Active Messages: a Mechanism for Integrated Communication and Computation

Thorsten von Eicken
David E. Culler
Seth Copen Goldstein
Klaus Erik Schauser

{tve,culler,sethg,schauser}@cs.berkeley.edu
Computer Science Division — EECS
University of California, Berkeley, CA 94720

Abstract

The design challenge for large-scale multiprocessors is (1) to minimize communication overhead, (2) allow communication to overlap computation, and (3) coordinate the two without sacrificing processor cost/performance. We show that existing message passing multiprocessors have unnecessarily high communication costs. Research prototypes of message driven machines demonstrate low communication overhead, but poor processor cost/performance. We introduce a simple communication mechanism, *Active Messages*, show that it is intrinsic to both architectures, allows cost effective use of the hardware, and offers tremendous flexibility. Implementations on nCUBE/2 and CM-5 are described and evaluated using a split-phase shared-memory extension to C, *Split-C*. We further show that active messages are sufficient to implement the dynamically scheduled languages for which message driven machines were designed. With this mechanism, latency tolerance becomes a programming/compiling concern. Hardware support for active messages is desirable and we outline a range of enhancements to mainstream processors.

1 Introduction

With the lack of consensus on programming styles and usage patterns of large parallel machines, hardware designers have tended to optimize along specific dimensions rather than towards general balance. Commercial multiprocessors invariably focus on raw processor performance, with network performance in a secondary role, and the interplay of processor and network largely neglected. Research projects address specific issues, such as tolerating latency in dataflow architectures and reducing latency in cache-coherent architectures, accepting significant hardware complexity and modest processor performance in the prototype solutions. This paper draws on recent work in both arenas to demonstrate that the utility of exotic message-driven processors can be boiled down to a simple mechanism and that this mechanism can be implemented efficiently on conventional message passing machines. The basic idea is that the control information at the head of a message is the address of a user-level instruction sequence that will extract the message from the network and integrate it into the on-going computation. We call this *Active Messages*. Surprisingly, on commercial machines this mechanism is an order of magnitude more efficient than the message passing primitives that drove the original hardware designs. There is considerable room for improvement with direct hardware support, which can be addressed in an evolutionary manner. By smoothly integrating communication with computation, the overhead of communication is greatly reduced and an overlap of the two is easily achieved. In this paradigm, the hardware designer can meaningfully address what balance is required between processor and network performance.

1.1 Algorithmic communication model

The most common cost model used in algorithm design for large-scale multiprocessors assumes the program alternates between computation and communication phases and that communication requires time linear in the size of the message, plus a start-up cost[9]. Thus, the time to run a program is $T = T_{compute} + T_{communicate}$ and $T_{communicate} = N_c(T_s + L_cT_b)$, where T_s is the start-up cost, T_b is the time per byte, L_c is the message length, and N_c is the number of communications. To achieve 90% of the peak processor performance, the programmer must tailor the algorithm to achieve a sufficiently high ratio of computation to communication that $T_{compute} \geq 9T_{communicate}$. A high-performance network is required to minimize the communication time, and it sits 90% idle!

If communication and computation are overlapped the situation is very different. The time to run a program becomes $T = \max(T_{compute} + N_cT_s,\ N_cL_cT_b)$. Thus, to achieve high processor efficiency, the communication and compute times need only balance, and the compute time need only swamp the communication overhead, *i.e.*, $T_{compute} \gg N_cT_s$. By examining the average time between communication phases ($T_{compute}/N_c$) and the time for message transmission, one can easily compute the per-processor bandwidth through the network required to sustain a given level of processor utilization. The hardware can be designed to reflect this balance. The essential properties of the communication mechanism are that the start-up cost must be low and that it must facilitate the overlap and co-ordination of communication with on-going computation.

1.2 Active Messages

Active Messages is an asynchronous communication mechanism intended to expose the full hardware flexibility and performance of modern interconnection networks. The underlying idea is simple: each message contains at its head the address of a user-level handler which is executed on message arrival with the message body as argument. The role of the handler is to get the message out of the network and into the computation ongoing on the processing node. The handler must execute quickly and to completion. As discussed below, this corresponds closely to the hardware capabilities in most message passing multiprocessors where a privileged interrupt handler is executed on message arrival, and represents a useful restriction on message driven processors.

Under Active Messages the network is viewed as a pipeline operating at a rate determined by the communication overhead and with a latency related to the message length and the network depth. The sender launches the message into the network and continues computing; the receiver is notified or interrupted on message arrival and runs the handler. To keep the pipeline full, multiple communication operations can be initiated from a node, and computation proceeds while the messages travel through the network. To keep the communication overhead to a minimum, Active Messages are not buffered except as required for network transport. Much like a traditional pipeline, the sender blocks until the message can be injected into the network and the handler executes immediately on arrival.

Tolerating communication latency has been raised as a fundamental architectural issue[1]; this is not quite correct. The real architectural issue is to provide the ability to overlap communication and computation, which, in-turn, requires low-overhead asynchronous communication. Tolerating latency then becomes a programming problem: a communication must be initiated sufficiently in advance of the use of its result. In Sections 2 and 3 we show two programming models where the programmer and compiler, respectively, have control over communication pipelining.

Active Messages is not a new parallel programming paradigm on par with send/receive or shared-memory: it is a more primitive communication *mechanism* which can be used to implement these paradigms (among others) simply and efficiently. Concentrating hardware design efforts on implementing fast Active Messages is more versatile than supporting a single paradigm with special hardware.

1.3 Contents

In this paper, we concentrate on message-based multiprocessors and consider machines of similar base technology representing the architectural extremes of processor/network integration. Message passing machines, including the nCUBE/2, iPSC/2, iPSC/860 and others, treat the network essentially as a fast I/O device. Message driven architectures, including Monsoon[17, 16] and the J-Machine[5], integrate the network deeply into the processor. Message reception is part of the basic instruction scheduling mechanism and message send is supported directly in the execution unit.

Section 2 examines current message passing machines in detail. We show that send/receive programming models make inefficient use of the underlying hardware capabilities. The raw hardware supports a simple form of Active Messages. The utility of this form of communication is demonstrated in terms of a fast, yet powerful asynchronous communication paradigm. Section 3 examines current message driven architectures. We show that the power of message driven processing, beyond that of Active Messages, is costly to implement and not required to support the implicitly parallel programming languages for which these architectures were designed. Section 4 surveys the range of hardware support that could be devoted to accelerating Active Messages.

2 Message passing Architectures

In this section we examine message passing machines, the one architecture that has been constructed and used on a scale of a thousand high-performance processors. We use the nCUBE/2 and the CM-5 as primary examples.

