section 3 describes Alewife's primitive mechanisms and uses microbenchmarks to measure the base performance of the mechanisms in terms of the latency and bandwidth of primitive functions. Section 4 presents detailed case-studies of two applications that illustrate the benefits of Alewife's approach. Section 5 discusses related work on parallel architectures. Finally, Section 6 summarizes the insight gained from implementing Alewife and describes plans for future research.

2 The Alewife Machine

The Alewife architecture is organized as shown in Figure 1. Memory is physically distributed over the processing nodes, which use a mesh network for communication.

Each Alewife node consists of a Sparcle[2] processor, 64K bytes of direct-mapped cache, 4M bytes of data and 2M bytes of directory (to support a 4M byte portion of shared memory), 2M bytes of private (unshared) memory, a floating-point coprocessor, and an Elko-series

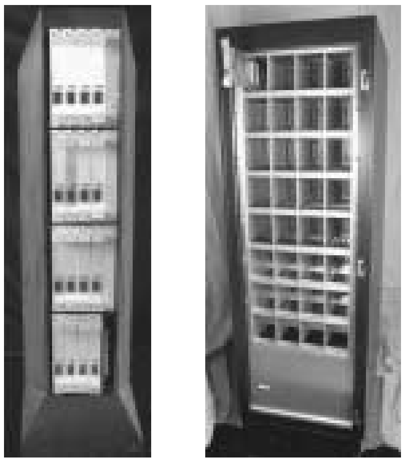
Figure 2: 16-node machine and 128-node chassis.

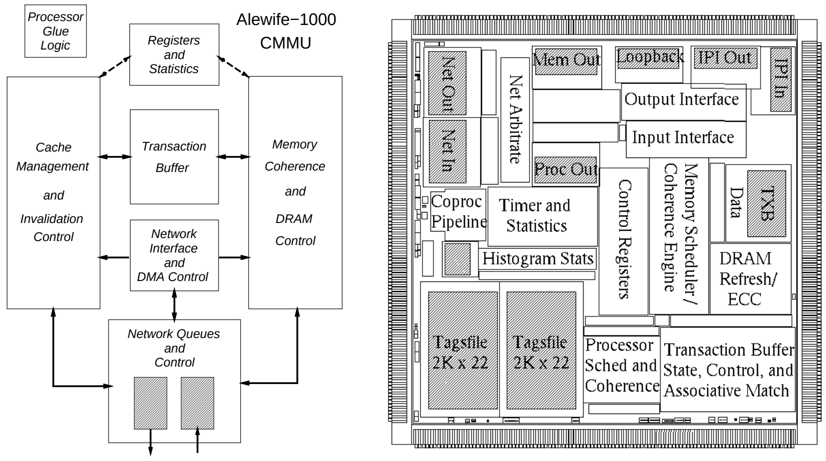

Figure 3: Block diagram and floorplan for the CMMU (15mm × 15mm).

mesh routing chip (EMRC) from Caltech. Both the cache memories and floating-point unit (FPU) are off-the-shelf, SPARC-compatible components. The EMRC network routers use wormhole routing and are connected to form a direct network with a mesh topology. The nodes communicate via messages through this network. A single-chip Communications and Memory Management Unit (CMMU) services data requests from the processor and network. I/O is provided by a SCSI disk array attached to the edges of the mesh network.

Figure 2 shows the physical realization of a 16-node Alewife system and the chassis for 128 nodes. The 16-node system, complete with two internal $3\frac{1}{2}$-inch disk drives, is about $74\times12\times46$ cm, roughly the size of a floor-standing workstation. Packaging for a 128-node machine occupies a standard 19-inch rack. Timing numbers in this paper reflect a 32-node Alewife machine, packaged in the lower right quarter of the 128-node chassis.

User access to an Alewife machine is through a host workstation. Client interface software connects to the Alewife machine via UNIX sockets to a server process running on the host. External NFS file access is also provided by the host.

The first Alewife machine became operational in May, 1994. Results in this paper were obtained using first-silicon versions of Sparcle and the CMMU. Although there are a few bugs in the CMMU, all of them have software work-arounds. However, one of the bugs involves a timing conflict with the FPU, requiring operation at 20 MHz when floating-point is in use. Integer codes run at 30 MHz. A planned respin of the CMMU will correct these bugs and boost performance to the intended 33 MHz. For consistency, the remainder of this paper will quote performance numbers at a 20 MHz system speed.

2.1 Sparcle processor

Sparcle was derived from an industry-standard SPARC (version 7) processor. It provides an efficient and tight coupling between the processor pipeline and the communications network. Many of the features of the underlying SPARC implementation are exploited directly by Alewife: for example, the SPARC external coprocessor interface is used for fast messaging, interrupt control, and fine-grained synchronization. SPARC also provides register windows that can be exploited as separate contexts for block multithreading.

Sparcle augments the basic SPARC architecture with a few simple mechanisms to facilitate rapid messaging, block multithreading, and fine-grain synchronization:

- **User-level colored loads and stores.** The SPARC architecture defines an 8-bit *Alternate Space Indicator* (ASI) that serves to tag all load and store operations with one of 256 different “colors”. Sparcle allocates the top 128 ASI values to the user and defines new load and store instructions that emit these ASI values.
- **Extra synchronous trap lines.** These lines enable unique trap vectors for context-switch and fine-grain synchronization traps.
- **Context management instructions.** New instructions allow rapid switching between active hardware contexts. The SPARC current window pointer is visible at the pins, permitting context-dependent state in the CMMU and FPU.

These changes require an increase of fewer than 2000 gates over the unmodified SPARC design. Together, they yield a processor with support for low-overhead communication, including a 14-cycle context-switch time for a remote data cache miss.

2.2 The Alewife CMMU

The Alewife CMMU[18] implements most of the unique functionality of Alewife. In an Alewife node, the CMMU is connected directly to the first-level cache bus and serves much the same functionality as a cache-controller/memory-management unit in a uniprocessor. It contains tags for the cache, provides DRAM refresh and ECC, and handles cache fills and replacements. In addition, it implements the architectural mechanisms described in this paper. The CMMU also provides asynchronous queueing for the EMRC network chips and a number of hardware statistics facilities.

