IN-61
179673
P-21

NASA Contractor Report 191479
ICASE Report No. 93-30

ICASE

20 Years of Excellence

LOW LATENCY MESSAGES ON DISTRIBUTED MEMORY MULTIPROCESSORS

Matthew Rosing
Joel Saltz

NASA Contract Nos. NAS1-19480, NAS1-18605
June 1993

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23681-0001

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

LOW LATENCY MESSAGES ON DISTRIBUTED MEMORY MULTIPROCESSORS

Matt Rosing[1]
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681
and
Joel Saltz[1]
Department of Computer Science
A.V. Williams Building
University of Maryland
College Park, MD 20742

ABSTRACT

This paper describes many of the issues in developing an efficient interface for communication on distributed memory machines and proposes a portable interface. Although the hardware component of message latency is less than one microsecond on many distributed memory machines, the software latency associated with sending and receiving typed messages is on the order of 50 microseconds. The reason for this imbalance is that the software interface does not match the hardware. By changing the interface to match the hardware more closely, applications with fine grained communication can be put on these machines. Based on several tests that we have run on the iPSC/860, we propose an interface that will better match current distributed memory machines. The model used in the proposed interface consists of a computation processor and a communication processor on each node. Communication between these processors and other nodes in the system is done through a buffered network. Information that is transmitted is either data or procedures to be executed on the remote processor. The dual processor system is better suited for efficiently handling asynchronous communications compared to a single processor system. The ability to send data or procedure invocations is very flexible for minimizing message latency, based on the type of communication being performed. This paper describes the tests performed and the proposed interface.

[1]This research was supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-19480 and NAS1-18605 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681

1 Introduction.

The goal of this paper is to discuss many of the issues involved in developing a highly efficient, portable software interface for sending messages on distributed memory machines. Our interest in this area stems from the fact that even though hardware latencies (the time for the hardware to send an empty message between two nodes) are on the order of 100ns on newer machines, the software component of the message latency is on the order of 50us when using a send/receive model of communication. By reducing this large discrepancy between hardware and software it will be possible to efficiently execute applications with fine grained communications and parallelism. Examples of applications that have these characteristics include unstructured mesh solvers, molecular dynamics codes, and some sparse iterative linear system solvers. These applications are characterized by having a large number of small messages. Because of this, the time to initiate messages becomes disproportionately dominant in the overall cost of the program.

One reason that it will be possible to specify a more efficient communication interface is that current message based libraries, although conceptually quite simple, provide a large amount of generality that in many cases is not needed. An example of this, that will be described in much more detail below, is pipelining. This technique costs a few thousand assembly instructions per element transmitted to implement on the iPSC/860 when using a library based on sends and receives. However, it is possible to implement this with only a few instructions when programming the hardware directly.

Thus, a new interface should be designed to take advantage of the hardware that is typically found in newer machines and allow the use of application specific knowledge to use this hardware more efficiently. This can be done by making the interface look more like the underlying hardware, which in turn will give the user more control of the hardware. An extremely low level interface like this will probably not be of interest to a broad range of users because of the added complexity that it will have when compared to the simple semantics of sends and receives. However, there are many users that will be able to take advantage of a lower level interface. Compiler writers, library writers and other tool builders that understand the hardware on these machines would be typical users. In this research, we are only interested in developing an interface for commercially available machines and do not address the multitude of research machines that exist.

A useful performance goal for such an interface would be to match the software component of the latency to the hardware component. That is, the time it takes to transmit data from a register on one processor to a register on another processor, including synchronization, should be roughly equal to the hardware latency. On a hypothetical machine with 100ns hardware latency and a 50Mhz clock this corresponds to roughly five instructions. This is an extremely aggressive goal and it is unlikely that a library can be used with these constraints. Possibly, some form of macros or pre-processor could be used.

In the next section we will give a very brief description of hardware trends and how this effects programming these machines. In section 3 we describe related work. In section 4 we describe the hardware of the Intel iPSC/860 and give a detailed example of implementing a pipeline algorithm when programming the hardware directly. In section 5 we describe a

typically difficult problem to implement on distributed memory machines. Finally, in section 6, we propose an interface that could efficiently support communications on many distributed memory machines.

2 Hardware Trends

The major trends of all distributed memory machines is that message latencies are going down and bandwidths are going up. The iPSC/860 has a bandwidth of 2.8 Mbytes per second and hardware latency of about 25μs between neighboring nodes. The Paragon will have a bandwidth per link on the order of 200 Mbytes per second and a hardware latency on the order of 100ns between neighboring nodes. The CM-5 has point to point communication as high as 20Mbytes per second and hardware latency roughly around 1μs. On the AP1000 [IHI+90], the network bandwidth is 25Mbytes per second with a hardware startup latency of 160ns between neighboring nodes.

A more interesting trend in the development of distributed memory machines is the addition of a processor on each node to handle communication. The Paragon has a general purpose processor to handle data transmission while the Meiko CS-2 has a custom processor for handling communications. A communication processor has the benefit of overlapping communication with computation. Another more important benefit of this is that an incoming message can be handled asynchronously with respect to the main processor without incurring an interrupt and paying the cost of disrupting the instruction and data cache on the main processor.

