Abstract

This report summarizes and compares the functionality of several portable message passing libraries. A message passing library contains explicit communication primitives for the exchange of messages among computing processes. A **portable** message passing library further attempts to provide a uniform communication interface on different types of machines. We survey ten message passing libraries, most of which are publicly available, and summarize their basic communication interfaces in this report. We also discuss the semantics of message passing primitives, how easy they are to use, and how the interface design and implementation can affect portability and performance.

1 Introduction

In spite of claims that writing shared memory programs is easier than writing message passing ones, message passing remains a popular programming model for large scale machines, particularly when the goal is to get the maximum performance out of a given machine. There are several reasons for this popularity. First, simple point-to-point communication primitives are cheaper to built in hardware than a globally shared memory, and scalability of shared memory machines continues to be a topic of debate. Second, message passing seems to be the appropriate programming paradigm for exploiting the non-uniformity of memory accesses in distributed memory machines, since all remote accesses are explicit in the program. Third, the nondeterminism in message passing programs is typically less than in a shared memory program, because a message constitute synchronization as well as communication. The behavior of a message passing program, at least in some models, is more predictable than with shared memory.

Nevertheless, programming with low-level message passing primitives can be an onerous task. Communication among computations has to be carefully coordinated, and the program is usually hard to understand. To cope with this difficulty, communication libraries (also called message passing libraries) that provide high-level programming abstractions have been built on top of the simple message passing primitives. Typical performance overhead of using such libraries, based on numbers provided by the developers, is about 25% relative to the cost of native message primitives [1, 2].

Aside from reducing programming complexity, a portable message passing library may abstract the architectural details of different parallel machines. The programmer can now use the same communication interface to program distributed memory multicomputers, cache-coherent multiprocessors, or even networks of workstations. Some implementations of the libraries further allow the programmer to spread computation in a heterogeneous environment. Heterogeneity may become an efficient way to exploit the power of future generation networks; it also eases the development of applications requiring many specialized computers such as high-speed computing devices and sophisticated graphical displays.

In this report we sample ten communication libraries that are built around message passing primitives. Most of them operate on multiple types of parallel computers. A few also have the capability of running in a heterogeneous network environment. The ten communication libraries are listed below:

- APPL[3]: a communication library developed at NASA Lewis Research Center. The goal of APPL is to simplify programming on the parallel machines there. APPL stands for "Application Portable Parallel Library".
- CMAM[19]: an *Active Message layer* for the Connection Machine (CM-5), developed at the University of California, Berkeley.
- CMMD[4]: a specialized communication library written for the Thinking Machines CM-5, a multiprocessor architected around a packet switching network.
- EXPRESS[1, 5]: a commercial parallel programming toolkit developed by Parasoft Corp., which was founded by a group from the Caltech hypercube project.
- P4[6]: a library of macros and subroutines developed at Argonne National Laboratory. It is meant to support "Portable Programs for Parallel Processors", from which it also takes its name.
- PARMACS[20]: a collection of macros described in the book *"Portable Programs for Parallel Processors"*. It is one of the predecessors of P4.
- PICL[7]: a "Portable, Instrumented Communication Library" developed by the researchers at Oak Ridge National Laboratory. PICL was used to aid the performance characterization research at ORNL[8].
- PVM[9, 10]: a communication package developed at Oak Ridge National Laboratory to address heterogeneity in a network based environment. PVM stands for "Parallel Virtual Machine".
- TCGMSG[11]: a communication library developed and used by the Theoretical Chemistry Group at Argonne National Laboratory.
- Zipcode[2, 12]: a communication library developed by the researchers at Lawrence Livermore National Laboratory.

The operating environments of these libraries are summarized in table 1. Note that many different ports of these libraries are currently under development, and the entries here are likely to be out-dated.

The list is by no means exhaustive. There are many others that provide similar functionality. In addition to the message passing primitives, some of these communication libraries also provide shared-memory primitives, performance monitoring tools, and user interfaces for programming and debugging. In this report we focus on the basic message passing aspect only.