The nCUBE/2 has up to a few thousand nodes interconnected in a binary hypercube network. Each node consists of a CPU-chip and DRAM chips on a small double-sided printed-circuit board. The CPU chip contains a 64-bit integer unit, an IEEE floating-point unit, a DRAM memory interface, a network interface with 28 DMA channels, and routers to support cut-through routing across a 13-dimensional hypercube. The processor runs at 20 Mhz and delivers roughly 5 MIPS or 1.5 MFLOPS.

The CM-5 has has up to a few thousand nodes interconnected in a "hypertree" (an incomplete fat tree). Each node consists of a 33 Mhz Sparc RISC processor chip-set (including FPU, MMU and cache), local DRAM memory and a network interface to the hypertree and broadcast/scan/prefix control networks. In the future, each node will be augmented with four vector units.

We first evaluate the machines using the traditional programming models. Then we show that Active Messages are well-suited to the machines and support more powerful programming models with less overhead.

2.1 Traditional programming models

In the traditional programming model for message passing architectures, processes communicate by matching a *send* request on one processor with a *receive* request on another. In the synchronous, or crystalline[9] form, send and receive are blocking — the send blocks until the corresponding receive is executed and only then is data transferred. The main advantage of the blocking send/receive model is its simplicity. Since data is only transferred after both its source and destination addresses are known, no buffering is required at the source or destination processors.

Blocking send/receive communication exacerbates the effects of network latency on communication latency[1]: in order to match a send with a receive a 3-phase protocol, shown in Figure 1, is required: the sender first transmits a request to the receiver which returns an acknowledgement upon executing a matching receive operation and only then is data transferred. With blocking send/receive, it is impossible to overlap communication with computation and thus the network bandwidth cannot be fully utilized.

To avoid the three-phase protocol and to allow overlap of communication and computation, most message passing implementations offer non-blocking operation: *send* appears instantaneous to the user program. The message layer buffers the message until the network port is available, then the message is transmitted to the recipient, where it is again buffered until a matching *receive* is executed. As shown in the ring communication example in Figure 2, data can be exchanged while computing by executing all sends before the computation phase and all receives afterwards.

[1]We call communication latency the time from initiating a send in the user program on one processor to receiving the message in the user program on another processor, *i.e.*, the sum of software overhead, network interface overhead and network latency.

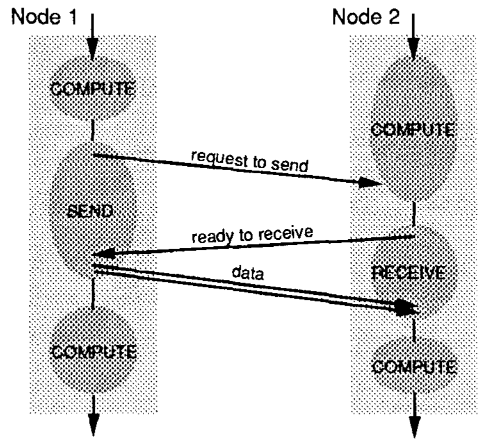


Figure 1: *Three-phase protocol for synchronous send and receive. Note that the communication latency is at best three network trips and that both send and receive block for at least one network round-trip each.*

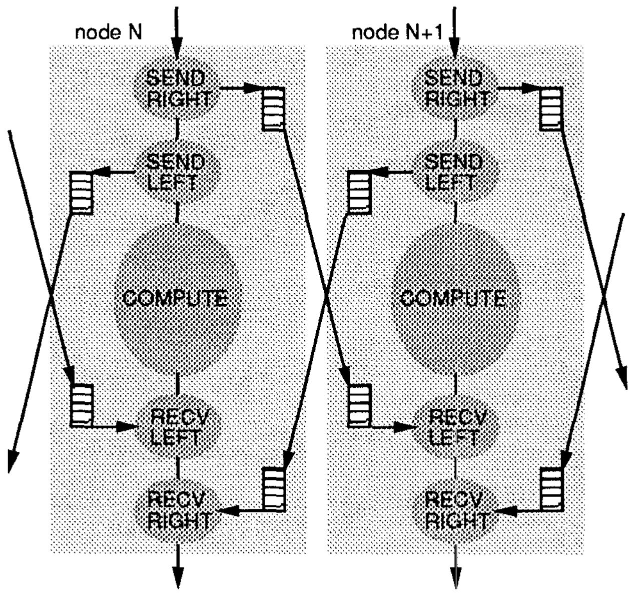


Figure 2: *Communication steps required for neighboring processors in a ring to exchange data using asynchronous send and receive. Data can be exchanged while computing by executing all sends before the computation phase and all receives afterwards. Note that buffer space for the entire volume of communication must be allocated for the duration of the computation phase!*

Table 1 shows the performance of send/receive on several current machines. The start-up costs are on the order of a thousand instruction times. This is due primarily to buffer management. The CM-5 is blocking and uses a three-phase protocol. The iPSC long messages use a three-phase protocol to ensure that enough buffer space is available at the receiving processor. However, the start-up costs alone prevent overlap of communication and computation, except for very large messages. For example, on the nCUBE/2 by the time a second send is executed up to 130 bytes of the first message will have reached the destination. Although the network bandwidth on all these machines is limited, it is difficult to utilize it fully, since this requires multiple simultaneous messages per processor.

Machine	T_s [μs/mesg]	T_b [μs/byte]	T_{fp} [μs/flop]
iPSC[8]	4100	2.8	25
nCUBE/10[8]	400	2.6	8.3
iPSC/2[8]	700	0.36	3.4
	390†	0.2	
nCUBE/2	160	0.45	0.50
iPSC/860[12]	160	0.36	0.033[7]
	60†	0.5	
CM-5‡	86	0.12	0.33[7]

†: messages up to 100 bytes
‡: blocking send/receive

Table 1: *Asynchronous send and receive overheads in existing message passing machines. T_s is the message start-up cost (as described in Section 1.1), T_b is the per-byte cost and T_{fp} is the average cost of a floating-point operation as reference point.*

2.2 Active Messages

Although the hardware costs of message passing machines are reasonable, the effectiveness of the machine is low under traditional send/receive models due to poor overlap of communication and computation, and due to high communication overhead. Neither of these shortcomings can be attributed to the base hardware: for example, initiating a transmission on the nCUBE/2 takes only two instructions, namely to set-up the DMA[2]. The discrepancy between the raw hardware message initiation cost and the observed cost can be explained by a mismatch between the programming model and the hardware functionality. Send and receive is not native to the hardware: the hardware allows one processor to send a message to another one and cause an interrupt to occur at arrival. In other words the hardware model is really one of launching messages into the network and causing a handler to be executed asynchronously upon arrival. The only similarity between the hardware operation and the programming model is in respect to memory address spaces: the source address is determined by the sender while the destination address is determined by the receiver[3].