A vaguely similar idea to that of a communication processor is that of specialized packet types on the EM-4 [KSY90]. The EM-4 is a coarse grained data flow machine that supports different message types in hardware. These different types of packets include remote write, remote read, and remote process invocation as well as others. This is an example of a specialized communication processor that, if made more general, could greatly aid in reducing message latencies.

Another interesting development is the combination of distributed memory machines with more traditional shared memory technology. One example of this is the Cray-MPP that has local memory for each processor and a global address space [MPM92]. Although this is a distributed memory machine it will have very low message latencies for word size messages.

The net effect of these developments is that message latencies are becoming very small. This includes the time to create a message, transmit it, have another processor synchronize with that message, and put it in a useful form before using it. Whether or not it is possible to develop a portable interface that can be efficiently used for all of these hardware platforms is an interesting research question.

3 Related Work

Recently, a simple interface, Active Messages [vECGS92] has been developed that is more efficient than sends and receives on distributed memory machines. An Active Message is essentially an asynchronous Remote Procedure Call [Nel81]. That is, the calling end of the

RPC does not wait for the remote procedure to complete before it continues executing from the call site.

One reason that Active Messages are more efficient than sends and receives is because an active message, upon arrival at a processor, is processed immediately by a specified routine that was designed explicitly for that message. Therefore, the operating system has very little to do with the message and latencies can be controlled by the specified routine.

Although Active Messages will probably operate more efficiently than the send receive model, it is not clear that there is not a more efficient model to use. The overhead required to select, verify, and call the correct routine to use, along with effects of interrupting the processor and the cache, will probably be considerably more than the hardware latency, a time that we would like to match with the software latency. An example of where such fine grained communication would be required is a pipeline algorithm that will be described in more detail in the next section. In this algorithm, a message consists of a single floating point value and the number of operations between communication is very small. A critical aspect of making the pipeline version run efficiently is that the communication channel was treated as part of the pipeline. i.e., values were read and written directly from and to the channel from the computation. In this case, a communication coprocessor would have slowed down the communication considerably because of processor synchronization and the loss of memory bandwidth in doing extra, unnecessary reads and writes.

4 Low Latency Messages on the iPSC/860

To study how latencies associated with messages can be reduced on distributed memory machines we modified the NX/2 operating system on the iPSC/860 and ran various tests. In this section we describe the underlying hardware and operating system on the iPSC/860 as well as the tests we ran.

The hardware associated with communication consists of the network, an input and output FIFO, status and control registers, and interrupt logic. The FIFOs and registers are memory mapped. In NX/2, these locations are accessible only by the operating system. Both of the FIFOs are each 4k bytes long consisting of 1024 4 byte words. The status register describes the state of the FIFOs. There are flags indicating such things as empty, full, and partially empty or full. The control registers, among other things, controls when interrupts associated with the FIFOs occur. This could be never, at the beginning or end of an incoming message, or when a FIFO becomes half full.

A message consists of, essentially, a header word and data words. The first word contains the route the message is to take and thus describes the destination node. The rest of the message contains data. The last word has a special end of data mark associated with it for use by the hardware.

Therefore, a message can be generated quite easily. This can be done with two writes to memory. The first contains the destination address and the second contains data. It may be possible to send a single word message but we have not tried this. If the receive interrupt logic is not enabled a receive consists of reading the status register to check that a message is in the input buffer and then reading the message out of the buffer. In the case of receiving a two word message this essentially consists of three memory reads.

The interrupt logic on the 860 chip, used to handle asynchronous communication events, is a very large component of the message latency. It should be noted, however, that in the test that is usually performed to measure latency, bouncing messages between neighboring nodes, interrupts will not occur. In this test each of two nodes repeatedly waits for an incoming message and then immediately sends a message to the other processor. When a processor waits for a message to arrive the interrupt logic is turned off and the processor sits in a very tight loop waiting for the status register to change before pulling the message from the input buffer. In a more realistic situation where a message arrives before it is needed, causing an interrupt, the message latency can increase from 70us to 130us. (The higher time was measured when each process would wait for a message by continuously executing the probe function until a message arrived.)

The cause of interrupts being so expensive on the 860 chip is mostly due to the large state of the processor. This includes 32 floating point registers, 32 integer registers, an add, multiply, and load pipeline, and fairly complex instruction modes that all must be saved and reconstructed before resuming normal processing. The result of this is that it takes on the order of 1000 instructions to handle an interrupt. This does not include any of the time to process a message.

Other sources of increased latency include the time to do a trap into the operating system and the effects on the cache of messages asynchronously arriving at a node. The time to execute an operating system call, although not as severe as communication interrupts, is roughly 50 instructions. We have not measured the cost that an interrupt will have on the instruction and data caches but we expect that it would be substantially more than the hardware latency.

