Decoupling Communication Services for Compiled Parallel Programs

CMU - Authors

March 8, 1994

Abstract

The message passing systems of parallel supercomputers typically bundle a rich set of services into their send and receive primitives. These services include the basic data transfer service, as well as message-level synchronization, flow control, selective receives, and buffer management. While this rich functionality greatly simplifies the job of the message passing programmer, it adds to communication overheads. Further, if the message passing program is generated by a parallelizing compiler, many of these services are not needed. We propose an approach for building communication systems based on the idea of unbundled communication. We identify the basic services provided by message passing systems, show how to combine these basic services to form simpler and more efficient message passing models that can be exploited by parallelizing compilers, and present performance results measured on a 128-node Cray T3D to validate the approach.

Keywords: Parallel computers, inter-processor communication, message passing, compilation for parallel systems

1 Introduction

Distributing the data for a parallel program enables the independent operation of the nodes of a parallel system but also introduces the need for communication, which is another form of overhead. If all data were local to a single node, there would be no need for communication, but also no parallelism.

The need to communicate during the execution of a parallel program is independent of the machine's memory model. If the parallel system supports a shared (or common) memory model, each node can access the local memory of other nodes, but to perform such a remote access, inter-node communication must take place. This inter-node communication is either directed by hardware (see e.g. [4, 17, 9]) or system software (e.g. [2, 3]). If the machine exposes the addressing of the private memory of each node, then explicit data transfers are required to access data, and either the programmer or the software tool chain assumes responsibility for these transfers. In either case, accesses to local memory are faster than accesses to the remote memory at another node, so there are many incentives to reduce the cost and number of communication steps.

In this paper, we describe an innovative approach that uses compiler knowledge to direct the communication services used for each communication step, so the resulting program only pays for the required functionality. The input language for our compiler is a variant of High Performance Fortran (HPF), which allows the programmer to specify a computation for a parallel system without explicitly specifying the communication. The compiler maps the global name space of the program into the address space(s) of the target system.

Traditionally, such compilers have generated code for each node of a system and used either the vendor-supplied message passing system (e.g., NX for Intel machines) or a generic message passing system (like PVM) for inter-node communication. Such message passing systems[1] provide many services that are not needed in all cases. The services add to the communication overhead, so these message passing systems are slow, especially for small messages. By unbundling the services of the communication substrate (e.g., buffer management, synchronization, and scheduling of transfers), a more efficient system can be built, since a compiler can customize the functionality of the communication system to the application demands. We used a parallel system (a Cray T3D [1]) to demonstrate the practicality of these ideas and to evaluate their impact on performance.

The outline of the paper is then as follows. In Section 2, we briefly introduce the basic terminology of message passing services, describe the range of services commonly included, and discuss other approaches to improve the performance of message passing. In Section 3, we discuss the relation between synchronization and data transfers and argue that these two services should be separate (unbundled). Using two applications, we show in Section 4 how their performance can be improved on a T3D parallel system with a message passing system that supports unbundled communication.

[1] Unfortunately, the term "message passing" has been overloaded: depending on the context, it may refer to the mechanisms of inter-node communication (e.g, as provided by the PVM or CMMD libraries, or the NX runtime system), or it may refer to a specific programming model in which the user specifies communication explicitly (e.g., when programming in Occam or Fortran 90 + PVM). In this paper, we use message passing in the spirit of the first meaning.

2 A reference model for common message passing

In this section we establish a *reference model* for message passing to describe common message passing services and the underlying mechanisms in uniform terms. Our reference model captures the essential features of the communication system of a wide range of systems (e.g., the AP-1000 [14], the NX kernels of iPSC and Paragon systems [5, 6], Express [8], PVM [21], the Cray T3D message passing system [16], and the CM5 message library [7]), as well as the proposed standard message passing interface library draft MPI [11] and the Virtual Channel Router (VCR) [15].

2.1 Common message passing steps

Figure 1 illustrates the different steps of traditional message passing implementations as they are performed for a typical parallel computation. On the sender side, we have:

S.T (Send Transfer): The data of a message is transferred from user space into system space. The send function may complete at this time and the user may or may not be free to overwrite the data.

S.P (Send Packetize): Message wrappers and routing headers are generated. This step potentially breaks large messages into smaller packets. All outgoing message packets are queued.

S.I (Send Inject): If the network is ready to accept another message, it is injected into the network.

Correspondingly, we have on the receiver side:

R.X (Receive eXtract): If a message is available, it is extracted from the network.

R.P: (Receive dePacketize): Packets are reassembled into messages if necessary. Incoming messages are buffered or matched with pending receives.

R.T: (Receive transfer): The data is transferred from system space to user space and the internal buffers are released.

From a compiler's point of view, Figure 1 is overly simplistic, since it assumes that parallel programs always exchange dense and contiguous blocks of data. In practice, the data to be exchanged are usually stored in (fields or elements of) program variables, and they are not generally stored in contiguous memory locations. Therefore, the basic steps must include two more:

S.C (Send Compose or local gather): The outbound data is gathered from its original location and put into blocks for more efficient transfers.

R.U (Receive Unpack or local scatter): The inbound data is stored in its final location.

Both operations include the corresponding address computations. Figure 2 augments Figure 1 with these operations.

