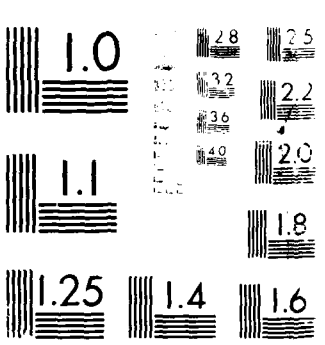
AD-A195 432 A FINE-GRAIN MESSAGE-PASSING PROCESSING NODE(U) 1/1
MASSACHUSETTS INST OF TECH CAMBRIDGE MICROSYSTEMS
RESEARCH CENTER W J DALLY NOV 87 VSLI-MEMO-87-421
UNCLASSIFIED N00014-80-C-0622 F/G 25/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(4)

VLSI Memo No. 87-421
November 1987

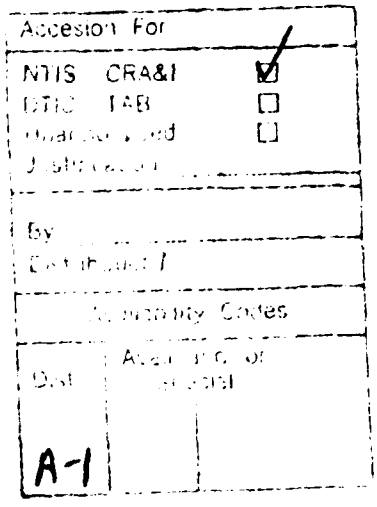
A FINE-GRAIN, MESSAGE-PASSING PROCESSING NODE

William J. Dally

Abstract

This paper describes a processing node for a fine-grain message-passing concurrent computer. The node consists of a processor, a communication unit, and a memory. To reduce the overhead of message passing and task switching to 5μs, the node incorporates a send instruction, a fast communication system, hardware message buffering and dispatch, and a general translation mechanism. These mechanisms work together to implement a fine-grain programming system.

Microsystems Research Center Room 39-321
Massachusetts Institute of Technology
Cambridge Massachusetts 02139
Telephone (617) 253-8138



Acknowledgements

This work was supported in part by the Defense Advanced Research Projects Agency under contract numbers N00014-80-C-0622 and N00014-85-K-0124, and by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation.

Author Information

Dally: Artificial Intelligence Laboratory and Laboratory for Computer Science, MIT, Room NE43-419, Cambridge, MA 02139, (617)253-6043.

A Fine-Grain, Message-Passing Processing Node[1]

William J. Dally
Artificial Intelligence Laboratory and
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

This paper describes a processing node for a fine-grain message-passing concurrent computer. The node consists of a processor, a communication unit, and a memory. To reduce the overhead of message passing and task switching to $5\mu s$, the node incorporates a send instruction, a fast communication system, hardware message buffering and dispatch, and a general translation mechanism. These mechanisms work together to implement a fine-grain programming system.

1 Introduction

The natural grain size of many parallel algorithms is about 20 instructions. To fully exploit the concurrency in such algorithms, we must be able to efficiently execute tasks of this length. The message transmission and reception overhead of existing systems is in excess of 200 instruction times. With such a large overhead, these systems must execute tasks at the artificially large grain size of about 1,000 instructions. If we can reduce the overhead and operate at the natural grain size, we can effectively apply 100 times as many processing elements to the problem.

At MIT we are developing the J-Machine [12], a fine-grain message-passing concurrent computer that will efficiently execute programs with 20-instruction tasks. The J-Machine is consists of a number (initially 4096) of possibly different processing nodes that communicate over a high speed network. The network is implemented using a Network Design Frame (NDF) [10], a communication controller integrated into the pad-frame of a chip. The center of the NDF chip is used to implement a node-specific processor. The primary processing node of the J-Machine consists of an NDF wrapped around a Message-Driven Processor (MDP) [9], a symbolic processing element.

