Appears in: Proceedings of the 9th International Conference on Supercomputing, July 3-7, 1995, Barcelona, Spain

Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers

T. Stricker[1], J. Stichnoth[1], D. O'Hallaron[1], S. Hinrichs[1], and T. Gross[1,2]

[1]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[2]Institut für Computer Systeme
ETH Zürich
CH 8092 Zürich

Abstract

Synchronization is an important issue for the design of a scalable parallel computer, and some systems include special hardware support for control messages or barriers. The cost of synchronization has a high impact on the design of the message passing (communication) services. In this paper, we investigate three different communication libraries that are tailored toward the synchronization services available: (1) a version of generic *send-receive* message passing (PVM), which relies on traditional flow control and buffering to synchronize the data transfers; (2) message passing with *pulling*, i.e. a message is transferred only when the recipient is ready and requests it (as, e.g., used in NX for large messages); and (3) the decoupled *direct deposit* message passing that uses separate, global synchronization to ensure that nodes send messages only when the message data can be deposited directly into the final destination in the memory of the remote recipient. Measurements of these three styles on a Cray T3D demonstrate the benefits of the decoupled message passing with direct deposit. The performance advantage of this style is made possible by (1) preemptive synchronization to avoid unnecessary copies of the data, (2) high-speed barrier synchronization, and (3) improved congestion control in the network. The designers of the communication system of future parallel computers are therefore strongly encouraged to provide good synchronization facilities in addition to high throughput data transfers to support high performance message passing.

1 Introduction

The trend to use off-the-shelf processors as compute engines for supercomputers implies that the communication system distinguishes parallel systems. For example, the Cray T3D, Intel Paragon, IBM SP2, TMC CM-5, and networks of workstations are all based on standard microprocessors but use dramatically different communication systems.

This research was sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152. Computational resources were provided in part by the Pittsburgh Supercomputing Center (PSC).

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

One of the key issues in the design of a communication system is what services or primitives to include. For example, both the T3D and the CM-5 include special hardware support for fast synchronization (in the form of hardwired barrier trees), whereas such support is absent in the Paragon and networks of workstations.

The importance of synchronization hardware goes beyond the obvious use for explicit synchronization. When compiling a High Performance Fortran (HPF) program, it is often necessary to insert synchronization steps to ensure that the communication operations of one statement (or group of statements) do not conflict with the communication operations of another. A programmer who directly controls the operation of the nodes of a parallel system (i.e., who does not use a parallelizing compiler) faces the same problems. For example, when implementing a transpose, judicious use of synchronization speeds up the routing and data transfers[8], and when switching between different connection setups for different phases of a computation, synchronization is necessary to avoid race conditions[6].

The availability of fast synchronization impacts the performance of the message passing system that provides the basic data transfer services. In the absence of a fast global synchronization facility, the message passing system must rely on transport-level acknowledgements to ensure that messages have been delivered.[1] Such transport-level acknowledgements are just another type of message. If there exists a fast global synchronization mechanism, the compiler or the programmer can in many cases replace such transport-level acknowledgement messages with a single global synchronization step, significantly reducing the overhead associated with inter-node communication.

To get a better understanding of the importance of synchronization in practice, we investigate three message passing systems that are adapted to three different scenarios. Section 2 describes these systems and discusses their reliance on synchronization. Section 3 further develops the ideas of data transfer and synchronization. We measure key parameters of these communication systems to quantitatively assess the impact of good or bad synchronization, and discuss how the data transfer and synchronization services can be decoupled.

To factor out the differences in implementation technology of the various parallel systems, we use a single parallel system as our testbed. It would be difficult to separate clock rate differences from architectural differences if we compared message passing libraries with synchronization on one system against message passing li-

[1]A system may also include link-level acknowledgements to deal with problems at the link level, e.g. loss or corruption during a transfer. However, only a transport-level acknowledgement can guarantee that a message has been buffered or stored at the recipient.

braries without synchronization on another system. Since we are interested in the impact of architectural support, we choose a powerful system that provides a wide range of options, the T3D. In Section 4, we report on the performance of the different communication systems on the T3D. We discuss the results and present our conclusions in Section 5.

2 Background

In message passing parallel systems, either the user or the compiler explicitly moves data from one node to another, thereby "renaming" the data. That is, as data is moved from one node to another, its name (address) is changed. In contrast, shared address space systems preserve the name of a data item as it is moved to another node. A data item may appear in the local memory (cache) of a node after a transfer, but its name (address) is still the same as it was before the transfer. The relative advantages of both machines have been discussed in numerous papers, and there exist a number of machines for either style. There are also proposals to integrate both approaches in a single machine. This paper concerns itself solely with message passing communication.

2.1 Control and data transfer messages

The key function of the communication system for a parallel computer is to transfer data, as well as to provide explicit synchronization. However, transferring data can be tricky: if data are sent too early, the data may have to be buffered. If they are sent too late, the receiver nodes must wait.

Conventional message passing program use the same mechanism for control and data transfers: they send messages. We classify messages as either control or data messages, based on their content and purpose in the program.

Most *control messages* are linked to synchronization. The reception of such a control message contains an assertion that some data is ready, that some buffer is available to receive more data, or even that some data made it or did not make to its destination, in case our communication system requires retransmission. Therefore, all messages implicitly issued by protocols are control messages. Despite its implementation in hardware, we classify a barrier synchronization as a form of control message, since it also communicates assertions between nodes.

Messages that contain data are called *data transfer messages*. They typically contain a significant amount of data that is moved between nodes within the parallel program. Since the amount of data is usually too large to be stored in special-purpose registers, the message passing system must pay attention to the *buffering* of data in memory. Typically the source of the data is in the user process' memory on one node, and the destination is in the user's memory at another node.

In some rare cases messages contain data as well as control. We call these *hybrid messages*, but we suggest that such a message is classified and handled either as a data transfer message or as control message, depending on the amount of data it contains. Hybrid messages can be generated in two cases. Many flow control and other protocol implementations piggyback control information such as acknowledgement or choke packets onto data messages flowing into the other direction. In this case, the messages contain data and must be handled like data transfer messages. Hybrid messages also occur in global data parallel operations that perform little data transfer and computation but synchronize the machine. An example of such a parallel operation is a scan or reduction. These messages can be handled like a control message, since they do not contain large amounts of data.

