Message Buffering and its Effect on the Communication Performance of Parallel Computers

William Saphir[1]

Report RNS-94-004 April 1994

Abstract

A primary barrier to obtaining high performance on distributed memory parallel computers is poor internode communication performance. Access to remote data has historically been orders of magnitude slower (higher latency and lower bandwidth) than access to local data. In recent generation parallel computers, however, internode bandwidth approaches local memory bandwidth. This change in system balance increases the effect of message buffering on communication performance, since the time to copy a message to or from a local buffer is comparable to the time to transfer the message across the network. The performance degradation introduced by message buffering is illustrated on the Intel Paragon (running OSF/1 R1.1.4), where bandwidth for unbuffered NX messages is approximately 35 megabytes per second (MB/s), bandwidth for buffered NX messages is less than 5 MB/s, and bandwidth for PVM messages, which are buffered at least twice, is approximately 2 MB/s. Buffering may also interfere with attempts to optimize performance by overlapping communication with computation.

Message passing libraries differ considerably in their buffering of messages. I discuss in detail the buffering characteristics of NX on the Intel Paragon and CMMD on the Thinking Machines CM-5, as well as buffering in the "portable" libraries PVM and MPI. I describe the strengths and weaknesses of these libraries with respect to message buffering, concluding that CMMD is well-designed for high performance on fast networks and MPI for portable performance on both slow and fast networks. NX suffers from design flaws but can be made to perform efficiently. PVM, while appropriate for ethernet-connected workstations, belies its reputation as a portable and robust de facto standard by performing poorly on parallel computers with fast networks.

1. Numerical Aerodynamic Simulation (NAS) Systems Division, MS 258-6, NASA Ames Research Center, Moffett Field, CA 94035-1000 (wcs@nas.nasa.gov). The author is an employee of Computer Sciences Corporation, NASA Contract NAS 2-12961

1.0 Introduction

Distributed memory parallel computers have not yet fulfilled their promise to replace vector supercomputers for general purpose scientific computation. While parallel computers solve certain types of problems efficiently, they don't perform as well on problems with global data dependencies, such as the simulation of fluid flow using implicit methods [1, 2]. The efficient solution of these problems requires very fast data transfer between the nodes of the parallel computer.

Internode communication performance is limited fundamentally by internode network hardware, but is reduced further by software overhead. One of the most important but overlooked sources of overhead is buffering. One says that transferred data (usually called a "message") is *buffered* when the transfer mechanism makes a temporary copy of the data between its source and destination.

Buffering has three important effects.

- Buffering reduces the effective bandwidth for data transfer, because copying data to or from a buffer takes extra time.
- Buffering reduces the performance benefit of overlapping communication and computation. While network data transfers can be performed independently of the processor, buffer copying can't be overlapped with computation because it is performed by the processor itself.
- Buffering increases memory use, effectively decreasing the amount of memory available to an application. This does not directly affect performance.

Buffering has historically had little effect on performance. In recent generation high performance parallel computers, however, internode network bandwidth approaches local memory bandwidth. With this change in system balance, the time to copy data to or from a buffer can be comparable to the time to transfer data across the network. This magnifies the performance effects of buffering to the point where they are intolerable for the most demanding applications.

An analysis of the performance implications of buffering is most straightforward for the so-called "message-passing" paradigm for parallel computation, where an independent thread of control is associated with each node of a parallel computer, and nodes exchange data using a cooperative send/receive mechanism. Most message passing codes are of the SPMD (Single Program, Multiple Data) type, in which each node runs (asynchronously) a separate copy of the same program.

I evaluate four important message passing libraries representing four completely different approaches to message buffering.

NX [3] is the native message passing library on Intel Parallel Supercomputers, including the Paragon, the iPSC/860 (Hypercube) and Delta. NX message buffering is complex and nondeterministic, applications can avoid buffering only by using explicit synchronization. Buffering has a moderate but unpredictable effect on performance.

CMMD [4] is the native message passing library on the Thinking Machines CM-5. CMMD allows applications to avoid buffering almost entirely. With respect to message buffering, it is well-designed for fast networks.

PVM (Parallel Virtual Machine) [5] is a portable library that runs on both tightly coupled multiprocessors and networks of workstations. PVM buffers messages twice explicitly and sometimes implicitly in the underlying transport layer. While this behavior is appropriate for networks of workstations, it makes PVM very inefficient on high performance parallel computers and unacceptable for applications that require very high network bandwidth.

MPI (Message Passing Interface) [6] is a recent message passing interface standard. MPI allows flexibility in buffering that, in principle, provides efficient performance on both slow and fast networks. However, the current publicly available implementation of MPI buffers messages unnecessarily on fast networks, producing very poor performance.

The specific performance data I discuss in this report will become out-of-date fairly rapidly as hardware and software improvements are made. The data are intended to illustrate the performance implications of buffering, not the performance of the computers being discussed.

2.0 Message Buffering

The distinguishing feature of the message passing paradigm for parallel computation is that nodes transfer data using a cooperative send/receive paradigm: one node "sends" data that is "received" by another. The data in the local memory of the sending node is copied into the local memory of the receiving node as shown in Figure 1.