A J-Machine constructed from MDP/NDF processing nodes combines features of both message-passing and shared memory machines. Like the Caltech Cosmic Cube

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-80-C-0622 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation.

[25], the Intel iPSC [18], and the N-CUBE [21], each node of the J-Machine has a local memory and communicates with other nodes by passing messages. The J-Machine can exploit concurrency at a much finer grain than these early message passing computers. Delivering a message and dispatching a task in response to the message arrival takes 5μs on the J-Machine as opposed to 5ms on an iPSC. Like the BBN butterfly [3] and the IBM RP3 [22] the J-Machine provides a global virtual address space. The same IDs (virtual addresses) are used to reference on and off node objects. Like the InMOS transputer [17] and the Caltech MOSAIC [20] the MDP/NDF is a single chip processing element integrating a processor, memory, and a communication unit.

The next section introduces the problems associated with fine-grain concurrent computation by means of a concurrent factorial program. In Section 3 the mechanisms provided by the MDP/NDF to support this style of programming are presented. The **SEND** instruction, message delivery, message buffering, scheduling and dispatching, and the MDP translation instructions are described. This section also discusses some of the features we didn't implement and the reasons why they were abandoned. Section 4 illustrates how the MDP/NDF mechanisms work together to execute the factorial program from Section 2.

2 The Problem

Concurrent programming is often considered harder than sequential programming because of partitioning, communication, and synchronization. If a machine is programmed at a very low level, with the programmer explicitly specifying the partition and the communication, concurrent programming can indeed be a difficult task, and the programs produced are rarely portable. However, with suitable programming abstractions [11], concurrent programming need be no harder than sequential programming.

Consider the factorial program shown in Figure 1. The program calculates $n!$ by recursively dividing the interval of integers to be multiplied. To compute $n!$, the depth of the recursion is $\log_2 n$. This program is patterned after one by Theriault [29] p.33. and is written in Concurrent Smalltalk (CST) [5], a concurrent programming language based on Smalltalk-80 [14]. A description of the programming language is beyond the scope of this paper.

The computation graph for this program is shown in Figure 2. Each node of this graph represents a context object created to hold the state of one activation of the `rangeProduct:` method. A message from the parent node in the tree creates the context, sends two `rangeProduct` messages, and then suspends awaiting the replies. This initial activation executes 16 *working*[2] instructions for internal nodes.

[2] These instruction counts consider only useful code. Message reception and address translation

```
(Integer) factorial
  ^ 1 rangeProduct: self '

(Integer) rangeProduct: n
  | mid |
  self = n ifTrue: [^self].
  mid <- self + n // 2.
  ^ (self rangeProduct: mid) * (mid+1 rangeProduct: n) '
```

Figure 1: A Factorial program in Concurrent Smalltalk. This program executes an average of 8 assembly instructions in response to a message.

Leaf nodes execute only 6 instructions to send their argument back. The first reply message resumes the context, saves its argument, and suspends (4 instructions). The second reply performs the multiply and sends a reply up the tree (8 instructions[3]). The average number of working instructions executed in response to a message is ≈ 8.

Operating at a fine-grain, in addition to exploiting more concurrency, also simplifies communication. Using the natural partition of the program, communication is implicitly specified by interactions between the named objects in the program. Synchronization is also implicit. The arrival of a message containing the required data schedules execution. For example, in Figure 1 the expression (`self rangeProduct: mid`) causes a `rangeProduct:` message to be sent. The arrival of this message schedules the execution of the method.

In some concurrent programming systems, communication is made difficult by non-uniform naming: local objects are referenced differently than non-local objects. In the Cosmic Kernel [28], for example, local objects may be referenced through a pointer, while global objects require an explicit message send and receive. Providing a global address space allows objects to be referenced via a single mechanism (the virtual address) regardless of their location, and relieves the programmer of the bookkeeping required to keep track of node numbers. Programs become both easier to write and more portable.

The logical partition of most programs is fine-grained (8 word objects, 20 instruction methods). Unifying this logical partition with the physical partition of a program

overheads have been factored out.