2.2 User models

As stated above, we consider all communication that *renames* the data to be *message passing*. We include the <shmem_put> remote store operation on a T3D[2] in this category, in contrast to operations on a shared memory machine like the Silicon Graphics Power Challenge, where node programs do not distinguish between local and remote addresses.

There are many models of message passing. We provide here a broad classification based on the degree of involvement by the sender and the receiver node.

Rendezvous model The sender and receiver perform a handshake to transfer a message. Consequently, every data transfer enforces a complete synchronization between the sender and receiver. The model is popular for theoretical work and forms the basis of Occam [12]. One of the drawbacks of this model is the overly tight synchronization imposed on sender and receiver for the transfer of every data word. Therefore, this model is not considered any further in this paper.

Postal model Both the sender and receiver participate in a message exchange. The sender performs a *send* operation and the receiver issues a *receive* operation. These operations can be invoked in either order. That is, messages can be sent at any time without waiting for the receiver; the data is buffered until the receiver accepts it [2].

The postal model is implemented in the message passing libraries of many parallel systems, e.g. PVM [4], MPI [17], and NX [11]. However, there are important implementation differences. For example, NX may delay the transfer of data for large messages until the receiver is ready. When a node issues a receive operation, the communication system *pulls* the data over from the sender. This optimization simplifies buffer management and improves performance (since there is never a need to copy data on the receiver side). The different optimizations and their impact on the precise semantics of the send and receive calls are beyond the scope of this paper and are discussed in [14].

Deposit/fetch models Only one of the two nodes (sender, receiver) actively participates in the data transfer. In the deposit model, the sender "drops" the data into the address space of the receiver, without participation of the receiver process. The fetch model is the dual of the deposit model: the receiver retrieves the message data. Both models allow a clean separation of control and data messages. In the deposit model, control messages or hardware barriers are used to deal with explicit synchronization, and data messages are sent only when the receiver has signaled its willingness to accept them. Similarly, in the fetch model, control messages establish when data is ready to be fetched, and then the data transfer can take place.

This model can be implemented in software with active messages [16], where the sender node just sends the data, and a handler is invoked on the receiver to move the data to its final destination. However, our model suggests that a general control transfer in the form of an RPC should be avoided and that implicit synchronization is sufficient. The benefits of separating control and data transfers as an optimization for RPC in client/server computing is argued in [15]. The direct deposit model suggests that the handler is implemented directly in hardware as, e.g., by the custom circuitry to handle

[2] This operation takes a local address in the address space of one node and a remote address in the address space of another, and moves *a copy* of the data from one address space to another. See [3] for more details.

<shmem_put> and <shmem_get> primitives on the T3D [3]. On this machine, custom hardware moves data across the network whenever a node explicitly copies data from the local address space to the remote address space of a selected other node. For this paper we use <shmem_put> exclusively and we refer to the hardware executing the transfers at the receiver side as the *deposit engine*.

We select three representative message passing libraries, implementing the postal and the deposit models:

PVM, standard message passing: An example of the classical postal model with traditional sends and receives.

RRMSG, request-response message passing: A variant of the postal model, optimized by buffering data at the sender and "pulling" large messages only when they are needed.

DMSG, deposit message passing: A message passing implementation that exploits the decoupling of control and data transfers and that relies on decoupled synchronization services.

This last aspect, separating synchronization and data transfer, is reviewed in more depth in the next sections.

2.3 Synchronization properties and buffering

Any message passing system inherently provides a simple one-way synchronization between the sending and the receiving process: data must be produced and sent before it can be received. But the synchronization does not go in the other direction; it is common in message passing systems for a node to finish sending a message and continue before the receiving process has invoked the receive routine. A long message cannot be allowed to fill the network and needlessly consume communication resources; in fact, most networks are only deadlock free under the assumption that the receivers continuously and unconditionally remove incoming messages [5]. Thus in practice, some storage space (on either the sender or the receiver) for messages is necessary to allow sends to complete before the corresponding receive is posted. Such storage space can be supplied through either user program or system buffers.

In practice, storage space for communication buffers is bounded, thus requiring long messages to be broken down into smaller packets. As a result, a flow control protocol must be used to send requests for buffers, replies, and acknowledgments in messages. With such a protocol, a message passing system can ensure that buffers do not overflow.

The flow control protocol can be used to provide synchronization (beyond buffer management), but doing so is expensive, because each data transfer has implications on the management of buffer storage. The coupling of data transfer and synchronization increases the cost of the data transfers, since each transfer may involve storage management decisions.

3 Decoupling synchronization and data transfers

Separation of synchronization and data transfer is the key to communication performance in parallel supercomputers. There are three principal reasons to separate synchronization from data transfers:

1. If synchronization and data transfers are coupled, a control message may involve costly storage management operations, since those are necessary for data transfers.
2. If no system buffers are provided and the data transfers are fully under user control (desirable to avoid copying), then we cannot rely solely on the synchronization provided by data transfers.
3. If we want to target the data directly to its final destination (desirable to avoid copying), synchronization is required prior to data transfers.

Decoupling synchronization from data transfers creates additional opportunities for improvement in the communication system. Compiler-generated parallel programs can manage the buffers involved themselves and can include synchronization code. Therefore, these programs do not have to rely on the synchronization provided by data transfers. It is no longer necessary to acknowledge each individual buffer allocation request; acknowledgements must be provided for a buffer *pool* that is managed as a unit by the compiler.

Combining protocol messages

With the global knowledge of a compiler, protocol messages of several nodes can be combined, resulting in a drastic reduction in the number of protocol messages (e.g., for flow control). For example, a complete exchange (or personalized all-to-all communication) among n nodes implies n^2 data transfers, one for each sender-receiver pair. A connection-oriented request/reply flow control protocol must send $O(n^2)$ messages to request and acknowledge the data transfers between all senders and receivers. However, since the compiler has global knowledge about the communication step, it can use a simple tree reduction to propagate the acknowledgements to all processors with a total of just $2n$ messages in $2 \log n$ steps. Figure 2 depicts the impact on the T3D; we delay the discussion until the end of this section.

