The J-Machine Architecture and Evaluation*

William J. Dally, John S. Keen and Michael D. Noakes
Artificial Intelligence Laboratory and Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
billd@ai.mit.edu, johnk@ai.mit.edu, noakes@ai.mit.edu

Abstract
The MIT J-Machine is a fine-grain distributed memory multiprocessor that provides low-overhead mechanisms for general purpose parallel computing. Each processing node consists of a Message-Driven Processor (MDP) and 1 MByte of DRAM. The MDP microprocessor integrates communication, computation and memory management functions in a single VLSI chip. A 512 node J-Machine is operational and is due to be expanded to 1024 nodes in January 1993. In this paper we discuss the architecture of the J-Machine and evaluate the effectiveness of the mechanisms embodied in the MDP.

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 from efforts to build concurrent machines by wiring together many sequential processors, however, that these highly-evolved sequential mechanisms are not adequate to support most parallel models of computation. These mechanisms do not support synchronization of events, communication of data, or global naming of objects efficiently. As a result, these functions, inherent to any parallel model of computation, must be implemented largely in software with prohibitive overhead. For example, sequential machines require hundreds of instructions to create a new process.

The J-Machine [2] is a prototype multiprocessor that combines a fast interconnection network with efficient mechanisms for fine-grain parallel computation. These two technologies enable a J-Machine multiprocessor to operate efficiently with only a small amount of memory per node. Machines ranging in size from workstations to supercomputers can be constructed by combining tens to thousands of these nodes. The J-Machine mechanisms support fine-grain parallel computation in which processes last only a few tens of instructions, exchange small data objects, and synchronize frequently. Breaking a parallel program into more, smaller tasks both exposes more parallelism than is otherwise available and simplifies the parallel programming task.

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, a memory management unit, a 3-D mesh network router, a network interface, a 4K-word x 36-bit SRAM, and an ECC DRAM controller in a single 1.1M transistor VLSI chip. The MDP was developed in collaboration with Intel Corp. Rather than being specialized for a single model of computation, the MDP incorporates primitive mechanisms for communication, synchronization and naming that permit it to support most proposed models of parallel programming effectively. A 512 node J-Machine is in operation at MIT and will be expanded to 1024 nodes in January 1993.

This paper briefly describes the architecture of the J-Machine and evaluates its performance. More comprehensive explanation of the architecture and implementation of the J-Machine can be found in [1, 2, 4].

2 J-Machine Architecture
The MDP instruction-set includes the usual arithmetic, data movement, and control instructions. It is distinguished by its special support for communication, synchronization, and naming. A series of SEND instructions is used to send a message of arbitrary length to any node in the machine. Upon arrival, the message is buffered in a configurable hardware queue located in the on-chip memory of the MDP. When the message arrives at the head of the queue, a task is dispatched to handle the message in as few as four cycles.

Synchronization in the MDP is performed using message dispatch and presence tags on all state. The dispatch of a process on each message arrival allows messages to be used to signal events on remote nodes. Presence tags are used to synchronize on the arrival of data. For example, suppose a producer task is to compute a value and write it into a memory location which a consumer task will subsequently read. The location is initially tagged as empty. If the consumer attempts to read this location before the producer writes it, the MDP will raise an exception and cause the operating system to suspend the consumer task until the value

*The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88-K-0738 and F19628-92-C-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.
is written by the producer. Synchronization on data availability in this manner is quite common in many parallel programs.

The MDP supports naming with segmented memory management and with translation instructions. A memory segment can be allocated for each distinct data object in a program. An associated physical segment descriptor indicates the object's location in memory and its size. A corresponding virtual address is assigned to each object. The binding from virtual address to segment descriptor is stored in an associative XLATE table and is supported by hardware. New bindings can be placed in the table using the ENTER instruction. The XLATE instruction returns the segment descriptor, if any, associated with a given virtual address. Objects may move around in memory, as a result of heap compaction for example. When an object is relocated, its segment descriptor is updated. The XLATE table can be employed to keep track of objects which migrate from one processor to another.