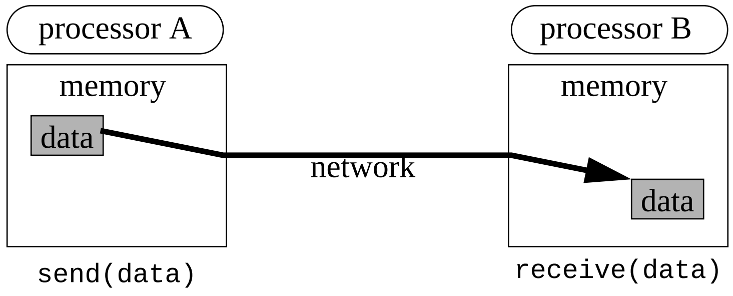

FIGURE 1. *Basic message passing.*

To describe buffering, it is convenient to divide process memory logically into two types. User memory is explicit in the user program and accessed directly by it. It consists mainly of user program data structures. System memory is not accessed explicitly by the user program but is managed by system libraries or possibly the kernel.

Although from the user's point of view data is transferred from user memory on one node to user memory on another node, the communication library may make a temporary copy in system memory. This is called buffering. Data may be buffered at the sending node, the receiving node, or both. Figure 2 shows the path of data that is buffered on the receiving node.

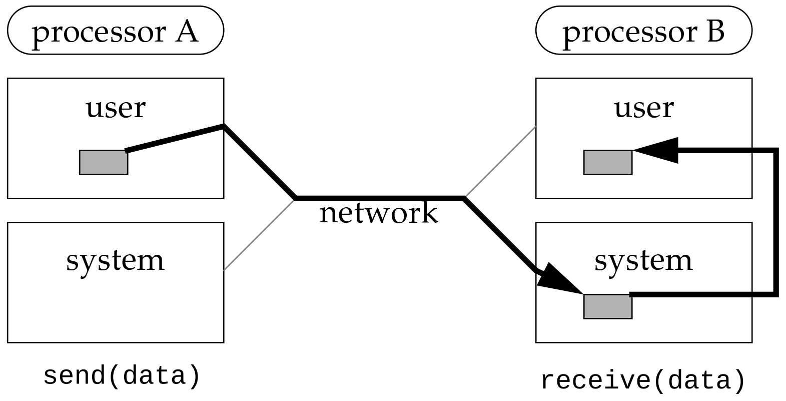

FIGURE 2. *Path of a message buffered at the receiving node*

There are two good reasons to buffer data. The first is when the semantics of `send()` specify that it must complete whether or not a corresponding `receive()` is posted. Since the system may not have been told where to put the data, it must make a copy. In SPMD applications, it is rarely necessary to have a *non-blocking* `send()` of this sort (see section 3.1). A second reason for buffering is to free the processor from having to wait on a slow network. In this case, the processor can make a local copy of the data, tell the network to transfer the data when possible, and then continue to perform useful work while the data is transferred.

Internode communication performance can be characterized by describing how long it takes to perform a single data transfer between two nodes and how the network handles multiple simultaneous data transfers. The latter issue is part of a vast subject sometimes called "scalability," and is not directly related to buffering.

Data transfer time is often a linear function of message size characterized by latency (start-up time) and bandwidth (transfer rate):

$$\text{transfer time} = \text{latency} + \frac{\text{message size}}{\text{bandwidth}}$$

Transfer time is smaller for lower latency and higher bandwidth. When messages are short, latency dominates the transfer time. When messages are long, bandwidth is most important. Messages are "long" or "short" relative to a $\text{crossover size} \equiv \text{bandwidth} \times \text{latency}$, for which latency and bandwidth are equally important.

In cases where transfer time is not linear in message size, it is common to define "latency" as the minimum time to send a message (usually a message of zero length) and "bandwidth" as the maximum effective bandwidth, that is, message size divided by total transfer time for very large messages.

Internode communication performance is usually orders of magnitude worse than local memory performance. In the most recent generation of parallel computers, however, internode bandwidth has begun to approach local memory bandwidth. This change in system balance (ratio of network bandwidth to local memory bandwidth) increases the effect of message buffering on communication performance.

The change in system balance results from improvements in network technology and the fact that RISC microprocessors, upon which parallel computers are often based, typically have poor bandwidth to main memory and a small cache. A memory to memory copy on a RISC processor is done using a sequence of load and store operations. This process is inefficient and very sensitive to the relative alignment of the source and destination.

The performance implications of buffering are twofold. First, the speed of copying is an upper limit on effective bandwidth for message transfers. More specifically, assuming the network transfer and buffer copy occur sequentially, the effective bandwidth b_e is the harmonic mean of the copy bandwidth b_c and the network bandwidth b_n:

$$b_e = \frac{b_c b_n}{b_c + b_n}$$

Even if the copy speed is equal to the network speed, effective bandwidth is reduced by a factor of two.

The second effect of buffering is that the buffer copy is done by the processor itself and therefore cannot be done asynchronously. This reduces the benefit of overlapping computation and communication - an important optimization in many codes. When copy bandwidth equals network bandwidth, the benefit is lost entirely.

Both effects are most important for long messages since they affect bandwidth but not latency. There is very little start-up cost associated with a memory-to-memory copy.

