Appears in Proceedings of the 20th International Symposium on Computer Architecture, May 1993.

The J-Machine Multicomputer: An Architectural Evaluation*

Michael D. Noakes, Deborah A. Wallach, and William J. Dally
Artificial Intelligence Laboratory and Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
noakes@ai.mit.edu, kerr@ai.mit.edu, billd@ai.mit.edu

Abstract

The MIT J-Machine multicomputer has been constructed to study the role of a set of primitive mechanisms in providing efficient support for parallel computing. Each J-Machine node consists of an integrated multicomputer component, the Message-Driven Processor (MDP), and 1 MByte of DRAM. The MDP provides mechanisms to support efficient communication, synchronization, and naming. A 512 node J-Machine is operational and is due to be expanded to 1024 nodes in March 1993. In this paper we discuss the design of the J-Machine and evaluate the effectiveness of the mechanisms incorporated into the MDP. We measure the performance of the communication and synchronization mechanisms directly and investigate the behavior of four complete applications.

1 Introduction

Over the past 40 years, sequential von Neumann processors have evolved a set of mechanisms appropriate for supporting most sequential programming models. It is clear, however, from efforts to build concurrent machines by connecting many sequential processors, that these highly-evolved sequential mechanisms are not adequate to support most parallel models of computation. These mechanisms do not efficiently support synchronization of threads, communication of data, or global naming of objects. As a result, these functions, inherent to any parallel model of computation, must be implemented largely in software with prohibitive overhead.

The J-Machine project [5] was developed to study how to best apply modern VLSI technology to construct a multicomputer. Each processing node of the J-Machine consists of a Message-Driven Processor (MDP) and 1 MByte of DRAM. The MDP incorporates a 36-bit integer processor (32 bits of data augmented with 4 bits of tag), a memory management unit, a router for a 3-D mesh network, a network interface, a 4K-word $\times$ 36-bit SRAM, and an ECC DRAM controller in a single 1.1M transistor VLSI chip. Rather than being specialized for a single model of computation, the MDP incorporates primitive mechanisms for communication, synchronization, and naming that permit it to efficiently support threads with 50 to 150 instructions which exchange small data objects frequently with low-latency and synchronize quickly. A 512 node J-Machine is in daily use at MIT and will be expanded to 1024 nodes in March 1993.

This paper describes a range of experiments performed on the J-Machine to study the effectiveness of the selected mechanisms in supporting parallel applications. These experiments are divided into micro-benchmarks, designed to isolate the effects of the primitive mechanisms, and macro-benchmarks, to demonstrate the cumulative effect of the mechanisms on application level codes. We investigate the sequential performance of the MDP, the message-passing mechanisms, the performance of the 3D-mesh network, and the behavior of parallel applications running on the J-Machine.

We use these studies to critique the effectiveness of the mechanisms and reflect on the impact of these design decisions in developing programming systems for the J-Machine. We contrast the effectiveness of the J-Machine with comparable multicomputers and consider the impact of alternative mechanisms to further enhance efficiency.

2 The J-Machine

This section describes the architecture of the J-Machine and the hardware prototype on which the studies were performed.

2.1 Architecture

The instruction set of the MDP includes the usual arithmetic, data movement, and control instructions. The MDP is unique in providing special support for communication, synchronization, and naming.

*The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and F19628-92C-0045, and by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation, and AT&T.

The MDP supports communication using a set of `send` instructions for message formatting, a fast network for delivery, automatic message buffering, and task creation upon message arrival. A series of `send` instructions is used to inject messages at a rate of up to 2 words per cycle. The format of a message is arbitrary except that the first word must contain the address of the code to run at the destination and the length of the message. Messages are routed through the 3D-mesh network using deterministic, e-cube, wormhole routing [4]. The channel bandwidth is 0.5 words/cycle and the minimum latency is 1 cycle/hop. Upon arrival, messages are buffered in a hardware queue. When a message arrives at the head of the queue, a task is dispatched to handle it in four processor cycles. During these cycles the Instruction Pointer is loaded from the message header, an address register is set to point to the new message so that the thread's arguments may be accessed, and the thread's first instruction is fetched and decoded.

Messages may be issued to one of two priorities. Priority one messages receive preference during channel arbitration, are buffered in a separate queue at the destination, and are dispatched before pending priority zero messages. Priority one threads may interrupt executing priority zero threads. There is also a background priority that runs whenever both message queues are empty. Fast interrupt processing is achieved through the use of three distinct register sets.

Synchronization is provided by the ability to signal events effectively using the low-latency communication primitives and by the use of data-tagging in both the register file and memory. Two of the possible sixteen data types, `cfut` and `fut`, are used to mark slots for values that have not yet been computed. If a thread attempts to read a slot before the value has been supplied, the processor will trap to a system routine to suspend the thread until the value is delivered. In this event, the arrival of the value is used to restart the thread. The `cfut` type provides inexpensive synchronization on a single slot, much like a full-empty bit. The `fut` type may be copied without faulting and thus supports the more flexible, but more expensive, future datatype [2]. Futures are first-class data objects and references to them may be returned from functions and stored in arrays, for example.

The MDP supports a global namespace with segmented memory management and with name translation instructions. Local memory is referenced using indexed accesses via segment descriptors that specify the base and length of each memory object. These objects may be relocated at will, to implement local heap compaction, for example, if we ensure that only global virtual addresses of objects are exported from nodes. A hardware name-translation table is provided to accelerate virtual address to physical segment descriptor conversion. Virtual-physical pairs are inserted in the table using the `enter` instruction and extracted using `xlate`. A successful `xlate` takes three cycles.

The register file includes four data registers and four address registers per priority. The instruction set of the MDP permits most operators to read one of the operands from memory to help reduce access pressure on the register file. The MDP instruction encodes two 17 bit instructions in each word. Most instructions can operate in one cycle if both operands are in registers and in two cycles if one operand is in internal memory. The peak execution rate is therefore 12.5 million instructions per second. Branch instructions, prefetching stalls, operands in memory, and instruction alignment issues reduce this rate to a typical value of 5.5 million instructions per second if code and data are in internal memory. This can fall to fewer than 2 million instructions per second if all code and data are in external memory.