Each processing node of the J-Machine occupies an area of 2" by 3" using conventional through-hole technology. A J-Machine processor board, which measures 20.5" by 26", contains an eight by eight grid of processing nodes. Standard ribbon cable connectors on the perimeter of the board can be used to expand the X and Y dimensions up to 32 nodes. The vertical interconnect between boards is implemented using 48 elastomer conductors. These strips, contained in 4 rows of custom holders, provide 1584 electrical connections between adjacent boards. Of these connections, 960 are used for signalling and the remaining are ground returns. The use of elastomeric connectors enables the J-Machine to achieve a very high network bisection bandwidth (16 Gbits/sec) in a very small cross sectional area (2 ft²) and at the same time keep the length of the longest channel in a 1024-node machine under four inches.

Up to 16 J-Machine processor boards are stacked in a chassis as shown in Figure 1 (next page). A complete stack contains 1024 processing nodes (14,150 MIPS peak) and 1 GByte of DRAM memory in a 4 ft³ volume. It dissipates 1.5KW. The chassis also contains space to mount up to 32 peripheral cards that interface J-Machine channels to standard peripherals. Each peripheral card supports up to four channels with a combined bandwidth of 113 MBytes/sec. The host-interface for the machine will be a Sun Sparc workstation and we have implemented an Sbus adapter to the J-Machine network. The 1024 node prototype will include 10 SCSI interface cards driving 20 SCSI channels with an aggregate bandwidth of 100 MBytes/sec. The initial installation will support 4 234 MByte (formatted) disks per channel with a total capacity of 18 GBytes. We will also support a high-performance graphics interface with a physically distributed frame buffer. This frame buffer will be distributed over four peripheral modules with an aggregate input bandwidth of 225 MBytes/sec.

The chassis is packaged in the upper portion of the J-Machine cabinet, shown in Figure 2. The base of the cabinet can hold up to 80 3.5-inch disk-drives mounted in card cages which form the secondary storage for the J-Machine.

3 Evaluation

Table 1 compares the asynchronous send and receive overheads of the J-Machine to that reported for several contemporary parallel computers. 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 four entries are the times reported by the vendor, and corroborated by independent researchers, based on their message libraries. The time for the CM-5 includes a three-phase protocol. The third pair of entries are for tuned implementations of the Active Message system [7] and gives a sense of the reduction in overhead that can be achieved when the programming model matches the available hardware more closely.

Table 2 shows the peak bandwidth among the major modules of the MDP. These figures are specified
Figure 1: J-Machine Chassis with Processor Stack and SCSI Adaptor.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Ts μs/msg</th>
<th>Tb μs/byte</th>
<th>Cycles/msg</th>
<th>Cycles/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPSC/860</td>
<td>&gt; 100 words (Vendor)</td>
<td>160.0</td>
<td>0.36</td>
<td>6400</td>
</tr>
<tr>
<td></td>
<td>&lt; 100 words (Vendor)</td>
<td>60.0</td>
<td>0.50</td>
<td>2400</td>
</tr>
<tr>
<td>nCUBE/2</td>
<td>(Vendor)</td>
<td>160.0</td>
<td>0.45</td>
<td>3200</td>
</tr>
<tr>
<td>CM-5f</td>
<td>(Vendor)</td>
<td>86.0</td>
<td>0.12</td>
<td>2838</td>
</tr>
<tr>
<td>nCUBE/2</td>
<td>(Active Msg)</td>
<td>23.0</td>
<td>0.45</td>
<td>480</td>
</tr>
<tr>
<td>CM-5</td>
<td>(Active Msg)</td>
<td>3.3</td>
<td>0.12</td>
<td>109</td>
</tr>
<tr>
<td>J-Machine</td>
<td></td>
<td>0.8</td>
<td>0.04</td>
<td>11</td>
</tr>
</tbody>
</table>

†: blocking send/receive

Table 1: One-way message overhead. Ts is the sum of fixed overheads of send and receive. Tb is the injection overhead per byte.

<table>
<thead>
<tr>
<th>Source</th>
<th>Network</th>
<th>Registers</th>
<th>Inem</th>
<th>Emem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles/w</td>
<td>MWords/s</td>
<td>Cycles/w</td>
<td>MWords/s</td>
</tr>
<tr>
<td>Network</td>
<td>2</td>
<td>7.0</td>
<td>14.1</td>
<td>56.7</td>
</tr>
<tr>
<td>Registers</td>
<td>0.5</td>
<td>28.3</td>
<td>2</td>
<td>7.0</td>
</tr>
<tr>
<td>Inem</td>
<td>2</td>
<td>7.0</td>
<td>3</td>
<td>4.7</td>
</tr>
<tr>
<td>Emem</td>
<td>8</td>
<td>1.8</td>
<td>9</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Table 2: Peak bandwidth between major modules of the MDP, in cycles per word and MWords per second.

