Low Latency Messages on Distributed Memory Multiprocessors

MATT ROSING[1] AND JOEL SALTZ[2]

[1]*Pacific Northwest Laboratory, Richland, WA 99352*
[2]*University of Maryland*

ABSTRACT

This article describes many of the issues in developing an efficient interface for communication on distributed memory machines. Although the hardware component of message latency is less than 1 μs on many distributed memory machines, the software latency associated with sending and receiving typed messages is on the order of 50 μs. 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. This article describes several tests performed and many of the issues involved in supporting low latency messages on distributed memory machines.

1 INTRODUCTION

The goal of this article 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 1 μs on newer machines, the software component of the message latency is on the order of 50 μs 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, which 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, new interfaces should be designed to take advantage of the hardware that is typically found in newer machines and allow the use of ap-

Received August 1993
Accepted October 1994

e-mail: m_rosing@pnl.gov

Scientific Programming, Vol. 4, pp. 35–43 (1995)
CCC 1058-9244/95/010035-09

plication-specific knowledge to use this hardware more efficiently. This can be done by making an interface look more like the underlying hardware, which in turn will give the user more control of the hardware. Low-level interfaces will probably not be of interest to a broad range of users because of the added complexity that they 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 lower-level interfaces. 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 distributed memory machines and do not address the multitude of other research and commercial machines that exist.

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. At the end of this section we describe a typically difficult problem to implement on distributed memory machines. Finally, in Section 5 we discuss many of the issues involved in building an interface that could efficiently support communications on many distributed memory machines, as well as potential hardware support to lower software latency.

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 25 μs between neighboring nodes. The Paragon has a bandwidth per link of 200 Mbytes per second and a hardware latency on the order of 1 μs between neighboring nodes. The CM-5 has point-to-point communication as high as 20Mbytes per second and hardware latency roughly around 1 μs. On the AP1000 [1], the network bandwidth is 25Mbytes per second with a hardware startup latency of 160 ns 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 whereas 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 [2]. 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 [3]. 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 [4], 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 [RPC, 5]. 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 that an active message, on 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. 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 by the application. In this case, a communication coprocessor would have slowed down the communication considerably because of processor synchronization and the loss of memory bandwidth is 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 1,024 four 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, control 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 messages 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 70 to 130 μs. (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 1,000 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. A related issue is that a user's application must not be able to jam the network. If this happens, both system messages and messages from other nodes will not be able to get through the network. This requires some form of flow control to be implemented and can be a substantial source of message latency.

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 on the iPSC/860 impossible. The only good solution for these problems is to handle them in hardware. It should be noted that the CM-5 is one machine that does not have these problems.

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 advantage of these special cases. This is done at the risk of adding complexity to the send-receive model but we believe that this added complexity is worth the increased efficiency.

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 communication 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 25 μs. 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 70 to 25 μs is not tremendously important, on future machines this type of programming may

reduce the latency from around 40 to less than 1 μs.

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 preconditioning Krylov linear solvers with incompletely factored matrices [6, 7]. 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} are computed, and so on. There are a variety of ways this problem can be mapped onto multiprocessors [8]. 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 $rhs_i - y_{i-1} * a1_{i-1} - y_{i-N} * an_{i-N}$. The first and third terms are computed locally because the processor that contains y_i also contains rhs_i, y_{i-N}, and an_{i-N}. The product $y_{i-1} * a1_{i-1}$ 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 challenging as we have set M equal to the number of processors we use and N equal to 2,048. Therefore, each processor, except the first and last, receives and sends 2,048 messages consisting of one floating point value each. In between each data transmission, each processor carries out at most four floating point operations. There are 32 processors.

All of the times described below are the times for the last processor to complete divided by the number of iterations N. These times are therefore the time per iteration. As a reference point, this algorithm was encoded with the NX/2 send and receive library. The time to do the computation was 5 μs. The time to do the communication was 474 μs.

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 89 μs. The computation time was still 5 μs. Thus, the bulk of the 89 μs 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 is used immediately. The large savings, however, have to do with leaving the channel open for the duration of the computation phase. By doing this the communication time was reduced to 4 μs while the computation time was still 5 μs.

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 overload 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 385 μs. Although 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 whereas 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.

4.1 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 run-time, 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 to 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 run-time and the corresponding communication patterns can therefore also not be computed until run-time. 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.

To implement these types of problems, it is critical to have the ability for one processor to modify the memory of another processor with minimal 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.

5 INTERFACE DESIGN ISSUES

In this section we describe many of the issues in developing support for low latency messages. We are primarily interested in developing software to run on existing machines but the issues raised can also be used to direct the development of hardware.

One of the major tasks in lowering latency is to eliminate interrupts. Based on a goal of a 1 μs software latency (approximately 50 assembly instructions), an interrupt on a typical super scalar RISC processor is much too large. This will probably get worse as the size of the processor state that must be saved and restored increases. To avoid interrupts and not require message buffering in the network, the executing process must poll the network or a dedicated processor must be added to process messages. Polling would be impractical without compiler support to automatically insert polling instructions. Even then, it will be difficult to control how often a poll is made. With respect to defining new hardware, a specialized processor could be tailored for handling communications, or a general purpose processor, like that on the Paragon, can be used. For example, we have implemented several commands, including remote write, read, and accumulate, on the Intel Paragon [9]. Initial timings for remote write and read are 30 μs and 55 μs, respectively. These timings, although not close to our goal of matching the software latency to the hardware latency, show how a

more specific communication model can run efficiently.