Using barriers and specialized messages

The flow control messages for many communication patterns are highly regular. Since the structure of the pattern is known, each node has detailed knowledge about which resources (buffers) are involved. Therefore, the flow control messages convey only sequencing information, so empty messages can be sent, provided they are identified as control messages. Specialized hardware for short messages and message combining exists in some supercomputers. Such reduction or synchronization trees are the optimal candidates to propagate the synchronization and flow control information among the computation nodes.

Traditionally, the message passing hardware is designed to transfer *large amounts* of data efficiently. Control and synchronization have different characteristics (small size, immediate use at receiver) and should be handled by different mechanisms; e.g., barrier synchronization hardware or combining trees.

Avoiding copying and buffering

Figure 1 shows all possible copies due to buffering steps within a general purpose library like PVM. The steps are labeled with the terms used later in our measurements of PVM and RRMSG overheads. Note that all buffers are in user space; there are no copies due to protection domain crossings on supercomputer nodes dedicated to a single user. PVM implementations require buffering steps either at the sender or the receiver node to maintain the semantics of a message passing model with implied synchronization properties of every data transfer. In such a model, send operations are legal even before the matching receive is posted. Furthermore, the built-in buffer management in PVM and RRMSG can only handle

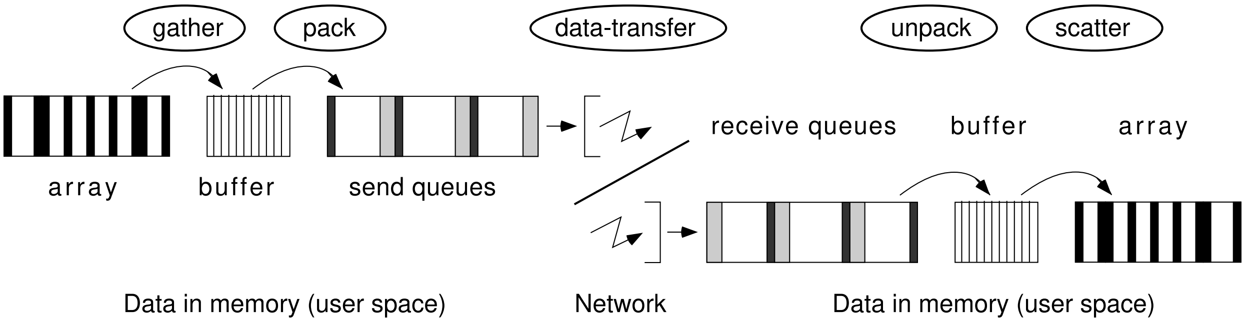

Figure 1: Possible copies of data occurring in traditional message passing (e.g. PVM) due to bad synchronization and buffering.

contiguous blocks of data, resulting in a mandatory copy to gather possibly strided or indexed data elements out of the array structures at the sender. Similarly, a scattering copy at the receiver is needed to deliver the data elements to their final destination.

On parallel supercomputers, interconnection networks can transfer data at rates close to the local memory bandwidth. Buffering at the end points through copying is therefore a limiting factor to communication performance, since the traffic to and from data buffers traverses the memory bus multiple times. A quantitative study of memory system performance in message passing system can be found in [**?**].

In our decoupled deposit message passing system, all messages are taken directly from memory (user space) at the sender and are automatically directed to their final destination in memory at the receiving end. If buffers are used, they are under compiler/user control while synchronization messages are generated separately. In the deposit model, we cannot rely on the synchronization properties associated with data transfers because there exists a possibility of live data being overwritten. Some additional form of synchronization is needed, and our DMSG library relies on hardware barriers for synchronization. The RRMSG library uses short protocol messages for the same purpose.

Based on these three observations, the Fx compiler for a dialect of HPF [7] achieves its best performance with the DMSG library. Using a global picture of the communication steps, the compiler back end replaces the synchronization effects of numerous data transfers by more efficient collective communication primitives (e.g., barrier synchronizations). The resulting message passing system is able to focus on and optimize the data transfers. No buffering services are provided in the message passing library, since the buffer management is taken care of by the compiler and most of the synchronization is done globally by barriers.

Cost of synchronization

Figure 2 depicts the cost of different synchronization options for all-to-all communication on a T3D. Three different ways to propagate synchronization information among all communication nodes are considered. In the first method (ctrl-msgs), each data transfer is accompanied by a request, a reply, and an acknowledge control message, thus ensuring that the buffers can be managed easily. In the second method (ctrl-msg-tree), all requests, replies, and acknowledgements are carried out in a collective communication operation, using combining trees. The third method (hw-barrier) invokes the subset barrier hardware of the machine.

These measurements of control message exchanges illuminate the importance of paying attention to the cost of control messages. Exchanging $O(\log n)$ messages instead of $O(n^2)$ resulted in significantly improved performance, as expected. Although both the hw-barrier and ctrl-msg-tree implementations use the same <shmem_put> remote store mechanism, the further improvement for hw-barrier is due to the dedicated barrier synchronization hardware. The results show that an optimized path for zero length messages is a worthwhile option to handle control messages.

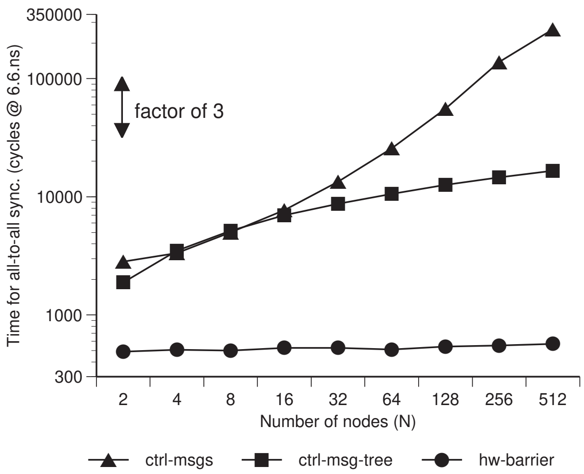

Figure 2: Costs of different mechanisms for flow control in an all-to-all communication step.

4 Evaluation

We chose two application kernels, 2D-FFT and SOR, to run on the T3D, to quantitatively verify our claim that decoupling synchronization and data transfer is the key to message passing performance and application scalability.