185
in both cycles/word and Mwords/sec assuming a cycle rate of 14.1 MHz. Some of the exchanges must be synthesized from multiple instructions. For example, there is no primitive operation to copy directly between two words of internal memory and so this is implemented as a load/store pair. This table highlights several key points. The first is that the bandwidth of the network is equal to the bandwidth between the register file and internal memory. We also note that it is possible to copy words from the internal memory to the network as quickly as they can be copied to the register file. These characteristics contribute to the MDP’s ability to access remote memory with exceptional efficiency. On the negative side, we note the significant imbalance between the bandwidth of the internal memory and the external memory. This problem is fundamentally an artifact of the package that was available for the MDP, and the bandwidth to capacity ratio of standard DRAM parts.

The tight coupling between the instruction pipeline and the network interface supports a round-trip remote null procedure call in as few as 43 cycles. A null procedure between the most distant nodes in a 1024-node J-Machine can be accomplished in as few as 101 cycles.

Overhead is an important factor in determining network terminal bandwidth. A remote operation incurs overhead due to message setup, channel acquisition, and message invocation. This overhead is traditionally amortized by ensuring that remote access transfers relatively large amounts of data [3]. Requiring coarse-grain communication complicates programming in general and makes fine-grain 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 with messages as short as 8 words.

We are currently investigating the behavior of a variety of small to medium-sized applications in order to understand the effectiveness with which the J-Machine’s mechanisms support user-level codes. We also wish to explore the basic issues of implementing algorithms that are scalable to thousands of processors. We summarize the significant features of four of these programs in this paper: Longest Common Subsequence, the Traveling Salesman problem, Radix Sort, and a Ray-Tracer.

Figure 3 indicates the non-scaled speedup achieved for these four applications. In each of these examples we constrain the problem size to permit it to be run entirely within the memory resources of a single processor to establish the reference speed for the application. We maintain a fixed data size as the number of processors is increased.

The Longest Common Subsequence (LCS) program finds the longest sequence of characters that appear in the same order in two strings. In this experiment the two reference strings each contain 4096 characters. Partial matches progress across the nodes in waves. Skew in the startup and shutdown times of the processors leads to non-ideal speedup.

A Ray Tracing program divides the user’s viewport into a number of patches and this work is partitioned dynamically over the nodes of the J-Machine. The absolute performance of the Ray Tracer is limited by a lack of floating point hardware in the J-Machine. The scalability of this implementation has been much improved recently but still suffers from sequentialization in the work distribution mechanism.

The Traveling Salesman problem is a classic path-searching problem. The program finds the optimal tour in which all the nodes of a graph with weighted edges can be traversed. Tasks are dynamically redistributed to accommodate different rates of pruning and thereby maintain an even load across the J-Machine. This application is currently constrained by the sequentialization of access to the global current best path.

Radix Sort orders a set of N L-digit numbers by performing a sequence of partial reorderings. Each iteration consists of three phases during which the data is copied from a source array to a destination array so as to reorder the numbers based on the value of a subset of the digits. This application is perhaps the most revealing of certain architectural limitations of the J-Machine. The initial step speedup from the idealized one node case to two nodes is only 1.28. This occurs because of the relatively high cost of a remote memory write to a local memory write even on the J-Machine. After that each doubling of the number of nodes halves the runtime but true speedup remains less than 1/2. At 128 nodes the average link utiliza-
tion during the global reordering phase approaches the onset of saturation. This issue is compounded by a difficulty in channel arbitration in the current network implementation.

4 Critique

We have demonstrated that the J-Machine’s end-to-end hardware support for communication reduces communication latency by an order of magnitude compared to conventional parallel computers. These low overheads are achieved not only by having a fast network, but also by providing hardware support for formatting and injection at the source node (the `SEND' instruction) and for storage allocation, task creation, and task dispatch at the receiving node. Using these mechanisms, J-Machine end-to-end latencies are comparable with machines hardwired for a particular model of computation (e.g., shared memory) while providing more generality. Low message startup overheads enable efficient use of very short messages, down to a single word in length.