The operating system on the iPSC/860 (NX/2) controls the communication hardware and interrupt mechanisms. The communication model is based on sending and receiving contiguous blocks of typed messages. NX/2 must handle the general case of having any message of any size arrive at any time without the operating system crashing. This requires a complex system that handles buffer management, handshake protocols, interrupts, security and other issues. The operating system, although quite complex, handles this general case very well.

Another aspect of the operating system that adds to the latency of communication is that the operating system uses the same communication network as the applications. Because of this, NX/2 must be reliable and secure. This requirement adds to the overall message latency. Outgoing messages must be checked for valid addresses and the processor must assume that any system message can arrive at any time and must be handled properly.

This aspect of using the same hardware for both the operating system and user applications, and not having hardware support for message security is a problem that will make the goal of reducing the latency to a few instructions very difficult if not impossible. The only good solution for these problems are to handle them in hardware. It should be noted that the CM-5 is one machine that does not have these problems. We will not consider this problem any further and will assume that a single user has control of the entire machine.

The software latency associated with sending messages is primarily due to a complex general purpose operating system that can handle any situation. However, most of the time an application does not need the power of a general operating system. Many times the application writer has specific information that can be used to take advantage of the

hardware. Examples of this include the knowledge of how large a buffer is required, that only a single type of message will be sent, that the data from a message will always be placed in a specific location, etc. Thus, we want to give the user more control of the hardware to take care of these special cases. This is done at the risk of adding complexity to the send-receive model but we feel that this added complexity will be worth the added potential for writing efficient programs.

In order to test this idea we modified the NX/2 operating system to give the user more control of the hardware. It should be noted that the modifications made were done quickly as a "hack" just to test our ideas. The resulting code is clumsy to use and does not provide security between users. The modified operating system, called MX, executes in one of two modes. The first is identical to NX/2 and must be used whenever any system generated communications occur (file IO, prints, process control, etc). The second essentially removes the operating system. It is important that a message intended to be handled by one mode not arrive at a node while it is in the other mode, otherwise MX will crash. It would be possible to circumvent much of this problem by rewriting all of the system generated communication in terms of the non-NX/2 mode. It might also be possible to use the diagnostic network for the operating system to communicate across the machine. We have not made any of these changes.

The MX interface consists of a call to switch between MX and NX/2 modes, a call to set the receive and send interrupt handlers, and a call to set the control register. In MX the input and output FIFOs and the status register are mapped into user space so there are no calls to access these objects, the user can do this directly.

We have written several programs to study how message latencies could be reduced. The first test is similar to many of the tests used to measure message latency, a short message is bounced between two neighboring nodes. In this program it is not necessary to use interrupts to notify the processor that a message has arrived, the processor has nothing to do and will block until the message has arrived. Therefore, the interrupt mechanism is turned off and each processor waits until the status register indicates that a message has arrived before the processor reads the value from the input FIFO. A message send consists of writing two words to the output FIFO. The resulting time to execute a message send and receive is approximately 25us. As the number of assembly instructions to execute this loop is approximately a dozen, we believe that this time reflects mostly the hardware latency. Although reducing the latency from 70us to 25us is not tremendously important, on future machines this type of programming may reduce the latency from around 40us to less than 1us.

The rest of the examples in this section are based on a pipelined solve of a linear system of equations involving a banded, lower triangular matrix of the type that arises in pre-conditioning Krylov linear solvers with incompletely factored matrices [MvdV81] [TDJ68]. These matrices arise in the five point discretization of partial differential equations on two dimensional,`N`×`M` grids. The matrix consists of the main diagonal of `NM` ones (which are not stored); a diagonal immediately below the main diagonal with `NM - 1` elements, of which each Nth element is zero due to the border effects of the grid; and a diagonal `N` rows below the main diagonal with `NM - M` elements. Due to the zeroes in the first off diagonal, a pipelined type of parallelism can be used to perform the solve. At step 1, y_0 is computed. At step 2, y_1 and y_N are computed. At step 3, y_2, y_{N+1}, and y_{2N} are computed, at step

4, y_3, y_{N+2}, y_{2N+1}, and y_{3N}, and so on. There are a variety of ways this problem can be mapped onto multiprocessors, [SCMB90]. We choose a cyclic mapping that optimizes load balance at the expense of increasing both communication volume and number of messages. We map the two diagonal vectors cyclically onto the machine. Each y_i is computed by $rhs_i - y_{i-1} * al_{i-1} - y_{i-N} * an_{i-N}$. The first and third terms are computed locally since the processor that contains y_i also contains rhs_i, y_{i-N}, and an_{i-N}. The product $y_{i-1} * al_{i-1}$ is received from the neighboring processor by blocking until a message has arrived and then reading this value. This product is sent to the neighboring processor via some form of message. This algorithm, although messy, is typical of fine grained applications.

Sparse triangular solves arising from incompletely factored matrices pose challenging performance problems for distributed memory architectures. Our particular model problem is particularly challenging as we have set M equal to the number of processors we use and N equal to 2048. Therefore, each processor, except the first and last, receives and sends 2048 messages consisting of one floating point value each. In between each data transmission, each processor carries out at most four floating point operations.