Parallel programs are sometimes classified as either embarrassingly parallel, coarse grain parallel, or fine grain parallel. In the first case, communication performance does not matter, and networks of workstations or arrays of cheap signal-processors are sufficient to achieve good performance. But in the latter two classes of programs, communication performance is a function of the problem size and has significant impact on overall performance. Our application kernels and their input sizes (both with sixteen million elements, i.e. a $4k \times 4k$ 2D-FFT and a $4k \times 4k$ SOR) are motivated by two supercomputing grand challenge applications that are currently under investigation at CMU. The two application kernels differ in the amount of data sent and most importantly in the complexity of the communication pattern. The SOR kernel has a larger amount of data to be exchanged, but the data is contiguous and only sent to two neighbors. The size of messages is not affected by the number of nodes used and the total amount of data sent increases with the machine size. By contrast, the 2D-FFT performs a dense all-to-all communication pattern with smaller messages. Appendix A provides more details.

We measure the computation and communication performance of our kernels running under three different message passing systems.

PVM 3.3 The PVM library provides the conventional *send* and *receive* primitives, which combine synchronization and data transfer services. It is important to note that PVM 3.3 is a specialized library version of PVM for communication completely within the T3D distributed memory parallel computer [9], as opposed to a more general version that is also capable of communicating over a network of heterogeneous systems. This PVM library was written and optimized by the vendor under the assumption of *exclusive processor use* and *full access to the communication system*. Thus, all our programs run in physical memory and communicate directly from user space to user space. The results reported for this implementation of PVM are also indicative of the performance that can be obtained with a native implementation of MPI, the evolving message passing standard.

DMSG: Deposit model message passing Our second communication service is based on the idea of service decomposition into control and data transfers. DMSG allows deposits into the address space of any other node. At the destination node, the deposit engine executes the remote store asynchronously, without any participation from the receiving node. The sender keeps track of when remote stores complete. For all further synchronization, the compiler relies on the hardware barrier, which can synchronize all processors within a few microseconds.

RRMSG: request/response message passing With the third library we study the case of machines where control and data transfers are optimized separately, but there is no direct support for synchronization through hardware barriers (e.g., Intel Paragon or SP2). We refer to this model as "RRMSG". This style mirrors closely the operation of NX for long messages on a Paragon running under OSF/1 or an iPSC860. For every data transfer there are three control messages: a *request* by the sender for transfer, a *response* in which the receiver confirms buffer reservation, and a final acknowledgement by the receiver confirming the reception of the data. These extra control messages are necessary to free the user from potentially complicated buffer management decisions. In some cases, buffering and copying can be avoided with the exchange of a few synchronization messages. The RRMSG model incorporates ideas from both worlds: the conventional message passing systems (e.g., PVM) and the highly efficient direct transfers of the deposit model (e.g., DMSG).

All three implementations rely ultimately on the built-in T3D *remote store* commands to implement data transfers. The differences that we report in the next section are due to the different synchronization schemes, emphasizing the importance of adequate synchronization support in parallel systems.

4.1 Performance impact of fast synchronization

The scalability of application kernels depends critically on the message passing style and hardware support for synchronization. In Figures 3, 4, 5, and 6 we graph the total aggregate performance and the Per-node performance for the different messages passing systems on different machine sizes. These performance figures include all communication overheads due to parallel execution.

The performance of the 2D-FFT kernel executing on PVM falls sharply as we move to machines with more than 64 nodes, due to lower communication speeds. Better scalability and good sustained performance can be achieved with the decoupled models that are the basis of DMSG and RRMSG. (See Figures 3 and 4.)

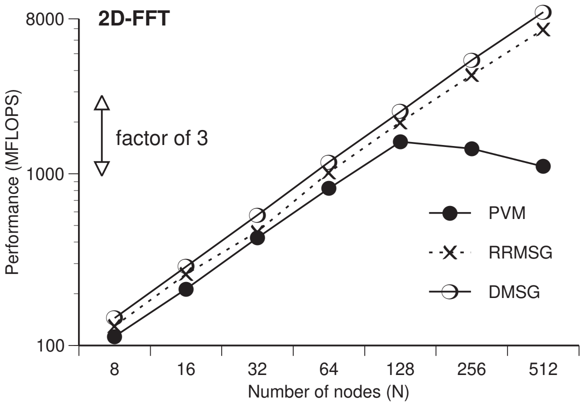

Figure 3: Aggregate performance of a 4096 $\times$ 4096 point 2D-FFT for different machine sizes, graphed on a log-log scale.

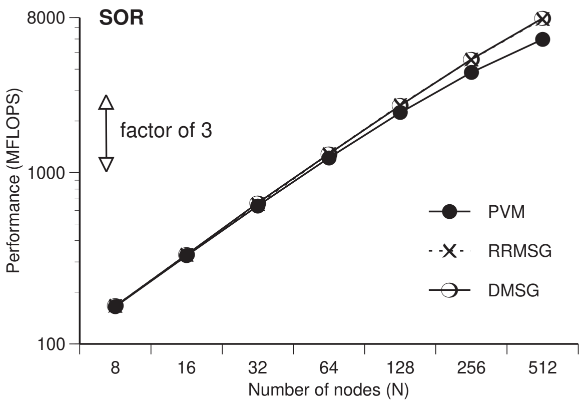

Figure 4: Aggregate performance of a 4096 $\times$ 4096 element SOR for different machine sizes, graphed on a log-log scale.

Due to separate synchronization, DMSG is able to make use of the good architectural support for hardware barriers and fast direct deposit data transfers. In DMSG, the deposit model library, the per-node performance remains near peak for both applications (2D-FFT and SOR), even at machines sizes as large as 512 nodes, as depicted in Figures 5 and 6.

For a more detailed analysis of communication performance, we measured the achievable throughput of both applications during communication phases independent of the computation part. These numbers can be used to relate the net throughput (after all overheads) to the peak communication performance of the T3D hardware, specified at about 130 MByte/s per link, or about 65 MByte/s for each of the paired nodes, if both processors communicatate simulatanously.