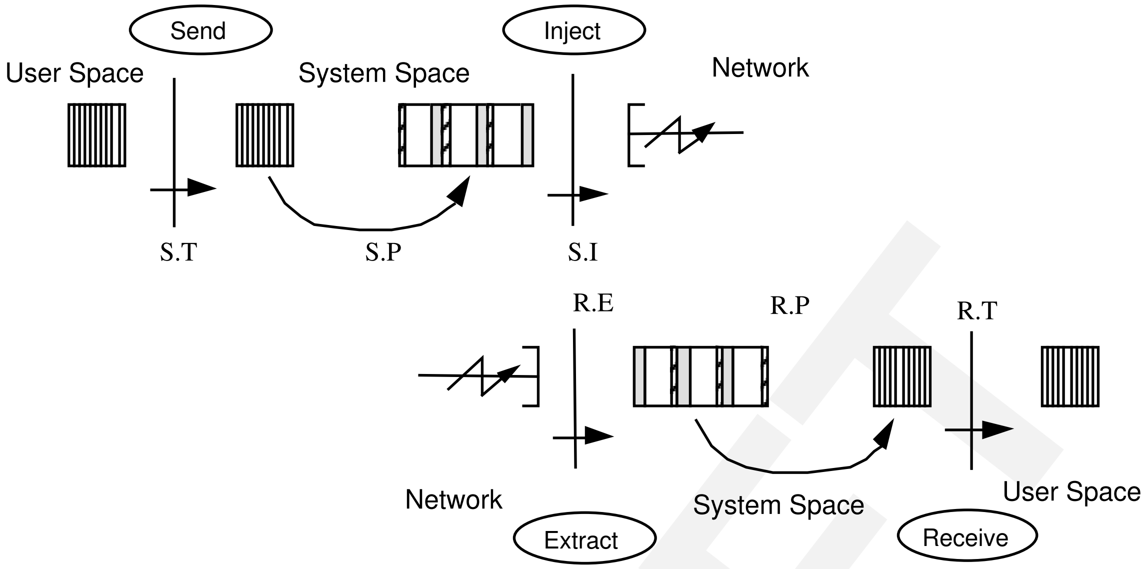

Figure 1: Reference model for traditional message passing.

2.2 Two common user models

We use the terms "send" and "receive" to refer to the user operations that exchange information between cooperating nodes in a parallel system. The steps of Section 2.1 can be combined in many different ways to create send and receive primitives with different semantics, which represent different user models. E.g., a message passing system can delay completion of S.C of one message until R.U of the previous message has been performed. Furthermore, there are many options if the receiver is unable to accept or process a message.

Since a send can be invoked *before* or *after* the matching receive is posted at the receiver node, deciding what constraints to impose on a pair of matching send and receive operations is an important design choice. One approach is to adopt the *rendezvous model*: sends never complete until the matching receive

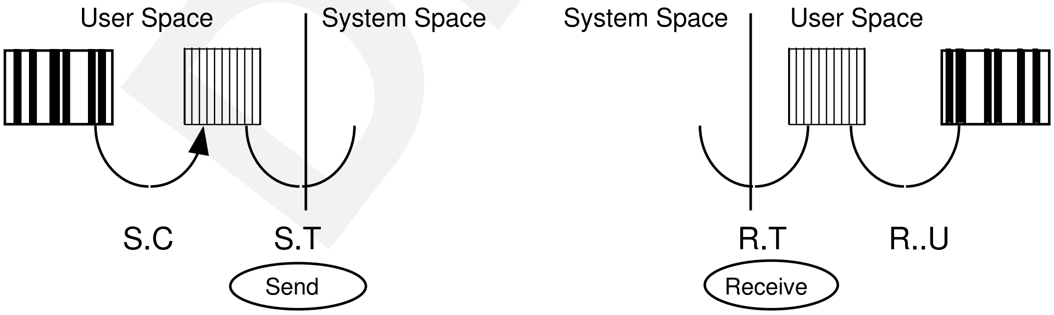

Figure 2: Composing and unpacking of message data at end points.

is executed [18, 20]. Another approach allows an unconstrained flow of messages from the sender to the receiver, which we call the *flow model*. The rendezvous model permits an efficient implementation of large data transfers, but it forces many parallel programs to unnecessarily sequentialize their communication. The flow model assumes the existence of some form of buffering to enable the decoupled operation of the sender and receiver.

A major choice for message passing systems is whether sends and receives are invoked with or without blocking. A non-blocking send returns immediately after transferring the message to the system (S.T), whereas a blocking send waits until the message is either queued (S.P) or injected (S.I). Similar options exist for receive operations; a non-blocking receive must check explicitly whether the data for that particular message buffer has already arrived. Both blocking and non-blocking variants are covered by the reference model.

2.3 Message passing services

Transferring data is the basic service of a message passing system, and different constraints on the steps of Section 2.1 provide either the rendezvous model, the flow model, or one of their variants. However, other services are often provided as well, either voluntarily or by the choices made in combining the basic steps.

For example, to avoid retransmission of messages that cannot be accepted or processed by the receiver, the message passing system may include some form of flow control to throttle the sender. Such implementation features may result in a subtle change of the user model: instead of providing a true flow model, suddenly properties of the rendezvous model become apparent.

Another reason for buffering is to cover up nondeterministic properties of the network (which can exist even if deterministic routing is used). Sends and receives can block temporarily, and messages between any two nodes may block other messages between unrelated nodes, depending on the internal state of the network.

Buffering avoids any unwanted *mutual* blocking between unrelated messages within the communication system. Ideally, receivers remove any incoming messages unconditionally from the network, and send operations always proceed and complete, regardless of the conditions at the receiver. Although send operations may be delayed due to congestion in the network in this model, the programmer can rely on a guarantee that they complete eventually when the load within the network evens out.