[3] If the multiply exceeds the machine precision additional time is required to perform a `bignum` multiply.

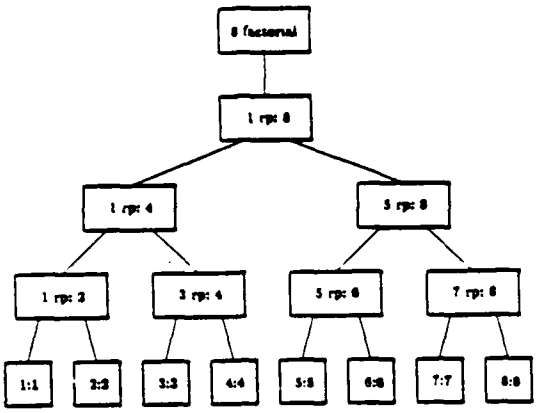


Figure 2: Computation graph for the factorial program.

simplifies programming and results in greater concurrency. To efficiently execute the resulting fine-grain program, a machine must have fast communication, low message reception overhead, and low scheduling overhead. To simplify communication between named objects, a machine must support a global address space. The mechanisms used by the J-Machine to meet these requirements are described in the next section. While we intend to use these mechanisms to execute object oriented [27] or actor [1] [2] programs. They are the same mechanisms that are required to support shared memory models of computation such as Multilisp [15].

3 Mechanisms

The following sequence of actions is involved in sending and receiving a message.

1. The originating node *translates* the ID (address or name) of the destination object into a destination node number.
2. The originating node *sends* the message.
3. The network *transmits* the message to the destination node.
4. The destination node *receives* the message.
5. The destination node *decides* whether to execute or buffer the message.
6. If the decision is to buffer, the node *buffers* the message.

7. Eventually, the message is executed and the node *translates* the receiver ID and message selector into a receiver address and a method address.
8. The method is executed.
9. The method suspends and transfers control to the next message.

On a 4K node J-Machine, this sequence of operations takes $\approx 5\mu s$[4]. The mechanisms implemented in the hardware of the system to accelerate this operation include.

1. A send instruction (2).
2. A fast communication mechanism, the NDF [10] (3).
3. A message unit that controls the reception and buffering of messages (4 and 6).
4. A scheduling mechanism that (5) decides when to preempt execution and (9) selects a message to be executed when a method suspends.
5. A general translation mechanism (1 and 7).

Send Instruction

The MDP injects messages into the network using a send instruction that transmits one or two words (at most one from memory) and optionally terminates the message. The first word of the message is interpreted by the network as an absolute node address (in x,y format) and is stripped off before delivery. The remainder of the message is transmitted without modification. A typical message send is shown in Figure 3. The first instruction sends the absolute address of the destination node (contained in R0). The second instruction sends two words of data (from R1 and R2). The final instruction sends two additional words of data, one from R3, and one from memory. The use of the SENDE instruction marks the end of the message and causes it to be transmitted into the network. In a Concurrent Smalltalk message, the first word is a message header, the second specifies the receiver, the third word is the selector, subsequent words contain arguments, and the final word is a continuation. On our register-transfer simulator, this sequence executes in 4 clock cycles.

Early in the design of the MDP we considered making a message send a single instruction that took a message template, filled in the template using the current addressing environment, and transmitted the message. Each template entry specified one word of the message as being either a constant, the contents of a data register, or a memory reference offset from an address register (like an operand

[4] This estimate assumes that all caches hit.

```
SEND    R0          ; send net address
SEND2   R1,R2       ; header and receiver
SEND2E  R3,[3,A3]   ; selector and continuation - end msg.
```

Figure 3: MDP assembly code to send a 4 word message uses three variants of the SEND instruction.

descriptor). The template approach was abandoned in favor of the simpler one or two operand SEND instruction because it did not significantly reduce code space or execution time. A two operand SEND instruction is nearly as dense as a template and can be implemented using the same control logic used for arithmetic and logical instructions.