3.0 NX Message Buffering

NX [3] is the native communication library on Intel parallel supercomputers. The Intel Paragon is a distributed memory parallel computer based on the Intel i860/XP microprocessor. The processors are connected by a network with the topology of a 2-dimensional grid. The Paragon can run either an enhanced version of the OSF/1 operating system, provided by Intel, or the SUNMOS [7] operating system, developed at Sandia National Laboratory and the University of New Mexico. SUNMOS provides higher performance, less overhead, but also less functionality. Both support NX, but implement it in slightly different ways. Unless noted otherwise, the description below applies to NX under OSF/1 AD, revision 1.1.4.

The four basic NX message passing primitives are `csend()`, `crecv()`, `isend()` and `irecv()`. Semantically, they behave as follows.

- `csend()` sends a message. It returns whether or not a matching receive has been called[1]. The calling process is free to modify the original data after `csend()` returns.
- `isend()` sends a message. It returns "immediately" whether or not a matching receive has been called. The program may not modify the data until verifying (using `msgwait()`) that the message has been sent. The network transfers the data asynchronously, that is, outside the flow of control of the sending process. The processor can do useful work while the message is being transferred.

1. This behavior is not documented explicitly, but is the way `csend()` has behaved historically and is the way users expect it to behave.

- `crecv()` receives a message. It blocks until a matching message arrives. When `crecv()` returns, the data may be used and modified.
- `irecv()` "posts a receive" and returns immediately. The program may not use or modify the data until verifying (using `msgwait()`) that the message has been received.

From the point of view of the calling process, `csend()` and `crecv()` initiate and complete a data transfer, while `isend()` and `irecv()` initiate a data transfer that proceeds independently of the processor.

The semantics of the NX message passing primitives do not specify the path taken by the data. Specifically, they do not say anything about buffering. `csend()` requires some sort of buffering for its implementation, but how that buffering is done is not specified. Message buffering under NX is complex, nondeterministic, and undocumented. The following description of buffering is based on an "experimental" investigation of NX. It is appropriate for long messages (more than 100 bytes).

The path of a message sent from node A to node B using the NX `csend()` or `isend()` routine is shown schematically in Figure 3. Data is either transferred directly into user memory (path 1) or buffered in system memory (path 2) depending on whether or not a corresponding receive (`crecv()` or `irecv()`) has been posted on node B.

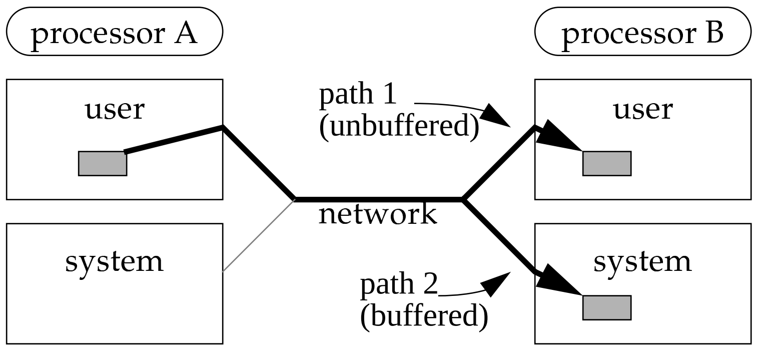

FIGURE 3. `csend()` ***and*** `isend()` ***data paths***

If node B has already posted a receive, NX knows where to place the incoming message from node A. If not, NX transfers the data into the system memory of node B. This message buffering (or some variant) is required by the semantics of `csend()`, because it must complete whether or not a matching receive has been posted. The semantics of `isend()` do not require buffering but `isend()` is implemented this way on the Paragon.

If a message has been buffered on the receiving side, it is copied into user memory when the receiving node calls `crecv()` (Figure 4). If, instead of `crecv()`, the receiving node calls `irecv()` followed by `msgwait()`, the details are much more complicated. The important fact is that the copy does not proceed asynchronously (overlapped with computation).

When a buffered message is received by `irecv()` followed by `msgwait()` the buffer copy is usually copied by `msgwait()`. If, however, there is an intervening `crecv()` for a (different) message that has not arrived, the `crecv()` copies (entirely) the buffered message into user space before waiting for its own message. This provides some optimization if the message being received takes a long time to arrive, so that the processor can make use of the idle time. It comes, though, at the expense of clear, predictable behavior. A consequence is that a `gsync()` (a global synchronization, implemented with `csend()` and `crecv()`) will cause buffered messages to be copied if a receive for them has already been posted. The price of copying buffered messages is paid in additional time for the `gsync()` to complete. A final complication is that an intervening `irecv()/msgwait()` pair, which is otherwise equivalent to `crecv()`, will not cause the buffered message to be copied.

Buffer copying on the iPSC/860 and on the Paragon under SUNMOS is different. On those platforms, buffer copying is done inside the `irecv()`, not the `msgwait()`. Although the iPSC/860 and Paragon documentation are unclear and contradictory, this behavior appears to be incorrect because `irecv()` (Immediate RECeiVe) should return immediately.