The communication performance is more important for applications with transpose steps (e.g., FFTs) than for applications with simple next-neighbor overlap exchanges (e.g., solvers like SOR). The performance improvement noted by such applications is mainly

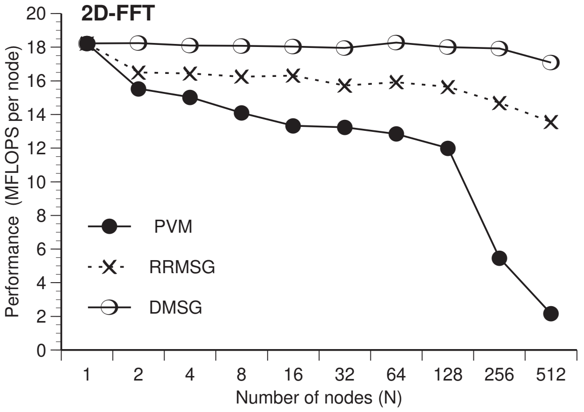

Figure 5: Per-node performance of a 4096 $\times$ 4096 point 2D-FFT for different machine sizes.

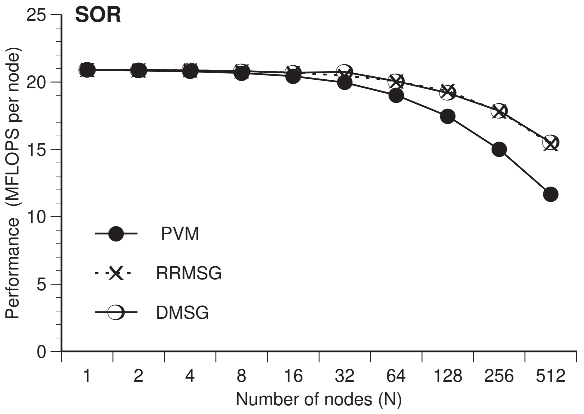

Figure 6: Per-node performance of a 4096 $\times$ 4096 element SOR for different machine sizes.

due to the substitution of a large number of control messages by hardware barriers in the all-to-all communication step. Furthermore, well-synchronized programs can benefit from the better performance of direct deposit data transfer for strided local access pattern.

The aggregate communication transfer rates are shown in Figures 7 and 8, and the per-node communication transfer rates are shown in Figures 9 and 10. Communication performance in PVM

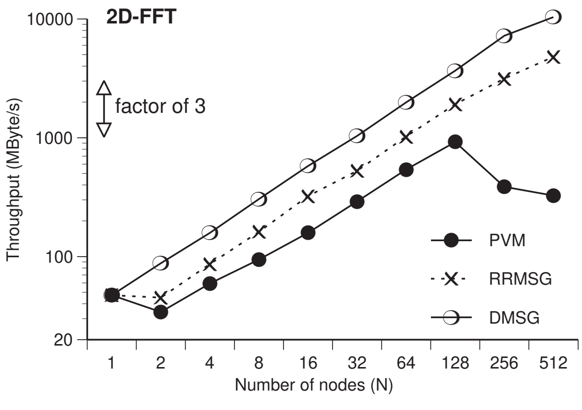

Figure 7: Aggregate communication performance of a 4096 $\times$ 4096 point 2D-FFT, graphed on a log-log scale.

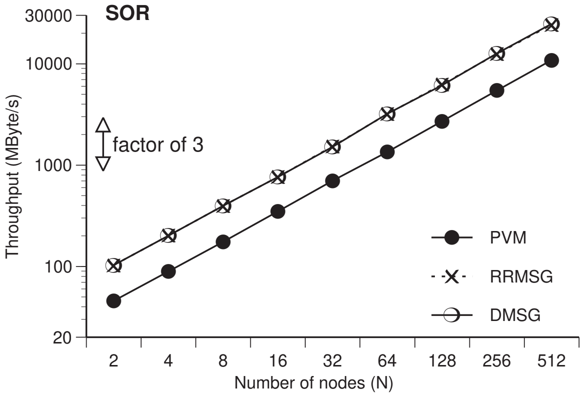

Figure 8: Aggregate communication performance of a 4096 $\times$ 4096 element SOR, graphed on a log-log scale.

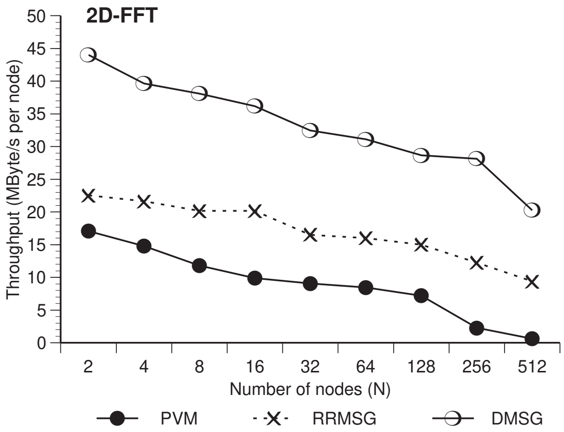

Figure 9: Per-node communication performance of a 4096 $\times$ 4096 point 2D-FFT for different T3D machine sizes.

seems limited even with large problem sizes and smaller machines. For larger machines, PVM seems completely limited by the constant per-message overhead spent on buffer allocation and the implicit synchronization of each data transfer. The overheads for control messages are also present in RRMSG, but the per element overheads can be avoided, and the constant overhead seems smaller. Therefore the performance does not drop as fast as the number of nodes increases, resulting in improved scalability.

In DMSG we incorporate all advantages of decoupled synchronization. We use hardware barriers, direct deposit for data transfers, and additional barriers for congestion control in the network. Direct deposit eliminates buffering as well as the gather/scatter copies. With fast barriers on the T3D, there is minimal synchronization overhead, and high message transfer rates make application performance scale up without loss as more processing nodes are added.

For the SOR application kernel, which uses a simple next neighbor pattern and transfers large contiguous blocks of memory, we still note significant performance differences. The reduced synchronization overheads of DMSG and RRMSG are less visible, but with PVM, the data is still copied several times due to the standard interface to the library. While DMSG and RRMSG achieve throughput numbers from 48 to 50 MByte/s, PVM is limited to 20 MByte/s, due to copying.