Message Communications

Communication between nodes is performed by the NDFs on the nodes along the route. The NDF performs routing, buffering, and flow control to deliver messages in a 2-D mesh connected network. These functions are performed entirely within the NDF. No memory bandwidth or CPU time on intermediate nodes is used by message delivery. NDFs are connected by 9-bit communication channels that are expected to operate at 50MHz for a throughput of (450Mbits/sec). The propagation delay through this self-timed router is 20ns. Wormhole routing [7] [8] [19] is used to give an idle-network latency of 20D+80L ns, where D is the distance in channels and L is the message length in 36-bit words. In a 4096 node machine, for example, the average distance is 42 and the worst case distance is 126 for respective latencies of 1.3μs and 3μs respectively. In [7] it is shown that these latencies increase only slightly as network traffic is increased to within 25% of saturation.

Figure 4 shows how the network of NDFs delivers a message in the J-Machine. The message is injected into the network at node (1,5). The source NDF converts the absolute destination address (4,1) into a relative address (3,-4). The message is then routed in the positive X direction with the head *flit* (flow control digit) containing the relative X address decremented at each stage. Message delivery is pipelined with each flit occupying one stage of the pipeline. At node (3,5), the relative X address is decremented to zero. Upon seeing this zero, Node (4,5) strips the X address and begins routing the message in the negative Y direction.

At node (4,3) the head of the message blocks for two cycles (because the channel

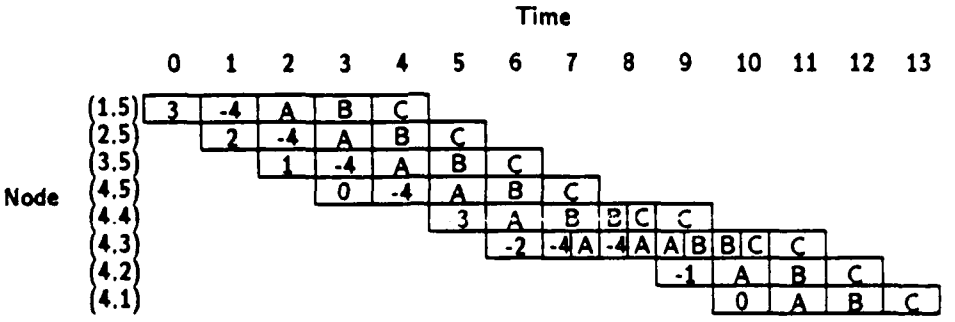


Figure 4: The NDFs deliver a message using wormhole routing. Buffering compresses the message when blockage occurs.

to (4,2) is in use). The tail of the message continues to advance compressing the message in the NDF buffers. After two network cycles[5] the blockage is removed and the entire message proceeds to the destination (4,1). The relative Y address (now 0) is stripped and the remainder of the message is delivered to the node.

A block diagram of the logic that performs this routing is shown in Figure 5. The NDF contains two priority levels that implement logically separate networks sharing the same set of physical wires. Figure 5 shows only one level. Each level consists of four direction data paths, one for each of the cardinal directions (+X, -X, +Y, -Y). A message from the processor (P channel) has its address converted from absolute to relative by subtracting the local node address. A two-way switch then selects the proper direction by examining the sign bit of the resulting relative X address. Each direction data path has two inputs (the direction input and the preceding dimension) and two outputs (the direction output and the next dimension). It arbitrates between the two inputs and performs a zero check on the head flit to select the appropriate output. If the direction output is selected, the head flit is incremented/decremented. The head flit is stripped if the dimension output is selected.

Partitioning the router into separate direction data paths significantly reduces both the area and delay as compared to previous designs based on a central crossbar switch [6]. In the most common case (a message continuing in the current direction) a flit sees only a single 2 way switch between the input and output. Additional

[5]The network is completely self-timed – there is no clock. The channel cycles are aligned here for purposes of illustration only.