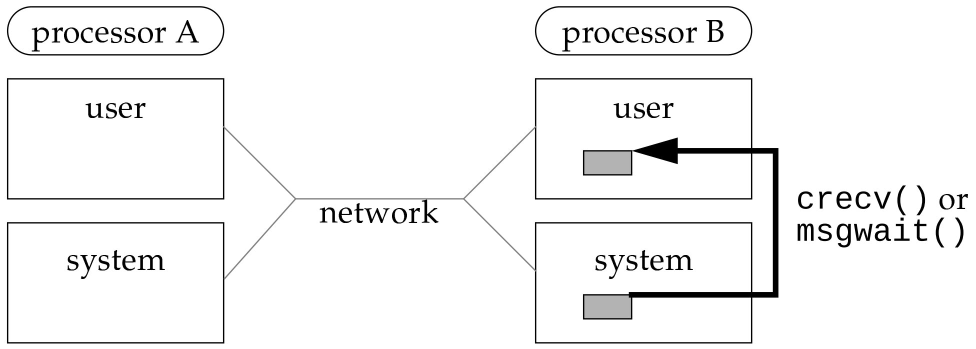

FIGURE 4. ***Memory-to-memory copy for a buffered message***

The Paragon has unusually high network bandwidth and low memory-to-memory copy bandwidth. Under OSF/1, the copy is performed by the library routine `bcopy()`, which performs at the slow speed of 4.8 MB/s, compared to a effective network bandwidth of 35 MB/s (for unbuffered messages). Under SUNMOS, message copy speed (copying is not done by `bcopy()`) is about 19 MB/s, compared to a network speed of 175 MB/s. Intel has recently provided an optimized `bcopy()` that can perform as fast as 70 MB/s (although copy speed is a very sensitive function of buffer size and alignment, and can be as low as 4.8 MB/s). The fast `bcopy()` temporarily leapfrogs OSF/1 network bandwidth but performs at less than half of what the network will eventually be capable. For comparison, NX on the Intel iPSC/860, a previous generation parallel computer from Intel, uses the same buffering scheme, but performance degradation due to buffering is much smaller because the system balance is different: the network bandwidth of 2.8 MB/s is much slower than the copy speed of about 19 MB/s.

3.1 NX Message Buffering in Real Applications

Because NX message buffering is complex and nondeterministic, avoiding buffering in real applications can be difficult. The problem is exacerbated by communication patterns commonly found in SPMD codes. Message passing in real applications is almost always part of a loosely synchronous data exchange. The logical flow of a typical SPMD program looks like:

```
compute   (local computation)
exchange  (communication)
compute
exchange
etc.
```

The exchanges include both sending and receiving data: message traffic is rarely unidirectional.

In the general case, a node may send to and receive from a large number of nodes, but all the interesting buffering behavior can be seen by examining a single exchange with one neighbor. Consider the exchange implemented by the following pseudocode.

```
node A                  node B
csend(nodeB, buf1)      csend(nodeA, buf1)
crecv(buf2)             crecv(buf2)
```

Assume that all nodes are exactly synchronized. In that case, the `csend()`s occur simultaneously and the `crecv()`s happen simultaneously. Both messages are buffered, since no receive is posted before the corresponding send. If, on the other hand, the nodes are not synchronized, it is possible that the `csend()` and `crecv()` on one node will happen before the other node reaches this block of code, resulting in both buffered and unbuffered communication.

In the more general case of an exchange among a large number of nodes, one send is always buffered. This happens no matter what the state of synchronization, and further delays those nodes on which buffering occurs. As information propagates in subsequent exchanges, the delay propagates as well so that the effective bandwidth is always limited approximately by the message copy speed.

There is one case when the exchange illustrated above is necessary. This is when `buf1` must be the same as `buf2`. Since in this case data must be sent before new data is received, buffering is inevitable. Usually this situation can be avoided, and it is rare in practice.

To explore truly interesting exchanges, we need to exploit the possibility of overlapping communication with computation through the use of `isend()` and `irecv()`. This is an important optimization on any system where the network can operate independently from the compute processor. A sophisticated version of the above exchange, which tries to post the receive before the send and also to overlap communication with computation, might look like the following:

```
node A                      node B
mid1=irecv(buf1)            mid1=irecv(buf1)
mid2=isend(nodeB, buf2)     mid2=isend(nodeA,buf2)
           .. perform computation..
msgwait(mid2)               msgwait(mid2)
msgwait(mid1)               msgwait(mid1)
```

If the nodes are perfectly synchronized, the receive will be posted before the send, and communication will be unbuffered. If the computation is long enough, the costs of communication can be hidden entirely, because the data will be available at the end of the computation (so that the `msgwait()` returns immediately). If, however, the nodes are even slightly out of sync (`irecv()` is very fast) the message will be buffered. This has several consequences:

1. The maximum effective bandwidth will be significantly reduced.
2. The benefits of overlapping communication with computation will be lost, as the message copy is done during the `msgwait()`, not during the computation.
3. The delay caused by the extra time to copy the buffered message further desynchronizes the nodes. The positive feedback effect makes a state of synchronization unstable.