2.2 Prototype

Each processing node of the J-Machine is composed of a single Message-Driven Processor (MDP) chip and three $1\text{M}\times 4$ DRAM chips. The resulting node is 2" by 3". Each J-Machine processor board contains 64 MDP chips along with their external memories. The board measures 26" by 20.5". Each pair of nodes shares a set of elastomeric conductors to communicate with the corresponding nodes on the boards above and below in the board stack. A total of 48 elastomeric connectors held in four connector holders provide 1420 electrical connections between adjacent boards. Of these connections, 960 are used for signalling and the remaining 460 are ground returns. The use of elastomeric conductors enables the J-Machine to achieve a very high network bisection bandwidth (a peak rate of 14.4 Gbits/sec) in a very small cross sectional area (2 ft^2) and at the same time keep the length of the longest channel wire in a 1024-node machine under four inches. A stack of sixteen processor boards contains 1024 processing nodes (12,500 MIPS peak) and 1Gbyte of DRAM memory in a volume of 4 ft^3, and dissipates 1500W (1.5W per node).

Figure 1 shows a pair of J-Machine cabinets. The cover has been removed from the cabinet on the left to reveal a partial processor board stack at the top and a pair of peripheral interface cards below. Cards will soon be available to connect to SCSI devices, a Sun Sparc host, and a distributed graphics frame-buffer. The maximum possible bandwidth into this peripheral stack is 3.2 Gbyte/sec using just one of the faces of the cube. The base of the cabinet holds an array of up to 80 3.5-inch disk-drives, power supplies, and the host.

The experiments reported in this paper were run on a 512-node machine arranged as an $8 \times 8 \times 8$ mesh, operating at a processor clock rate of 12.5Mhz (derived from a 25Mhz input clock).

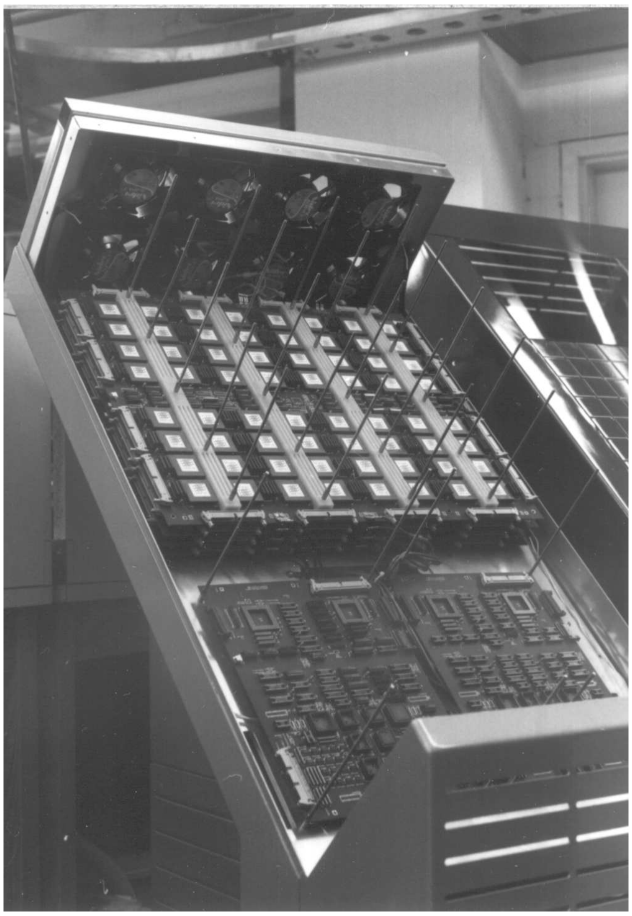

Figure 1: J-Machine.

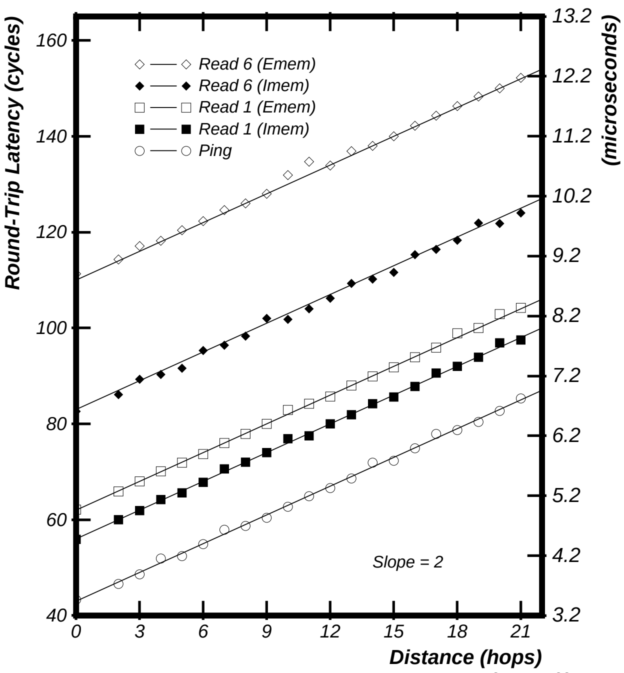

Figure 2: Round Trip Latency vs Distance for different types of remote reads, explained in Section 3.1.

3 Micro-Benchmark performance

To characterize the performance of the J-Machine's communication and synchronization mechanisms, we have written a number of small, synthetic programs designed to measure various aspects of J-Machine performance. Our studies focus on the tight coupling between the processor core and the network to support fine-grained communication, the impact of the differing communication bandwidths among the modules of the MDP, and the variation of message latency with network load.

3.1 Network performance

Communication performance is characterized by throughput, which is the number of messages per unit time that can be delivered, and latency, which is the end-to-end time required to deliver one message. Throughput limits the performance of communication intensive programs with large amounts of parallelism. Latency adds to the length of the critical path through a program with limited parallelism. Both latency and throughput depend on the end-to-end performance of the communication system, not just on network performance. The performance of the network interface and the coupling between the processor and the network interface are key factors in determining overall communication performance.

Network latency Figure 2 shows the round-trip latency of a null remote procedure call (RPC) as a function of the distance traversed and the data transferred. These measurements are taken on an unloaded network. The indicated data points show measured values for a 512 node machine.

Each line corresponds to a different type and size of data transfer. The line labeled *Ping* gives the latency for sending a two-word request message to the remote node and waiting for and receiving a single word acknowledgment. The remaining lines give the latencies for remote reads of 1 or 6 words from either the on-chip memory (Imem) or the off-chip memory (Emem). The remote read is performed by sending a three-word request message to a remote node. That node reads the specified address and replies with a 2 or 7 word message containing the requested data.