At transfer rates of 20 MByte/s per node, there is no network congestion for all-to-all communication on a T3D, except for the ba-

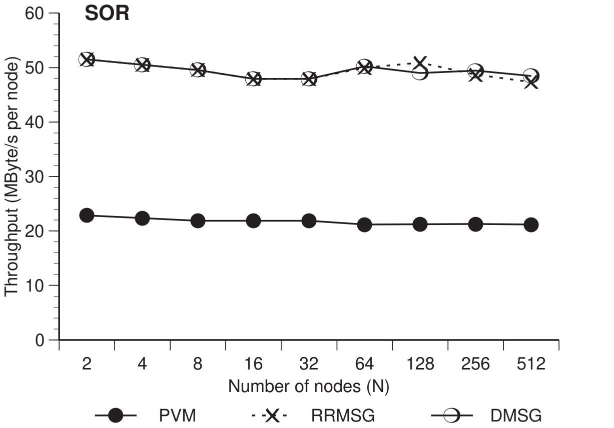

Figure 10: Per-node communication performance of a 4096 $\times$ 4096 element SOR for different T3D machine sizes.

sic reduction of the link speed by a factor of two.[3] Despite the higher transfer rates of DMSG, the congestion in the router can be minimized with additional barriers for machine sizes up to 1024 according to a method described in [8]. Congestion control is impractical for PVM and RRMSG because the protocol messages synchronize each sender and receiver pair independently rather than with a global barrier.

4.2 Detailed analysis of communication time

To understand the full impact of the message passing style on the performance of our applications, we must quantify the times spent in communication-related work in more detail and investigate whether performance is lost through constant per-message overhead (e.g., startup or protocol overheads) or through linear per-byte cost (e.g., copies during buffering). To see the impact of data copies in the different styles, we examine *large* problem sizes in particular. We expect the constant per-message overheads to be more more visible at *small* problem sizes.

Figures 11, 12, 13, and 14 show the fraction of time our application kernels spend in computation and in communication. On the left, each figure depicts the total execution time in seconds. On the right, the communication time is further broken down into: *data transfer*, the time to actually transfer the data across the network; *barriers/control*, the time spent in synchronization; *gather/scatter*, the time required to gather all data into one contiguous block and scatter it into its final location; and *pack/unpack*, the fraction of time PVM spends to prepare the messages (pvm_fpack, pvm_funpack calls). Figure 1 in Section 3 illustrates this process for PVM.

2D-FFT: large problem size

As Figure 11 indicates, for the larger problem size of 2D-FFT, the DMSG communication is almost a factor of four faster than PVM. The RRMSG case without hardware barriers is still about a factor of two faster than PVM.

The amount of time spent doing the actual transfer of data across the network is virtually the same in all cases. The data transfer part of the DMSG library is somewhat slower since it includes the gather/scatter step, depositing strided data directly to its final destination. The RRMSG and PVM cases pay a significant additional cost to separate the two data transfer and gather/scatter. Furthermore,

[3] This reduction is due to the node architecture of the T3D. A single network access point is shared among a pair of processing nodes.

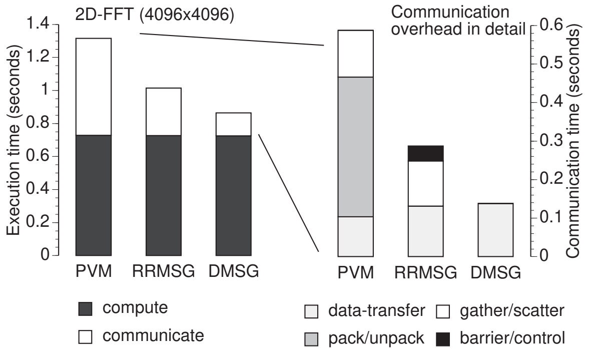

Figure 11: 2D-FFT (4096 $\times$ 4096) for different message passing models: detailed breakdown of execution time [128 nodes].

in DMSG and RRMSG, all buffer management can be done by the compiler, thus incurring no additional overhead during runtime. For PVM, the measurements quantify the cost of buffer management and copy overheads in the pack/unpack times.

The amount of time spent in synchronization and protocol processing is so small for DMSG, which uses fast hardware barriers for synchronization, that it is not visible in the figure. The time is larger for PVM, but because the PVM data transfer functions are integrated, we cannot separate the synchronization/protocol costs from the pack/unpack costs. The RRMSG synchronization cost includes the cost of a control message that synchronizes every data transfer in advance, which adds up to much more time than a single barrier.

SOR: large problem size

Not all applications have dense and communication-intensive patterns like the transpose in 2D-FFT. In some applications, the nodes just exchange an overlapping region of data with their immediate neighbors; SOR is such an application. With very few messages exchanged in the large SOR case, the benefits of decoupled synchronization and fast barriers come indirectly through less copying rather than directly through elimination of overheads, as depicted in Figure 12.

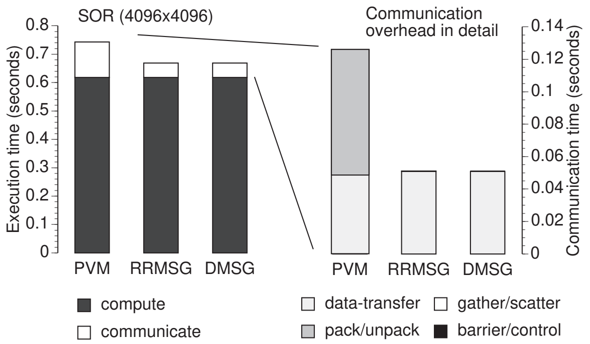

Figure 12: SOR (4096 $\times$ 4096) for different message passing models: detailed breakdown of execution time [128 nodes].

While DMSG and RRMSG transfer the contiguous blocks of data end-to-end without copying, PVM does not have enough synchronization information to store data directly into its destination (i.e., it cannot risk overwriting live data), and it seems to make at least one copy to an intermediate buffer. This is visible in the measured overall communication time. For the vendor implementation of PVM, the measured breakdown into transfer time and buffering overhead in the graph is meaningless. The measurements indicate that some of the actual data transfers and buffering are delayed until the pvm_unpack calls are made.