An inadequate "solution" is to move the `isend()` below the computation so that the receive is more likely to be posted before the send. While it reduces the likelihood of buffered communication, overlapping of computation and communication has been lost. Furthermore, this scheme may fail due to load imbalance or other factors. Finally, a state of loose synchronization resulting in unbuffered communication is unstable, since the positive feedback discussed in item 3 above still exists. In order to overlap computation and communication it is necessary to separate both `isend()` and `irecv()` from their corresponding `msgwait()`s by intervening computation.

One could try to patch the "solution" by doing computation after both `isend()` and `irecv()`. This scheme suffers from all of the above problems to varying degrees. Although it may be possible to reduce the likelihood of buffering, it is not possible to guarantee buffering won't occur. Whether or not a particular scheme will work "most of the time" depends on the application.

The only scheme that is guaranteed to prevent buffering is explicit synchronization of nodes after the `irecv()` but before the `isend()`. This ensures that all receives are posted before their corresponding sends.

Synchronization is unappealing practically as well as aesthetically. It affects performance both because it takes time to send the messages to synchronize and because synchronization forces all nodes to wait for the slowest. At a minimum, the time to synchronize is the latency for a single message. It is tempting to dismiss the latency cost as small for the case when messages being sent are large, but in certain well-tuned applications both latency and message transfer time may be hidden completely by doing asynchronous communication.

The importance of the second performance penalty of synchronization — forcing all nodes to wait for the slowest — depends entirely on the application. For SPMD codes on dedicated multiprocessors, the cost is probably minimal. In other environments, such as a network of workstations, synchronization can be costly. This is not an issue for NX on Intel computers, but can be an issue for NX "portability packages" running on workstations, and may be an issue for MPI.

NX programmers are familiar with two other important reasons to synchronize, neither of which is directly related to message buffering. On the Intel iPSC/860 it is necessary to synchronize to take advantage of the bidirectional capabilities of the hypercube network [8]. This scheme has the side-effect of avoiding buffering, but the performance gain from bidirectionality is far greater than that due to the lack of buffering. A second reason to synchronize is to make use of so-called "force type" messages. These messages are never buffered and are discarded if no receive is posted. They provide a slight reduction in latency on the iPSC/860. Although reduction in latency is usually cited as the reason to use forced messages, the "side effect" of eliminating buffering gives a comparable performance improvement on the iPSC/860 for message sizes exceeding approximately 3000 bytes. On the Paragon, where lower latency for forced messages has not been implemented, force types are a useful tool to ensure that messages aren't buffered. For normal message types, it is very difficult to tell when a message has been buffered. Programs using force types will fail (possibly intermittently) if not all receives are posted before their corresponding sends.

4.0 CMMD Message Buffering

CMMD [4] is the native message passing library on the Thinking Machines CM-5. CMMD provides nearly complete control over buffering. The five basic CMMD message passing primitives are `CMMD_send_block()`, `CMMD_send_noblock()`, `CMMD_send_async()`, `CMMD_receive_block()`, and `CMMD_receive_async()`.

- `CMMD_send_block()` blocks until a matching receive is posted. The calling process is free to modify the original data after it completes.
- `CMMD_send_noblock()` returns whether or not a matching receive has been posted. The calling process is free to modify the original data after it completes. It is semantically equivalent to `csend()`.
- `CMMD_send_async()` is semantically equivalent to `isend()`.
- `CMMD_receive_block()` is semantically equivalent to `crecv()`.
- `CMMD_receive_async()` is semantically equivalent to `irecv()`.
- `CMMD_msg_wait()` waits for completion of a transfer initiated with an `_async()` call.

While the CMMD routines appear similar to NX routines, they behave very differently with respect to buffering. `CMMD_send_block()` and `CMMD_send_async()` never buffer their messages. The message path for these routines is shown in Figure 5.

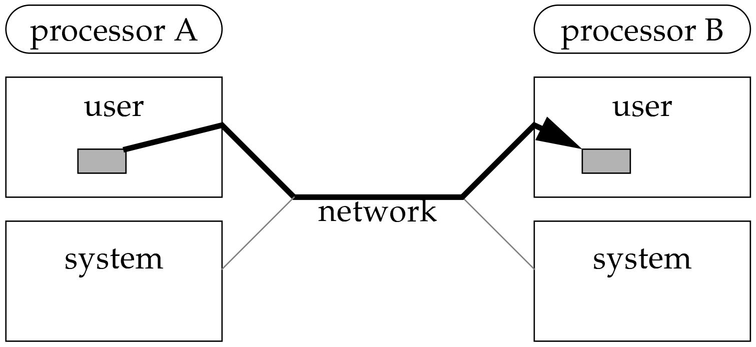

FIGURE 5. *CMMD unbuffered communication*

Because `CMMD_send_block()` does not return until a matching receive is posted, there is no need for it to buffer its messages. `CMMD_send_async()` is implemented so that its message is not sent until the destination node posts a receive, thereby eliminating the need for buffering. There is no reason why `isend()` could not or should not be implemented this way.

Applications use `CMMD_send_noblock()` when non-blocking communication is required and the buffer needs to be reused immediately. If a receive has been posted, `CMMD_send_noblock()` sends its data immediately. In contrast to `csend()`, `CMMD_send_noblock()` buffers its data on the sending side if necessary. It then sends the buffered data asynchronously and returns. This scheme avoids the problem of what routine should unbuffer the data on the receiving side (e.g., `irecv()` or `msgwait()`). `CMMD_send_noblock()` has the restriction that it cannot operate on parallel arrays, which further restricts its applicability.