Most vendor libraries attempt to implement the unrestricted flow model but "enhance" the basic data transfer services with end-to-end flow control. With the flow-control protocol, the receiver can stop the sender as soon as the receive buffers overflow. Some message passing systems leave it to the user to handle overflows in the receive buffers and simply abort the computation when system resources run out. Hand-coded programs often find application-specific ways to prevent overflow and insert additional communication (synchronization) to avoid this potentially catastrophic failure.

The core functionality of every message passing system includes transfer of data and (one-way) synchronization:

Transfer of data: As its most important function, message passing libraries copy data from one processor to another on a parallel system. Different levels of that service provide support for contiguous and non-contiguous data transfers.

Synchronization: A receive operation can never complete until the corresponding send operation is invoked. This provides a one-way synchronization of the receiving process with the sending process.

However, the services provided by different message passing systems go further, and not all of these services may be required for all cases. This list of services commonly includes:

End-to-end flow control: The synchronization service can be extended so that the receiver can also synchronize the sender by forcing the sender to slow down (or even stop) sending. This service is independent from the link-level flow control usually found in the network hardware.

Selective receives: The programmer needs to match corresponding send and receive operations to assure proper handling of the data transfers. On systems where multiple sends and receives can be outstanding, the message passing library must have a mechanism to select the proper message for each receive operation.

Buffer management: Some message passing systems accept the data in a simple blocking send primitive but move it into a system buffer before injecting it into a network. Other systems extract data from the network into system buffers and hand the data over to the user only after the receive primitive is invoked.

Protection: Operating and runtime system messages must be decoupled and protected from user data messages. In some machines, messages from different users must be protected from each other.

One service that is generally taken for granted in a parallel computer is reliable data transport. Unlike in network computing, the interconnect of a parallel supercomputer can either correct or flag all transmission errors, and the rare case of a transmission error is considered a machine failure.

2.4 Implementation models

Figures 1 and 2 depict the actions of a typical message passing system. However not all of these steps have to be taken if the message passing system is the target of a compiler. Figure 3 depicts the steps that are essential for such use. The streamlined communication system can be faster since it provides fewer services; this is acceptable since the compiler can manage several aspects of communication that are normally handled by the complex message passing systems.

In its simplest form, the message passing system just provides synchronization and user-space to user-space data transfers, as shown in Figure 3. Notice that data transfers can be initiated by the sender as well as the receiver, and this observation motivates two implementation models for message passing, the *deposit model* and the *fetch model*.

In the deposit model, a block is simply deposited into the memory of the receiver without requiring involvement of the receiver node. The processor on the receiver node does not have to execute any code for the deposit operation. This mechanism is efficient and easy to implement, provided architectural problems like address resolution and cache coherency are addressed. If the destination addresses are computed at the sender and shipped with the data, the deposit can even be made directly into the final memory location.

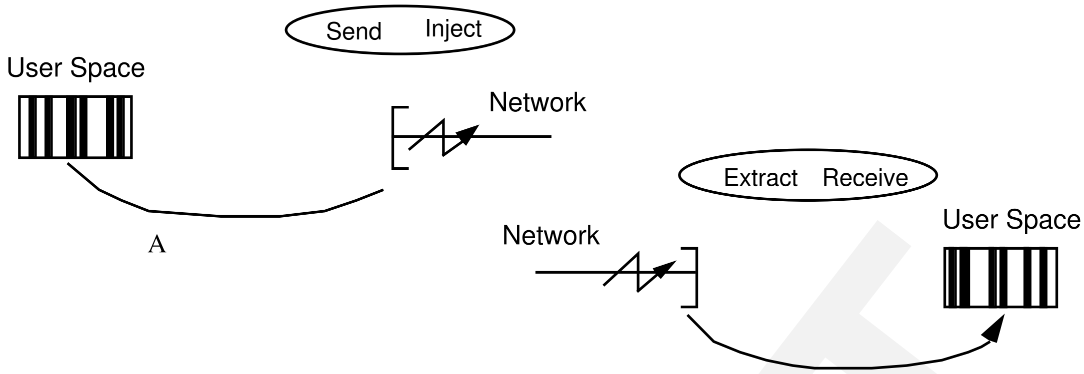

Figure 3: Message passing as a compiler target.

The mechanism for the fetch model is quite similar. The receiving processor needs to know that the data is ready and issues a fetch request to the sender and pulls the data over.

The *deposit model* and the *fetch model* are particularly interesting with respect to compiler generated communication. In both models, the sender or receiver use precise knowledge about the state of the computation at each communication partner. This knowledge is required to determine when data are ready (fetch) or when buffers are free (deposit). One of the challenges for a compiler is to translate the dependences of a data parallel program into such knowledge for the sending and receiving nodes.

2.5 Related approaches to message passing optimization

Other approaches to improving the performance of message passing systems can be classified as improvements to the runtime system and application specific protocols.

The operating system adds overhead to the processing of messages at the end points. Transferring data between different protection domains involves copying and/or protection domain faults. Druschel and Peterson propose the *fbuf* data structure [12] to reduce the cost of this protection domain crossing. In micro-kernel operating systems, the protocol stack may be spread between the user process, the network server, and the base kernel. Transferring control between these processes also adds to the overhead of processing messages. Maeda and Bershad [19] propose linking the protocol stack with the user process, creating an application-specific instance of the communication software that requires far fewer domain transfers. While these operating system optimizations are beneficial, they only solve part of the problem. Yet systems that have no operating system overhead like the T3D still show unnecessary message passing overhead (see Section 4).