Active Messages simply generalize the hardware functionality by allowing the sender to specify the address of the handler to be invoked on message arrival. Note that this relies on a uniform code image on all nodes, as is commonly used (the SPMD programming model). The handler is specified by a user-level address and thus traditional protection models apply. Active Messages differ from general remote procedure call (RPC) mechanisms in that the role of the Active Message handler is not to perform computation on the data, but to extract the data from the network and integrate it into the ongoing computation with a small amount of work. Thus, concurrent communication and computation is fundamental

[2]On the nCUBE/2, each of the 13 hypercube channels has independent input and output DMAs with a base-address and a count register each. Sending or receiving a message requires loading the address and the count.

[3]Shared-memory multiprocessor advocates argue that this is the major cause of programming difficulty of these machines.

to the message layer. Active Messages are not buffered, except as required for network transport. Only primitive scheduling is provided: the handlers interrupt the computation immediately upon message arrival and execute to completion.

The key optimization in Active Messages compared to send/receive is the elimination of buffering. Eliminating buffering on the receiving end is possible because either storage for arriving data is pre-allocated in the user program or the message holds a simple request to which the handler can immediately reply. Buffering on the sending side is required for the large messages typical in high-overhead communication models. The low overhead of Active Message makes small messages more attractive, which eases program development and reduces network congestion. For small messages, the buffering in the network itself is typically sufficient.

Deadlock avoidance is a rather tricky issue in the design of Active Messages. Modern network designs are typically deadlock-free provided that nodes continuously accept incoming messages. This translates into the requirement that message handlers are not allowed to block, in particular a reply (from within a handler) must not busy-wait if the outgoing channel is backed-up.

2.2.1 Active Messages on the nCUBE/2

The simplicity of Active Messages and its closeness to hardware functionality translate into fast execution. On the nCUBE/2 it is possible to send a message containing one word of data in 21 instructions taking $11\mu s$. Receiving such a message requires 34 instructions taking $15\mu s$, which includes taking an interrupt on message arrival and dispatching it to user-level. This near order of magnitude reduction ($T_c = 30\mu s$, $T_b = 0.45\mu s$) in send overhead is greater than that achieved by a hardware generation. Table 2 breaks the instruction counts down into the various tasks performed.

Task	Instruction count send	receive
Compose/consume message	6	9
Trap to kernel	2	–
Protection	3	–
Buffer management	3	3
Address translation	1	1
Hardware set-up	6	2
Scheduling	–	7
Crawl-out to user-level	–	12
Total	21	34

Table 2: *Breakdown into tasks of the instructions required to send and receive a message with one word of data on the nCUBE/2. "Message composition" and "consumption" include overhead for a function call and register saves in the handler. "Protection" checks the destination node and limits message length. "Hardware set-up" includes output channel dispatch and channel ready check. " Scheduling" accounts for ensuring handler atomicity and dispatch. "Crawling out to user-level" requires setting up a stack frame and saving state to simulate a return-from-interrupt at user-level.*

The Active Message implementation reduces buffer management to the minimum required for actual data transport. On the nCUBE/2 where DMA is associated with each network channel, one memory buffer per channel is required. Additionally, it is convenient to associate two buffers with the user process: one to compose the next outgoing message and one for handlers to consume the arrived message and compose eventual replies. This set-up reduces buffer management to swapping pointers for a channel buffer with a user buffer. Additional buffers must be used in exceptional cases to prevent deadlock: if a reply from within a handler blocks for "too long", it must be buffered and retried later so that further incoming messages can be dispatched. This reply buffering is not performed by the message layer itself, rather REPLY returns an error code and the user code must perform the buffering and retry. Typically the reply (or the original request) is saved onto the stack and the handlers for the incoming messages are nested within the current handler.

The breakdown of the 55 instructions in Table 2 shows the sources of communication costs on the nCUBE/2. A large fraction of instructions (22%) are used to simulate user-level interrupt handling. Hardware set-up (15%) is substantial due to output channel selection and channel-ready checks. Even the minimal scheduling and buffer management of Active Messages is still significant (13%). Note however, that the instruction counts on the nCUBE/2 are slightly misleading, in that the system call/return instructions and the DMA instructions are far more expensive than average.

The instruction breakdown shows clearly that Active Messages are very close to the absolute minimal message layer: only the crawl-out is Active Message specific and could potentially be replaced. Another observation is that most of the tasks performed here in software could be done easily in hardware. Hardware support for active messages could significantly reduce the overhead with a small investment in chip complexity.

2.2.2 Active Messages on the CM-5

The Active Messages implementation on the CM-5 differs from the nCUBE/2 implementation for five reasons[4]:

1. The CM-5 provides user-level access to the network interface and the node kernel time-shares the network correctly among multiple user processes.
2. The network interface only supports transfer of packets of up to 24 bytes (including 4 bytes for the destination node) and the network routing does not guarantee any packet ordering.
3. The CM-5 has two identical, disjoint networks. The deadlock issues described above are simply solved by using one network for requests and the other for replies. One-way communication can use either.
4. The network interface does not have DMA. Instead, it contains two memory-mapped FIFOs per network, one for outgoing messages and one for incoming ones. Status bits indicate whether incoming FIFOs hold messages and whether the previous outgoing message has been successfully sent by the network interface. The network interface discards outgoing messages if the network is backed-up or if the process is time-sliced during message composition. In these cases the send has to be retried.
5. The network interface generally does not use interrupts in the current version due to their prohibitive cost. (The hardware and the kernel do support interrupts, but their usefulness is limited due to the cost.) For comparison, on the nCUBE/2 the interrupt costs the same as the system call which would have to be used instead since there is no user-level access to the network interface.

[4]The actual network interface is somewhat more complicated than described below, we only present the aspects relevant to this discussion.

Sending a packet-sized Active Message amounts to stuffing the outgoing FIFO with a message having a function pointer at its head. Receiving such an Active Message requires polling, followed by loading the packet data into argument registers, and calling the handler function. Since the network interface status has to be checked whenever a message is sent (to check the send-ok status bit), servicing incoming messages at send time costs only two extra cycles. Experience indicates that the program does not need to poll explicitly unless it enters a long computation-only loop.

Sending multi-packet messages is complicated by the potential reordering of packets in the network. For large messages, set-up is required on the receiving end. This involves a two-phase protocol for GET, and a three-phase protocol for PUT (discussed below). Intermediate-sized messages use a protocol where each packet holds enough header information (at the expense of the payload) that the arrival order is irrelevant.