2D-FFT: small problem size

Figure 13 shows the 2D-FFT performance on a small problem size (256×256). Because of the large number of small messages, the constant per-message overhead for synchronization protocol and buffer management dominates the PVM and RRMSG times. With a relatively small amount of computation to do, the overall performance of the 2D-FFT is critically dependent on the message passing system used: DMSG is about 30 times faster than PVM, while RRMSG outperforms PVM by a factor of about 10.

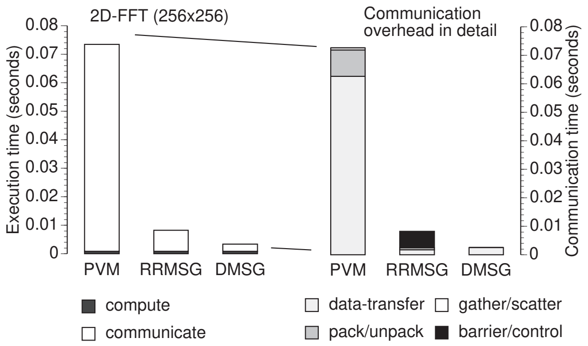

Figure 13: 2D-FFT (256 × 256) for different message passing models: detailed breakdown of execution time [128 nodes].

PVM spends most of its communication time in the basic data-transfer routines pvm_send and pvm_receive, which include both synchronization and actual data transfer. The extra cost of scattering/gathering and buffer packing is relatively small but visible. RRMSG incurs a significant synchronization overhead, because synchronization with control messages is quite costly in dense communication patterns. DMSG is faster because it relies on a global synchronization with fast hardware barriers; even with a small 2D-FFT problem size, the time of the barriers is not visible.

SOR: small problem size

In SOR, each node exchanges data only with two neighbors. Therefore, in comparison to the 2D-FFT communication pattern, we expect to see a reduced impact of per-message synchronization overhead, but a bigger impact of extra copies in PVM, which always buffers data. Figure 13 illustrates the SOR performance for a small (256 × 256) problem size.

For SOR with smaller problem sizes, PVM seems to transfer data a bit faster to its internal buffers than DMSG and RRMSG can transfer data end-to-end. However, PVM incurs a large overhead due to internal copying triggered by the pvm_pack and pvm_unpack calls. For simple, sparse patterns like SOR, synchronization can be done with control messages at a reasonable extra cost on machines that do not have built-in barriers. Thus the extra synchronization overhead in RRMSG is small compared to the overall communication time.

Summary

For each application, the amount of data transferred is identical regardless of the message passing library used. We can therefore compare the time spent in communication to explain why DMSG has the highest transfer rates and the lowest communication overhead in

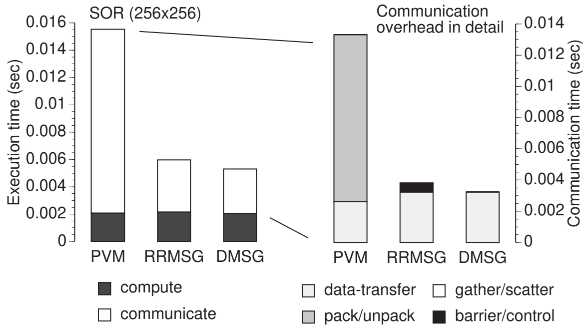

Figure 14: SOR (256 × 256) for different message passing models: detailed breakdown of execution time [128 nodes].

both applications with both problem sizes. The scalability curves for our application kernels follow from the characteristic overheads of the different message passing libraries.

5 Conclusions

Our data show that by decoupling synchronization and data transfer, the message passing library can use good architectural support for fast synchronization and high throughput in data transfers to significantly reduce the time parallel programs spend in communication. Such architectural features are essential for the scalability of application programs to large parallel machines. Without decoupling and good architectural support, even programs that use scalable algorithms cannot exploit large machine sizes in practice. The communication overheads of the message passing libraries put a ceiling on performance on large machines.

Separating synchronization and data transfers opens the door to a more efficient implementation of inter-node communication. As long as traditional message passing systems like PVM or MPI tie synchronization, buffer management, and data transfer together, communication will be expensive. A streamlined message passing model, like deposit message passing, reduces the communication cost by efficiently using hardware support for fast global barriers and direct deposit data transfer. This model also provides the option for a compiler to implement and optimize buffer management and synchronization protocols using detailed information about the structure and dependences in the program. Such information is readily available in modern compilers for data parallel programs.

The foundation of this fast communication system is a de- dicated synchronization subsystem that handles all control transfers and makes sure that the data itself and the storage location at the destination node are ready before any data is transferred. We identified three benefits of good synchronization support. First, unnecessary copies of data can be avoided. In the deposit model, even non-contiguous, strided data elements can be transferred directly from source location to destination location without coalescing, packing, or buffering. On the T3D, the remote stores provide excellent hardware support for this kind of data transfer. The high communication throughput figures of deposit message passing underline this advantage. Second, the overheads of protocol processing and handshaking are drastically reduced by replacing the large number of empty control messages with a global barrier tree. On the T3D, there is hardware support for such barriers. Third, in dense communication patterns, like all-to-all communication (e.g., due to array transposes), the deposit message passing system implementation inserts additional barriers for congestion control in the network. These barriers cost little, but are responsible for increased throughput in large machines with 128, 256, or 512 nodes.

On modern machines like the T3D, using the deposit model for compiler generated communication takes advantage of existing hardware support to improve the communication performance of applications. In the benchmark case of a sixteen million point 2D-FFT on 512 nodes, the deposit communication performs 30 times better than PVM, the conventional, vendor-supplied message passing library, and a factor of 3 times better than RRMSG, a highly optimized request/response library. For the 2D-FFT, the improved communication translates into an overall performance gain of a factor of almost ten, from 1103 MFLOPS (PVM) to 10112 MFLOPS (DMSG). Such improvements provide a strong incentive to include appropriate synchronization support in future parallel machines.

References

[1] D. Adams. Cray T3D System Architecture Overview. Technical report, Cray Research Inc., September 1993. Revision 1.C.

[2] A. Bar-Noy and S. Kipnis. Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems. In *Symposium on Parallel Algorithms and Architectures*, pages 13–22, San Diego, June 1992. ACM.