Active messages emphasizes the fast processing at the receiver node [22]. In terms of the reference model, active messages provides fast mechanisms for message extraction and transfers. Each message contains a pointer to instructions (a handler) for the receiver, so the receiver need not spend time deciding what to do with the incoming data. Issues like synchronization and buffer management are left to the user or a compiler. Central to the idea of active messages is the notion that the receiver node executes a handler.

One alternative to runtime system improvements is application-specific protocol development. In this case, a specialized message passing protocol for a specific application is developed by analyzing the communication patterns of the program. This approach has been pursued by Felten [13] and is certainly a first step away from the default message passing primitives. Felten's system takes as input a representation of the communication pattern of the program, described in terms of send and receive primitives. The system then tries to reduce the synchronization and buffering requirements of the communication. One advantage of this approach is that it works even if the parallel program was developed without a parallelizing compiler. However, since this approach starts with the final node program instead of the initial single threaded parallel program, the global information that was available to the creator of the program is lost.

Our approach integrates the design of the message passing system with the compiler backend by providing exactly the functionality required by the compiler. This includes the case that the sender drops the data into buffers provided by the receiver, without executing a message handler code on the receiver node. In the next section we justify the separation of data transfer services from synchronization.

3 Synchronization of message passing programs

In a general message passing system such as PVM or MPI, data elements and synchronization information are both carried by the same kind of messages. In this section we discuss the relationship between such data transfers and synchronization, and how they can be dealt with by separate mechanisms.

3.1 Synchronization properties and buffering

A message passing system provides a simple one-way synchronization mechanism between the sending and the receiving process: data must be produced and sent before it is received. Unless the rendezvous model is used, some storage space for messages is necessary to allow sends to complete before the corresponding receive is posted. Otherwise, a message might be long enough to fill the network and delay the sender until the receiver starts processing the message. Moreover, in a realistic network, unrelated messages might share a communication channel and can therefore block each other. Most networks are only deadlock-free under the assumption that the receivers continuously and unconditionally remove incoming messages [10]. Such storage space can be supplied either through user program or system buffers.

If the storage space is bounded (as it is in practice), then some form of flow control between sender and receiver is required. Furthermore, limitations on pre-allocated buffer space mandate that long messages are broken down into smaller packets, and that a flow control protocol is used to send requests for buffers, replies, and acknowledgments in messages. With these protocols, 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. Each data transfer has implications on the management of storage, and this coupling increases the cost of the data transfers, since each transfer may involve storage management decisions.

3.2 Copying and buffering

Since both the size of messages and the data layout are application-specific, it is often impossible to use memory mapping hardware to avoid copying. Furthermore, some parallel computers do not provide any virtual memory. Buffering at the end points is particularly costly, since the traffic to and from data buffers traverses the memory bus multiple times, which increases the latency.

It is therefore advantageous to direct all messages to their final destination in memory. That is, a high performance message passing system must extract the data from the network and store it directly into the user space. In our implementation of the deposit model, the block transfer engine at the receiver achieves exactly this. However, since the sender and receiver operate independently, some form of synchronization is needed to avoid overwriting live data.

Since messages are directed to user space, all buffers are under user control. Any synchronization messages must be generated separately, since we cannot rely on the synchronization properties associated with data transfers. Therefore, we propose to separate synchronization and the actual transfer of data.

3.3 Separating synchronization and data transfers

If all messages are directed to their final destination (deposit model), or data are directly fetched from their source (fetch model), synchronization is required *prior* to the data transfer(s). Therefore, synchronization must be separate from data transfer.

Also, compiler-generated parallel programs can manage the buffers involved themselves and therefore 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.

Decoupling synchronization from data transfers creates additional opportunities for improvement in the communication system.

1. With the global knowledge of a compiler, flow control messages of several nodes can be combined, resulting in a drastic reduction in the number of flow control messages. For example, a personalized all-to-all communication[2] among n processors implies n^2 data transfers, one for each sender-receiver pair. A connection-oriented Ack/Nack flow control protocol must send n^2 messages to acknowledge the data transfers between 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 just $2n$ messages in $2 \log n$ steps. A study showing the precise tradeoffs for the T3D is given in Figure 4 and discussed in Section 4.

2. 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; 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 synchroniza-

[2] All nodes of a system send a different message to every other node.

tion trees are the optimal candidates to propagate the synchronization and flow-control information among the computation nodes.

Non-blocking receives provide one way to realize the benefits of decoupling synchronization from data transfers. With non-blocking receives, receives are posted ahead of the send. Incoming messages can then be matched with their corresponding receives at the time of their extraction from the network, thus saving the overhead of copying the data to a buffer. However, the opportunity to place incoming messages in their final destinations does not remove the need to allocate buffer space, unless *all* receives are posted in advance (e.g., due to global timing guarantees). With blocking, this optimization can be exploited at best for a single receive.

Traditionally, the message passing hardware is designed to *transfer 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.

Based on the two observations above, our compiler for a dialect of HPF uses a communication system that separates synchronization from the data transfers. Using a global picture of communication steps, the compiler backend 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 required, since the buffer management is taken care of by the compiler and most of the synchronization is done globally.