The performance of Active Message on the CM-5 is very encouraging: sending a single-packet Active Message (function address and 16 bytes of arguments) takes $1.6\mu s$ ($\approx$ 50 cycles) and the receiver dispatch costs $1.7\mu s$. The largest fraction of time is spent accessing the network interface across the memory bus. A prototype implementation of blocking send/receive on top of Active Messages compares favorably with the (not yet fully optimized) vendor's library: the start-up cost is $T_c = 23\mu s$ (vs. $86\mu s$) and the per byte cost is $T_b = 0.12\mu s$ (identical). Note that due to the three-phase protocol required by send/receive, T_c is an order of magnitude larger than the single packet send cost. Using different programming models such as Split-C, the cost off communication can be brought down to the Active Message packet cost.

2.3 Split-C: an experimental programming model using Active Messages

To demonstrate the utility of Active Messages, we have developed a simple programming model that provides split-phase remote memory operations in the C programming language. The two split-phase operations provided are PUT and GET: as shown in Figure 3a, PUT copies a local memory block into a remote memory at an address specified by the sender. GET retrieves a block of remote memory (address specified by sender) and makes a local copy. Both operations are non-blocking and do not require explicit coordination with the remote processor (the handler is executed asynchronously). The most common versions of PUT and GET increment a separately specified flag on the processor that receives the data. This allows simple synchronization through checking the flag or busy-waiting. Operating on blocks of memory can yield large messages which are critical to performance on current hardware as seen below.

The implementations of PUT and GET consist of two parts each: a message formatter and a message handler. Figure 3b shows the message formats. PUT messages contain the instruction address of the PUT handler, the destination address, the data length, the completion-flag address, and the data itself. The PUT handler simply reads the address and length, copies the data and increments the flag. GET requests contain the information necessary for the GET handler to reply with the appropriate PUT message. Note that it is possible to provide versions of PUT and GET that copy data blocks with a stride or any other form of gather/scatter[5].

To demonstrate the simplicity and performance of Split-C, Figure 4 shows a matrix multiply example that achieves 95% of peak

[5]Split-C exposes the underlying RPC mechanism the programmer as well, so that specialized communication structures can be constructed, e.g., enqueue record.

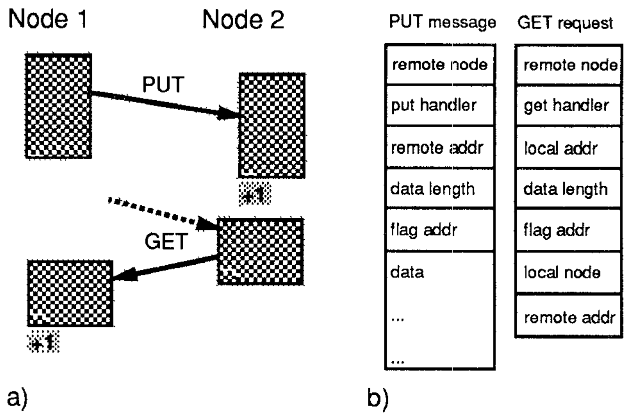


Figure 3: *Split-C* PUT *and* GET *perform split-phase copies of memory blocks to/from remote nodes. Also shown are the message formats.*

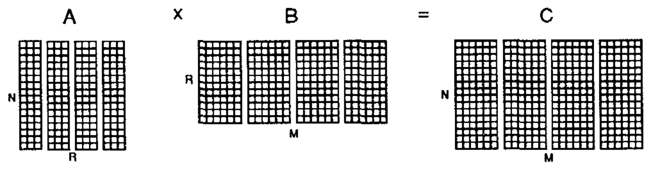
performance on large nCUBE/2 configurations. In the example, the matrices are partitioned in blocks of columns across the processors. For the multiplication of $C = A \times B$ each processor GETs one column of A after another and performs a rank-1 update (DAXPY) with the corresponding elements of its own columns of B into its columns of C. To balance the communication pattern, each processor first computes with its own column(s) of A and then proceeds by getting the columns of the next processor. Note that this algorithm is independent of the network topology and has a familiar shared-memory style. The remote memory access and its completion are made explicit, however.

The key to obtaining high performance is to overlap communication and computation. This is achieved by GETting the column for the next iteration while computing with the current column. It is now necessary to balance the latency of the GET with the time taken by the computation in the inner loops. Quantifying the computational cost is relatively easy: for each GET the number of multiply-adds executed is Nm (where m is the number of local columns of B and C) and each multiply-add takes $1.13\mu s$. To help understand the latency of the GET, Figure 6 shows a diagram of all operations and delays involved in the unloaded case.

The two top curves in Figure 5 show the performance predicted by the model and measured on a 128 node nCUBE/2, respectively, as the number of columns per processor of A is varied from 1 to 32. N is kept constant ($N = 128$) and R is adjusted to keep the total number of arithmetic operations constant ($R = 262144/M$). The matrix multiply in the example is computation bound if each processor holds more than two columns of A (*i.e.*, $m > 2$). The two bottom curves show the predicted and measured network utilization. The discrepancy between the model and the measurement is due to the fact that network contention is not modeled. Note that while computational performance is low for small values of m, the joint processor and network utilization is relatively constant across the entire range. As the program changes from a communication to a computation problem the "overall performance" is stable.

2.4 Observations

Existing message passing machines have been criticized for their high communication overhead and the inability to support global



The matrices are partitioned in blocks of columns across the processors. For the multiplication of $C_{N\times M} = A_{N\times R} \times B_{R\times M}$ each processor GETs one column of A after another and performs a rank-1 update (DAXPY) with its own columns of B into its columns of C. To balance the communication pattern, each processor first computes with its own column(s) of A and then proceeds by getting the columns of the next processor. This network topology independent algorithm achieves 95% of peak performance on large nCUBE/2 configurations.

```
int N, R, M;  /* matrix dimensions */
double A[R/P][N], B[M/P][R], C[M/P][N];
int i, j, k;  /* indices */
int j0, nj;   /* initial j, next j */
int dj;       /* delta j (j=j0+dj) */
int P, p;     /* num of procs, my proc */
int Rp = R/P;
double V0[N], V1[N]; /* remote col bufs */
double *V=V0;  /* current column */
double *nV=V1; /* next column */
double *tV;    /* temp column */
static int flag = 0; /* sync. flag */
extern void get(int proc, void *src, int size,
                void *dst, int &flag);

j0 = p * Rp;          /* starting column */
get(p, &A[0][0], N*sizeof(double),
    nV, &flag);       /* get first col of A */
/* loop over all columns of A */
for(dj=0; dj<R; dj++) {
  j = (j0+dj)%R;      /* this column index */
  nj = (j0+dj+1)%R; /* next column index */
  /* wait for previous get to complete */
  while(!check(1, &flag)) ;
  tV=V; V=nV; nV=tV;/* swap curr&next col */
  /* if not done, get next column */
  if(nj != j0) get(nj/Rp, &A[nj%Rp][0],
         N*sizeof(double), nV, &flag);
  /* accum. V into every column with scale */
  for(k=0; k<M/P; k++)
        for(i=0; i<N; ++) /* unroll! */
            C[i][k] = C[i][k] + V[i]*B[j][k];
}
```

Figure 4: Matrix multiply example in Split-C.

memory access. With Active Messages we have shown that the hardware is capable of delivering close to an order of magnitude improvement today if the right communication mechanism is used, and that a global address space may well be implemented in software. Split-C is an example of how Active Messages can be incorporated into a coarse-grain SPMD (single-program multiple-data) programming language. It generalizes shared memory read/write by providing access to blocks of memory including simple synchronization. It does not, however, address naming issues.