All of the times described below are the times for the last processor to complete. As a reference point, this algorithm was encoded with the NX/2 send and receive library. The time to do the computation was 5us. The time to do the communication was 474us.

In the first version of the pipelined solve routine using MX, the interrupts were disabled and each send and receive was implemented in a manner similar to the bounce program described above. By doing this, the software component of the latency was reduced to the bare minimum. By leaving the incoming messages in the input FIFO until they were required, this also reduced the memory traffic. The resulting communication time for this version was 89us. The computation time was still 5us. Thus, the bulk of the 89us was the time to set up the channel.

In the next version of the solve routine, the hardware latency time was circumvented by just opening the channel once at the start of the loop and leaving it open for the duration of the iterations, after which point it was closed. By doing this a send consisted of writing one word to memory and a receive was done by reading the status register until something was in the input FIFO, and then reading the value into a register where it can be used immediately. The large savings, however, has to do with leaving the channel open for the duration of the computation phase. By doing this the communication time was reduced to 4us while the computation time was still 5us.

Although this technique of leaving the channel open will not be as useful on future machines that have very low hardware message latencies, it will still be useful in that a send has been reduced to a single write and a receive has been reduced to two memory reads. This is accomplished by not having to add control information to the message. In this case a message is just data and the overhead of figuring out what kind of data is not required.

As a final test, the code was written as if the communication pattern could not be predetermined. Although this is not the case for this program, it is for many irregular problems, as will be described in the next section. In this version of the program each node, instead of waiting for a value to arrive, will send a fetch request to the node that contains the value. The request is handled by an interrupt routine that is called whenever a message has arrived in the input FIFO. The communication component of this program executed in 385us. While this is not nearly as good as the previous program, it is still better than the program written

using the Intel primitives. In the modified version the bulk of the time spent was in the trap handler saving and restoring the processor state.

This final program is much more complicated than the other programs because of the complex nature of how the processes synchronize. In the other programs the synchronization is done based on the FIFOs while the synchronization involved in the last program is similar to that found in shared memory systems. Although it was not measured, we also expect that the time required to handle the synchronization is significantly more than that in the other programs.

In summary, these tests imply several characteristics for any proposed communication system. Firstly, interrupts are very expensive and should be avoided. This will become more important as the size of the processor state that must be saved and restored is growing with the complexity of the microprocessors used. Second, the operating system should be minimally involved with communications. As seen above where communications can be generated with one or two assembly instructions, any interaction with the operating system will be very expensive. This removal of the operating system implies that issues typically handled by the underlying system, such as security and IO, should be handled elsewhere. Finally, many numerical algorithms have a repetitive communication pattern, such as a pipeline, and the ability to setup the communication paths once and then reuse them can make very efficient use of the hardware.

5 Implementing a Sparse Matrix Solve

In this section we discuss some of the issues in implementing a more complex, fine grained problem, a lower triangular sparse matrix solve. Although simple, this problem illustrates many typical problems in implementing sparse problems.

A code segment that describes the basic computation is shown below.

```
do i=1,n
    y(i) = rhs(i)
    do j=col(i),col(i+1)-1
        y(i) = y(i) - a(j)*y(col(j))
    end do
end do
```

In this example, the dependency pattern is determined by the integer array **col**. We can determine, at runtime, which row substitutions (i.e. iterations of the outer loop) can be carried out independently. This leads to a natural way of parallelizing the code; we can simply carry out the computation as a sequence of parallel loops. Edges in this dependency graph that cross processor boundaries correspond to communication between different loops. Unfortunately, this process typically creates a large number of parallel loops so startup latencies tend to have a serious performance impact.

This is a very difficult problem to efficiently implement on machines with high message latencies because of the small message size inherent in this problem (each y(i) that is needed

on multiple processors). This is compounded by the fact that, in general, the values assigned to the **col** array are not determined until runtime and the corresponding communication patterns can therefore also not be computed until runtime. Thus, the technique of opening up a channel and leaving it open for the duration of a computation, as used in Section 4, will not be useful in this situation.

In situations where the above section of code is repeatedly called, it is possible to compute, once before the first iteration of the loop, schedules that are used to combine messages that are transmitted between the same nodes [SCMB90]. This technique was implemented using sends and receives and should benefit from a more efficient communication mechanism as described in Section 4. The main savings would be to remove the cost of interrupts by disabling them. This can be done because, by precomputing the communication schedules, there is no need for generating asynchronous communications. It will be important however, to add additional communications in the schedule for the purpose of flow control; it is preferable that the receiving FIFOs will always have room to store incoming data, otherwise an interrupt will be required to spill data from the FIFO into memory.

There are situations in which it is not feasible to carry out the kind of preprocessing described above. When a computation such as the sparse triangular solve is carried out only a small number of times, it may not be possible to amortize preprocessing costs. Furthermore, there are a number of computations in which the data access patterns cannot be known in advance. An example of this is sparse factorization with pivoting. In this case, a processor A needs to be able to fetch data from processor B, after processor B has calculated values required by processor A. In the example of the sparse lower triangular solve, processor A needs the value of **y(i)** from processor B only after processor B carries out the forward solve for **y(i)**. The asynchronous and fine grained nature of these problems make it impractical to implement using typical sends and receives.