[3] R. Barriuso and Knies A. Shmem User's Guide for C. Technical report, Cray Research Inc., June 20 1994. Revision 2.1.

[4] A. Beguelin et al. *A User's Guide to PVM*. Oak Ridge National Laboratories, Oak Ridge, TN, 1991.

[5] W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. on Computers*, C-36(5):547–553, May 1987.

[6] A. Feldmann, T. Stricker, and T. Warfel. Supporting Sets of Arbitrary Connections on iWarp Through Communication Context Switches. In *Proc. SPAA*, pages 203–212. ACM, June 1993.

[7] T. Gross, D. O'Hallaron, and J. Subhlok. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, Fall 1994.

[8] S. Hinrichs, C. Kosak, D. O'Hallaron, T. Stricker, and R. Take. An Architecture for Optimal All-to-All Personalized Communication. Technical Report CMU-CS-94-140, Carnegie Mellon University, School of Computer Science, 1994.

[9] Cray Research Inc. *PVM and HeNCE Programmer's Manual*. Mendota Heights, MN, manual v4 for release 1.1 edition, 1994. SR-2501-4.

[10] R. Numrich, P. Springer, and J. Peterson. Measurement of Communication Rates on the Cray T3D Interprocessor Network. In *Proc. HPCN Europe '94, Vol. II*, pages 150–157, Munich, April 1994. Springer Verlag. Lecture Notes in Computer Science, Vol. 797.

[11] P. Pierce and G. Regnier. The Paragon Implementation of the NX Message Passing Interface. In *Proc. Scalable High Performance Computing Conference*, pages 184–190, Knoxville, TN, May 1994. IEEE Computer Society Press.

[12] D. Pountain and D. May. *A Tutorial Introduction to Occam Programming*. BSP Professional Books, Oxford, GB, 1987.

[13] T. Stricker and T. Gross. Optimizing Memory System Performance for Communication in Parallel Computers. In *Proc. 22nd Intl. Symposium on Computer Architecture*, to appear, Santa Marguerita di Ligure, June 1995. ACM.

[14] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling Communication Services for Compiled Parallel Programs. Technical Report CMU-CS-94-139, Carnegie Mellon University, School of Computer Science, 1994.

[15] C. Thekkath, H. Levy, and E. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of ASPLOS VI*, pages 1–12, San Jose, October 1994. ACM.

[16] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. 19th Intl. Conf. on Computer Architecture*, pages 256–266, May 1992.

[17] D. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–73, April 1994.

A Experimental setup

A.1 The host system

The network of the T3D is a 3-dimensional bidirectional torus with a measured peak speed of 126 MByte/s between any two nodes in the torus. The nodes do not have paged virtual memory, and the operating system on the nodes is limited to a small runtime kernel [1].

The key low-level hardware primitive for communication is <shmem_put>, a remote store or block transfer operation. This operation forms the core of all our message passing communication systems, so performance differences are due to design differences, not artificial implementation properties. In the published literature the maximal measured transfer bandwidth is reported to be 126 MByte/s and an message overhead or latency to be 2.7 μs based on the <shmem_put> primitives [10]. We verified the maximal measured bandwith but achieved a lower pure latency figure of 1.33 μs with the underlying hardware primitives, but we measured the software overhead and latency to be 3.80 μs. We include these numbers to allow the reader to calibrate our data with measurements on his or her own T3D.

A.2 Applications used in the evaluation

To evaluate the impact of different passing implementation targets, we chose two kernels: one with a dense communication pattern and one with a sparse, nearest neighbor pattern.

2D-FFT The first kernel is a two-dimensional Fast Fourier Transformation (2D-FFT) on an $N \times N$ array of single precision complex numbers. The columns of the array are distributed by block.[4] During the first computation phase, each node independently performs a one-dimensional FFT operation on each column residing on that node. Next comes a communication phase, in which we transpose the array. This transpose results in a dense all-to-all communication, in which each node sends a distinct data block to every other node. After the transpose, we once again perform a set of 1D-FFT operations on each column. Finally, we transpose the array again, resulting in

[4] We assume a column-major memory layout of arrays, as in Fortran. In C, we would distribute by rows.

another all-to-all communication. This example also captures the communication behavior of large one-dimensional FFTs. On parallel systems, these are often broken up in this way into artificial "rows" and "columns" for better performance.

A typical application then proceeds with a series of filtering operations and possibly performs another 2DFFT to transform the data back to the original domain at the end.

SOR Our second application kernel is an example of such a filter, the computation of a k-point stencil over a two-dimensional $N \times N$ block-distributed array. This kernel is also used in successive over relaxation (SOR) iterative solvers. The stencil computations have simple communication patterns, exchanging data only with a few neighbors (typically one in each direction). The algorithm usually iterates multiple times until convergence is reached; our kernel is measured with 10 iterations of a 5-point stencil, width 4 in each direction. As in the FFT, we mapped only one dimension of the 2D array onto the nodes regardless of the three dimensional physical structure of the T3D. A better mapping could improve the application kernel further.

Table 1 shows for both applications the overall performance per node and the aggregate performance for 512 nodes. We also list the data rates to show the impact of communication performance in more detail. The relative fraction of communication and local computation work and the effect of communication style on scalability is studied with common problem sizes of one to sixteen million elements (e.g., 4096 $\times$ 4096 for 2D-FFT and SOR). We determined this problem size by looking at grand challenge applications in earthquake and airshed modeling.

Two-dimensional FFT Problem Size: 4096 $\times$ 4096 on a 512 node T3D				
Msg Passing System	MFLOPS per node	MFLOPS total	MByte/s per node	MByte/s total
PVM	2.1	1103	0.64	326
RRMSG	13.6	6938	9.39	4809
DMSG	17.1	8755	20.27	10379

Two-dimensional SOR (5pt stencil) Problem Size: 4096 $\times$ 4096 on a 512 node T3D				
Msg Passing System	MFLOPS per node	MFLOPS total	MByte/s per node	MByte/s total
PVM	11.7	5977	21.1	10845
RRMSG	15.4	7882	47.3	24243
DMSG	15.5	7939	48.4	24822

Table 1: Results for the application kernels for message passing system comparison.