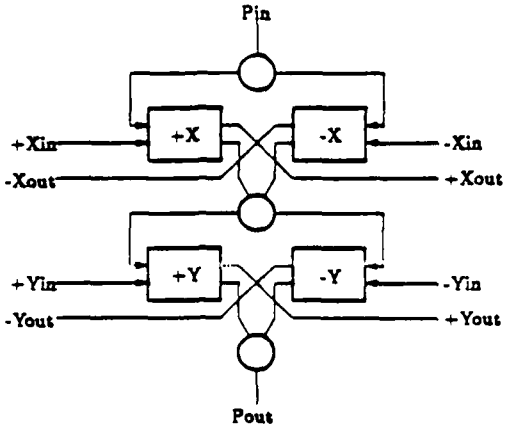


Figure 5: The NDF consists of separate dimension data paths that forward a message in its current direction or switch it to the next dimension.

switching is only required when a message switches dimensions.

Absolute integer node addressing was selected to simplify passing node numbers through the network – viz. with absolute addressing a node's address is the same on each node. Relative integer addressing is used internally by the NDF. Rather than converting from absolute addressing at the input to the network, we considered using relative addresses everywhere and adjusting them on-the-fly as they passed through the network. The incrementers and decrementers required to perform this adjustment are already in the NDF data paths. However, it proved cumbersome to detect which flits passing through the network contained network addresses in a particular dimension. We also considered representing addresses as polynomials over GF(2) and using a Galois incrementer [4] (the combinational equivalent of a linear feedback shift register) to adjust addresses. This approach had the advantage of requiring only a single gate delay to perform an increment/decrement however handling polynomial addresses proved difficult for some parts of the system software.

Message Reception

Message reception overhead is reduced to $\approx 1\mu s$ by buffering, scheduling, and dispatching messages in hardware. The MDP maintains two message/scheduling queues (corresponding to two priority levels) in its on-chip memory. As messages arrive over the network, they are buffered in the appropriate queue. The queues are implemented as circular buffers. It is important that the queue have sufficient

```
MOVE    [1,A3],R0   ; get method id
XLATE   R0,A0       ; translate to address descriptor
RES     2           ; transfer control to method
```

Figure 6: MDP assembly code for the CALL message.

performance to remove words from the network as they arrive. Otherwise, messages would backup into the network causing congestion. To achieve the required performance, special addressing hardware is used to enqueue or dequeue a message word with wraparound and full/empty check in a single clock cycle. A queue row buffer allows enqueuing to proceed using one memory cycle for each four words received. Thus a program can execute in parallel with message reception with little loss of memory bandwidth.

The message queues schedule the tasks associated with messages. At any point in time, the MDP is executing the task associated with the first message in the highest priority non-empty queue. If both queues are empty, the MDP is idle – viz., executing a background task. Sending a message implicitly schedules a task on the destination node. The task will be run when it reaches the head of the queue. This simple two-priority scheduling mechanism removes the overhead associated with a software scheduler. More sophisticated scheduling policies may be implemented on top of this substrate.

Messages become *active* either by arriving while the node is idle or executing at a lower priority, or by being at the head of a queue when the preceding message *suspends* execution. When a message becomes active, a handler is dispatched in one clock cycle. The dispatch forces execution to a physical address specified in the message header. This mechanism is used directly to process messages requiring low latency (e.g., combining and forwarding). Other messages (e.g., remote procedure call) specify a handler that locates the required method (using the translation mechanism described below) and then transfers control to it. For example, the call handler is shown in Figure 6. The first instruction gets the method ID (offset 1 into the message). Hardware initializes register A3 to contain an address descriptor (base/length) for the current message. The next instruction translates the method ID into an address descriptor for the method. If the translate faults, because method is not resident or the descriptor is not in the cache, the fault handler *fixes* the problem and reschedules the message. If the translation succeeds, the final instruction (resume) transfers control to the method.