The figure shows that there are four components to round-trip remote-procedure-call latency on the J-Machine. In addition to a base latency, there are components due to distance, message length, and computation. The base latency of 43 cycles is the round-trip latency for a node

Machine		T_s μs/msg	T_b μs/byte	cycles/msg	cycles/byte
nCUBE/2	(Vendor)	160.0	0.45	3200	9
CM-5†	(Vendor)	86.0	0.12	2838	4
DELTA	(Vendor)	72.0	0.08	2880	3
nCUBE/2	(Active)	23.0	0.45	460	9
CM-5	(Active)	3.3	0.12	109	4
J-Machine		0.9	0.04	11	0.5

†: blocking send/receive

Table 1: One-way message overhead. T_s is the sum of the fixed overheads of send and receive. T_b is the injection overhead per byte [6], [17].

pinging itself. This latency consists of two trips through the network at a total cost of 24 cycles, and the execution of two threads at a total cost of 19 cycles. All of the lines in the figure have a slope of two, indicating that one-way latency increases by one clock per hop as the distance traveled increases. Increasing the message length also increases the latency proportionally. The time required to compute the result (*i.e.* access the appropriate memory) adds 2 cycles per word for internal memory and 8 cycles per word for external memory.

Table 1 compares the asynchronous send and receive overheads of the J-Machine to those reported for several other current multicomputers. These overheads are generally composed of times to format a message and inject it into the network on the sender side, and to either poll or interrupt the processor and absorb the message on the receiver side. Note that network latency is not included in this table. The first three rows in the table are the times reported by the vendor based on their message libraries. The second set of rows are for tuned implementations of the Active Message system [17] on the same hardware and give a sense of the reduction in overhead that can be achieved when the programming model matches the available hardware more closely. We observe that the J-Machine reduces the overhead per byte by almost an order of magnitude and the overhead per message by up to two orders of magnitude. This is a striking difference whose repercussions have a dramatic impact on communication costs in the machine.

Network latency vs load The latency statistics reported above were measured on an unloaded network. In a loaded network there is an additional component of latency due to contention. We developed an experiment to evaluate the total one-way message latency for messages of different lengths at different average network loads. In this experiment, every node in the 512 node machine repeats a simple loop in which it selects a random destination, sends a message of length N to the target, waits for an N word acknowledgment, and then idles for L cycles. The idle phase simulates computation and its duration has an inverse relation to the average traffic load placed on the network. We run this loop for between 15 and 30 seconds, and then compute the average message rate and thereby the bisection traffic. We establish a base case in which no messages are transmitted in order to determine the number of cycles that execute in the loop itself. This allows us to measure the total number of cycles for the round-trip exchange. We divide this by two to obtain the one-way message latency.

The left-hand side of Figure 3 shows the one-way message latency in a 512 node machine as a function of network bisection traffic. A family of four curves represent messages of length 2, 4, 8, and 16 words, respectively. Again, the difference in latencies for the zero load case is due to both message overhead costs and propagation delays. The idle time L is varied to generate the full range of possible loads. The basic loop of the application takes 45 cycles without any idling. This sets an upper bound on the possible message rate for this experiment that is quite adequate for our purposes. The figure shows that random traffic causes the network of a 512-node J-Machine to saturate with a bisection traffic of 6Gbits/sec, nearly half of the bisection capacity (14.4Gbits/sec). This measurement is consistent with analytical predictions and simulation studies [1] [4]. The figure also shows that the component of latency due to contention increases in the expected manner [1].

The right-hand side of Figure 3 uses the same data to illustrate how network contention affects the achievable ratio of communication to computation. We consider an experiment in which each node picks a random destination within the machine, sends an N word message and receives an N word acknowledgment, performs a computation that is modeled by an idle loop, and then repeats the sequence. The time spent in computation as a fraction of the total

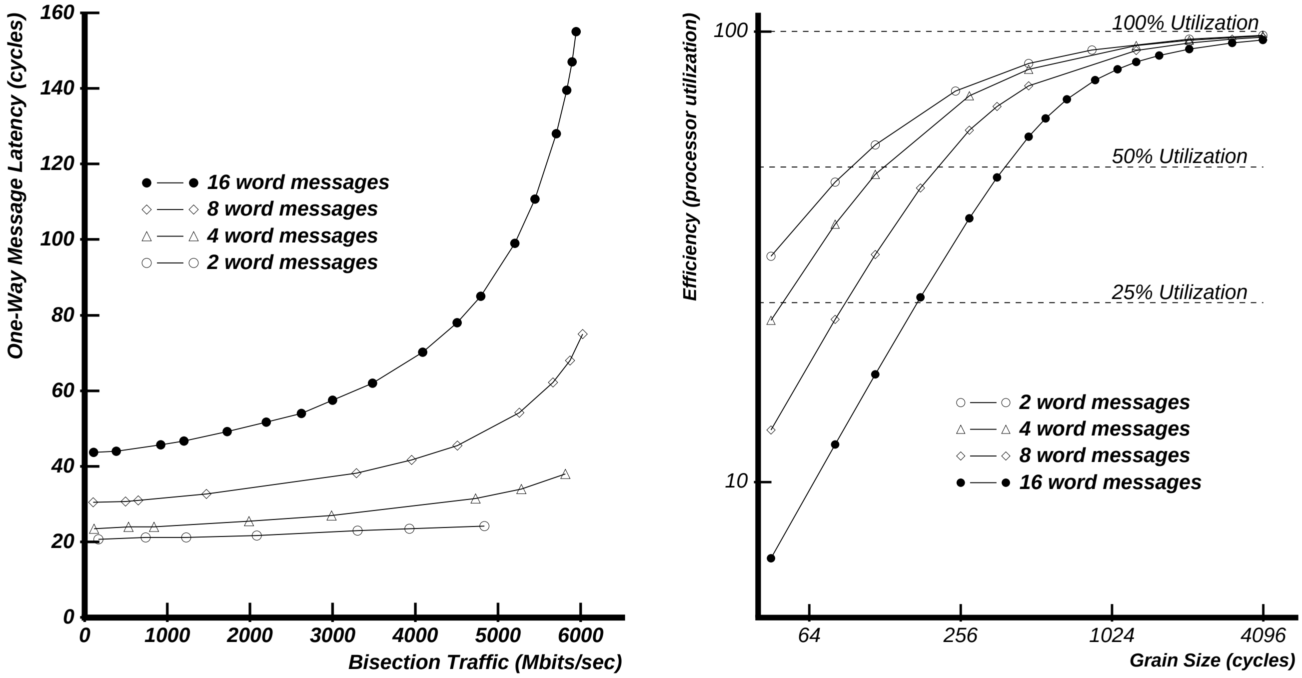

Figure 3: The left side of this figure shows message latency as a function of bisection traffic for an 8x8x8 J-Machine. The right side shows the processor efficiency vs grain size.