Figure 3 shows a block diagram of this chip. The *Processor Glue Logic* is responsible for interpreting colored memory operations and coprocessor instruction requests. The *Cache Management and Invalidation Control* and *Memory Coherence and DRAM Control* blocks comprise, respectively, the processor and memory portions of the cache coherence protocol. In addition, both blocks service requests from the *Network Interface and DMA Control* block, which provides user-level message passing with locally coherent DMA[16]. Since the processor and memory sides of the cache coherence protocol

Category	Gate Count	%	Mechanism			
			SM	MP	LT	FG
Processor Requests	11686	12	√		√	√
Full/Empty Decode	2157	2				√
Memory Service	13351	13	√	√		
DRAM Control	8720	9	√	√		
Livelock Removal	2108	2	√		√	
Transaction Buffer	17399	17	√	√	√	
IPI interface	11805	12	√	√		
Network Queueing	7363	7	√	√	√	√
CMMU Registers	9308	9	√	√		
Statistics	11958	12				
Miscellaneous	4627	5				

Table 1: Functional block sizes (in gates) for the Alewife CMMU, as well as contributions to shared memory (SM), message passing (MP), latency tolerance (LT), and fine-grain synchronization (FG). Total chip resources: 100K gates and 100K bits of SRAM.

as well as the message-passing interfaces share the same network queues, message passing and shared memory are integrated[14].

The *Transaction Buffer* is a 16-entry, fully-associative data store that tracks outstanding cache coherence transactions, holds prefetched data, and stages data in transit between the cache, network and memory. It is integrated closely with the mechanism for removing livelock in the face of block multithreading [17]. The *Registers and Statistics* block contains a dedicated cycle counter, a timer, and a number of statistics facilities. The *Network Queues and Control* block contains asynchronous interfaces for the EMRC network routers.

Figure 3 also shows a floorplan of the CMMU. This chip is implemented with three layers of metal in the LEA-300K hybrid gate-array technology from LSI Logic. Shaded blocks are standard-cell memories. The rest of the chip is implemented in a sea-of-gates style; costs for the gate-array portion of the chip are given in Table 1. In this technology, a NAND gate is one (1) gate, while a scan flip-flop takes nine (9) gates.

2.3 Programming model

Although the fast-message capability of Alewife makes it a good vehicle for executing programs written in a message-passing style, it is better viewed by the programmer as a shared-memory machine. The Alewife hardware mechanisms, including fast messages, are combined in support of the shared-memory programming model. To facilitate programming, Alewife provides tools that inform programmers when poor performance is caused by widely-shared data objects, and which parts of the application are affected. Programmers can then fine-tune performance by using the direct message-passing interface integrated with shared memory.

Alewife has compilers for a parallel version of ANSI C and a parallel version of LISP called Mul-T [13]. For parallel C, Alewife supports the `p4` library from Argonne National Laboratory as well as parallel loops and distributed arrays. Automatic partitioning can be used when a program uses parallel loops and arrays [1].

Parallelism in Mul-T is specified with the `future` construct. Low thread creation overhead is achieved using *lazy task creation* [22], a method for dynamic partitioning and load balancing. The Alewife runtime system includes a parallel stop-and-copy garbage collector.

2.4 Alewife debugging and tuning

Alewife provides a number of facilities to aid in program debugging and performance tuning. An Alewife version of GDB allows symbolic program debugging, complete with the ability to set breakpoints, examine data and registers on individual nodes, and inspect threads, both active and blocked.

The programmer can make use of two distinct facilities in Alewife for performance debugging. First, the LimitLESS cache coherence mechanism can be configured to collect information about which memory locations are being shared and accessed in a pattern that causes poor performance. A tool is available that traces errant memory behavior directly to source variables.

Second, the Alewife CMMU provides extensive facilities for performance monitoring. Four 32-bit statistics counters and a histogram array can be configured to measure a wide variety of hardware events: examples include cache hits and misses, instruction counts, and network throughput statistics. A graphical interface allows users to specify a set of statistics and displays static and dynamic views of the results.

3 Mechanisms and Microbenchmarks

This section describes the implementation of the mechanisms introduced in Section 1. It presents the cost and the raw performance of each of the mechanisms in isolation.

3.1 Shared memory

The Alewife machine provides hardware support for distributed, cache-coherent shared memory. Cache lines in Alewife are 16 bytes in size and are kept coherent through a software-extended scheme called LimitLESS[8, 9]. This scheme implements a full-map directory protocol by supporting up to five readers per memory line directly in hardware and by trapping into software for more widely-shared data. Consequently, LimitLESS involves a close interaction between hardware and software. The hardware invokes software handling for remote requests by making use of the Alewife message-passing interface: faulted coherence requests are forwarded to the processor in the same way as any other message. The queueing inherent in the message-passing interface permits multiple pending LimitLESS coherence requests.

Shared memory is distributed, in the sense that the shared address space is physically partitioned among nodes. Each 16-byte memory line has a *home node* that contains storage for its data and coherence directory. All coherence operations for a given memory line, whether handled by hardware or software, are coordinated by its home node. Each Alewife node contains the data and coherence directories for a 4M byte portion of shared memory.

The Alewife directory entry format is shown in Figure 4. Directories are 64-bits wide and are stored in off-chip DRAM. Each entry contains five 9-bit pointers, two bits of state, two bits of meta-state, and four full/empty bits (one for each word in the line). The *Local Bit* provides an optimization which guarantees that the local node can always acquire a pointer. The *Pointers In Use* field indicates the number of other pointers that are in use. The number of pointers available to the hardware may be adjusted from two to five with the *Pointers Available* field. Since the *Pointers In Use* field can be set by software, the cost of the LimitLESS read handler is amortized

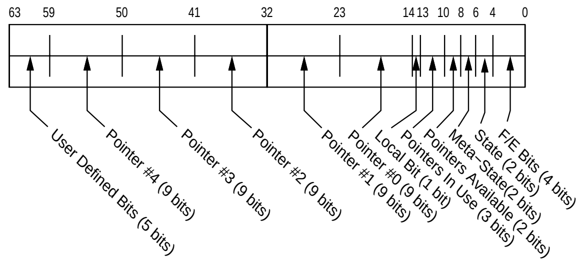

Figure 4: A hardware directory entry in Alewife.

Name of Meta-State	Description
`Normal`	dir. entry under hardware control
`Trap-On-Write`	reads handled by hw, writes by sw
`Write-In-Progress`	dir. entry software interlocked
`Trap-Always`	all requests handled by sw

Table 2: The four directory entry meta-states of the CMMU.

over up to six different read requests: when invoked to handle a read request, the handler resets the *Pointers In Use* field, allowing the next five requests to be handled without software intervention.

Table 2 describes the four LimitLESS meta-states. Two of these states, `Normal` and `Trap-On-Write`, are persistent. `Normal` indicates that a directory entry is entirely under hardware control – the four states associated with this meta-state form a base write-invalidate protocol. `Trap-On-Write` indicates that the identities of some of the readers are unknown to the hardware; consequently, a write request must be handled by software. Read requests, however, take advantage of a "read-ahead" optimization: the CMMU simultaneously forwards read requests to the local processor *and* returns data to the requesting nodes at hardware speed. `Trap-Always` permits experimental protocols to be constructed in entirely software.