An early version of the MDP had a fixed set of message handlers in microcode. An analysis of this code showed that it was limited by memory accesses and there

was little performance advantage in implementing it in microcode. The microcode was eliminated, the handlers were recoded in assembly language, and the *message opcode* was defined to be the physical address of the handler routine. Frequently used handlers are contained in an on-chip ROM. This approach simplifies the control structure of the machine and gives us flexibility to redefine message handlers to fix bugs, for instrumentation, and to implement new message types.

The message queue originally allocated storage from the heap for each incoming message. This eliminated the need to copy messages when a method suspended for intermediate results. However, the cost of allocating and reclaiming storage for each message proved to be prohibitive. Instead, we settled on the preallocated circular buffer. When a method suspends for intermediate results, message arguments are copied into a context object. The overhead of this copying is small since the context must be created anyway to specify a continuation and to hold live variables. The fixed buffer also provides a convenient layering. Priority zero messages are sent when the memory allocator runs out of room and priority one messages are sent when the priority zero queue fills.

Translation

The MDP is an experiment in unifying shared-memory and message-passing parallel computers. Shared-memory machines provide a uniform global name space (address space) that allows processing elements to access data regardless of its location. Message-passing machines perform communication and synchronization via node-to-node messages. These two concepts are not mutually exclusive. The MDP provides a virtual addressing mechanism intended to support a global name space while using an execution mechanism based on message passing.

The MDP implements virtual addressing using a very general translation mechanism. The MDP memory allows both indexed and set-associative access. By building comparators into the column multiplexer of the on-chip RAM, we are able to provide set-associative access with only a small increase in the size of the RAM's peripheral circuitry.

The translation mechanism is exposed to the programmer with the `ENTER` and `XLATE` instructions. These instructions make use of the set-associative mechanism of the MDP memory. `ENTER Ra,Rb` associates the contents of `Ra` (the key) with the contents of `Rb` (the data). The association is made on the full 36 bits of the key so that tags may be used to distinguish different keys. `XLATE Ra,Ab` looks up the data associated with the contents of `Ra` and stores this data in `Ab`. The instruction faults if the lookup *misses* or if the data is not an address descriptor. `XLATE Ra,Rb` can be used to lookup other types of data. This mechanism is used by our system code to cache ID to address descriptor (virtual to physical) translations, to cache ID to node number (virtual to physical) translations, and to cache class/selector to

```
MOVE    [1,A3],R0          ; Receiver ID
XLATE   R0,A2              ; Receiver descriptor
MOVE    [0,A2],R0          ; Receiver object header
AND     R0,CLASS_MASK,R0   ; Isolate class
OR      R0,[2,A3],R0       ; combine w/ selector
XLATE   R0,A0              ; get method
RES     2
```

Figure 7: Send handler translates class and selector into address descriptor for method and transfers control.

address descriptor (method lookup) translations.

Tags are an integral part of our addressing mechanism. An ID may translate into an address descriptor for a local object, or a node address for a global object. The tag allows us to distinguish these two cases and a fault provides an efficient mechanism for the test. Tags also allow us to distinguish an ID key from a class/selector key with the same bit pattern.

Most computers provide a set associative cache to accelerate translations. We have taken this mechanism and exposed it in a pair of instructions that a systems programmer can use for any translation. Providing this general mechanism gives us the freedom to experiment with different address translation mechanisms and different uses of translation. We pay very little for this flexibility since performance is limited by the number of memory accesses that must be performed.

4 Fine-Grain Programming

To illustrate how these mechanisms work together to execute a concurrent program recall the factorial example from Section 2. When the factorial message is received it is immediately buffered. When the message becomes active, control is dispatched to the `SEND` handler. Shown in Figure 7, this handler forms a key from the class of the receiver and the message selector and looks up the associated method using the XLATE instruction. After executing 7 instructions, control is transferred to the factorial method for class `Integer`.