time determines the efficiency. The right side of the figure shows the coarse-grain case where efficiency is 100% and performance is determined only by computation. As one moves to the left along each curve, a point is reached where communication latency becomes the limiting factor and the efficiency decreases since increasingly more time is spent awaiting the completion of each message exchange. The *half-power* point, where efficiency reaches 50%, occurs when the computation to communication ratio is between 100 and 300 cycles/message for the message lengths shown.

Throughput Network saturation bandwidth, discussed above, is a limiting factor for programs that perform distant communications. For programs that exploit locality and tend to communicate mainly with neighboring nodes, however, terminal network bandwidth often becomes the limiting factor to performance.

A remote operation incurs overhead due to message setup, channel acquisition, and message invocation. This overhead is traditionally amortized by ensuring that remote accesses transfer relatively large amounts of data [9]. Requiring coarse-grain communication complicates programming in general and makes fine-grained synchronization and effective load-balancing particularly difficult. The efficient communication mechanisms of the J-Machine enable us to approach the effective terminal bandwidth of the network using small messages.

Figure 4 shows the maximum data-transfer rate that can be sustained between two nodes for a given message size. The data is generated at the source by loading dummy data directly from the register file. When the message is dispatched at the destination, it may be discarded, copied into internal memory, or copied into external memory. The figure shows that even for the highest bandwidth case (discarding the delivered data), the bandwidth is already within a 90% of its peak rate with messages as short as eight words. In all of the cases, two word messages achieve a throughput that is more than half of the eventual peak.

3.2 Synchronization

Synchronization involves enforcing the ordering of events in a parallel program. The J-Machine provides hardware support for synchronization via efficient message dispatch and the use of presence tags in the memory and register file. The task that is created and dispatched in response to each arriving message is a natural means to signal a waiting task that the event it is waiting on has occurred. Presence tags on every storage location are used to signal when a value is not yet available. Attempting to read a not-present value raises an exception. Since synchronization is often performed in conjunction with the production of data, it is natural to associate the synchronization tag with the data.

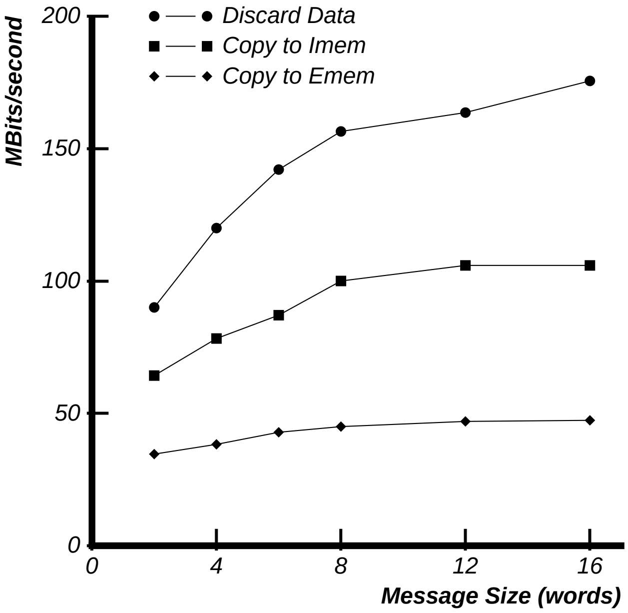

Figure 4: The terminal network bandwidth.

Event	Tags	No Tags	Save/Restore
Success	2	5	
Failure	6	7	30 - 50
Write	4	6	
Restart	0	0	20 - 50

Table 2: A comparison of synchronization times, in cycles, with and without hardware support for presence tags.

Software barrier synchronization (μsec)					
Nodes	EM4	J	KSR	IPSC/860	Delta
2	2.7	4.4	60	111	109
4	3.6	6.5	90	234	248
8	4.7	8.7	180	381	473
16	5.4	11.7	260	546	923
32		14.4	525	692	1816
64	7.4	16.5		847	3587
128		20.7			
256		24.4			
512		27.4			

Table 3: Barrier synchronization [6], [7], [14].

Producer-Consumer synchronization Table 2 compares the cost in cycles of performing a local producer-consumer synchronization both with and without the J-Machine presence tags. In the absence of tags, a separate synchronization variable must be tested before (or set after) accessing data. The table shows the costs of four events: reading data that is ready (Success), attempting to read unavailable data (Failure), writing data without needing to restart a thread (Write), and restarting a thread when the data is available (Restart). All data is stored in on-chip memory. We separate the general cost of thread suspension and restart, for which we provide a range of values reflecting different possible policies within the runtime and compiler system, from the issues directly related to the data operations. This table shows that the use of hardware provides a modest advantage when the test passes and that task suspension and restart overhead, rather than the synchronization overhead itself, is the dominant cost of failure. For the MDP, this synchronization mechanism is naturally integrated with our general mechanisms for runtime data tagging and fault management.

Barrier synchronization To test the ability of the J-Machine to synchronize sets of tasks, we measured the performance of a barrier synchronization library routine written in assembly language. Table 3 indicates the time taken in microseconds per barrier for a range of machine sizes and compares the performance with a number of contemporary parallel computers. It is difficult to compare the performance of our algorithm to similar routines on other machines as presented in the literature since sufficient detail to determine what is being included is rarely provided. Our numbers represent the time taken from the point at which the current thread calls the barrier routine until the time this single thread is resumed.

The barrier synchronization library routine is implemented in a *scan* [3] style. For an n processor machine, $O(n \log_2 n)$ messages are sent, n per wave. The pattern formed by the messages is that of a butterfly network mapped onto a 3-d grid, with the messages sent each wave representing one stage of the butterfly. Incoming messages invoke a different handler for each wave; this matching is done quickly through the use of the fast hardware dispatch mechanism.