The report is organized as follows. Section 2 outlines the programming models provided by the communication libraries. Section 3 introduces the basic send and receive

Table 1: Summary of operating environments

Name of library	hardware configurations	languages
APPL	workstation network, iPSC/860, NASA Hypercluster, Alliant FX	C, Fortran
CMAM	CM5	C
CMMD	CM5	C, Fortran
Express	workstation network, Alliant, Transputers, iPSC/860, Ncube, CRAY	C, Fortran
P4	workstation network, Multimax, Alliant, Cray, iPSC/860, Delta, BBN, Sequent Symmetry, Encore	C, Fortran
PARMACS	Alliant, Sequent, Encore Cray-2, Vax, Intel Hypercube workstation network	C
PICL	iPSC/2, iPSC/860, Ncube/3200, iPSC/1, Symult S2010, Cosmic Environment, Linda, Cogent	C, Fortran
PVM	workstation networks, CRAY, iPSC/860, iPSC/2, CM2, Titan, Alliant FX/8, Symmetry	C, Fortran
TCGMSG	workstation network, Alliant FX(MPP), Titan, Convex, CRAY, iPSC/860, Delta, Encore	C, Fortran
Zipcode	workstation network, Delta, BBN TC2000, CM-5, iPSC's, Ncube, Symult S2010	C, Fortran

primitives and defines various forms of message passing semantics. Section 4 compares the different messages passing semantics, giving examples to illustrate their differences in use and performance. Section 5 categorizes the ten message passing libraries in terms of their differences in basic semantics and discuss how one can port applications written in one library to another. Section 6 summarizes the high-level communication primitives other than simple sends and receives. In section 6 we summarize the survey and give our conclusions.

2 Programming Model

Parallel programs written in the message passing paradigm are characterized by a collection of *processes*, coordinated by messages bearing explicit names of the recipient processes. It is important to distinguish between processes and processors. A process is a piece of computation capable of sending and receiving messages; it's an instantiation of a program text. In the message passing model different processes do not communicate via shared objects, and therefore they have distinct address spaces. A processor, in contrast, is a physical resource that actually carries out the computation specified by a process. Depending on the machine configuration a processor can be a processing element of a multiprocessor or a workstation connected to a general purpose network. The mapping from processes to processors is highly machine-dependent. For example, the mapping is static (fixed) and one-to-one on a CM-5, whereas it is dynamic and many-to-one on a Sequent Symmetry. For the sake of portability the programmer should not be concerned with the process-processor mapping, and the rest of our discussions will be about the processes only.

A parallel program can be written in various programming models, which are sometimes enforced by languages and sometimes a matter of programming style.

The *data-parallel* model consists of a single program thread that executes over parallel data structures, particularly arrays. Languages for SIMD (Single Instruction Multiple Data) machines, such as Connection Machine Fortran and Maspar Fortran, are classic examples of data-parallel languages. The execution model is that instructions are executed in lock step on all the proessors, with data-depended branches performed by having some processes abstain from executing instructions along the branch. Recently, the term "data-parallel" has been expanded to encompass any program written in a data-parallel style, namely one in which the parallelism comes from performing the same operation on all the elements of an aggregate structure.

In the SPMD model, which stands for Single Program and Multiple Data streams, the programmer write a single program text and runs is on all processes. The difference between the data-parallel model and SPMD model is the number of threads of control: there is one thread of control in a data-parallel language and multiple threads in an SPMD language. A compiler for a data-parallel language must take computations on a data structure of size N and map it onto P processes. In SPMD programs, this mapping is done by the programmer. An SPMD is often provided, along with a message passing library as the lowest level programming model on MIMD (Multiple Instruction Multiple Data) distributed memory multiprocessors. A generalization of the SPMD model allows the programmer to load different programs on different processes. This is sometimes called an MIMD programming model. Other than saving a certain amount of code space, the difference between SPMD and MIMD programming is very small, since an SPMD program can branch to the relevant code segment. Another variant is the FPMD (Few Programs Multiple Data) model, where multiple instances of the same program are used for a group of processes. A final variation is the Host/Node model, in which there is a distinguished processor called the *host*. Two programs are used in this model: one for the host processor and one for the other *node* processors. The host program is usually used to interact with the user and to invoke the appropriate node functions; the node programs are written in a SPMD manner. For the rest of our discussion we will assume the MIMD model is used.