MDP assembly code for the factorial method is shown in Figure 8. This code performs no computing. It reformats its message for the first rangeproduct and

```
MOVE    1,R1            ; receiver is 1
CALL    ID_TO_NODE      ; translate to node number in R0
SEND    R0              ; send address
DC      SEND_HEADER     ;
SEND2   R0,R1           ; send header and receiver
DC      RANGE_PRODUCT   ; send selector
SEND    R0              ;
SEND    [1,A3]          ; send argument
SENDE   [3,A3]          ; continuation
SUSPEND
```

Figure 8:

transmits it. Thus it is a good indicator of the overhead associated with sending a message; this 12 instruction sequence takes 18 clock cycles (estimated) to send a five word message. The total time from `factorial` message in to `rangeProduct:` message out is 30 clock cycles (1.5μs on a 20MHz MDP).

This code uses continuation passing to return its result. The factorial code does not create a context to await the result of the `rangeProduct:`. Instead, it passes the continuation (where to reply to) from its message in the `rangeProduct:` message. The `rangeProduct` method then returns to the original sender.

The `rangeProduct:` code (not shown) creates a context to await the results of its two message sends[6]. The ID of this context is included in the continuation field of each of these messages. After the messages are sent, a `SUSPEND` instruction is executed to pass control to the next message in the queue while awaiting the replies.

When the first reply arrives, it stores its value in the context and tests for the presence of the other reply value. Finding that value absent, execution is again suspended. The arrival of the second reply reactivates the context, stores its value, and (finding the other result present) performs the multiply and sends the result in a reply message to its continuation. The scheduling of operations during this combining is performed by dataflow. As soon as both operands are present, the operation is performed.

[6] A four-instruction inline sequence allocates this context off a free list.

5 Conclusion

The MDP/NDF processing node efficiently executes fine-grain concurrent programs by providing mechanisms that reduce the overhead of message-passing, translation, and context switching to $\approx 5\mu s$. Reducing overhead to a time comparable with the natural grain size of many concurrent programs allows the programmer to exploit all of the concurrency present in these programs rather than grouping many grains together – reducing the concurrency to improve the efficiency.

The MDP provides very general hardware mechanisms that can support many different concurrent programming models including conventional message-passing [28], actors [1] [2], futures [15], communicating processes [16], and dataflow [13]. All of these programming models require the same execution mechanisms: communication, synchronization, and translation. Specializing a machine for a particular model of computation results in only a small increase in performance.

At the time of this writing the NDF design is complete and a chip has been submitted for fabrication. Instruction and register transfer level simulations of the MDP have been written and used to test the architecture. Transistor level design, and artwork design for the MDP are underway.

There are many promising directions for future research. The mechanisms described here efficiently execute concurrency at a grain size of $5\mu s$. Many numerical programs, however, have potential concurrency at the level of single operations. Architectures must be developed that can exploit this concurrency without incurring the overhead of message delivery or synchronization.

Another critical problem is the development of (communication, processor, and memory) resource management policies for concurrent operating systems. It is quite easy to write a program with sufficient concurrency to swamp any concurrent machine. A concurrent operating system must provide a means to *throttle back* such massively concurrent applications to match the concurrency to the available resources.

Acknowledgement

The following MIT students have contributed to the work described here: Linda Chao, Andrew Chien, Stuart Fiske, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Michael Larivee, Paul Song, Brian Totty, and Scott Wills.

I thank Tom Knight, Gerry Sussman, Steve Ward, Dave Gifford, and Carl Hewitt of MIT, and Chuck Seitz and Bill Athas of Caltech for many valuable suggestions, comments, and advice.

References

[1] Agha, Gul A., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[2] Athas, W.C., and Seitz, C.L., *Cantor Language Report*, Technical Report 5232:TR:86, Dept. of Computer Science, California Institute of Technology, 1986.

[3] BBN Advanced Computers, Inc., *Butterfly Parallel Processor Overview*, BBN Report No. 6148, March 1986.

[4] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Addison-Wesley, 1983, pp. 65-90.