Although faster synchronization times can be achieved by hardware barrier synchronization mechanisms, software routines can be used to synchronize several sets of threads simultaneously.

4 Macro-Benchmark performance

We have implemented a number of programs on the J-Machine and profiled their execution to determine the role that mechanisms play in enhancing performance. We discuss the programming tools we used and the general characteristics of the applications we studied.

4.1 Programming systems

Two J-Machine programming systems were used to conduct the benchmark studies reported in this section: Tuned J and Concurrent Smalltalk (CST). Two other programming systems are also running on the machine: Id [12] [15] and PCN [16].

Tuned J The J language serves as a system-level programming language for the J-Machine. It extends a per-node ANSI C environment with a small number of additional constructs for remote function invocation and synchronization. The user is responsible for data placement and task coordination, providing the ability to tune and to readily understand the runtime behavior of the application at the cost of increased programming effort. The J compiler currently produces inefficient code, so we perform modest hand-tuning of a few of the critical code sequences to reflect the expected future behavior of the compiler. Applications are profiled by static evaluation of the basic blocks of key routines and the addition of counters to determine the dynamic frequencies for each block.

Three of the applications reported here were written in both C and either equivalent hand-tuned J or assembly language. The run times on a single node of the J-machine running at 12.5 MHz varied from 8% faster to 20% slower than the C versions compiled with a C compiler and run on a Intel 386SX processor at 33 MHz.

CST The Concurrent Smalltalk programming system [8] supports object-based abstraction mechanisms and encourages fine-grained program composition. It extends sequential Smalltalk by supporting asynchronous method invocation, distributed objects, and a small repertoire of control constructs for explicit specification of parallel control flow. The compiler and runtime system provide the programmer with a global object namespace. The CST system includes several simple tools for analyzing the dynamic behavior of an application. We augment these tools with hand-placed counters as in Tuned J.

4.2 Parallel applications

Longest Common Subsequence This program determines the longest subsequence of characters common to two source strings, where a subsequence is the string remaining after removing arbitrary characters from a source string. In this implementation, one string is distributed evenly across the nodes of the machine; the other string is placed on Node 0 and the characters are passed across the nodes in a systolic fashion. We studied the application for the case where the first string is 1024 characters long and the second 4096. This program was written directly in assembly language.

Radix Sort This application sorts a set of integer keys. The keys are sorted one digit at a time, with digits represented by b-bits, from the least-significant to the most-significant digit. A stable, three-phase counting sort was used to sort each digit. In the first phase, a single scan of the source array determines the number of keys that hash to each of the digits (0 to $2^b - 1$). These keys will appear consecutively in the destination array at the end of the iteration. The counts are used to compute the initial offset for each key in the destination array. Finally, the array is scanned again and the data is reordered. In the parallel version, the data is distributed evenly across the machine. The counts computed by each node are combined and the initial offsets are generated using a binary combining/distributing tree. This program is written in Tuned J and sorts a set of 65,536 28-bit integers 4 bits at a time. This data set is small enough that it can be contained entirely within the local memory of one node to serve as a challenging base case.

N-Queens This familiar program determines the number of ways in which N queens may be placed on an NxN chess board so that no queen may attack another. It represents a range of graph-searching problems, all of whose central challenge is to control the potentially explosive parallelism that is available. This program is written in Tuned J and we explore its behavior as it solves the 13 queens problem.

Traveling Salesperson Problem The Traveling Salesperson Problem (TSP) benchmark is a classic path-searching program which finds the optimal tour in which the nodes of a graph with weighted edges can be traversed. A task is defined as a unique subpath of a given length; the tasks are initially distributed evenly over all the nodes. To process a task, a node explores all possible tours containing the subpath in depth-first order, while maintaining the shortest tour seen so far. If the length of the task's subpath is larger than the current bound, the node does not explore the subpath. This program is written in Concurrent Smalltalk and solves a 14 city configuration.

4.3 Application analysis

We present three sets of results that summarize the behavior of each of these applications. We will use these figures in the more detailed discussions that follow.

Figure 5 shows the speedup of each application. The apparent speedups that occur for a given application are strongly affected by the definitions of speedup used and the base case for the analysis. In all of these experiments we

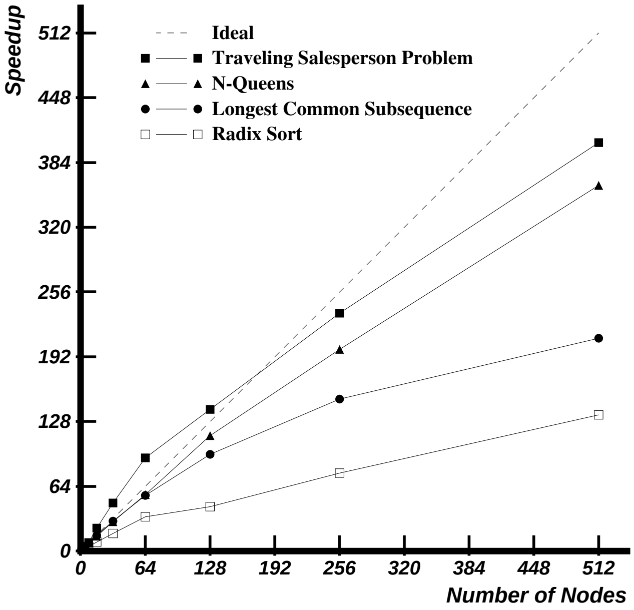

Figure 5: Speedup of applications on the J-Machine.

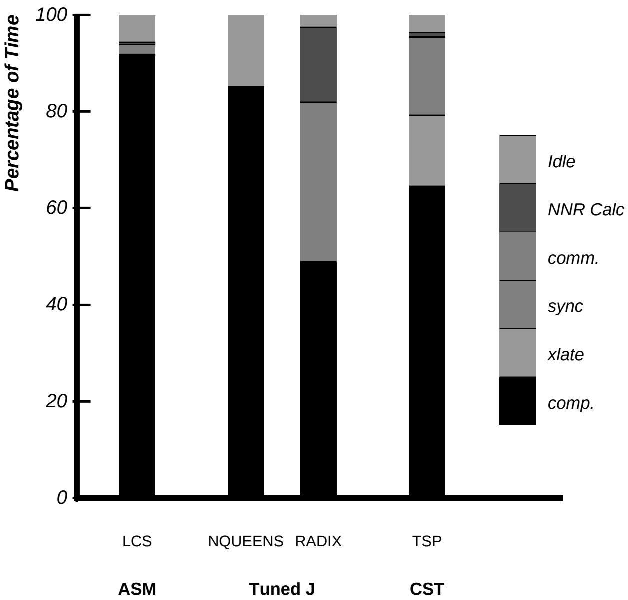

Figure 6: Breakdown of the total functions performed in each application running on a 64 node J-Machine.

hold the problem size constant. The base case for LCS, Radix-Sort, and N-Queens is a good sequential implementation; for TSP it is the parallel code. We view these curves as being primarily useful in directing attention to interesting trends rather than being of significance in themselves.