4 Evaluation

4.1 Target system

To verify our observations regarding performance of different message passing systems, we implemented and measured two application kernels on the Cray T3D. The T3D is a massively parallel system; each node is based on a DEC Alpha microprocessor and runs at 150 MHz. The network is a 3-dimensional bidirectional torus with a raw, unformatted communication speed of 150 MBytes/second between the nodes. The nodes do not have paged virtual memory, and the operating system on the nodes is limited to a small runtime kernel. The machine is attached to Cray C90, which acts as a front-end [1].

To put the observations of Section 3.3 in perspective, Figure 4 depicts a series of measured data points and interpolations to show the cost of synchronization options for an all-to-all communication pattern. Three different ways to propagate synchronization information among all communication nodes are considered. In the first method (ack-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 (ack-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.

4.2 Application kernels

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 type pattern. The first kernel is

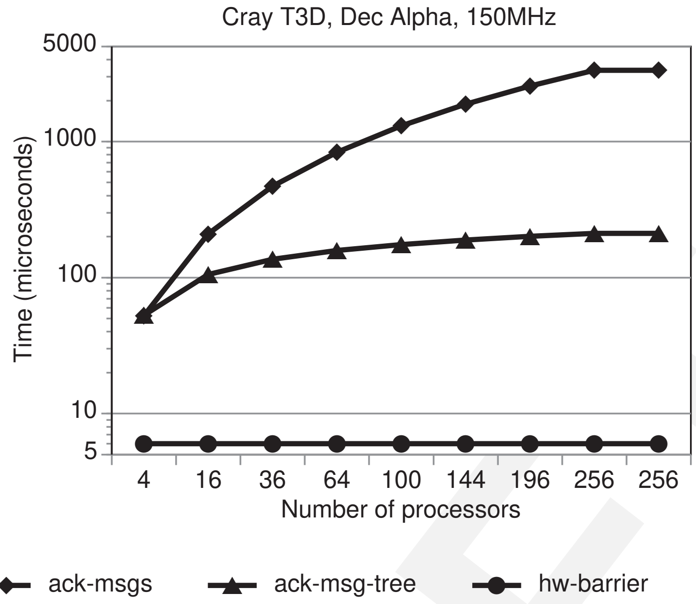

Figure 4: Costs of different mechanisms for flow control in an all-to-all communication step.

a two-dimensional fast Fourier transformation (2DFFT) on an $N \times N$ array of single-precision complex numbers. The columns of the array are distributed by block.[3] 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 1DFFT operations on each column. Finally, we transpose the array again, resulting in another all-to-all communication. 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.

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; our kernel is measured with 100 iterations of a 5-point stencil with width 4 in each direction.

4.3 Message passing targets

We use our compiler to map the two application kernels mentioned above for three styles of message passing. Each target employs a different style of message passing:

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

PVM: Cray PVM 3.0 The PVM/MPI target of our compiler generates generic message passing "send" and "receive" calls. It relies on the message passing library for all synchronization and data transfer services using selective receives. In our measurements all the send and receive calls execute using the commercial PVM 3.0 implementation of the supercomputer vendor [16], based on the T3D's block transfer primitives. Note that this is a specialized library version of PVM for communication completely within the T3D distributed memory parallel computer, as opposed to a more general one that is also capable of communicating with other standalone systems on the internet. This PVM library is written and optimized under the assumption of exclusive processor use and full access to the communication system.

DMSG: Deposit model message passing The second target is based on our insights about service decomposition and relies completely on the built-in T3D block transfer commands. These commands allow deposit writes into the address space of any other node, given a destination node, the block size, a local source memory address, and a remote destination memory address. At the destination processor, the block transfer engine executes the deposit asynchronously, without any participation from the receiving processor. Additionally, the compiler provides a semaphore to recognize when the transfer is 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 Our third target uses a more typical method of message passing and buffer management, similar to that used by Intel's NX message passing system for long messages. 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 required to free the user from potentially complicated buffer management decisions. The model incorporates ideas from both worlds: the conventional message passing systems (e.g., PVM), and the highly efficient deposit model (e.g., DMSG), which is specialized for compiler use. Both control messages and data transfers in RRMSG are implemented directly on the same efficient block transfer engine as DMSG or PVM.

Table 1 shows the overall and per-node performance achieved for both application kernels. The computations were done on a Cray T3D, and the performance on the 2DFFT exceeds 1.5 GFLOPS on 128 processors. Because of time constraints, the SOR code was only executed on a small partition of 16 nodes. For the final paper, we expect to incorporate results of a some dedicated runs on an unpartitioned, large-scale parallel machine (256 or 512 nodes).[4]

To appreciate the full impact of message passing styles on the performance of our applications, we must look at the times spent in communication-related work. Figure 5 shows a breakdown of execution times of the 2DFFT and the SOR kernel for each of the three message passing models. *Communicate* includes all time spent in moving the data from one node to another, including the per-message startup times, the transfer time, and all waiting time due to the network congestion.

[4]These performance numbers are given solely for a better understanding of the subsequent relative comparisons and to relate our computations properly to other experiments. They are not meant to be benchmarking results for the Cray T3D. We measured them using preliminary versions of the compiler and did not spend much time on fine-tuning single node computation performance.

Two-dimensional FFT				
Msg Passing System	Problem Size	# of nodes	MFlops per node	MFlops total
PVM	1024x1024	128	5.5	706
RRMSG	1024x1024	128	12.2	1607
DMSG	1024x1024	128	13.8	1770

Two-dimensional SOR				
Msg Passing System	Problem Size	# of nodes	MFlops per node	MFlops total
PVM	256x256	16	15.1	241
RRMSG	256x256	16	18.6	298
DMSG	256x256	16	18.8	301

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

Figure 6 shows a breakdown of only the communication times for the kernels. The execution time of the barrier synchronizations is not shown, as it is several orders of magnitude lower than the communication and computation times. The execution times are broken down according to the reference model. The *gather* time is the time required by the sending node to put all data into one contiguous block to meet the requirements of the block transfer engine. Gathering is necessary for the 2DFFT but not for the SOR, as the data is already contiguous in memory. The *scatter* time represents the work required to store the data into its final location on the receiver.