[5] Dally, William J., *A VLSI Architecture for Concurrent Data Structures*, Kluwer, Hingham, MA, 1987.

[6] Dally, William J. and Seitz, Charles L., "The Torus Routing Chip," *J. Distributed Systems*, Vol. 1, No. 3, 1986, pp. 187-196.

[7] Dally, William J. "Wire Efficient VLSI Multiprocessor Communication Networks," *Proceedings Stanford Conference on Advanced Research in VLSI*, March 1987, pp. 391-415.

[8] Dally, William J. and Seitz, Charles L., " Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, Vol. C-36, No. 5, May 1987, pp. 547-553.

[9] Dally, William J. et.al., "Architecture of a Message-Driven Processor," *Proceedings of the* 14^{th} *Symposium on Computer Architecture*, June 1987, pp. 189-196..

[10] Dally, William J., and Song, Paul., "Design of a Self-Timed VLSI Multicomputer Communication Controller," To appear in, *Proc. ICCD-87*, 1987.

[11] Dally, William J., "Concurrent Data Structures," Chapter 7 in [26].

[12] Dally, William J., "The J-Machine: A Concurrent VLSI Message-Passing Computer for Symbolic and Numeric Processing," to appear.

[13] Dennis, Jack B., "Data Flow Supercomputers," *IEEE Computer*, Vol. 13, No. 11, Nov. 1980, pp. 48-56.

[14] Goldberg, Adele, and Robson, David, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[15] Halstead, Robert H., "Parallel Symbolic Computation," *IEEE Computer*, Vol. 19, No. 8, Aug. 1986, pp. 35-43.

[16] Hoare, C.A.R., "Communicating Sequential Processes," *CACM*, Vol. 21, No. 8, August 1978, pp. 666-677.

[17] Inmos Limited, *IMS T424 Reference Manual*, Order No. 72 TRN 006 00, Bristol, United Kingdom, November 1984.

[18] Intel Scientific Computers, *iPSC User's Guide*, Order No. 175455-001, Santa Clara, CA, Aug. 1985.

[19] Kermani, Parviz and Kleinrock, Leonard, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, Vol 3., 1979, pp. 267-286.

[20] Lutz, C., et. al., "Design of the Mosaic Element," *Proc. MIT Conference on Advanced Research in VLSI*, Artech Books, 1984, pp. 1-10.

[21] Palmer, John F., "The NCUBE Family of Parallel Supercomputers," *Proc. IEEE International Conference on Computer Design, ICCD-86*, 1986, p. 107.

[22] Pfister, G.F. et. al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proceedings ICPP*, 1985, pp. 764–771.

[23] Seitz, Charles L., "System Timing" in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Addison-Wesley, 1980, Ch. 7.

[24] Seitz, Charles L., et al., *The Hypercube Communications Chip*, Display File 5182:DF:85, Dept. of Computer Science, California Institute of Technology, March 1985.

[25] Seitz, Charles L., "The Cosmic Cube", *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 22-33.

[26] Seitz, Charles L., Athas, William C., Dally, William J., Faucette, Reese, Martin, Alain J. , Mattisson, Sven, Steele, Craig S., and Su, Wen-King, *Message-Passing Concurrent Computers: Their Architecture and Programming*, Addison-Wesley, publication expected 1987.

[27] Stefik, Mark and Bobrow, Daniel G., "Object-Oriented Programming: Themes and Variations," *AI Magazine*, Vol. 6, No. 4, Winter 1986, pp. 40-62.

[28] Su, Wen-King, Faucette, Reese, and Seitz, Charles L., *C Programmer's Guide to the Cosmic Cube*, Technical Report 5203:TR:85, Dept. of Computer Science, California Institute of Technology, September 1985.

[29] Theriault D.G., *Issues in the Design and Implementation of Act2*, MIT Artificial Intelligence Laboratory, Technical Report 728, June 1983.

END

DATE

FILMED

8-88

DTIC