Figure 6 indicates the coarse breakdown of the functions performed by each node of a 64 processor machine for these applications. The calculations are in terms of cycles for LCS, N-Queens, and Radix Sort, allowing overhead such as the dispatch time to be considered, and in instructions for TSP, de-emphasizing the importance of the disparity between using internal and external memory.

Finally, Table 4 summarizes a number of metrics for the assembly language and Tuned J applications running on the 64 node machine. Each of these applications relies on two major classes of threads, for which the statistics are shown separately. For each application, we present the 64 node machine run time. For each thread, we additionally show the number of times the thread is invoked, the number of instructions run, the average thread length, and the message length used when invoking the thread.

4.3.1 Longest Common Subsequence

With 64 nodes, each node contains 16 characters of the base string and receives 4096 messages, each containing one character of the second string. The application is dominated by the behavior of this single message handler. The

	LCS	NQueens	RadixSort
RunTime (ms)	153	775	63
Thread 1	NxtChar	NQueens	Sort
# Threads	262,000	1,030	64
# K Instr	60,800	305,000	17,600
Instr/Thread	232	296,000	276,000
Msg Length	3	8	8
Thread 2	StartUp	NQDone	Write
# Threads	1	1,180	452,000
# K Instr	86	25	1,810
Instr/Thread	86,000	21	4
Msg Length	1	3	3

Table 4: Application statistics for a 64 node J-Machine.

application is initiated by Node 0 sending 4096 messages to itself, to process each character. After each node handles a character, it sends a message containing the character and the partial match information to its neighbor.

The key message handler consists of a fixed prologue section which generates an index into the state tables that maintain the match information, a loop over each of the characters in the base string on the node, and an epilogue section which forwards the partial result. As the size of the machine is increased, the time spent in the loop scales down proportionally, but the thread entry and exit cost remains constant. At 64 nodes, the handler entry and exit account for 9% of the total time. This overhead grows to 24% and then 33% as the machine grows to 256 and then 512 nodes respectively, becoming the dominant effect on scaling.

Performance is also reduced by idle cycles arising from two effects: load imbalance due to initialization and systolic skew. Work is distributed evenly except that Node 0 generates the initial set of 4096 messages. These messages appear one at a time and begin to traverse the machine almost immediately. Node 0, however, has significantly more work to do than the other nodes, causing them to periodically idle waiting for it to catch up. This load imbalance accounts for 4%, 13%, and 17% of the total runtime for machines of sizes 64, 256, and 512 nodes respectively. Nodes also idle due to systolic skew (pipeline end effects) during which the work front has either not reached a given node or has passed the node but has not reached the final node. This systolic skew represents a maximum of 11% of the total runtime across the range of machines studied.

4.3.2 Radix Sort

Radix sort is composed of two principal functions: the main loop that iterates seven times across the three phases, and the message handler that supports remote writing of values during the reorder phase. We regard the outer function as a single thread which suspends twice per iteration, once at the end of the counting phase and once at the end of each iteration, and each WriteData message as a separate thread. Table 4 shows that the outer-loop threads run for 310,000 instructions on a 64 node machine while the WriteData messages are only 4 instructions (16 cycles) each. We have written this application in a "fine-grained" style in which each value is written to its new slot as soon as the location has been computed. In machines lacking efficient communication primitives, one would need to collect and send large blocks of data to each node in order to amortize the overhead of injection.

Figure 5 shows three regimes of execution. The speedup from the tuned sequential one node version to the parallel two node version is 1.3. This is because the critical path through the inner loop is slightly longer and because the cost of a remote memory write is over three times greater than a local memory write. The latter effect is the dominant term and becomes even more pronounced as the machine size grows. Between 2 and 64 nodes, the performance is limited by local processor throughput, and the runtime halves for each doubling in machine size. From 64 to 128 nodes, the bisection bandwidth remains constant but the offered traffic is approaching the onset of saturation. This results in the glitch observed. Thereafter, the bisection bandwidth again doubles with each doubling in the number of nodes and the relative runtimes halve accordingly.

The discontinuity between 64 and 128 nodes is also affected by unfairness in the MDP router. Arbitration for output channels occurs at a fixed priority and nodes may be unable to inject a message into the network for an arbitrarily long period of time during periods of high congestion. Network congestion causes back-pressure on the nodes in the form of send-faults that indicate that a word cannot be accepted for injection. We have verified that certain nodes experience fault rates that are as much as two orders of magnitude higher than average. These nodes are correlated with physical position in the machine and logical position within the tree of combining/distributing messages used in the phase immediately preceding the reorder phase. This problem will always be resolved by the inevitable synchronization points that occur in parallel applications, but any nodes that have been blocked will need to catch up with their peers that have been able to make progress.

To achieve good performance our radix sort implementation had to compensate for the MDP's inability to accept and process messages at the peak rate of the network. The peak network to message buffer bandwidth is 0.5 words/cycle. The processor core is able to read words from the message queue at the same rate but it takes at least 3 cycles to relocate the value into internal memory and 6 into external memory. Typically, message handlers also update local data or synchronization state resulting in additional cycles per word. These operations lead to a rate mismatch which can cause the message buffer to experience transient overflows. This problem is especially challenging in an application like this one in which thousands of messages arrive during a single phase. This effect forced us to reduce the amount of work performed in the handler and to relocate this work to a different phase.

4.3.3 N-Queens

N-Queens is a graph search problem, with a state space exponential in the problem size. The key difficulty in an efficient N-Queens implementation is to control the explosive parallelism that tends to cause a large number of threads, overwhelming the machine's buffering resources.