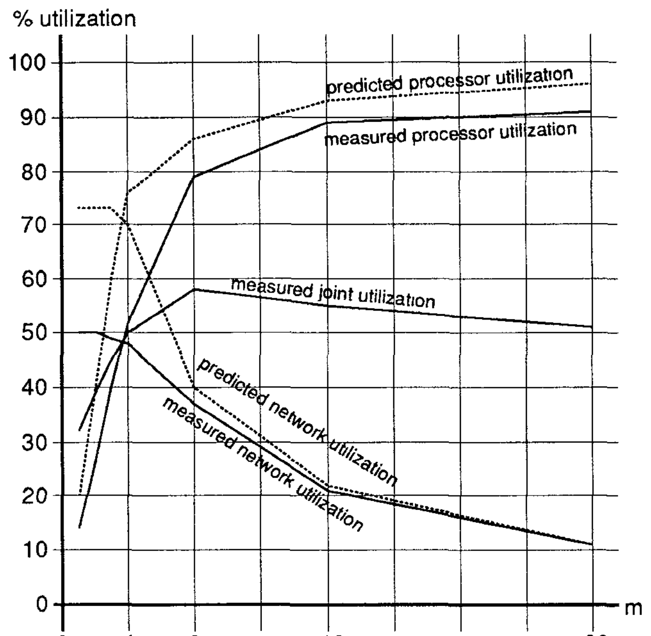


Figure 5: *Performance of Split-C matrix multiply on 128 processors compared to predicted performance using the model shown in Figure 6.*

Using Active Messages to guide the design, it is possible to improve current message passing machines in an evolutionary, rather than revolutionary, fashion. In the next section, we examine research efforts to build hardware which uses a different approach to provide another magnitude of performance improvement.

3 Message driven architectures

Message driven architectures such as the J-Machine and Monsoon expend a significant amount of hardware to integrate communication into the processor. Although the communication performance achieved by both machines is impressive, the processing performance is not. At first glance this seems to come from the fact that the processor design is intimately affected by the network design and that the prototypes in existence could not utilize traditional processor design know-how. In truth, however, the problem is deeper: in message driven processors a context lasts only for the duration of a message handler. This lack of locality prevents the processor from using large register sets. In this section, we argue that the hardware support for communication is partly counter-productive. Simpler, more traditional, processors can be built without unduly compromising either the communication or the processing performance.

3.1 Intended programming model

The main driving force behind message driven architectures is to support languages with dynamic parallelism, such as Id90[14],

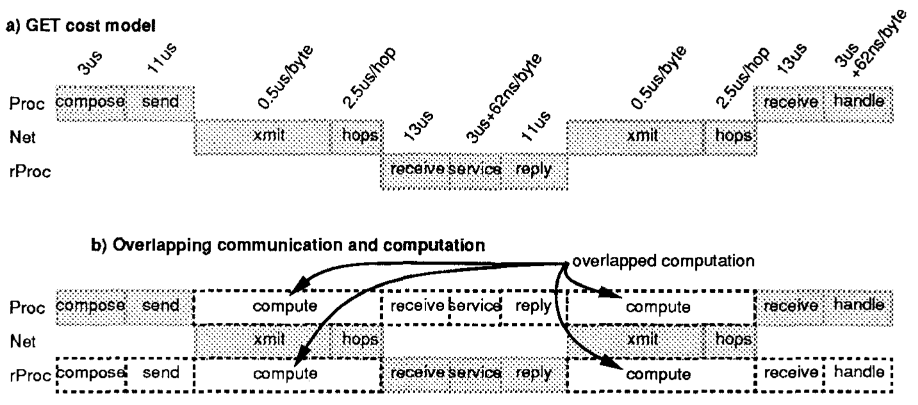


Figure 6: *Performance model for* GET. Compose *accounts for the time to set-up the request.* Xmit *is the time to inject the message into the network and* hops *is the time taken for the network hops.* Service *includes for copying the data into the reply buffer and* handle *for the time to copy the data into the destination memory block.*

Multilisp[10], and CST[11]. Computation is driven by messages, which contain the name of a handler and some data. On message arrival, storage for the message is allocated in a scheduling queue. When the message reaches the head of the queue, the handler is executed with the data as arguments. The handler may perform arbitrary computation, in particular it may synchronize and suspend. This ability to suspend requires general allocation and scheduling on message arrival and is the key difference with respect to Active Messages.

In the case of the J-Machine, the programming model is put forward in object-oriented language terms[6]: the handler is a method, the data holds the arguments for the method and usually one of them names the object the method is to operate on. In a functional language view, the message is a closure with a code pointer and all arguments of the closure. Monsoon is usually described from the dataflow perspective[17] and messages carry tokens formed of an instruction pointer, a frame pointer and one piece of data. The data value is one of the operands of the specified instruction, the other is referenced relative to the frame pointer.

The fundamental difference between the message driven model and Active Messages is where computation-proper is performed: in the former, computation occurs in the message handlers whereas in the latter it is in the "background" and handlers only remove messages from the network transport buffers and integrate them into the computation. This difference significantly affects the nature of allocation and scheduling performed at message arrival.

Because a handler in the message driven model may suspend waiting for an event, the lifetime of the storage allocated in the scheduling queue for messages varies considerably. In general, it cannot be released in simple FIFO or LIFO order. Moreover, the size of the scheduling queue does not depend on the rate at which messages arrive or handlers are executed, but on the amount of excess parallelism in the program[4]. Given that the excess parallelism can grow arbitrarily (as can the conventional call stack) it is impractical to set aside a fraction of memory for the message queue, rather it must be able to grow to the size of available memory.

Active Message handlers, on the other hand, execute immediately upon message arrival, cannot suspend, and have the responsibility to terminate quickly enough not to back-up the network. The role of a handler is to get the message out of the network transport buffers. This happens either by integrating the message into the data structures of the ongoing computation or, in the case of remote service requests, by immediately replying to the requester. Memory allocation upon message arrival occurs only as far as is required for network transport (*e.g.* if DMA is involved) and scheduling is restricted to interruption of the ongoing computation by handlers. Equivalently, the handlers could run in parallel with the computation on separate dedicated hardware.