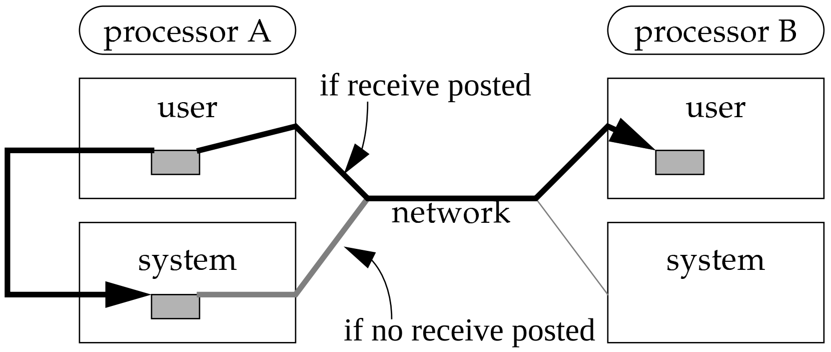

FIGURE 6. *Message path for CMMD_send_noblock()*

By using `CMMD_send_block()` and `CMMD_send_async()`, and using `CMMD_send_noblock()` only when necessary, CMMD applications can avoid buffering almost entirely.

While CMMD gives necessary control over buffering on an fast network, it would not run efficiently on a parallel computer with a slow network, such as ethernet-connected workstations. Why buffering might be desirable on such a network is explored in the next section.

5.0 PVM Message Buffering

PVM [5] is a freely-available message passing library developed at Oak Ridge National Labs and the University of Tennessee. Although it was originally developed for networks of workstations, it now runs on several tightly coupled parallel computers, including the Paragon and the CM-5.

The design of PVM reflects the fact that it was intended to run on networks of workstations. Networks connecting workstations are typically quite slow compared to those connecting nodes in high performance multiprocessors. Latency can also be high. Under these conditions, time to copy a message to or from a buffer is small compared to network transfer time. Moreover, the synchronization required by CMMD-style routines is expensive.

To operate efficiently on slow networks, PVM messages are buffered at both the sending and receiving ends. Data is explicitly "packed" at the sending node, sent, received, and then "unpacked" at the receiving node. The actual data transfer proceeds asynchronously. This scheme has a number of advantages from the point of view of performance and design. For instance, no negotiation is required with the destination node before a message is sent and no acknowledgment is required that it has been received. Furthermore, multiple non-contiguous pieces of data can be packed into the same message, avoiding the need to pay a high latency cost for each of several messages. PVM can automatically convert between data representations on different architectures. On very slow networks, the performance cost of the extra copy is relatively small. The message path for PVM message is shown in Figure 7.

However, when PVM is run on a machine with a very fast network, such as the Paragon, buffering at both ends is a bottleneck. A message is always buffered at least twice. Furthermore, PVM is sometimes built on a transport layer that buffers messages (such as NX), so that messages are often buffered three times. Indeed, peak message transfer rates using PVM on the Paragon are about 2 MB/s (7 MB/s if the fast `bcopy()` is used), an order of magnitude slower than NX messages, even though PVM uses NX as the underlying communication mechanism.

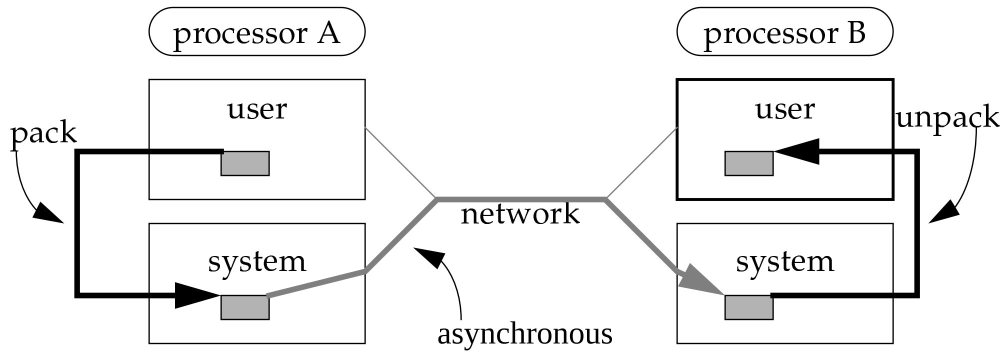

FIGURE 7. ***Message path for PVM messages***

An apparent counterexample to the above argument, that PVM is doomed to poor performance on fast networks, is the T3D, a parallel computer made by Cray Research. PVM is the native message passing library on the T3D [9] and presumably would not have been chosen if it had fundamental performance problems. The explanation is that "PVM" on the T3D is not the same as the publicly available version described above.