LimitLESS interrupt handlers directly manipulate hardware directories. Thus, although all hardware directory entry manipulations are atomic, interlock mechanisms are necessary to prevent the hardware from modifying directories that are in the process of being altered by software. To implement this functionality, Alewife provides two instructions for directory entry manipulation, `rldir` (read and lock directory) and `wudir` (write and unlock directory). While the mutual exclusion provided by these instructions is sufficient for simple atomicity, Alewife provides a second mechanism for greater performance: the `Write-In-Progress` meta-state. This state marks a directory entry as requiring software manipulation and unavailable to the hardware. This additional state is necessary for the read-ahead optimization described above.

Sparcle employs a single-ported, unified first-level cache, with no on-chip instruction cache. Consequently, 32-bit loads and stores that hit in the cache take two and three cycles, respectively (one cycle for the instruction fetch). Doubleword (64-bit) loads and stores that hit in the cache take one additional cycle.

Table 3 shows the cost incurred when memory references miss in the cache. These values were obtained with a sequence of experiments run on an otherwise idle Alewife system. All remote misses or invalidations are between adjacent nodes. Each additional "hop" of communication distance increases these latencies by approximately 1.6 cycles.

For a simple load miss to remote memory handled in hardware, 58% of the 38-cycle miss penalty is due to network latency (1.1 out of 1.9 microseconds). Roughly three-quarters of the network latency is time spent transferring flits between the CMMU and the interconnection network (36 flits at 22.5 nanoseconds/flit). Table 4 gives a breakdown of the various latencies involved in satisfying a remote read-miss.

Miss Type	Home Location	# Inv. Msgs	hw/ sw	Miss Penalty	
				Cycles	μsec
Load	local	0	hw	11	0.55
	remote	0	hw	38	1.90
	remote (2-party)	1	hw	42	2.10
	remote (3-party)	1	hw	63	3.15
	remote	–	sw[†]	425	21.25
Store	local	0	hw	12	0.60
	local	1	hw	40	2.00
	remote	0	hw	38	1.90
	remote (2-party)	1	hw	43	2.15
	remote (3-party)	1	hw	66	3.30
	remote	5	hw	84	4.20
	remote	6	sw	707	35.35

[†] This sw read time represents the throughput seen by a single node that invokes LimitLESS handling at a sw-limited rate.

Table 3: Typical shared memory miss penalties.

Action	Count
Cache-miss to request in network	2
Request transit time (8 bytes)	7
Request at memory to output header transmit	7
Data return in network (24 bytes)	15
Response arrival to beginning of cache fill	3
Cache fill time	4

Table 4: Rough breakdown of 38-cycle clean read-miss to neighboring node.

Misses handled in software represent the access time seen when a cache line is shared more widely than is supported in hardware (five pointers), so that the home node processor must be interrupted to service the request. In the case of a load, the software time represents the maximum throughput available when every request requires software servicing. Because of the read-ahead optimization and amortized read-handling, this latency number will rarely be experienced by a requesting node. The software store latency represents an actual latency seen by a writer; it includes the time required for the software handler to send six invalidations, for these invalidations to be received by the hardware, and for an exclusive copy to be returned.

3.2 Message passing

Message passing is both a crucial component of the LimitLESS cache coherence protocol and a mechanism to be used in cooperation with software-extended shared memory. Some communication operations, such as file I/O, remote task dispatch, and the inner loops of typical scientific codes, can often be implemented more efficiently with message passing than with shared memory. Further, since Alewife provides a protected user-level message-passing interface, compilation targets such as active messages[30] are naturally supported.

```
stio header,   $ipiout0
stio dataword, $ipiout1
stio address,  $ipiout2
stio length,   $ipiout3
ipilaunch 2, 1
```

Figure 5: Machine code implementing a message send. In addition to the required header, this message includes one explicit data word, and one block of data from memory.

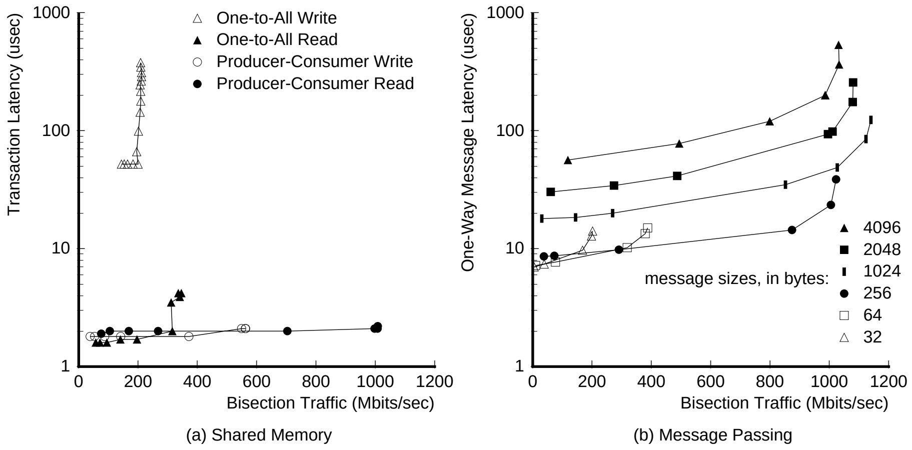

Figure 6: Latency versus bisection traffic for 16 nodes. (a) shared memory, two different data types. (b) message passing, random destinations.

Messages in Alewife are sent through a two phase process: first *describe*, then *launch*. A message is described by writing directly to an output descriptor array with a colored store instruction called `stio`. The output descriptor array consists of 16 memory-mapped network registers in the CMMU. Writes into this array incur the same cost as write hits in the cache. Once a message is described, it is launched via an atomic, single-cycle instruction called `ipilaunch`. This two phase process permits direct, user-level access to the network queues.

Figure 5 illustrates code for launching a message that consists of a header, one word of data from a register, and a block of data from memory (to be transferred via DMA). `header`, `dataword`, `address`, and `length` are aliases for arbitrary Sparcle registers. On entry to this code sequence, `header` contains the packet header, `dataword` contains the word of data, `address` points to the start of the data block, and `length` gives the number of double-words in the data block. This packet descriptor is two double-words long and contains one double-word of explicit data (`header` and `dataword`). Alewife maintains *local coherence* for the data block specified by `address` and `length`: data is acquired from the local cache at the source and invalidated from the local cache at the destination.

When a message arrives at its destination, it typically causes an interrupt. The CMMU overlaps message arrival with interrupt processing by posting the interrupt as soon as it has received the header of a message. Since the operating system reserves one of the four Sparcle hardware contexts for message processing (as in [24, 27]), no register saves or restores are necessary. The first 16 words of an incoming message are presented in the memory-mapped input packet array. Consequently, an interrupt handler may either load words directly from this array via the `ldio` instruction, or initiate a DMA sequence to store the message into memory.