We have performed experiments to measure the terminal bandwidth, the rate at which a node can transfer data to a remote memory. Experiments of this nature suggest the need for a more general messaging system that can deposit an arriving message directly in its final destination. With the current system, the message is first deposited in a circular queue in internal memory. If the data in the message is destined for memory it must then be copied there explicitly, resulting in a total of three memory crossings. The terminal bandwidth would be greatly increased by avoiding the intermediate step.

Our application results also highlight the deleterious effect of the low bandwidth of the external memory interface of each J-Machine node. While register and internal memory bandwidth are well-matched to the processor and the network, external memory bandwidth is slower by a factor of eight. This can be attributed to pin-limitations on the physical package we used, which reduced the external memory data bus width, and to the low bandwidth to capacity ratio of standard DRAM components. The 1M×4 parts used in the J-Machine have a ratio of 8.3sec⁻¹. To build a high-bandwidth node memory with a small capacity would, in the short term, require memory components that have a higher bandwidth to capacity ratio, such as the recently announced RAMBus parts [6]. In the long term, the processing node will need to be integrated with its main memory on a single chip.

Global synchronization can be achieved using a software-implemented barrier built from the basic J-Machine communications mechanism with a latency that is competitive with other software approaches. While somewhat slower than dedicated hardware, the software approach has the advantage that it can be used to synchronize several arbitrary sets of tasks simultaneously without having to arbitrate for the use of global synchronization hardware.

In analyzing the performance of our applications we have become painfully aware of a number of limitations of the J-Machine and MDP architectures.

An automatically managed memory hierarchy (i.e., organizing the internal memory as a cache) would have greatly reduced the access penalty for locating data in external memory since frequently used data would be automatically migrated.

The J-Machine network is not fair in that a busy channel can lock out a competing channel for an arbitrary amount of time. This affected the scalability of the Radix Sort application. A gating protocol for fairness would improve the predictability of some programs with only a modest increase in network cost.

The small register set of the MDP, while giving a fast context switch time, resulted in many 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 [5].

The MDP lack of floating-point hardware dramatically reduces its overall performance on numerical programs and results in an inflated computation to communication ratio for this class of application.

With the exception of the fairness issue, none of these problems are due to the communication and synchronization mechanisms that are the contribution of the J-Machine project.

Our experience suggests marrying the J-Machine communication, synchronization, and naming technology with a contemporary high-performance RISC processor and 4 MBytes of high-bandwidth DRAM memory. Such a processing node could be used in systems ranging from a workstation with 16 nodes to a server with hundreds of nodes to a supercomputer with thousands of nodes. For each class of system, the fine-grain parallel computer would have about the same amount of total memory as its sequential counterpart. The performance on purely sequential applications would be no better than a conventional machine, but it would outperform its sequential (or modestly parallel) counterpart by one to two orders of magnitude on parallel programs.

Our studies have 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 dynamically migrating data structures throughout the computation.

To extract more parallelism out of an application of a given size, we are exploring ways to combine compile-time and run-time scheduling. Using J-Machine-like mechanisms we can dynamically schedule tasks with a few tens of instructions. Our hypothesis is that with appropriate hardware support we can exploit all the levels of parallelism that a compiler can detect.

We have determined that the J-Machine communication, synchronization, and naming mechanisms are effective in supporting programs with small task sizes (150 instructions). The use of a segment-based addressing architecture provides protected accesses to

---

1The bandwidth to capacity ratio is defined as the number of times per second that all of the DRAM memory can be read out through the pins.
small data objects. Our applications 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. The biggest performance problem seen today with our macro-benchmark programs is load balancing which can be addressed by reducing the task size and dynamically balancing the load.

Acknowledgments
A system design project of the scope of the J-Machine project necessarily builds on the efforts of a large number of co-workers who contributed to the construction of the machine, the software infrastructure, and the development of evaluation programs. We also gratefully acknowledge many generous colleagues in other groups who assisted us in comparing and contrasting our efforts with their research directions.

References