Figures 7 and 8 show how the performance of our application kernels scales with different machine sizes, from a single node to 128 nodes. The absolute performance numbers of 2DFFT and PVM message passing indicate that the per-node performance drops sharply past 16 nodes, and there is not much gain with larger machines. The 2DFFT numbers for the more efficient deposit model demonstrate that speedups can be sustained up to 128 processors on the same problem size. On the larger machine the limited memory system performance becomes evident in that the scatter and gather times become a bottleneck, as can be derived from the breakdown figures given in Figure 6. Note that an implementation of the 2DFFT for a modern uniprocessor must also pay for a scatter (or a local memory transpose) because this operation also reestablishes the locality of reference required by the column FFTs.

4.4 Discussion

The measurements of control message exchanges verified our assumptions about the costs of treating control messages as data messages. Exchanging $O(\log n)$ messages instead of $O(n^2)$ resulted in the expected improved performance. The further improvement of the barrier synchronization hardware shows that an optimized path for zero length messages is worthwhile for control messages.

We compared the performance of the DMSG and RRMSG message passing targets to isolate the effects

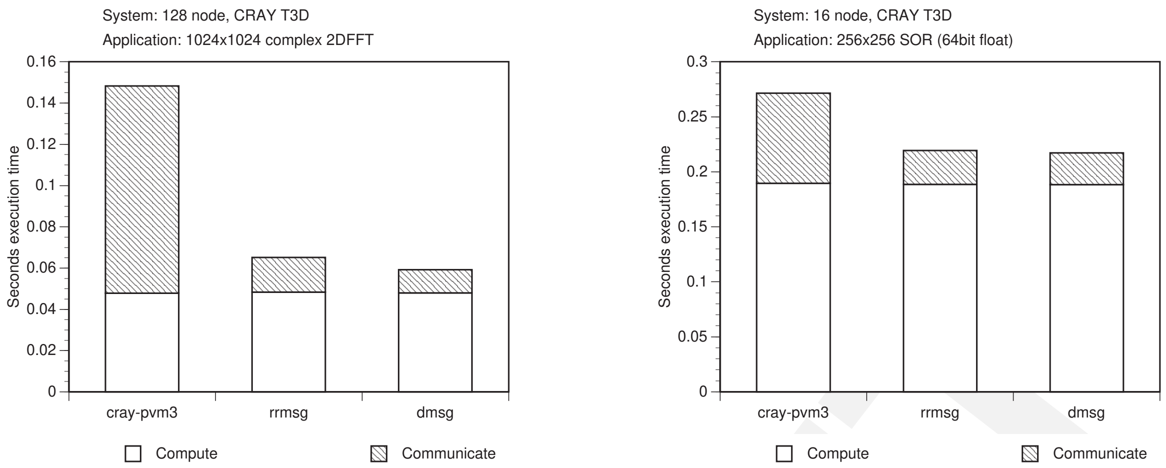

Figure 5: Breakdown of the total execution times for two application kernels executing on the Cray T3D with three different message passing targets for our compiler.

of additional control messages on performance. Both models use the same deposit model without additional buffering, but the RRMSG message passing uses control messages instead of a barrier synchronization to allocate the buffer space. For the 128-node 2DFFT example, the use of a barrier reduces the communication time by a factor of two. The barrier does not impact the smaller 16-node SOR example as much. In this case fewer control messages need to be exchanged, because each node need only communicate with four neighboring nodes, instead of exchanging with every other node as in the 2DFFT example.

The performance differences between DMSG and PVM are more significant. In addition to using control messages instead of using barrier synchronization, the PVM implementation uses intermediate system buffers between the user space and the network (i.e., in the S.T and R.T steps of the reference model, the data is copied between user and system space). In addition, the PVM implementation makes no compile-time assumptions about the amount of buffer space required, so at runtime the PVM kernel must allocate buffer space. For the 2DFFT kernel, the RRMSG version is 9.5 times faster than the PVM version; the DMSG version is 19 times faster. For the SOR kernel, DMSG and RRMSG are both roughly 2.6 times faster than the PVM version.

In both applications, the cost of barrier synchronization is negligible. However, for this machine, avoiding the costs of additional copies appears to be more important than avoiding extra control messages. For less regular communication patterns, this may not be the case. For systems with less communication bandwidth than the T3D, or higher latency compared to computation power and memory bandwidth, optimizing control messages should definitely be more important. In addition to offering a performance advantage for 2DFFT, using barriers simplifies the deposit model protocol.