The Alewife message-passing interface has low overhead. A simple, 2-word message (one header, one data-word), can be transmitted with 3 instructions, or 7 cycles. Message reception can use polling or interrupts. The cost of reception is more expensive when an interrupt must be fielded at the receiving end. Using interrupts, a system-level handler for a 2-word message can be entered in approximately 35 cycles. This time includes reading the message from the network, dispatching on an opcode in the header, and setting up for a general call to handler routines written in C.

Adding user-level message protection increases this entry time by another 15 cycles to approximately 50 cycles. A null user-level message handler requires a total of 95 cycles. Much of this time is associated with saving and restoring the system-level timer (to time out an errant user-level handler and prevent a user from locking up the machine), preventing access to shared memory before the current message has been removed from the queue, and checking for user-requested atomicity. Simple modifications planned for the respin of the CMMU will combine these three functions into a single mechanism and reduce the overhead of protected message passing considerably.

3.3 Shared memory versus message passing

Measurements of Alewife's mesh network show that each channel provides a peak bandwidth of approximately 356 Mbits/second (22.5 nanoseconds per 8-bit flit). For a sixteen node machine, this rate yields a maximum possible bisection bandwidth of 2.8 Gbits/sec. Synthetic workload generators measure the capacity of Alewife's network in more realistic environments.

Figure 6 characterizes Alewife's network performance in terms of latency and bandwidth of both shared memory and message passing. A shared-memory workload generator simulates two different types of data. Figure 6(a) shows the results of running this microbenchmark on a 16-node Alewife machine. The horizontal axis shows the bisection traffic achieved by each run; the vertical axis uses a logarithmic scale to plot the average latency of a memory transaction (a read or a write). This experiment measures actual network traffic, as opposed to the bandwidth available to user data. For shared memory, user data accounts for roughly half of the traffic. The latency is measured from the time a node requests a memory block until the time that the block is ready to be accessed from a transaction buffer.

The curves in Figure 6(a) marked with circles show the base performance of shared memory. During these experiments, every processor accesses memory in a *producer-consumer* fashion: each processor writes to a number of blocks in its local memory and then reads the same amount of data from another processor's memory. The read phase of this benchmark generates bisection traffic of up to 1 Gbit/sec, over 35% of the maximum possible bandwidth. The shared memory coherence protocol produces half as much traffic during the write phase of this benchmark. All of the producer-consumer transactions require about 2 microseconds (μsec) latency across the entire traffic range.

The experiments plotted with triangles show the performance when each processor writes to a number of memory blocks and all of the other processors in the system read the blocks. This *one-to-all* scenario invokes Alewife's software-extended shared memory. Due to the directory read-ahead feature (see Section 3.1), the read latency at low bandwidths is similar to the producer-consumer experiments. However, the read traffic saturates at 346 Mbits/sec, with a 4.2 μsec latency; the base write latency lies at 52 μsec, and increases to 380 μsec as bandwidth demands increase.

Figure 6(b) shows the results of a synthetic message-passing workload: this microbenchmark consists of a single loop in which each processor transmits a "ping" message to another node, selected randomly. The other node responds with an acknowledgment message. In this experiment, almost all of the network traffic (measured by the horizontal axis) consists of user data. The vertical axis measures the average latency of half of a ping/ack round-trip (including both hardware and software delays). The highest bisection traffic achieved with random ping destinations is 1.2 Gbits/sec, over 40% of the maximum possible bandwidth. This traffic corresponds to the right-most point on the graph, with 1024-byte messages and a 100 μsec/message latency. The lowest latency is 7.0 μsecfor 32 byte messages.

Contrasting Figures 6(a) and 6(b) shows the benefits of each mechanism. Shared memory provides a fast, convenient abstraction for orchestrating interprocessor communication, but message passing makes more efficient use of bandwidth.

3.4 Fine-grain synchronization

The primary advantage of fine-grain synchronization is that more parallelism can be exposed to the underlying hardware or software system than if coarse-grain synchronization techniques, such as barriers, were employed. For example, a thread synchronizing at a barrier has to wait for the arrival of *all* other synchronizing threads before proceeding, regardless of whether that thread depends on results computed by the other threads. By synchronizing on exactly the data words to be consumed, fine-grain synchronization eliminates false dependencies and allows a thread to proceed as soon as the data it needs is available.

The Alewife machine provides both hardware and software support for fine-grain synchronization. Hardware support consists of a full/empty bit[29] for each 32-bit data word. To access these bits, colored load and store instructions are provided that perform full/empty test-and-set operations. Table 5 presents a sample of Alewife data-access instructions. All of these instructions return the original full/empty bit in the coprocessor status word. Two Sparcle instructions, `bempty` (branch on empty) and `bfull` (branch on full), can then be used to examine this bit.

In Alewife, the odd data width introduced by full/empty bits does not impact DRAM, cache, or network data widths. At memory side, full/empty bits are stored in the bottom four bits of the coherence directory entry (see Figure 4). At cache side, they are stored as an extra field in the cache tags. In data packets, they are transmitted in the bottom four bits of the address, and take advantage of the 16-byte cache-line width.

The system provides several language extensions for fine-grain synchronization in the form of J-structures and L-structures. Patterned after I-structures[6], J-structures support producer-consumer style synchronization on vector elements, with full/empty bits associated with each vector element. A J-structure read waits until the element is full before returning its value. A J-structure write updates the element and sets it to full. An L-structure supports mutual-exclusion style synchronization on vector elements with full/empty bits associated with each vector element. L-structures support 3 operations: a locking read, an unlocking write, and a non-locking read.

Name	Meaning
`ldn`	Load and examine full/empty bit.
`stn`	Store and examine full/empty bit.
`lden`	Load, set empty, examine original value.
`stfn`	Store, set full, examine original value.
`ldet`	Load, set empty, and trap if already empty.
`stft`	Store, set full, and trap if already full.

Table 5: Examples of Alewife's data-access instructions.

Event	hw	sw
Read success	2	6
Read failure	21	6
Write	3	6
Unload thread	120–130	
Reload thread	90–100	

Table 6: Costs in cycles of fine-grain, producer-consumer synchronization in Alewife. "hw" represents the use of full hardware support; "sw" represents explicit checking in software.

Table 6 compares the costs (in cycles) of implementing J-structure read and write operations, with and without hardware support. The hardware implementation (hw) relies on traps to signal a failed read, and uses a separate, centralized waiting queue. It allows successful reads and writes to proceed at the speed of normal Sparcle loads and stores. The software-based implementation (sw) uses an additional memory word to emulate a full/empty bit for each J-structure element.

3.5 Latency tolerance

Latency tolerance in Alewife takes two forms: block multithreading and non-binding software prefetching. By supporting both block multithreading and prefetching, Alewife provides a platform for directly comparing these two latency tolerance mechanisms.