To implement these types of problems, it is critical to have the ability for one processor to affect the memory of another processor with a minimum of effort. In the above example this requires the ability to fetch data from a processor once a specific condition has been met on that processor. To model this using NX/2 is difficult and the resulting costs associated with this are extremely high. Even using the modified operating system, it is doubtful if an efficient implementation can be built because it will be difficult to avoid the cost of an interrupt, as was required in the final experiment described in Section 4. On a machine such as the Paragon where it is possible to have no interrupts associated with such a memory fetch, it may be possible to have an efficient implementation.

6 A Proposed Interface

In this section we propose a machine model and an interface for generating low latency messages in a distributed memory environment. The machine consists of a set of nodes that are connected via a low latency network. Figure 1 describes the hardware of a single node. Each node contains a main processor for handling computation and a second processor for handling communication. The reason for having two processors is to eliminate interrupts and to support the overlap of communication and computation. Hiding communication in this manner is important when developing efficient programs. In this model, both processors

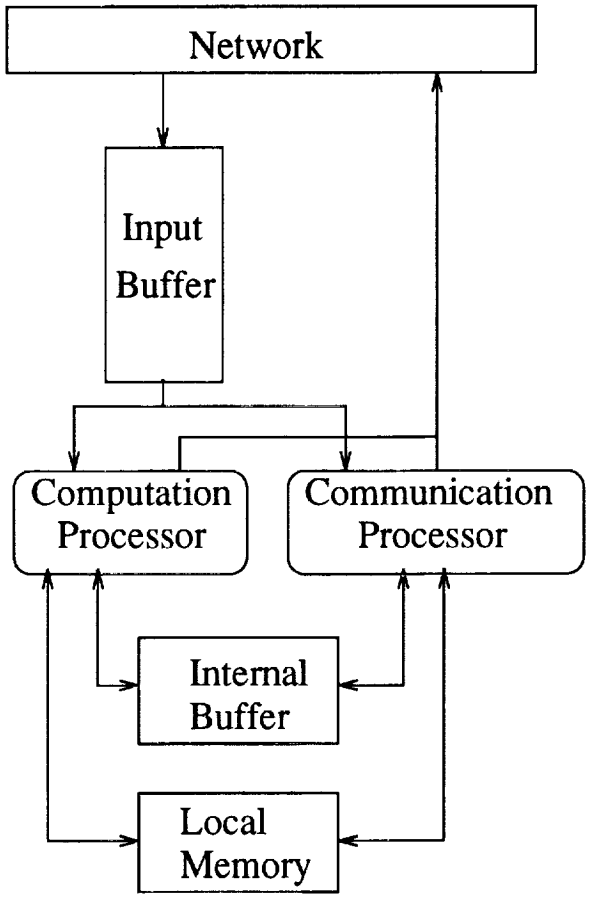

Figure 1. Hardware model for communication interface

are general purpose processors but in reality the communication processor could be a custom processor for handling communications.

An interesting problem with using two processors as opposed to one is the effect of cache. If a single processor handled both communication and computation and the data to be sent were residing in the cache at the send, then there would be no traffic between the memory and the processor; the processor would send the data directly to the network. If two processors are used then the data will have to first be flushed from the main processor's cache so that the second processor can use it. For small messages it will probably be more efficient to have the computation processor send the data.

The processors communicate with other nodes by receiving information from the network using the input buffer or by sending information to other nodes by writing directly to the network. The two processors can also communicate with each other through an internal buffer. The buffers are an important aspect of the model because they preclude the need for interrupts when data asynchronously arrives before it is needed. The buffers contain either of two types of information. The first is a pointer to a procedure that should be called when this information is "received" by either of the two processors. This is similar to an Active Message and efficiently supports application specific communications more than typed messages.

The second type of information is data composed of a sequence of bytes. It is intended that this type of data is sent between processors only when the two processors are cooperating with each other. In such a case the added complexity of calling a procedure to remove the data from the buffer and place it in memory where it can then be used by the local processor is cumbersome and inefficient. This type of communication is designed to support iterative communication patterns, such as the pipeline example described in section 4, by allowing incremental pieces of data to be sent as they become ready.

The network in the proposed model is similar to many current machines and has the following characteristics. It can send information packets with very little startup latency. The packets consists of procedure calls or blocks of data. Furthermore, information can be broadcast to a subset of nodes. This is done to minimize traffic in the network on machines that support broadcasts. Another important characteristic is that the network will ensure that information can not be sent to nodes that belong to other users. This is critical in order to remove the need for having the operating system involved with generating communications. Furthermore, packets are guaranteed to arrive in the order they are sent when multiple packets are sent between two nodes. This is important to prevent the receiving node from having to reconstruct the information before using it. Finally the network supports the ability for a processor to retain a path in the network and send several packets to another processor without packets from any other processor to arrive in mid stream. This makes it easier to send data that is not contiguous in memory or information that is spatially separated in time, such as the intermediate stages in a pipeline.