Cray has made some sophisticated optimizations that allow programs to avoid buffering. The first of these is "in place" message packing. When data is packed on the T3D, a program can specify that it not be copied, i.e., that it should be left "in place." This eliminates a buffer copy at the sending node. The price paid is that PVM/T3D provides no mechanism to tell when a message has been delivered so that a program cannot tell when it is safe to modify a buffer (this is equivalent to having `isend()` without `msgwait()` under NX, or `CMMD_send_async()` without `CMMD_msg_wait()` under CMMD). Programs must rely on explicit synchronization to ensure that it is safe to modify a buffer. This makes PVM programs written for the T3D nonportable. The second optimization made by PVM/T3D is that data is not actually transferred across the network until it is unpacked. This eliminates buffering on the receiving side (for both normal and in place messages). In this scheme, it is not possible to overlap computation and communication. On the other hand, buffering has already eliminated most of the benefits of PVM's asynchronous communication for fast networks, so this isn't much of a loss.

6.0 MPI Message Buffering

MPI (Message Passing Interface) [6] is a new message passing standard developed by representatives of industry, academia, and government laboratories. It is designed to be portable, and to run efficiently on a wide variety of parallel computers.

MPI provides a number of communication "modes," each with potentially different buffering characteristics. These are the standard mode, which will presumably be the most commonly used, synchronous mode, and ready mode. For each mode, there are two `send()` routines, corresponding to `csend()` and `isend()` or `CMMD_send_block()` and `CMMD_send_async()` — a program is free to reuse a buffer after the first type of send but must check for message completion after using the second kind.

The synchronous mode sends behave identically to `CMMD_send_block()` and `CMMD_send_async()`. Messages sent in ready mode are discarded unless a receive has already been posted (this functionality is equivalent to that provided by NX force types). These send modes will presumably be used only for very specialized applications and will give performance improvements on few architectures.

The discussion in the MPI document of standard mode buffering is deliberately vague. An MPI implementation may provide an arbitrary amount of buffering (including zero buffering) for standard mode sends. This allows the full range of behavior from `csend()`, which never blocks for a matching receive, to `CMMD_send_block()`, which always does. In order to be portable, programs written in MPI must assume that a standard send always blocks for a receive.

While this scheme originally appears to be a nightmare, providing the user no control over buffering, it actually may allow optimal performance on all platforms. All decisions about whether or not data should be buffered can be made by the MPI implementor. On a fast network, presumably MPI will be implemented without buffering. On a slow network, it may buffer extensively. All that is required of an application programmer is that he or she write a program which does not deadlock even if all sends block for receives. What about cases where a program requires a non-blocking send? The final MPI specification (not available at the time of this writing, which is based on the November 1993 MPI draft) includes a buffered send to address this issue.

Whether or not MPI will actually be implemented efficiently on a variety of platforms remains to be seen. One problem is that MPI may be built on top of a transport layer that buffers messages. For instance, the publicly available, portable version of MPI runs on top of NX on the Intel Paragon, so that it inherits the buffering characteristics of NX.

A potentially far worse problem, however, arises because MPI generally has to communicate more information than the underlying transport layer provides. For instance, NX provides only a 32-bit tag to identify messages. MPI messages are identified by a tag, a group and a context. They may also contain information on the type of data in the messages. Thus an MPI message cannot be translated directly into a single NX message. To send a single MPI message, an MPI implementation must either send more than one NX message or must copy MPI information and user data into a larger buffer. This must be done at both the sending and receiving ends, so that a message is buffered twice. This is what happens for the portable version of MPI running on the Intel Paragon and the CM-5. As shown in Section 7, corresponding effective bandwidth for MPI is much lower than that of native message passing libraries on these platforms.

The problems described above apply only to the publicly available portable version of MPI. Native implementations of MPI can achieve excellent performance. In particular, the implementation of MPI on the IBM SP-1 achieves essentially the same latency and bandwidth as IBM's proprietary message passing library, EUI-H [10].

7.0 Benchmarks

Benchmarks that measure communication performance can be misleading because they usually measure transfer rates for unbuffered messages. It is quite difficult to measure directly buffered communication rates when buffering is nondeterministic. In the following, I report (when possible) measured rates for unbuffered communication and for buffer copying, and compute an effective buffered communication rate, which is the harmonic mean of the other two. In the case of PVM, I measured a communication rate that includes explicit buffering (packing and unpacking). For MPI, the measured bandwidth reflects the implicit buffering discussed in the previous section.

The benchmark I used to measure latency and bandwidth is a token ring benchmark, where a message is passed around a loop of several processors. The benchmark is structured so that receives are always posted before the corresponding sends so that messages are never buffered implicitly. Of course PVM messages are always buffered twice explicitly: this scheme eliminates only unnecessary buffering in the underlying transport layer. Message copy rates are measured by timing the routine that does the copying - `msgwait()` in the case of NX and `CMMD_send_noblock()` for CMMD.

Table 1 summarizes the results from the CM-5 (CMMD 3.1), Paragon (OSF/1 R1.1.4), and iPSC/860 at the Numerical Aerodynamic Simulation facility at NASA Ames Research Center. PVM 3.2 runs on top of NX and CMMD on the Paragon and CM-5, respectively. The version of MPI used was the freely available (and unoptimized) version that is under development at Argonne National Laboratory and Mississippi State University. On the Paragon, MPI runs on top of a device independent layer that is built on NX. On the CM-5, the device independent layer runs on top of Chameleon, another portable message passing library, which runs on top of CMMD. Latency and bandwidth results are "best case," rather than derived a best fit the message size/bandwidth relation. Effective bandwidth for buffered messages is computed from bandwidth and copy speed. All results are accurate to within 5%.