Three different mechanisms in the Alewife CMMU help support block multithreading. First, the CMMU takes advantage of as much parallelism as possible when servicing a remote cache miss by generating a context-switch trap in parallel with message generation. Second, the CMMU implements lockup-free caches. Third, the CMMU implements a livelock avoidance technique, to prevent the livelock that can arise when cache-coherent shared memory is coupled with context switching and LimitLESS.

Software prefetch is implemented in Alewife as two different colored load instructions, one for read prefetch and the other for write prefetch; the value returned from the prefetch instructions is ignored. Prefetched data is returned in the transaction buffer.

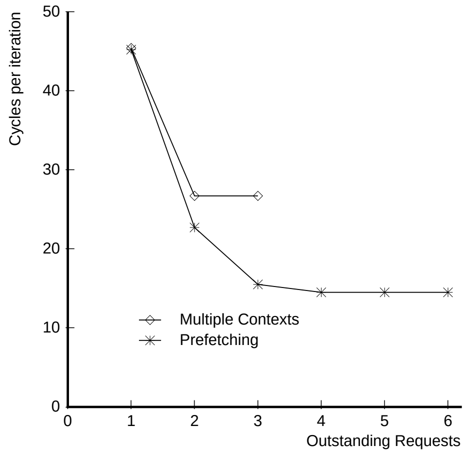

Figure 7: Effectiveness of latency tolerance mechanisms.

Program	Input Size
MP3D	18k particles, 6 iterations
Barnes-Hut	16k bodies, 4 iterations
LocusRoute	3817 wires, 20 routing channels (Primary2)
Cholesky	3948 $\times$ 3948 floats, 56934 non-zeros (TK15)
Water	729 molecules, 5 iterations
Appbt	20 $\times$ 20 $\times$ 20 floats
Multigrid	56 $\times$ 56 $\times$ 56 floats
CG	1400 $\times$ 1400 doubles, 78148 non-zeros
EM3D	20000 nodes, 20% remote neighbors
Gauss	512 $\times$ 512 floats
FFT	80k floats
SOR	512 $\times$ 512 floats, 50 iterations
MICCG3D	32 $\times$ 32 $\times$ 32 and 64 $\times$ 64 $\times$ 64 floats

Table 7: Main application characteristics.

To measure the benefit of latency tolerance using context switching and prefetching, an experiment runs a small loop on one processor that adds numbers fetched from the memory of another processor. Figure 7 shows the number of cycles per loop iteration as a function of the number of outstanding requests. As expected, one outstanding request incurs the same overhead using either prefetching or context switching. As the prefetch depth is increased, the performance improves until the limit of network bandwidth is reached. For context switching, the limiting factor is the overhead of the mechanism, not bandwidth. Because the loop performs remote reads which have a relatively low latency ($\sim$40 cycles), the 14 cycle context switch time hides all of the latency with two contexts. For longer remote latencies that can occur in real programs, three contexts may be beneficial.

Although the absolute performance of prefetching is better due to low overhead, its use is limited to places where cache-miss behavior can be predicted statically. Results in Section 4 show that context switching can increase the performance of a parallel application, even when prefetching has been carefully used.

4 Application Performance

This section presents the performance of a number of applications and demonstrates the efficacy of the mechanisms in the machine. It starts by summarizing the performance of a dozen applications written in a shared-memory style. The following subsection compares the performance of an application written in a message-passing style on Alewife and on the Thinking Machines' CM-5 multiprocessor. The last two subsections present details of two application case studies, MP3D and MICCG3D.

4.1 Shared memory performance

Shared-memory applications perform well on Alewife, proving the viability of both the software-extended coherence mechanism and the low-dimensional communication substrate provided by the mesh network. Table 7 summarizes the main characteristics of the applications evaluated on Alewife. The first five applications shown in the table are from the SPLASH suite[28], the three following ones are from the NAS parallel benchmarks[7], the next four are engineering kernels, and the last solves a numerical problem.

Table 8 presents the running time and speedup performance of these applications on Alewife. The table includes results for "Mod MP3D", which is a version of the original MP3D application that eliminates some useless code and improves locality by modifying the mapping of particles to processors. Section 4.3 discusses both the original and modified versions of MP3D in detail.

All the speedups presented in Table 8 are based on the parallel implementation of each program running on one processor except those that are marked in the table with asterisks and the different versions of MICCG3D. These exceptions ran with input sizes that do not fit on a single node's memory[1]. The experiments with an asterisk assume that the speedup is linear at the smallest number of nodes that can hold the data set. The MICCG3D speedups are computed using a best sequential running time that is obtained by assuming sequential running time grows linearly with problem size. The Alewife compiler, used for all applications, produces code with a sequential running time that is within 10% of `gcc` at the "`-O2`" level of optimization.

The results show that Alewife usually achieves good application performance, especially for the computational kernels, even for relatively small input sizes. In particular, MP3D (an application with a difficult shared-memory workload) achieves extremely good results. In contrast, a comparison between the two entries for Cholesky in the table demonstrates the importance of the input size for the performance of this application; a 5-fold input size increase leads to a significant improvement in speedup. The modest speedups of CG and Multigrid result from load imbalance and bad cache behavior, which can be addressed by using larger input sizes and the latency tolerance mechanisms in Alewife.

Table 8 presents the performance of the $32\times32\times32$ and $64\times64\times64$ input sizes for MICCG3D (labelled MICCG3D-32 and MICCG3D-64, respectively) using coarse-grain and fine-grain synchronization. The speedups appear low because they are measured against the best sequential implementation of the application, rather than a uniprocessor run of the parallel algorithm. The performance of MICCG3D will be explained in detail in Section 4.4.

As a whole, experience with porting a variety of applications in a short period of time shows that Alewife provides a good environment for applications written in a shared-memory style. Programs can be easily ported to the machine and can achieve good performance.

[1]Barnes-Hut was run with 32k bodies as input, while Cholesky was run with five times as many non-zeros as the base input size.