These measurements back up our original assertion that a reduced message passing system like the deposit model shows improved communication performance compared to the more general message passing system. To achieve the streamlined model shown in Figure 3, the communication system architecture must provide services in a different manner than is found in today's vendor message passing libraries (i.e., the reference model). The services of the traditional message passing system must be unbundled, so

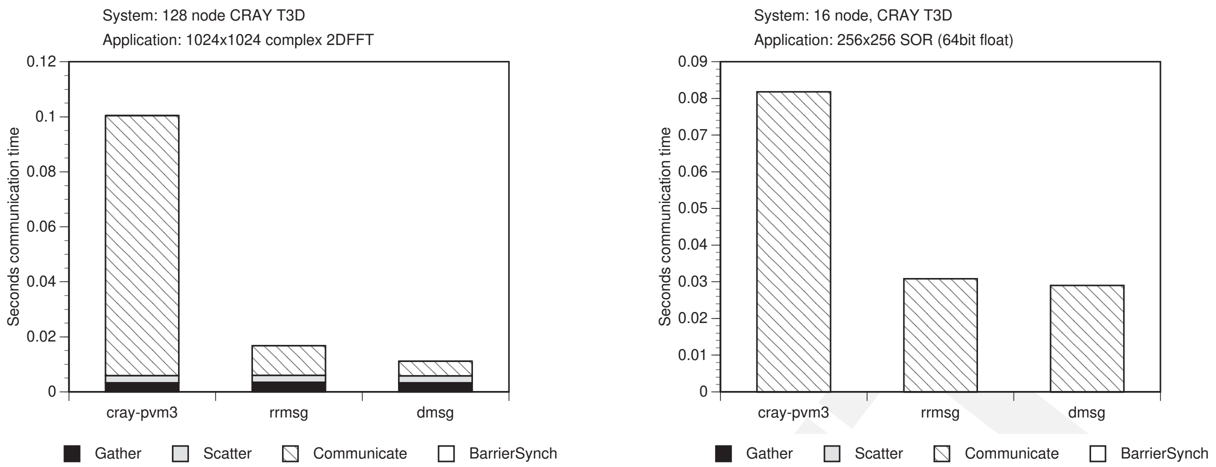

Figure 6: Detailed breakdown of the time spent in communication-related work for two application kernels executing on the Cray T3D with three different message passing targets for our compiler.

buffering, data transfer, and synchronization services can be offered separately. Then the application need only use the services it requires without paying for *all* services.

Through communication analysis,[5] the data parallel compiler determines what communication patterns must occur in the program. Each source statement is translated into a communication step that performs all the communication patterns needed for that statement and a computation step that performs the actual computation. The compiler can map these communication patterns into services provided by the target communication system. This communication code must i) identify the elements to send, ii) possibly gather them into buffers, iii) transfer the data, and iv) maintain the necessary synchronization between sender and receiver. Therefore, the compiler has sufficient knowledge to take advantage of separable communication services.

Our discussion of communication models derived the deposit model for the distributed memory model, but the deposit model is quite similar to the remote shared memory push model. For good performance in both cases, the compiler must identify the non-local data at each step, retrieve that data before the computation can begin, send data directly to the final memory destination, and use a synchronization method to ensure the source or destination data are ready. In fact, on the T3D, both the deposit and the remote memory put use the same underlying hardware mechanisms.

5 Conclusions

Our data have shown that unbundling the communication services opens the door to a more efficient implementation of inter-node communication in a parallel system. As long as traditional message passing systems tie the service components of flow control, synchronization, and buffer management together, communication will be expensive. Even the custom-tailored implementation of PVM on the T3D, a

[5]Due to space limitations and anonymity constraints we omit the details of our compiler's communication analysis.

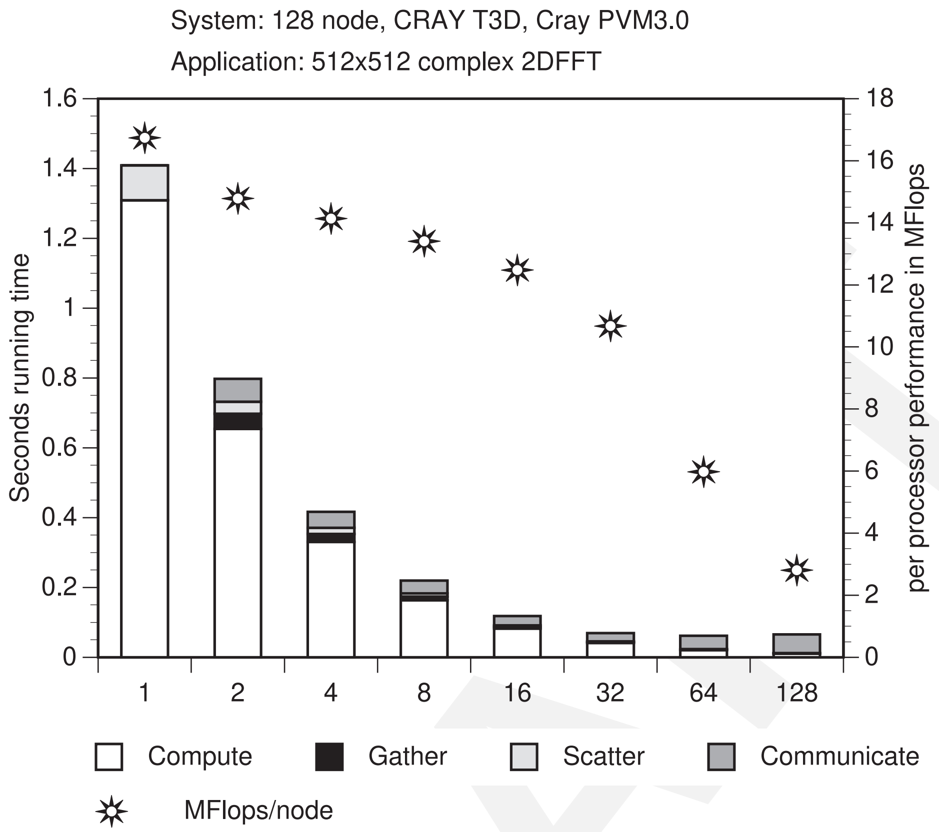

Figure 7: Execution times of the two-dimensional FFT application kernel for different Cray T3D machine sizes and the PVM message passing target for our compiler.

state-of-the-art parallel system with support for remote memory operations, cannot escape this reality.