3.2 Hardware Description

The Monsoon and J-Machine hardware is designed to support the message driven model directly. The J-Machine has a 3-D mesh of processing nodes with a single-chip CPU and DRAM each. The CPU has a 32-bit integer unit with a closely integrated network unit, a small static memory and a DRAM interface (but no floating-point unit). The hardware manages the scheduling queue as a fixed-size ring buffer in on-chip memory. Arriving messages are transferred into the queue and serviced in FIFO order. The first word of each message is interpreted as an instruction pointer and the message is made available to the handler as one of the addressable data segments. The J-Machine supports two levels of message priorities in hardware and two independent queues are maintained. Each message handler terminates by executing a SUSPEND instruction that causes the next message to be scheduled.

In Monsoon, messages arrive into the token queue. The token queue is kept in a separate memory proportional in size to the frame store. It provides storage for roughly 16 tokens per frame on average[6]. The queuing policy allows both FIFO and LIFO scheduling. The ALU pipeline is 8-way interleaved, so eight handlers can be active simultaneously. As soon as a handler terminates or suspends by blocking on a synchronization event, a token is popped from the queue and a new handler starts executing in the vacated pipeline interleave.

A common characteristic of both machines is that the amount of state available to an executing handler is very small: four data and three address registers in the J-Machine, an accumulator and three temporary registers in Monsoon. This reflects the fact that the computation initiated by a single message is small, typically less than ten arithmetic operations. This small amount of work cannot

[6]A token queue store of 64K tokens for 256K words of frame store and an expected average frame size of 64 words.

utilize many registers and since no locality is preserved from one handler to the next, no useful values could be carried along.

It is interesting to note that the J-Machine hardware does not actually support the message driven programming model fully in that the hardware message queue is managed in FIFO order and of fixed size. If a handler does not run to completion, its message must be copied to an allocated region of non-buffer memory by software. This happens for roughly 1/3 of all messages. The J-Machine hardware does support Active Messages, however, in which case the message queue serves only as buffering. Close to 1/3 of the messages hold a request to which the handler immediately replies and general allocation and scheduling is not required.

In Monsoon, the fact that tokens are popped from the queue means that the storage allocated for an arriving message is deallocated upon message handler execution. If a handler suspends, all relevant data is saved in pre-allocated storage in the activation frame thus, unlike the J-Machine, Monsoon does implement the message driven model, but at the cost of a large amount of high-speed memory.

3.3 TAM: compiling to Active Messages

So far, we have argued that the message driven execution model is tricky to implement correctly in hardware due to the fact that general memory allocation and scheduling are required upon message arrival. Using hardware that implements Active Messages, it is easy to simulate the message driven model by performing the allocation and scheduling in the message handler. Contrary to expectation this does not necessarily result in lower performance than a direct hardware implementation because software handlers can exploit and optimize special cases.

TAM[3] (Threaded Abstract Machine), a fine-grain parallel execution model based on Active Messages, goes one step further and requires the compiler to help manage memory allocation and scheduling. It is currently used as a compilation target for implicitly parallel languages such as Id90. When compiling for TAM, the compiler produces sequences of instructions, called *threads*, performing the computation proper. It also generates handlers, called *inlets*, for all messages to be received by the computation. Inlets are used to receive the arguments to a function, the results of called (child) functions, and the responses of global memory accesses. All accesses to global data structures are split-phase, allowing computation to proceed while requests travel through the network.

For each function call, an *activation frame* is allocated. When an inlet receives a message it typically stores the data in the frame and schedules a thread within the activation. Scheduling is handled efficiently by maintaining the scheduling queue within the activation frame: each frame, in addition to holding all local variables, contains counters used for synchronizing threads and inlets, and provides space for the *continuation vector* — the addresses of all currently enabled threads of the activation. Enabling a thread simply consists of pushing its instruction address into the continuation vector and possibly linking the frame into the ready queue. Figure 7 shows the activation tree data structure.

Service requests, such as remote reads, can typically be replied-to immediately and need no memory allocation or scheduling beyond what Active Messages provides. However, in exceptional cases requests must be delayed either for a lack of resources or because servicing inside the handler is inadequate. To amortize memory allocation, these requests are of fixed size and queue space is allocated in chunks.

Maintaining thread addresses in frames provides a natural two-level scheduling hierarchy. When a frame is scheduled (*activated*),

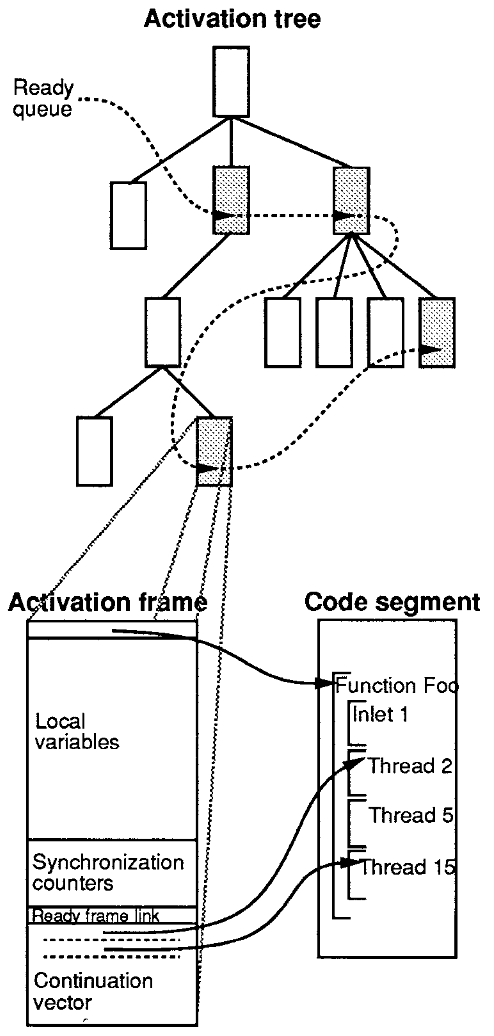


Figure 7: *TAM activation tree and embedded scheduling queue. For each function call, an activation frame is allocated. Each frame, in addition to holding all local variables, contains counters used to synchronize threads and inlets, and provides space for the* continuation vector — *the addresses of all currently enabled threads of the activation. On each processor, all frames holding enabled threads are linked into a ready queue. Maintaining the scheduling queue within the activation keeps costs low: enabling a thread simply consists of pushing its instruction address into the continuation vector and sometimes linking the frame into the ready queue. Scheduling the next thread within the same activation is simply a pop-jump.*

enabled threads are executed until the continuation vector is empty. When a message is received, two types of behavior can be observed: either the message is for the currently active frame and the inlet simply feeds the data into the computation, or the message is for a dormant frame in which case the frame may get added to the ready queue, but the ongoing computation is otherwise undisturbed.