Program	Running Time (Mcycles)						Speedup					
	1P	2P	4P	8P	16P	32P	1P	2P	4P	8P	16P	32P
Orig MP3D	67.6	41.7	24.8	13.9	7.4	4.3	1.0	1.6	2.7	4.9	9.2	15.7
Mod MP3D	47.4	24.5	12.4	6.9	3.5	2.2	1.0	1.9	3.8	6.9	13.4	21.9
Barnes-Hut	9144.6	4776.5	2486.9	1319.4	719.6	434.2	1.0	1.9	3.7	6.9	12.7	21.1
Barnes-Hut *	–	10423.6	5401.6	2873.3	1568.4	908.5	–	2.0	3.9	7.3	13.3	22.9
LocusRoute	1796.0	919.9	474.1	249.5	147.0	97.1	1.0	2.0	3.8	7.2	12.2	18.5
Cholesky	2748.1	1567.3	910.5	545.8	407.7	398.1	1.0	1.8	3.0	5.0	6.7	6.9
Cholesky *	–	–	2282.2	1320.8	880.9	681.1	–	–	4.0	6.9	10.4	13.4
Water	12592.0	6370.8	3320.9	1705.5	897.5	451.3	1.0	2.0	3.8	7.4	14.0	27.9
Appbt	4928.3	2617.3	1360.5	704.7	389.7	223.7	1.0	1.9	3.6	7.0	12.6	22.0
Multigrid	2792.0	1415.6	709.1	406.2	252.9	165.5	1.0	2.0	3.9	6.9	11.0	16.9
CG	1279.2	724.9	498.0	311.1	179.0	124.9	1.0	1.8	2.6	4.1	7.1	10.2
EM3D	331.7	192.1	95.5	46.8	22.4	10.7	1.0	1.7	3.5	7.1	14.8	31.1
Gauss	1877.0	938.9	465.8	226.4	115.7	77.8	1.0	2.0	4.0	8.3	16.2	24.1
FFT	1731.8	928.0	491.8	261.6	136.7	71.8	1.0	1.9	3.5	6.6	12.7	24.1
SOR	1066.2	535.7	268.8	134.9	68.1	32.3	1.0	2.0	4.0	7.9	15.7	33.0
MICCG3D-32-Coarse	–	36.6	21.7	11.7	6.9	4.4	–	0.5	0.8	1.5	2.5	3.9
MICCG3D-32-Fine	–	–	11.7	5.8	2.9	1.5	–	–	1.5	3.0	5.9	11.5
MICCG3D-64-Coarse	–	–	–	–	–	32.2	–	–	–	–	–	4.3
MICCG3D-64-Fine	–	–	–	–	–	12.5	–	–	–	–	–	11.1

Table 8: Performance of shared-memory applications on Alewife.

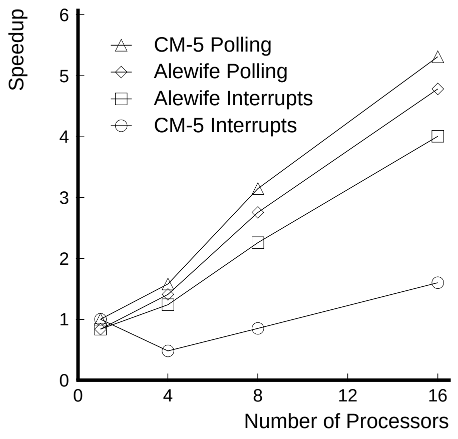

Figure 8: Performance of a message-passing implementation of the sparse triangular matrix solver.

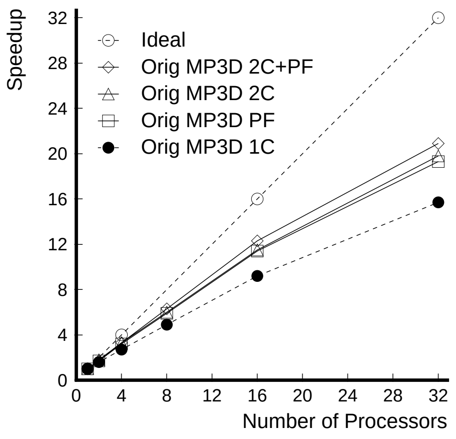

Figure 9: Speedups for various versions of MP3D: single-context (1C), two contexts (2C), and prefetching (PF).

4.2 Message-passing performance

This section compares Alewife's support for short messages with that of the CM-5. It shows that for a sparse matrix application with irregular, fine-grain communication requirements, Alewife delivers performance that is comparable to the CM-5 under polling and superior under interrupts. The application is a power grid benchmark from a sparse matrix suite [11] which uses the techniques of [10].

Figure 8 presents speedups of message-passing implementations of this application on Alewife and the CM-5, under both polling[2] and interrupts. Speedups are computed based on the running time (in cycles) of an optimized sequential code running on a single CM-5 node. The difference between the two polling implementations is 10%, and can be entirely attributed to the use of an experimental compiler on Alewife.

[2]Unlike the CM-5, user polling on Alewife allows system messages to continue to generate interrupts while user messages are received via polling.

More importantly, the difference between polling and interrupt versions on Alewife is only 16%. Since this application has extremely fine-grained communication (one or two floating-point numbers per message), this implies that interrupt-driven message passing, which excels at unpredictable message traffic, is more than sufficient for coarser-grained applications. In contrast, the interrupt-driven version of this application on the CM-5 suffers more than a factor of three degradation over the polling version. This illustrates the benefits of Alewife's fast interrupt handling, discussed in Section 3.2. In summary, these results show that message passing on Alewife has comparable performance to the CM-5 for small messages while handling a wider variety of message traffic (including DMA) efficiently.

4.3 MP3D

On Alewife, MP3D achieves the largest speedup ever reported for this application. There are two reasons for this result. First, most of

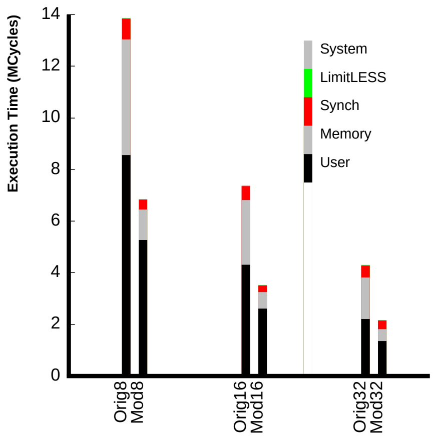

Figure 10: Orig and Mod MP3D running times, with costs.

the communication traffic in the benchmark is caused by migratory data, and Alewife's coherence protocol is optimized for this type of data. Second, Alewife has relatively low (∼60-cycle) latency for 3-party remote read transactions, which results from Alewife's pipelined memory system and its simple, flat network hierarchy. This low latency pays off when the whole hierarchy must be traversed frequently.

MP3D also serves as a good vehicle for assessing the performance of Alewife's latency tolerance mechanisms. The original MP3D code is a good candidate for latency tolerance, since improvements in locality for this program are difficult to obtain without significant code restructuring. Accordingly, this section considers the effect of using multiple contexts, software prefetching, and a combination of these two. Figure 9 presents the speedups of different versions of MP3D. All speedups in this graph are computed with respect to the non-prefetching parallel implementation running on one processor.

In order to investigate the maximum possible benefit of prefetching, software prefetching was inserted by hand. The prefetch instructions concentrate on the data causing the majority of the cache misses in MP3D. As seen in Figure 9, prefetching achieves a 23% improvement in speedup at 32 processors over the non-prefetching version.