The interface to this model consists of three types of routines for generating communications. A conceptual description of these routines are given in figure 2. These include routines to send information, to receive information, and to see if there is information that can be received. This last type is needed because the receiving routines block until information has been received. The sending routines never block unless the network becomes full.

The three type of rountines include:

send(destination, information, is_more)
receive(source, information)
is_information(source, information_type)

where

destination ::= internal | node | node_list | previous_chain
information ::= function_pointer | (location, number_bytes)
source ::= internal_buffer | network_buffer
is_more ::= boolean

Figure 2. A Description of the Interface Routines.

The send routines have three parameters. The first describes the destination of the information, the second describes the information being sent, and the third is used to chain together several sends so as to send multiple packets of information separated over space or time. The destination parameter can indicate that the information is sent to a buffer on one or more nodes, the internal buffer on the present node connected to the other processor, or the same location that a previous chain of messages has been sent to. For example, data can directly be sent to another node (corresponding to a csend on the iPSC/860), or a procedure can be sent to the communication processor that will in turn send the data (corresponding to an isend on the iPSC/860). The final parameter is a boolean variable that indicates whether this message corresponds to a chain that will continue at a later time. This is essentially a mechanism to allow a channel to be opened between the current processor and another processor (or set of processors). The ability to create a channel and send multiple packets of information through it has two advantages. The first is that the cost of setting up the channel can be spread out over multiple messages and the second is that it can be used to ensure that multiple messages will arrive without interruption by messages from other nodes. A similar mechanism exists in hardware on the iWarp machine, developed at Intel. In the iWarp, a channel can be created between two nodes in the machine and data can then be sent through the channel with very low latency.

The receive routines consist of two types of functions. The first describes where the information will come from and is either the network buffer or the internal buffer. The second describes the type of information that is to be received. This can be either data or a procedure. If it is the former, then a pointer to where the data should be stored, along with the number of bytes that are to be read, is also in the parameter. If the information is a procedure then there is no additional information. In this case the receive call will not return until after the procedure has been received. Note that procedure messages can have parameters following immediately after the message using the chaining facility described in section 6.

The final type of routine is used to see if there is information of the correct type in one of the two buffers. There are two types of parameters in this call. The first describes which buffer is to be examined and the second describes the type of data that is expected. The output of this routine describes the state of the specified buffer to be one of the following: there is information of the correct type, there is no information, or there is information but

it is the wrong type.

6.1 Examples

In this section we give a brief description of how the described model can be used to build higher level communication models. The first example is to add asynchronous capabilities to the Active Messages model. The primary reason for adding asynchronous communications is to support the ability to overlap communications and computation. To do this, the communication processor on each node continuously looks for tasks to perform from both buffers on the node. The main processor asynchronously sends a task to a remote node by sending a message to the communication processor, which in turn sends the task to the remote processor. By having the communication processor handle the messages, the messages will be handled immediately and the main processor will be able to proceed doing something else. If the task is to be synchronously sent, then the main processor waits for a reply from the communication processor.

The asynchronous version of this interface can be built with four functions and is shown in figure 3. The first function, `AM_asynch`, is called by the user to initiate an asynchronous active message. The parameters include the destination node, a procedure to load data into the network and a procedure to unload the network. The procedure just transmits this data to the communication processor via the internal buffer. The procedure to load the network can load contiguous blocks of memory into the network using the `AM_data_put` routine. The channel is created before the routine to load the network is called. The `AM_data_get` routine is used on the receiving end to unload the data from the network. The fourth function is not called by the user but continuously runs on the communication processor. It looks for requests from both the local computation processor and remote communication processors. If a request is made from the local processor to send an active message then the three parameters are first received from the internal network. The data is sent to the remote node by opening a channel to it and placing the address of the procedure to receive the data, calling `ProcSend` so as to fill the channel, and finally closing the channel. (It is assumed that all nodes have an identical text image so the pointers to the functions are identical.) If a communication processor receives a request from a remote processor on the network buffer, the communication processor receives the pointer to the procedure to unload the network and then calls it.