The commands supported by a message processor also affect message latency. A fixed set of message commands will be easier to implement efficiently than allowing the execution of general code. However, these commands should support low-level operations such as direct or indirect reads and writes, accumulations into memory, or appending data to a buffer. The use of a general purpose processor is more flexible with respect to implementing these commands but may not be as efficient as custom hardware. With respect to defining an interface for existing machines, modeling a general purpose processor may be difficult if not impossible to implement because the user might not be allowed unrestricted access to the network. This is the case on the Paragon, where the message processor executes in system space, but not on the CM-5, where the user can access the network.

In general, there are few constraints on what types of operations can be supported. For developing interfaces to existing machines, there is one major constraint; the execution of a command cannot block the network. For example, a synchronization command that spins on a memory location until it is a certain value could block the network for an extended period. On machines where there is only one network to support both the operating system and user applications, it is critical that the network not block and prevent, for example, operating system signals from getting to a user application. One solution to this is to implement a virtual network for each user task and the operating system.

The inclusion of a second processor to handle communications, along with the need for this processor to directly access user memory, suggests other requirements of the message processor. The first of these is that the message processor executes in user space. This allows virtual addresses for reads and writes to be automatically translated by hardware as opposed to doing this in software. The translation of these addresses manually by the operating system is somewhat expensive (on the order of a cache miss). If there is a page fault, then the message processor must be able to resolve this while the rest of a message is backed up in the network. This implies that page requests are supported on another, potentially virtual, network.

Another source of latency is the operating system. As shown in Section 4 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, IO, and system support should be handled elsewhere.

The next issue that influences latency is the use of buffers. Buffers are useful for overlapping communication and computation in asynchronous tasks but require a fair amount of overhead to manage unless they are implemented in hardware. Buffering data may involve unnecessary copying. This effects both the memory bandwidth and cache. A larger source of latency is due to the asynchronous behavior between the user's application, the network, and user requests for communication. To support efficient use of the network and overlap communication with computation, applications have the ability to spawn communication requests that run asynchronously with the network and user's program. Processing such requests requires a relatively large amount of time in managing data structures and handling flow control. All of this can be simplified if the number of active requests is limited to a few per node.

Flow control is another issue associated with synchronizing processes that is a source of latency. Some form of flow control is required wherever data are sent between processes and there is some delay in processing the data at the remote end. An example of this is a remote read of large memory blocks. In such a case, requests can show up faster than they can be processed and the network could block. Another example of blocking the processing of network messages is having some form of mutual exclusion between the application on the main processor and the network.

Flow control requires extra messages be passed that are used to request and allocate space on the target node. Increased latency is caused by extra messages and managing whatever resource is being allocated. It is possible that supporting virtual networks in hardware that can block would work well enough in this type of situation. However, low-level support for requesting and allocating space may be needed to prevent deadlock. This is typically implemented in some form of enquire/acknowledge hand shake.

There are several issues associated with the network that influence message latency. As described above, the network should have flow control mechanisms built into it. Related to this is the

required support for virtual networks. That is, each application and the operating system should have its own network. Built into each virtual network are security mechanisms that prevent an application from sending messages to tasks not associated to them and simultaneously allow direct access to the network. Without this requirement, the operating system will not be able to depend on the network, or applications will have to go through the operating system to use the network.

The network should also guarantee that packets arrive in the order they are sent. This is important to prevent the receiving node from having to reconstruct the information before using it. Another characteristic of the network is that it 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 arriving in midstream. Many numerical algorithms have a repetitive communication pattern, such as a pipeline, and the ability to setup the communication mechanism once and then reuse it can make very efficient use of the hardware. 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.

6 CONCLUSIONS

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. Although 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. It is also important that messages can be processed without interrupting the computation processor. This suggests some form of processor be used for handling communication. This processor should execute in user space. Finally, each task group should have its own virtual network that has built in flow control and security.

One of our goals is to develop communication interfaces that are more specific and more efficient than that of the send and receive model. In the process of developing a few such interfaces we have found that there are only a few, hardware dependent, interfaces that have low software latencies. Although we have developed more efficient interfaces than send and receive, they are not substantially faster because of the many tasks such as flow control and security that must be done to build a flexible communication interface. More importantly, though, we believe that the inclusion of a few well-known hardware mechanisms, such as support for a communication processor, virtual networks for each user task and the operating system, and security built into the network, could make the development of relatively more general, low latency interfaces possible.

ACKNOWLEDGMENTS

Matt Rosing was supported by the National Aeronautics and Space Administration under NASA contract NAS1-18605 and NAS1-19480 while in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, and currently by Pacific Northwest Laboratory, operated for the U.S. Dept. of Energy (DOE) by Battelle Memorial Institute under contract DE-AC06-76RLO 1830.

REFERENCES

[1] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato, "Cap-ii architecture," Technical Report, Fujitsu Laboratories LTD, Kawasaki Japan, 1990.

[2] Y. Kodama, S. Sakai, and Y. Yamaguchi, *Proceedings of the International Conference of Information Technology, Japan, 1990.*

[3] T. MacDonald, D. Pase, and A. Meltzer, *Proceedings of the Third Workshop on Compilers for Parallel Computers, Vienna, July 1992.*

[4] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, May 1992*, pp. 256–266.

[5] B. Nelson, "Remote procedure call," PhD thesis, Carnegie-Mellon University, 1981.

[6] 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," *J. Computational Physics*, vol. 44, pp. 134–155, 1981.

[7] R. P. Kendall, T. Dupont, and H. H. Rachford Jr., "An approximate factorization procedure for solving self-adjoint elliptic difference equations," *SIAM J. Numerical Anal.* vol. 5, pp. 559–573, 1968.

[8] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, "Run-time scheduling and execution of loops on message passing machines," *J. Parallel Distrib. Computing*, vol. 8, pp. 303–312, 1990.

[9] M. Rosing and P. Pierce, *Proceedings of the 1994 Intel Supercomputer Users Group Conference*, June 1994.