Block multithreading allows MP3D to perform marginally better than hand-crafted software prefetching (26% vs 23%), proving that context switching can help applications achieve performance comparable to versions generated by sophisticated compilers and/or programmers. An interesting observation is that the combination of prefetching and multithreading for MP3D approaches the speedup performance of the hand-optimized version of the application, Mod MP3D (see Table 8). One possible explanation for this effect is that multithreading can tolerate the latency of replacement cache misses, which are difficult to predict when implementing software prefetching.

Figure 10 presents the cost breakdown (measured by the Alewife statistics hardware) for MP3D and Mod MP3D for 8, 16, and 32 processors. As shown in this figure, Mod MP3D significantly reduces both the busy time and the memory wait overhead of MP3D. Another interesting observation is that the overhead of handling widely shared cache blocks in software (the LimitLESS component) and the scheduler costs (the system component) are always negligible for the two programs. In fact, none of the shared-memory applications suffers significantly from these two types of overhead.

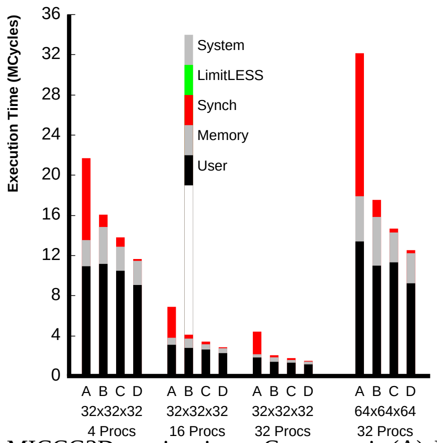

Figure 11: MICCG3D running times. Coarse-grain (A). Fine-grain: no hw (B), f/e bits with sw checks (C), and full hw support (D).

4.4 MICCG3D

MICCG3D solves Laplace's equation on a three-dimensional grid using a preconditioned conjugate gradient method. A central operation in this method has been difficult to parallelize, due to the computation's complex data dependencies. This section reports results for a single iteration of four different parallel implementations of MICCG3D. The first is coarse-grain: the data is block partitioned, and each partition is assigned to a single thread. The data blocks and threads are statically placed in the mesh, and barriers sequence the computation.

The other three implementations are all fine-grain, and allocate shared data in J-structures. They differ in the way they implement the J-structures. In the first, J-structures consist of two separate arrays: one for data, and one for synchronization variables. On each J-structure reference, software checks the synchronization variable. The second eliminates the need for the synchronization array by using full/empty bits as provided in Alewife. Checking the bit at each reference is still done in software. The last implementation uses the full capability of Alewife's full/empty bits, checking synchronization state in hardware. (For additional information, see [31]).

Figure 11 shows a cost breakdown of various MICCG3D execution times. The system and LimitLESS components are negligible and are not visible on the graph. Synchronization overhead in the coarse-grain implementation is the time spent waiting in barriers; in the fine-grain implementations, it is the time spent both in J-structure operations and in waiting for pending J-structure values. For each problem size and machine size, the figure shows breakdowns for all four implementations of MICCG3D. Results for 4, 16, and 32 processors are shown for a $32\times32\times32$ grid; 32 processor results are also shown for a $64\times64\times64$ grid. A large difference in synchronization waiting time is apparent between the coarse-grain and fine-grain implementations. Synchronization overhead continues to be significant with increasing machine size in the coarse-grain implementation for several reasons. The amount of useful work between barriers decreases as more of the parallelism is exploited. As the number of processors increases, the number of barriers increases to enforce the data dependencies, and each barrier itself is more expensive since barriers require global communication. The fine-grain implementations do not suffer from these effects because J-structures require synchronization only where data dependencies occur.

The three fine-grain implementations show the impact of hardware support for fine-grain synchronization on performance. The implementation with only software support has the highest memory overhead and synchronization overhead. The fine-grain implementation that uses software checking of full/empty bits has similiar synchronization overhead, but has a lower memory overhead. The comparison shows that full/empty bits afford compact storage and communication of synchronization state, resulting in lower demands on memory and network bandwidth. The implementation with full hardware support for fine-grain synchronization shows the smallest memory overhead.

In addition, using hardware to check synchronization state lowers synchronization overhead. However, the reduction in synchronization overhead is not as significant as the reduction in memory overhead. In Figure 11, the "Memory" segments (in the B bars versus C bars) are reduced by a greater amount due to full/empty bits than the "Synch" segments (in the C bars versus D bars) due to hardware checking. Hardware checking in the MICCG3D application still has a significant impact on overall performance because the software checking code increases register pressure, resulting in more register spilling. This is reflected by smaller "User" segments in the D bars. For codes with low register pressure, the overall gains from hardware checking of full/empty bits would be much less than the gains of having full/empty bits in the first place.

This study leads to two conclusions about fine-grain computation. First, the ability to express synchronization at a fine granularity has a first-order impact on performance for the MICCG3D application; providing support for the fine-grain synchronization primitives in hardware or software is a second-order effect. Second, the reduction in memory and network bandwidth due to full/empty bits impacts performance more significantly than hardware checking. In general, the degree to which full/empty bits are more important than hardware checking depends on register usage in the application code.

5 Related Work

A number of other systems provide a shared address space entirely in hardware. DASH [20] is a cache-coherent multiprocessor that uses a full-map directory-based cache coherence protocol. It includes prefetching and a mechanism for depositing data directly in another processor's cache. The KSR1 and DDM [12] provide a shared address space through cache-only memory. These machines also allow prefetching. The Scalable Coherent Interface [5] also specifies mechanisms for implementing large, shared address spaces.

Both the J-machine [24] and the CM-5 export hardware message-passing interfaces directly to the user. These interfaces differ from the Alewife interface in several ways. First, in Alewife, messages are normally delivered via an interrupt and dispatched in software, while in the J-machine, messages are queued and dispatched in sequence by the hardware. On the CM-5, message delivery through interrupts is expensive enough that polling is normally used to access the network. Second, neither the J-machine, nor the CM-5 allow network messages to be transferred through DMA. Third, the J-machine does not provide an atomic message send like Alewife does; this omission complicates the sharing of a single network interface between user code and interrupt handlers.

The Cray T3D integrates message passing and hardware support for a shared address space. Message passing in the T3D is flexible and includes extensive support for DMA. However, the T3D does not provide cache coherence.

Several recently proposed architectures are based on the integration of shared memory and message passing in some form. FLASH [19] includes a microcoded, kernel-level coprocessor for message handling including shared-memory protocol messages. Bulk transfers in FLASH avoid using the receiving processor, but require pre-negotiating memory allocation. FLASH provides a multi-user environment. Typhoon [26] offers user-level message handling and cache coherence, using a second processor dedicated to the network interface. The *T [23] architecture uses a memory coprocessor model as well.