The second example is to support the pipeline described in section 4. In this problem the communications processor would not be of any benefit so it would not be used. The programming of the hardware would therefore be very similar to what was done in section 4. The main processor would send data by using the send option with the flag set indicating that there will be more data in this message. This ensures that the remote node will not have other messages arriving in the middle of the sequence of data flowing through the pipe. By ensuring this exclusive use of the communication link, each packet of data can be reduced to the bare minimum. A send will consist of writing to the network and a receive will consists of a read from the network buffer. The four functions required to implement this are shown in figure 4. It should be noted that the sends and receives in this example work on such small data sizes that, for optimal efficiency, the put and get routines are defined as macros. Furthermore, the `Send` and `Receive` functions should also be implemented using some form

```
AM_asynch( int node, void (*ProcSend)(), void (*ProcRecv)()){
  Send( internal, node, 1);
  Send( internal, ProcSend, 1);
  Send( internal, ProcRecv, 0);
  }

AM_data_put( char *buf, int count){
  Send( previous_chain, buf, count, 1);
  }

AM_data_get( char *buf, int count){
  Receive( network, buf, count);
  }

AM_comm_proc(){
  int node;
  void (*ProcSend)(), (*ProcRecv)();
  for(;;)
    if( IsInformation( internal)){ /*request to send*/
      Receive( internal, &node, sizeof(node));
      Receive( internal, &ProcSend, sizeof(ProcSend));
      Receive( internal, &ProcRecv, sizeof(ProcRecv));
      Send( node, ProcRecv, sizeof(ProcRecv), 1);
      (*ProcSend)();       /*fill channel*/
      Send( previous_chain, 0,0, 0); /*close channel*/
      }
    if( IsInformation( network)){ /*incoming message*/
      /*get pointer to function*/
      Receive( network, &ProcRecv, sizeof(ProcRecv));
      (*ProcRecv)(); /*get parameters*/
      }
```

Figure 3. Example Interface for Asynchronous Active Messages.

```
pipe_create(int node){
  Send( node, 0,0, 1);
  }

#define pipe_put_float( x)  Send( previous_chain, &x, sizeof(float), 1)
#define pipe_get_float( pt) Receive( network, pt, sizeof(float))

pipe_kill(){
  Send( previous_chain, 0,0,0);
  }
```

Figure 4. Example Interface for Pipelined Communication.

of preprocessor to avoid the overhead of the function call.

The next example is to support the example described in section 5. In this example, each communication processor must do remote reads and writes. To handle a remote write, the main processor sends a task to the designated processor that contains the address to write to and the data to be written. A remote read can either be asynchronously or synchronously done. The asynchronous version can be done if multiple remote reads and writes can be done in parallel. To do this, the main processor sends a task to the appropriate processor that will do a remote write back to the main processor, and set a flag, indicating that the value has been received on the original processor. The main processor can then block on the flag until the value has been received. As remote writes are handled by the communication processor, this will be an efficient use of the processors. The interface to this is shown in figure 5. In this example a remote read or write sends the information to the communication processor, which in turn sends the information to the appropriate node. A remote write is implemented by the communication processor and a remote read reads the value and then does a remote write back to the original processor. In both cases a flag is set indicating that the operation has completed.

A final example is to implement sends and receives of buffered typed messages. A send consists of sending to packets of information. The first is a header describing the message type and length, and the second is the data itself. To ensure that these two packets arrive consecutively, the two are chained together using the mechanism described in section 6. The communication processor is programmed to receive these packets and place the data in a hashed buffer, based on the message type. A receive is then performed by the main processor by looking to see if a message of the appropriate type has arrived.

6.2 Implementation

The efficient implementation of this interface depends on several issues. The first of these is the ability to do some form of preprocessing on the calls described in section 6. To illustrate this, if the calls to the interface were in the form of a library then the overhead of making a library call would, in some cases, be more expensive then the actual action done by the

```
Remote_read_asynch( int node, float *remote, float *local, int *flag){
  /* Send to internal the values node, remote, local, flag, and that*/
  /* this will be a remote read                                      */
  }

Remote_write_asynch( int node, float *remote, float value, int *flag){
  /* Send to internal the values node, remote, value, flag, and that*/
  /* this will be a remote write                                     */
  }

comm_proc(){
  int node;
  float *remote, value, *local;
  int *flag;
  int type;
  for(;;){
    if( IsInformation( internal)){
      Receive( internal, &type, sizeof(type));
      if( type==READ)
        /*receive node, remote, local, flag   */
        /*transmit this to processor node along with this node number*/
      else /*type == WRITE*/
     /*receive node, remote, value, flag   */
     /*transmit this to processor node
      }
    if( IsInformation( network)){
      Receive( internal, &type, sizeof(type));
      if( type==WRITE){
        /*receive remote, value, flag*/
        *remote = value;
     *flag = 1;
        }
      else /*type==READ*/{
        /*receive remote, local, flag, and source_node*/
     value = *remote;
     /*transmit WRITE, source_node, value, local, flag*/
     }
    }
```

Figure 5. Example Interface for Asynchronous Remote Reads and Writes.

communication system. For example, if a single four byte word were to be sent to another processor then the implementation could take the form of a single write to the network. However, if this were handled with a library, then there would be overhead associated with the function call, determining that only a single word is to be sent, and probably moving the data from memory to the network instead of directly from a register. Because of this, it would apparently be more efficient to implement this system using a preprocessor that would translate the calls into more precise code.