Using the TAM scheduling hierarchy, the compiler can improve the locality of computation by synchronizing in message handlers and enabling computation only when a group of messages has arrived (one example is when all prerequisite remote fetches for an inner loop body have completed). This follows the realization that while the arrival of one message enables only a small amount of computation, the arrival of several closely related messages can enable a significant amount of computation. In cases beyond the power of compile-time analysis, the run-time scheduling policy dynamically enhances locality by servicing a frame until its continuation vector is empty.

As a result of the TAM compilation model, typically no memory allocation is required upon message arrival. Dynamic memory allocation is only performed in large chunks for activation frames and for global arrays and records. Locality of computation is enhanced by the TAM scheduling hierarchy. It is possible to implement TAM scheduling well even without any hardware support: on a uniprocessor[7] the overall cost for dynamic scheduling amounts to doubling the number of control-flow instructions relative to languages such as C. However, the overall performance depends critically on the cost of Active Messages. Table 3 summarizes the frequency of various kinds of messages in the current implementation. On average, a message is sent and received every eight TAM instructions (equivalent to roughly 20 RISC instructions). Note that these statistics are sensitive to optimizations. For example, significant changes can be expected from a software cache for remote arrays.

Message types	data words	frequency
Frame-frame	0	1%
	1	10%
	2	1%
Store request	1	8%
Fetch request	0	40%
Fetch reply	1	40%

Table 3: *Frequency of various message types and sizes (represented by the number of data values transmitted) in the current implementation of TAM. On average, a message is sent and received every 8 TAM instructions. These statistics are sensitive to compiler optimizations and, in some sense, represent a worst case scenario.*

4 Hardware support for Active Messages

Active messages provide a precise and simple communication mechanism which is independent of any programming model. Evaluating new hardware features can be restricted to evaluating their impact on Active Messages. The parameters feeding into the design are the size and frequency of messages, which depend on the expected workload and programming models.

Hardware support for active messages falls into two categories: improvements to network interfaces and modifications to the processor to facilitate execution of message handlers. The following subsections examine parts of the design space for each of these points of view.

4.1 Network interface design issues

Improvements in the network interface can significantly reduce the overhead of composing a message. Message reception benefits from these improvements as well, but also requires initiation of the handler.

[7] Id90 requires dynamic scheduling even on uniprocessors.

Large messages: The support needed for large messages is a superset of that for small messages. To overlap computation with large message communication, some form of DMA transfer must be used. To set-up the DMA on the receiving side, large messages must have a header which is received first. Thus, if small messages are well supported, a large message should be viewed as a small one with a DMA transfer tacked-on.

Message registers: Composing small messages in memory buffers is inefficient: much of the information present in a small message is related to the current processor state. It comes from the instruction stream, processor registers and sometimes from memory. At the receiving end, the message header is typically moved into processor registers to be used for dispatch and to address data. Direct communication between the processor and the network interface can save instructions and bus transfers. In addition, managing the memory buffers is expensive.

The J-Machine demonstrates an extreme alternative for message composition: in a single SEND instruction the contents of two processor registers can be appended to a message. Message reception, however, is tied to memory buffers (albeit on-chip). A less radical approach is to compose messages in registers of a network coprocessor.

Reception can be handled similarly: when received, a message appears in a set of registers. A (coprocessor) receive instruction enables reception of the next message. In case a coprocessor design is too complex, the network interface can also be accessed as a memory mapped device (as is the case in the CM-5).

Reuse of message data: Providing a large register set in the network interface, as opposed to network FIFO registers, allows a message to be composed using portions of other messages. For example, the destination for a reply is extracted from the request message. Also, multiple requests are often sent with mostly identical return addresses. Keeping additional context information such as the current frame pointer and a code base pointer in the network interface can further accelerate the formatting of requests.

Single network port: Multiple network channels connected to a node should not be visible to the message layer. On the nCUBE/2, for example, a message must be sent out on the correct hypercube link by the message layer, even though further routing in the network is automatic. The network interface should allow at least two messages to be composed simultaneously or message composition must be atomic. Otherwise, replies within message handlers may interfere with normal message composition.

Protection: User-level access to the network interface requires that protection mechanisms be enforced by the hardware. This typically includes checking the destination node, the destination process and, if applicable, the message length. For most of these checks a simple range check is sufficient. On reception, the message head (*i.e.*, the handler address and possibly a process id) can be checked using the normal memory management system.

Frequent message accelerators: A well-designed network interface allows the most frequent message types to be issued quickly. For example in the *T[15] proposal, issuing a global memory fetch takes a single store double instruction (the network interface is memory mapped). The 64-bit data value is interpreted as a global address and translated in the network interface into a node/local-address pair. For the return address the current frame pointer is cached in the network interface and the handler address is calculated from the low-order bits of the store address.

4.2 Processor support for message handlers

Asynchronous message handler initiation is the one design issue that cannot be addressed purely in the network interface: processor modifications are needed as well. The only way to signal an asynchronous event on current microprocessors is to take an interrupt. This not only flushes the pipeline, but enters the kernel. The overhead in executing a user-level handler includes a crawl-out to the handler, a trap back into the kernel, and finally the return to the interrupted computation[8]. Super-scalar designs tend to increase the cost of interrupts.

Fast polling: Frequent asynchronous events can be avoided by relying on software to poll for messages. In execution models such as TAM where the message frequency is very high, polling instructions can be inserted automatically by the compiler as part of thread generation. This can be supported with little or no change to the processor. For example, on Sparc or Mips a message-ready signal can be attached to the coprocessor condition code input and polled using a branch on coprocessor condition instruction.

User-level interrupts: User-level traps have been proposed to handle exceptions in dynamically typed programming languages[13] and floating-point computations. For Active Messages, user-level interrupts need only occur between instructions. However, an incoming message may not be for the currently running user process and the network interface should interrupt to the kernel in this case.

PC injection: A minimal form of multithreading can be used to switch between the main computational thread and a handler thread. The two threads share all processor resources except for the program counter (PC). Normally instructions are fetched using the computation PC. On message arrival, instruction fetch switches to use the handler PC. The handler suspends with a `swap` instruction, which switches instruction fetch back to the computation PC. In the implementation the two PCs are in fact symmetrical. Switching between the two PCs can be performed without pipeline bubbles, although fetching the `swap` instruction costs one cycle. Note, that in this approach the format of the message is partially known to the network interface, since it must extract the handler PC from the message.

Dual processors: Instead of multiplexing the processor between computation threads and handlers, the two can execute concurrently on two processors, one tailored for the computation and a very simple one for message handlers (*e.g.*, it may have no floating-point). The crucial design aspect is how communication is handled between the two processors. The communication consists of the data received from the network and written to memory, *e.g.*, into activation frames, and the scheduling queue.