A few architectures incorporate multiple contexts, pioneered by the HEP [29], switching on every instruction. These machines, including Monsoon [25] and Tera [4], do not have caches and rely on a large number of contexts to hide remote memory latency. In contrast, Alewife's block multithreading technique switches only on synchronization faults and cache misses to remote memory, permitting good single-thread performance and requiring less aggressive hardware multithreading support. A number of architectures — including HEP, Tera, Monsoon, and the J-machine — also provide support for fine-grain synchronization in the form of full/empty bits or tags.

6 Conclusion

Alewife represents a step in the maturation of multiprocessing technology. Specifically, it augurs the end of the religious war between proponents of the shared-memory and message-passing models of parallel computation. The working machine demonstrates that both models permit efficient and scalable implementations; moreover, the two models may – and should – be integrated into a unified multimodel framework. Although previous systems have implemented some of Alewife's mechanisms independently, Alewife is unique in its combination of coherent caches for shared memory, integrated message passing, support for fine-grained computation, and latency tolerance. These four mechanisms provide an integrated solution to the problems of communication and synchronization in parallel systems.

This integration of architectural features results in a multiprocessor that is both programmable and scalable. The case-study using the MP3D application illustrates this conclusion: it was easy to port this demanding workload to the architecture, and the application worked and realized acceptable speedups almost immediately. Subsequent performance tuning and invoking Alewife's latency tolerance mechanisms significantly improved MP3D's performance.

Experience with a variety of other workloads confirms this anecdotal evidence. More broadly, experience with applications indicates that a globally shared address space, cache coherence, and a message-based runtime system is instrumental in the quick development of working applications that perform well. Latency tolerance mechanisms, fine-grain synchronization, and explicit message passing help improve performance further.

At this time, effort is underway to respin the CMMU and to build a 128-node machine. Although Alewife addresses many of the issues of large-scale multiprocessing, it is essentially a single-user machine. Our future work will investigate mechanisms for protection and virtual memory in multimodel multiprocessors. Implementing a virtual machine model in the face of streamlined user-level communication mechanisms is challenging, and forms the basis of the new FUGU architecture [21].

7 Acknowledgments

The following members of the Alewife team contributed significantly to the success of the project: Jonathan Babb, Rajeev Barua, Fred Chong, David Hoki, Ed Hurley, Gino Maa, Anne McCarthy, Sramana Mitra, Dan Nussbaum, and John Piscitello. Fred Chong wrote the sparse matrix code described in Section 4.2.

Others outside of MIT contributed as well: we would like to thank Mike Marchetti for the Multigrid code, and Leonidas Kontothanassis for the FFT code and for assistance with writing Mod MP3D. The Alewife machine was built in cooperation with LSI Logic Inc., Sun Microsystems Inc., and the Information Sciences Institute at USC.

The Alewife project is funded in part by ARPA contract # N00014-87-K-0825, in part by a NSF Experimental Systems grant # MIP-9012773, and in part by NSF Presidential Young Investigator Award. Ricardo Bianchini was supported in part by Brazilian NUTES/UFRJ and CAPES/MEC (grant # 2038/90-2) fellowships, and by the Office of Naval Research Contract # N00014-92-J-1801 in conjunction with ARPA order # 8930.

References

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *The 22nd International Conference on Parallel Processing*, August 1993.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[3] A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *The 17th Annual International Symposium on Computer Architecture*, pages 104–114, June 1990.

[4] G. Alverson, R. Alverson, and D. Callahan. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. In *Workshop on Multithreaded Computers, Proceedings of Supercomputing*, November 1991.

[5] *ANSI/IEEE Std 1596-1992 Scalable Coherent Interface*, 1992.

[6] Arvind, R. Nikhil, and K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transcations on Programming Languages and Systems*, 11(4):598–632, October 1989.

[7] D. Bailey et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[8] D. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *The 21st Annual International Symposium on Computer Architecture*, pages 314–324, April 1994.

[9] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234. ACM, April 1991.

[10] F. Chong, S. Sharma, E. Brewer, and J. Saltz. Multiprocessor Runtime Support for Irregular DAGs. In R. Kalia and P. Vashishta, editors, *Toward Teraflop Computing and New Grand Challenge Applications*. Nova Science Publishers, Inc., 1995.

[11] I. Duff, R. Grimes, and J. Lewis. User's Guide for the Harwell-Boeing Sparse Matrix Collection. Technical Report TR/PA/92/86, CERFACS, October 1992.

[12] E. Hagersten, A. Landin, and S. Haridi. DDM — A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[13] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *The Symposium on Programming Languages Design and Implementation*, June 1989.

[14] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.H. Lim. Integrating Message-Passing and Shared-Memory; Early Experience. In *The 4th Annual Symposium on the Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[15] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *The 8th Annual Symposium on Computer Architecture*, pages 81–87, June 1981.

[16] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *The International Conference on Supercomputing*, July 1993.

[17] J. Kubiatowicz, D. Chaiken, and A. Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *The 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–284. ACM, October 1992.

[18] J. Kubiatowicz, D. Chaiken, A. Agarwal, A. Altman, J. Babb, D. Kranz, B.H. Lim, K. Mackenzie, J. Piscitello, and D. Yeung. The Alewife CMMU: Addressing the Multiprocessor Communications Gap. In *HOTCHIPS*, August 1994.

[19] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *The 21st Annual International Symposium on Computer Architecture 1994*, April 1994.

[20] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

[21] K. Mackenzie, J. Kubiatowicz, A. Agarwal, and M. Frans Kaashoek. FUGU: Implementing Protection and Virtual Memory in a Multiuser, Multimodel Multiprocessor. Technical Memo MIT/LCS/TM-503, October 1994.

[22] E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[23] R. Nikhil, G. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *The 19th Annual International Symposium on Computer Architecture*, pages 156–167, May 1992.

[24] M. Noakes, D. Wallach, and W. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *The 20th Annual International Symposium on Computer Architecture*, pages 224–235, May 1993.

[25] G. Papadopoulos and D. Culler. Monsoon: An Explicit Token-Store Architecture. In *The 17th Annual International Symposium on Computer Architecture*, pages 82–91, June 1990.

[26] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *The 21st Annual International Symposium on Computer Architecture*, April 1994.

[27] C. Seitz, N. Boden, J. Seizovic, and W.K. Su. The Design of the Caltech Mosaic C Multicomputer. In *Research on Integrated Systems Symposium Proceedings*, pages 1–22, Cambridge, MA, 1993. MIT Press.

[28] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[29] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *Society of Photo-optical Instrumentation Engineers*, 298:241–248, 1981.

[30] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, May 1992.

[31] D. Yeung and A. Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. In *The 4th Annual Symposium on Principles and Practice of Parallel Programming*, pages 187–197, May 1993.