TABLE 1. Performance of NX, CMMD, PVM and MPI on the Paragon, CM-5 and iPSC/860

	latency (asymptotic)	**bandwidth**	**copy speed**	**effective bandwidth (buffered)**
Paragon OSF/NX	92 us[a]	35 MB/s	4.8 MB/s	4.2 MB/s
Paragon OSF/NX (fast bcopy)	94 us[a]	35 MB/s	69 MB/s[b]	23 MB/s
Paragon SUNMOS/NX	63us[a]	167 MB/s	19 MB/s	17 MB/s
Paragon PVM 3.2	387 us[c]	2.0 MB/s (buffered)	NA	NA
Paragon PVM 3.2 (fast bcopy)	388 us[c]	7.8 MB/s (buffered)	NA	NA
Paragon MPI	342 us	2.2 MB/s (buffered)	NA	NA
Paragon MPI (fast bcopy)	343 us	7.7 MB/s[b] (buffered)	NA	NA
iPSC/860 NX	72 us[d] (minimum)	2.8 MB/s	19 MB/s	2.4 MB/s
CM-5 CMMD[e]	80 us	8.7 MB/s	9 MB/s[b]	4.4 MB/s
CM-5 PVM	573 us	3.2 MB/s	NA	NA
CM-5 MPI	405 us	2.6 MB/s	NA	NA

a. Zero-byte latency. Asymptotic latency is about 117 us under OSF/1 and 90 us under SUNMOS.

b. Peak bandwidth. Bandwidth varies depending on exact size of buffer (as low as 4.8 MB/s Paragon/OSF buffer copy, 1.5 MB/s CM-5/CMMD buffer copy, 2.0 MB/s Paragon/PVM bandwidth)

c. Asymptotic latency. Latency for zero byte messages is 474 us.

d. Latency for message to neighboring node.

e. CM-5 results are for serial arrays, not parallel arrays.

The results in Table 1 show communication performance for contrived benchmarks designed to measure low-level performance. The performance implications of buffering for real applications are highly complex and vary widely with type of application, algorithm, implementation, hardware platform, etc. Applications which are most affected by buffering are those which have large communication requirements, which use long messages, and which overlap computation with communication. Other factors, such as whether an application hides latency by overlapping computation with communication, are also important. Because the performance effects of buffering are so application- and platform-dependent, and because it can be quite difficult to isolate the effects of buffering, specific numbers do not have much meaning.

8.0 Conclusions

I have shown that message buffering can be an important performance issue when network speed approaches the speed at which a processor can handle data. How important buffering is depends on the system balance (ratio of network bandwidth to local memory bandwidth), the design of the message passing library being used, and the application being run.

Message passing libraries differ widely in how much control over buffering they give to the user, and in how much buffering is required, at a minimum, to send a message. I have discussed NX message buffering in detail because it is the most complex, and illustrates a range of issues. However, the effect of buffering on Paragon application performance is modest.

Buffering is probably most important for MPI and PVM, message passing libraries that promise portability over a wide range of systems. On fast networks, however, their portability comes with a price, part of which is a significant increase in message buffering. In the case of MPI, it may be possible to eliminate buffering, but the current portable implementation does not do that. PVM, on the other hand, does not appear to offer the possibility of excellent performance on fast networks.

9.0 Acknowledgments

I had useful discussions with Thanh Phung, Rob van der Wijngaart, Ed Hook, Bernard Traversat, Bill Nitzberg and Sam Fineberg at NAS which helped clarify some of the issues described in this report.

References

[1] Bailey, D., Barton, J., Lasinski, T., Simon, H., editors. *The NAS Parallel Benchmarks* NASA Technical Memorandum 103863 NASA Ames Research Center, Moffett Field, CA, July 1993.

[2] Bailey, D., Barszcz, E., Dagum, L., Simon, H., *NAS Parallel Benchmark Results*, Technical Report RNR-94-006, NASA Ames Research Center, Moffett Field, CA, March 1994.

[3] Intel Supercomputing Systems Division, *Paragon OSF/1 C System Calls Reference Manual*, Intel Corporation, Beaverton, OR, 1994.

[4] Thinking Machines Corporation, *CMMD Reference Manual*, Cambridge MA, 1993.

[5] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory Report TM-12187, Oak Ridge, TN, 37831.

[6] Message Passing Interface Forum, *Document for a Standard Message Passing Interface*, University of Tennessee, November 1993.

[7] Maccabe, B., and McCurley, K., *SUNMOS for the Intel Paragon*, Sandia National Laboratory, 1993.

[8] Seidel, S., Lee, M., and Fotedar, S., *Concurrent Bidirectional Communication on the Intel iPSC/860 and iPSC/2*, CS-TR 90-06, Michigan Technological University, Houghton, Michigan 49931-1295.

[9] Cray Research, Inc., *PVM and HeNCE Programmer's Manual*, SR-2501 3.0, Mendota Heights, MN, November 1993.

[10] Sam Fineberg, personal communication.