We have demonstrated for a few example problems the performance benefits that can be obtained from this approach to message passing. However, managing all the details of inter-node communication should not be attempted by the programmer. To take advantage of the opportunities of unbundled communication, detailed information about the structure and dependences in a program is necessary. This information is readily available in modern compilers for data parallel programs, as illustrated by our prototype compiler.

Exploiting the boundaries between compiler and communication system is crucial to realize the performance potential of parallel computers. An approach that exploits the compiler's knowledge about a program to minimize expensive operations or to remove unnecessary synchronization is superior to a model where the compiler considers the communication system as a separate component (or even as an adversary). Uniprocessors underwent considerable streamlining and simplification once it was recognized how to couple compilation and architecture. Our experience to date suggests that there exist similar opportunities for parallel systems to couple communication and computation.

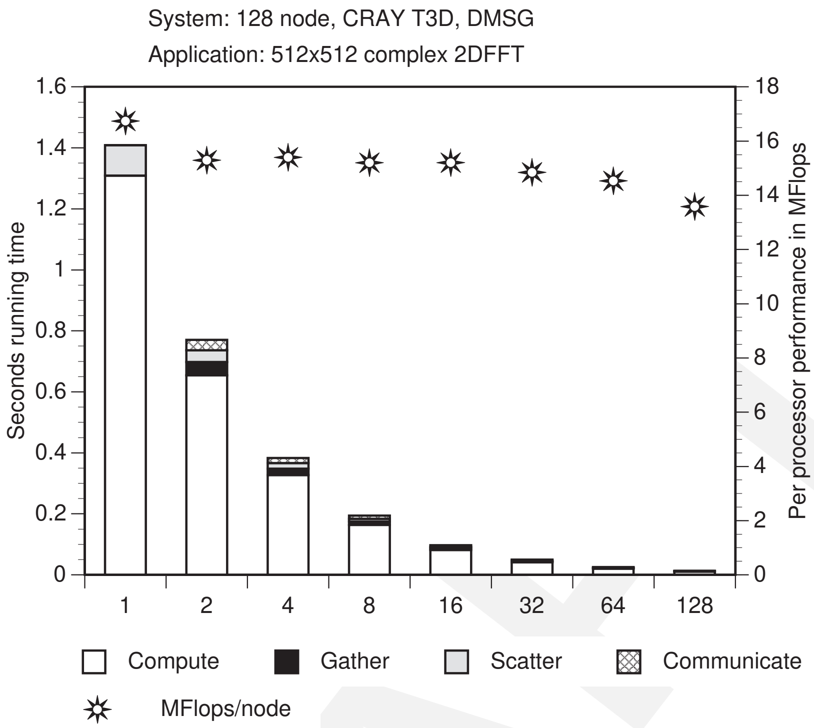

Figure 8: Execution times of the two-dimensional FFT application kernel for different Cray T3D machine sizes and the DMSG message passing target for our compiler.

References

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

[2] J. Bennett, J. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 125–134. ACM/IEEE, May 1990.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Procceedings of COMPCON*, 1993.

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. Technical report, Princeton University, Department of Computer Science, 1993. To appear in Proc. 21th Intl. Conf. on Computer Architecture.

[5] Intel Corp. iPSC Programmer's Reference Manual. Report, March 1987. Order Number 310612-002.

[6] Intel Corp. *Paragon X/PS Product Overview*. Intel Corp., March 1991.

[7] Thinking Machines Corp. *The Connection Machine System - CMMD User Guide and Reference Manual*. TMC, Boston, MA, 3.0 edition, 1993.

[8] Parasoft Corporation. *Express User's manual*. Parasoft Corp., 1990.

[9] A. Cox and R. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proc. 20th Intl. Symp. on Computer Architecture*, pages 98–108. ACM/IEEE, May 1993.

[10] 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.

[11] J. Dongarra, R. Hemoel, A. Hey, and D. Walker. A Proposal for a User-Level, Message Passing Interface in a Distributed Memory Environment. Technical Report TM-12231, ORNL, 1993.

[12] P. Druschel and L. L. Peterson. FBufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Asheville, NC, December 1993.

[13] E. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs*. PhD thesis, Univ. of Washington, Seattle, WA, 1993.

[14] T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata. Improving AP1000 Parallel Computer Performance with Message Communication. In *Proc. 20th Intl. Symp. Computer Arch.*, pages 314–325. ACM/IEEE, May 1993.

[15] J. Hsu and P. Banerjee. Performance Evaluation of Hardware Support for Message Passing in Distributed Memory Multicomputers. Technical Report RC 17153, IBM T. J. Watson Research Laboratory, April 1991.

[16] Cray Research Inc. *PVM and HeNCE Programmer's Manual*. Mendota Heights, MN, draft edition, 1992. SR-2501.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[18] INMOS Limited. *Transputer Reference Manual*. Prenctice Hall, 1988.

[19] C. Maeda and B. N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Asheville, NC, December 1993.

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

[21] V. S. Sunderam. PVM : A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[22] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19th Intl. Conf.. on Computer Architecture*, May 1992.