The N-Queens application attacks this problem by expanding the number of boards first in a breadth-first manner, then switching to a depth-first traversal of the rest of the state space. The amount of breadth-first expansion depends on the machine size and the problem size. Most of the N-Queens threads are coarse-grained tasks that perform a depth-first traversal locally. For a 64 node machine running a 13 queens problem size, most threads are 300,000 instructions long, as can be seen in Table 4. Small messages are used to communicate boards to be expanded (eight words) and results from finished computation (three words).

The MDP provides one primitive mechanism for low-level task scheduling: the message queue. This queue can contain no more than 256 minimum-length messages (four words) and is configured for 128 of these messages in Tuned-J. This buffer is only large enough for at most 64 board-distribution messages. In this implementation, all of the work is generated at the start of program and spread relatively evenly across the machine. It is not possible to determine a priori how long each thread will run for and there are insufficient threads to rely on the laws of averages to balance the load. This results in an observed idle time of 15% (Figure 6). We can distribute more task descriptors that represent less work per thread by buffering the messages in memory using a system-level queue overflow fault handler but it is relatively expensive and is intended to be used for transient traffic overruns rather than as a general task management mechanism. Alternatively, we could implement a user-level "scheduler" that spreads the creation of smaller threads across the duration of the application.

4.3.4 Traveling Salesperson Problem

The traveling salesperson algorithm is the only application we discuss that is implemented in CST. As such, there are several key differences between this application and the others, because the style of machine usage that CST and the COSMOS runtime promote is different than that of J. There are no procedure calls per se; all calls become message invocations, either on the local node or a remote node. All data structures are *objects*, which can migrate to other nodes, or even change position on a single node, and are always referred to by a global virtual name which must be translated at every use. Finally, no priority 1 messages may be sent in the current CST/COSMOS system.

Nevertheless, an experienced CST programmer can take steps to eliminate certain sources of overhead. These efforts have led to an implementation that scales well and that has a single-node performance that is comparable to published run times for other implementations. Procedure calls were inlined in many cases to reduce context switching overhead. Data objects were declared to be non-migratable for the duration of the program. Incomplete tours of the cities can be redistributed to balance the load; they are represented, however, by primitive objects, and as such are not referred to with global IDs.

TSP		
	User	O/S
Run Time (msec)	26300	
# Threads (Msgs)	9.1×10^6	8.9×10^6
# Instructions	2.8×10^9	5.4×10^8
# `xlates`	5.1×10^8	
# `xlate` Faults	1.6×10^4	
Instr/Thread (mean)	309	61
Avg Msg Length	5.1	4

Table 5: Major components of cost for TSP.

The TSP algorithm, like the N-Queens algorithm, is a graph-searching problem. The main difference between them is that TSP can prune the amount of work that needs to be done based on the work already done: a shortest path cannot be longer than any already calculated path. Pruning dominates the behavior of the TSP application, as can be seen in the speedup curve shown in Figure 5. The multi-node version happens to find better paths using less total work than the sequential one for this particular city configuration, and therefore exhibits super-linear speedup for the smaller machines. The TSP application's use of dynamic task balancing results in an idle time of only 3.8% as compared to 15% in N-Queens (Figure 6).

The TSP application, also like N-Queens, is very coarse-grained in its current implementation. There are three main types of threads: task-processing, bounds-distributing, and work-requesting. A task-processing thread can run as a single 32,000 instruction task; however, it must be interrupted periodically so that the messages that propagate updated bounds can be processed. These messages could, in principle, be handled using priority one threads but CST/COSMOS does not currently support this. Instead, we cause the path-tracing thread to suspend periodically by performing a null procedure call. Sixteen percent (the synchronization overhead from Figure 6) of the time that TSP runs is currently spent in this operation.

The `xlate` instruction is used to translate global object names into local physical addresses. The name is generally stored in the current context frame, and `xlate` places the physical address in the specified address register. Because the instruction takes only three cycles if the name is found in the name-cache, it is also used to reload an address register after a spill. This accounts, in part, for the large number of `xlate` instructions and the low `xlate` miss ratio (the percentage of time an `xlate` misses and thus needs to fault is insignificant) as noted in Table 5.

5 Critique

This paper has reported the results of an experimental evaluation of the J-Machine multicomputer. We have measured the communication and synchronization performance of the J-Machine using synthetic micro-benchmarks and compared these numbers to published figures for other contemporary parallel computers. The behavior of several application programs has also been studied.

The focus of the J-Machine design was on the tight coupling between a conventional integer-processing core and a high-bandwidth, low-latency network. Fast message injection, automatic message buffering, and fast message dispatching form the foundation of a system that offers efficient communication and synchronization. The J-Machine also includes hardware-based presence tags to enhance fine-grained data synchronization and a general name translation mechanism (`enter` and `xlate`) to support a global name space.

The communication micro-benchmarks demonstrate that the J-Machine reduces one-way message overhead by one to two orders of magnitude compared to contemporary multicomputers constructed from off-the-shelf processors (Table 1). Combining effective injection and dispatching with a low-latency network enables a node in one corner of a 512-node J-Machine to read a word from the memory of its nearest neighbor in 60 cycles and from the opposite corner node in 98 cycles (Figure 2). Figure 3 shows that threads achieve processor utilization of over 50% with task lengths between 25 and 50 cycles per word of message data even for random message patterns. A terminal network bandwidth of 90% of the eventual peak is achieved with messages as short as 8 words (Figure 4).

The impact of the communication mechanisms and hardware-based message dispatching on synchronization primitives is clearly seen in our implementation of a barrier synchronization library routine (Table 3), in which the J-Machine is again one to two orders of magnitude faster than contemporary microprocessor-based multicomputers.

Each macro-benchmark application has provided insight into the interplay of the mechanisms of the J-Machine and the task of producing effective parallel programs. The performance of N-Queens, LCS, and TSP is set by factors such as the nature of the problem, the length of the sequential path length through the code, and load balancing, rather than by constraints imposed by the low-level mechanisms. The performance of radix sort is limited by the global bandwidth available and by the use of messages to pass every word of data. This is our only application to date that has stressed our fine-grain communication and synchronization mechanisms.

In running these benchmarks we became aware of a number of limitations of the J-Machine and MDP architectures. Including statistics collection hardware in the machine design would have greatly simplified and enhanced the measurement collection process for this paper. The inclusion of a cycle counter, for example, would have enabled the time-stamping of events.