A dual-processor design is proposed for the MIT *T project. It uses an MC88110 for computation and a custom message processor. In the *T design, the two processors are on separate die and communicate over a snooping bus. If the two processors were integrated on a single die, they could share the data cache and communication would be simpler. The appealing aspect of this design is that normal uniprocessors can be used quite successfully.

[8]It may be possible for the user-level handler to return directly to the computation.

For coarse-grain models, such as Split-C, it is most important to overlap computation with the transmission of messages into the network. An efficient network interface allows high processor utilization on smaller data sets. On the other extreme, implicitly parallel language models that provide word-at-a-time access to globally shared objects are extremely demanding of the network interface. With modest hardware support, the cost of handling a simple message can be reduce to a handful of instructions, but not to one. Unless remote references are infrequent, the amount of resources consumed by message handling is significant. Whether dual processors or a larger number of multiplexed processors is superior depends on a variety of engineering issues, but neither involves exotic architecture. The resources invested in message handling serve to maintain the efficiency of the background computation.

5 Related work

The work presented in this paper is similar in character to the recent development of optimized RPC mechanisms in the operating system research community[18, 2]. Both attempt to reduce the communication layer functionality to the minimum required and carefully analyze and optimize the frequent case. However, the time scales and the operating system involvement are radically different in the two arenas.

The RPC mechanisms in distributed systems operate on timescales of 100s of microseconds to milliseconds, and operating system involvement in every communication operation is taken for granted. The optimizations presented reduce the OS overhead for moving data between user and system spaces, marshaling complex RPC parameters, context switches and enforcing security. Furthermore, connecting applications with system services is a major use of operating system RPCs, so the communication partners must be protected from one another.

In contrast, the time scale of communication in parallel machines is measured in tens of processor clock cycles (a few μs) and the elimination of all OS intervention is a central issue. Security is less of a concern given that the communication partners form a single program.

Another difference is that in the distributed systems arena the communication paradigm (RPC) is stable, whereas we propose a new mechanism for parallel processing and show how it is more primitive than and subsumes existing mechanisms.

6 Conclusions

Integrated communication and computation at low cost is the key challenge in designing the basic building block for large-scale multiprocessors. Existing message passing machines devote most of their hardware resources to processing, little to communication and none to bringing the two together. As a result, a significant fraction of the processor is lost to the layers of operating system software required to support message transmission. Message driven machines devote most of their hardware resources to message transmission, reception and scheduling. The dynamic allocation required on message arrival precludes simpler network interfaces. The message-by-message scheduling inherent in the model results in short computation run-lengths, limiting the processing power that can be utilized.

The fundamental issues in designing a balanced machine are providing the ability to overlap communication and computation and to reduce communication overhead. The active message model presented in this paper minimizes the software overhead in message passing machines and utilizes the full capability of the hardware. This model captures the essential functionality of message driven machines with simpler hardware mechanisms.

Under the active message model each node has an ongoing computational task that is punctuated by asynchronous message arrival. A message handler is specified in each message and serves to extract the message data and integrate it into the computation. The efficiency of this model is due to elimination of buffering beyond network transport requirements, the simple scheduling of non-suspensive message handlers, and arbitrary overlap of computation and communication. By drawing the distinction between message handlers and the primary computation, large grains of computation can be enabled by the arrival of multiple messages.

Active messages are sufficient to support a wide range of programming models and permit a variety of implementation tradeoffs. The best implementation strategy for a particular programming model depends on the usage patterns typical in the model such as message frequency, message size and computation grain. Further research is required to characterize these patterns in emerging parallel languages and compilation paradigms. The optimal hardware support for active messages is an open question, but it is clear that it is a matter of engeneering tradeoffs rather than architectural revolution.

Acknowledgements

We thank our good friends at MIT for the work on Id90, Monsoon and the J-Machine, which stimulated this investigation. nCUBE/2 time was gratefully provided by Sandia National Laboratories and nCUBE Corp. nCUBE also provided access to Vertex kernel sources and a mini nCUBE/2 configuration to reboot at leisure. Guidance and encouragement from hard working folks at Thinking Machines Corp. was instrumental throughout the development of Active Messages on the CM-5.

This work was supported by National Science Foundation PYI Award (CCR-9058342) with matching funds from Motorola Inc. and the TRW Foundation. Thorsten von Eicken is supported by the Semiconductor Research Corporation Grant 91-DC-008. Seth Copen Goldstein is supported by Sandia National Laboratories Contract 87-0212. Klaus Erik Schauser is supported by an IBM Graduate Fellowship. CM-5 computational resources were provided by the NSF Infrastructure Grant CDA-8722788.

References

[1] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proc. of DFVLR - Conf. 1987 on Par. Proc. in Science and Eng.*, Bonn-Bad Godesberg, W. Germany, June 1987.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight Remote Procedure Call. *ACM Trans. on Computer Systems*, 8(1), February 1990.

[3] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991. (Also available as Technical Report UCB/CSD 91/591, CS Div., University of California at Berkeley).

[4] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proc. of the 15th Ann. Int. Symp. on Comp. Arch.*, pages 141–150, Hawaii, May 1988.

[5] W. Dally and et al. Architecture of a Message-Driven Processor. In *Proc. of the 14th Annual Int. Symp. on Comp. Arch.*, pages 189–196, June 1987.

[6] W. Dally and et al. The J-Machine: A Fine-Grain Concurrent Computer. In *IFIP Congress*, 1989.

[7] J. J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, Computer Science Dept., Univ. of Tennessee, Knoxville, TN 37996, December 1991.

[8] T. H. Dunigan. Performance of a Second Generation Hypercube. Technical Report ORNL/TM-10881, Oak Ridge Nat'l Lab, November 1988.

[9] G. Fox. *Programming Concurrent Processors*. Addison Wesley, 1989.

[10] R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[11] W. Horwat, A. A. Chien, and W. J. Dally. Experience with CST: Programming and Implementation. In *Proc. of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989.

[12] Intel. Personal communication, 1991.

[13] D. Johnson. Trap Architectures for Lisp Systems. In *Proc. of the 1990 ACM conf. on Lisp and Functional Programming*, June 1990.

[14] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Proc. Workshop on Massive Parallelism, Amalfi, Italy, October 1989*. Academic Press, 1991. Also: CSG Memo 313, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.

[15] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Killer Micro for A Brave New World. Technical Report CSG Memo 325, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, January 1991.

[16] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report TR432, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[17] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, Seattle, Washington, May 1990.

[18] A. Thekkath and H. M. Levy. Limits to Low-Latency RPC. Technical Report TR 91-06-01, Dept. of Computer Science and Engineering, University of Washington, Seattle WA 98195, 1991.