The second issue concerning efficiency is security. On machines like the Intel Paragon and the Intel iPSC/860, the same network that is used for applications is used for operating system calls. Therefore, in order to ensure the integrity of the operating system, the application must first make a trap into the operating system before using the communication hardware. There are three potential solutions to this problem. The first is to ignore this problem when an application has sole use of the machine. In this case, if the operating system is corrupted it will have to be rebooted. The second is a more general solution and simply requires the trap handler to be reduced to minimize the overhead associated with traps. The current overhead of doing a trap on the iPSC/860 is roughly 50 instructions. This could possibly be reduced to 15 or 20. This would still be too large to send a large number of single words at a time. Therefore, to send noncontiguous data, there would also need to be commands to send multiple blocks of contiguous data and data separated by a fixed stride.

Another problem with implementing this interface efficiently has to do with the types of networks found on some distributed memory machines. In order to transmit large blocks of data with a high bandwidth, the data must arrive in the order in which it is sent. On the CM-5 this is not guaranteed in hardware and must therefore be handled in software. This has two consequences. The first is that the data must be handled an extra time, which uses memory bandwidth, and the second is that the data can not be left in the buffer until it is needed. This also causes extra memory traffic.

The final issue is implementing this interface on a system that has no form of communication processor. In this case the second processor must be emulated on the first processor using interrupts. Assuming that the cost of interrupts is not very expensive this would be a valid solution. However, on a machine such as the iPSC/860, where interrupts are very expensive, this would not be practical.

7 Conclusion

As the communication hardware improves on distributed memory machines it is important that the software improve equally as well. Bandwidth and latency improvements will make it possible to efficiently implement more fine grained applications, such as those found in sparse matrix computations. Currently, these applications are difficult to implement because the software associated with the usual send and receive model of communication is preventing the efficient use of the hardware. While send and receive libraries must handle the most general case, often it is possible to use application specific knowledge that allows for better use of the hardware.

It is clear that certain hardware configurations can greatly aid in supporting low latency messages. It is important that the operating system not be required when sending and

receiving data. This implies that there must be hardware support for isolating users from each other. Further, it is important that messages can be processed without interrupting the computation processor. This suggests some form of processor be used for handling communication. Finally, there must be relatively fast synchronization between the communication system and the computation processor.

Based on this hardware model we have defined a low level interface that more closely approximates the hardware found in many distributed memory machines.

References

[BSG90] H. Berryman, J. Saltz, and W. Gropp. Krylov methods with incomplete factorization preconditioners on the cm-2. *Journal of Parallel and Distributed Computing*, 8:186–190, Feb 1990.

[IHI+90] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. Cap-ii architecture. Technical report, Fujitsu Laboratories LTD, Kawasaki Japan, 1990.

[KSY90] Y. Kodama, S. Sakai, and Y. Yamaguchi. A prototype of a highly parallel dataflow machine em-4 and its preliminary evaluation. In *Proceedings of the Int. Conf. of Information Technology, Japan*, 1990.

[MPM92] T. MacDonald, D. Pase, and A. Meltzer. Addressing in cray research's mpp fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, July 1992.

[MvdV81] J. A. Meijerink and H. A. van der Vorst. Guidelines for the usage of incomplete decompositions in solving sets of linear equations as occur in practical problems. *Journal of Computational Physics*, 44:134–155, 1981.

[Nel81] B. Nelson. *Remote Procedure Call.* PhD thesis, Carnegie-Mellon Univ., 1981.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[TDJ68] Richard P. Kendall Todd Dupont and H. H. Rachford Jr. An approximate factorization procedure for solving self-adjoint elliptic difference equations. *SIAM Journal on Numerical Analysis*, 5:559–573, 1968.

[vECGS92] T. von Eicken, D. E. Culler, S.C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communications and computation. In *Proceedings or the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1993	Contractor Report

4. TITLE AND SUBTITLE
LOW LATENCY MESSAGES ON DISTRIBUTED MEMORY MULTIPROCESSORS

5. FUNDING NUMBERS
C NAS1-19480
C NAS1-18605

WU 505-90-52-01

6. AUTHOR(S)
Matthew Rosing
Joel Saltz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 93-30

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-191479
ICASE Report No. 93-30

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
Submitted to Scientific Programming

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 60, 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper describes many of the issues in developing an efficient interface for communication on distributed memory machines and proposes a portable interface. Although the hardware component of message latency is less than one microsecond on many distributed memory machines, the software latency associated with sending and receiving typed messages is on the order of 50 microseconds. The reason for this imbalance is that the software interface does not match the hardware. By changing the interface to match the hardware more closely, applications with fine grained communication can be put on these machines. Based on several tests that we have run on the iPSC/860, we propose an interface that will better match current distributed memory machines. The model used in the proposed interface consists of a computation processor and a communication processor on each node. Communication between these processors and other nodes in the system is done through a buffered network. Information that is transmitted is either data or procedures to be executed on the remote processor. The dual processor system is better suited for efficiently handling asynchronous communications compared to a single processor system. The ability to send data or procedure invocations is very flexible for minimizing message latency, based on the type of communication being performed. This paper describes the tests performed and the proposed interface.

14. SUBJECT TERMS
parallel computers, distributed memory, communication, message latency

15. NUMBER OF PAGES
20

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std Z39-18
298-102

☆U.S. GOVERNMENT PRINTING OFFICE: 1993 - 728-064/86025