The MDP's external memory bandwidth limits performance. The network can deliver data at a rate three times greater than the external memory can accept it. External memory latency (6 cycles) also limits the sequential performance of each node. This limitation could be addressed by using a RAM technology with a higher bandwidth to capacity ratio and automatically managing the on-chip memory as a cache.

The naming mechanisms of the MDP are inadequate to transparently and inexpensively provide a global name space. CST applications spend time `XLATE`ing virtual addresses to segment descriptors and some applications spend considerable time converting virtual addresses or linear node indices to router addresses. Automatic translation from virtual memory addresses to physical memory address and from virtual node id's to physical router addresses would eliminate the need for explicit name management. This mechanism could be implemented with a pair of TLBs. It would both eliminate much of our current naming overhead and, by protecting the translation of node addresses, provide greater protection between programs running on different partitions of the machine.

A method of applying "backpressure" to sending nodes is needed to avoid overflowing the MDPs on-chip queue into off-chip memory. Such a flow-control mechanism would also avoid blocking the network completely when one node stops accepting messages to handle a queue overflow. One simple method of flow control is to adopt a "return-to-sender" protocol that refuses messages when the queue is above a certain threshold by returning them to the sending node.

Future directions Our experience with the J-Machine has shown that global bandwidth is a critical resource that limits the computation to communication ratio for highly parallel programs. To reduce the demand for bandwidth, we are exploring methods for building parallel software systems that minimize communication by exploiting locality [11].

The MDP's paucity of registers, while reducing context switch time, resulted in more memory references than were necessary. We are currently researching methods of increasing the number of architectural registers without increasing context switch time for short-lived tasks [13].

To extract more parallelism out of an application of a given size, we are exploring ways to combine compile-time and run-time scheduling to exploit inter-thread and instruction-level parallelism [10].

The data presented here demonstrates that the J-Machine's communication, synchronization, and naming mechanisms are effective in supporting programs with small task sizes (150 instructions). These programs also demonstrate that a fast communication facility makes it possible to operate with only a small amount of memory per node (1MB) by fetching instructions and data from remote nodes when needed.

Conventional parallel computers offer no cost/performance benefits when compared to the workstations from which they are derived. Rather, they extend the absolute performance. Providing efficient mechanisms allows the construction of physically small nodes, which would offer the potential of better cost/performance across a range of machine sizes. It would be straightforward to incorporate the J-Machine mechanisms with the improvements described above into a conventional RISC processor core without affecting sequential performance. Combining such a core with a small, high-bandwidth memory and a router would result in an integrated processing node suitable for use in machines ranging from personal workstations (a few nodes) to departmental servers (tens to hundreds of nodes) and supercomputers (thousands of nodes). By reducing the memory and hence cost per node by an order of magnitude, such physically fine-grain machines offer proportionally better cost/performance while retaining the same total memory capacity.

Acknowledgments

The existence of a 512 node J-Machine with an infrastructure for basic performance evaluation is a credit to a large team of researchers; all are extended our gratitude. In addition we wish to acknowledge those people who contributed directly to this paper: Kirk Johnson and Steve Keckler wrote micro-benchmark programs. Shaun Kaneshiro, John Keen, and Kathy Knobe wrote and instrumented macro-benchmark programs. We thank them for their generous assistance.

References

[1] AGARWAL, A. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems 2*, 4 (Oct. 1991), 398–412.

[2] BAKER, H. C., AND HEWITT, C. The incremental garbage collection of processes. In *Conference Record of the Conference on AI and Programming Languages* (Rochester, New York, Aug. 1977), ACM, pp. 55–59.

[3] BLELLOCH, G. Scans as primitive parallel operations. In *International Conference on Parallel Processing* (1987), pp. S355–362.

[4] DALLY, W. J. Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans. Comput. 39*, 6 (June 1990).

[5] DALLY, W. J., FISKE, J. S., KEEN, J. S., LETHIN, R. A., NOAKES, M. D., NUTH, P. R., DAVISON, R. E., AND FYLER, G. A. The Message-Driven Processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro 12*, 2 (Apr. 1992), 23–39.

[6] DUNIGAN, T. Communication performance of the Intel Touchstone Delta mesh. Tech. Rep. ORNL/TM-11983, Oak Ridge National Laboratory, Jan. 1992.

[7] DUNIGAN, T. Kendall Square multiprocessor: early experiences and performance. Tech. Rep. ORNL/TM-12065, Oak Ridge National Laboratory, Mar. 1992.

[8] HORWAT, W. A Concurrent Smalltalk compiler for the Message-Driven Processor. AI Memo, MIT, 545 Technology Sq., Cambridge, MA 02139, May 1988. SB Thesis.

[9] HSU, J.-M., AND BANERJEE, P. Performance measurement and trace driven simulation of parallel and numeric applications on a hypercube multicomputer. In *17th Annual International Symposium on Computer Architecture* (1990), IEEE Press, pp. 260–269.

[10] KECKLER, S. W., AND DALLY, W. J. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture* (Queensland, Australia, May 1992), ACM, pp. 202–213.

[11] KNOBE, K., LUKAS, J. D., AND DALLY, W. J. Dynamic alignment on distributed memory systems. In *The Third Workshop on Compilers for Parallel Computers* (Vienna, Austria, July 1992), Austrian Center for Parallel Computation.

[12] NIKHIL, R. S., AND ARVIND. Id language reference manual version 90.1. Tech. Rep. 284-2, Computation Structures Group, MIT, Cambridge, MA 02139, 1991.

[13] NUTH, P. R., AND DALLY, W. J. A mechanism for efficient context switching. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors* (Oct. 1991), IEEE, pp. 301–304.

[14] SHAW, A. Implementing data-parallel software on dataflow hardware. Master's thesis, MIT, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, January 1993.

[15] SPERTUS, E. Execution of dataflow programs on general-purpose hardware. MS Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Aug. 1992.

[16] TAYLOR, S., ET AL. Scalable concurrent programming project. Semiannual technical report, Dept. of Computer Science, California Institute of Technology, Apr. 1992.

[17] VON EICKEN, T., CULLER, D., GOLDSTEIN, S., AND SCHAUSER, K. Active messages: A mechanism for integrated communication and computation. In *Proceedings of 19th Annual International Symposium on Computer Architecture* (1992), IEEE, pp. 256–266.