Communication between the processes can be categorized as follows: **point-to-point** messages transport data between a pair of processes and are the basis of other high-level communication primitives; **one-to-many** messages distribute information from one source process to multiple recipients; **many-to-one** messages combine information from multiple sources to a single recipient; **many-to-many** messages combine and then distribute information among multiple sources/recipients; **many-to-many personalized communication** consists of a set of independent point-to-point messages. More discussion on the functionality of these communication schemes can be found in section 5. Most multiprocessors have hardware support for fast inter-process communication. Communication in a heterogeneous environment is usually performed via the general purpose network interface (sockets) or the file system (pipes). The performance of these various types of communication is highly dependent on the machine architecture. For example, CM-5 has dedicated hardware support for carrying out some particular forms of one-to-many and many-to-many communication. These machine-dependent performance issues are beyond the scope of this report.

3 Definitions of Message Passing Semantics

In this section we define the message passing semantics existent in the communication libraries. We introduce these definitions via the most essential message passing primitives, namely the point-to-point sends and receives. These definitions can be easily extended to other types of communication primitives. In-depth analysis of the semantics is left to the next section.

3.1 Basic Primitives

The following primitives are found at the bottom layers of most communication libraries. Although these primitives perform very simple functions, they are sufficient to expose all the subtle differences in message passing semantics.

- **point-to-point send**: deliver a message from a given user buffer to some user buffer of the recipient process. The arguments for the send primitive include the address of the user buffer, the length of the message to be delivered, the recipient's process id, and possibly the message's tag.
- **point-to-point receive**: pick up a message and store it in a given user buffer. The arguments for the receive primitive include the address of the user buffer to store the incoming message, the length of the message to be received, and possibly the sender's process id or the message's tag.
- **active messages**: deliver a message from a given user buffer and invoke some procedure at the recipient process, using the message as its argument. The arguments for an active message include the user buffer storing the message, the length of the message, the address of the remote procedure (referred to as the *handler*), and the identifier of the recipient process.

In addition to the message content, some other information must be sent along to ensure the message is picked by the proper recipient. This is done by specifying the id of the sending process and a *tag* describing what the message is about. The message is then picked up by a receiving statement specifying some matching sender id and tag. The programmer may choose to ignore this information by using *wildcards* in the receive statements.

Note that the message types we discuss here is not related to the type of the message data as defined by the programming language. The language type is not used by the communication library to enforce message selection. It is used to perform data conversion in a heterogeneous networking environment. Some libraries supports data conversion based on the XDR (External Data Representation) standard. This functionality eliminates the machine dependent code from the user program. PVM and P4 provide interfaces to enforce such type conversion.

3.2 Semantics of Basic Message Passing Primitives

We now define the semantics of simple message passing primitives. Unfortunately there is no standard terminology for some of the concepts described here; we use the common terminology whenever it exists.

- **Synchronous send:** send a message and wait for an acknowledgment from the recipient.
- **Synchronous receive:** receive a message (wait if necessary) and send back an acknowledgment.
- **Blocking send:** send a message and continue execution without waiting for the reception of the message. However, the send statement may block in some circumstances, and the execution of subsequent statements is not guaranteed.

- **Blocking receive:** receive a message (wait if necessary). Subsequent statements will not be executed until the receive is done.
- **Non-blocking send:** send a message and continue execution without waiting for the reception of the message. The latency of the send statement is bounded and subsequent statements will be executed in any case. The user may not reuse the user buffer before the send operation is performed.
- **Non-blocking receive:** receive a message if one is available. Some variants of the non-blocking receive return information about the message (such as the sender's id and the message tag) without retrieving the message itself. The receive statement returns with failure immediately if there is no message to be received.

Note that the synchronous primitives use a different protocol than the asynchronous ones (blocking or non-blocking). Therefore synchronous sends must be paired with the synchronous receives. On the other hand, blocking and non-blocking sends/receives can be mixed.

Communication via sends and receives is *cooperative* in that the delivery of a message requires actions on both the sending and the receiving processes. Active messages, on the other hand, requires an action from the sending process only. The recipient thus has no role in specifying the time, space, and method of delivery.

4 Comparison of Message Passing Semantics

We now explain the differences between the various message passing semantics – synchronous vs. asynchronous, blocking vs. non-blocking, and active vs. cooperative. A series of examples are used to contrast their use and performance implication.

4.1 Synchronous vs. asynchronous message passing

The main use of a message is to transport a piece of information. A message can also be used to synchronize the execution of two processes. Synchronization requires one process to wait (i.e., to suspend execution) for the other; the recipient waits until the expected message is sent, and the sender waits until the sent message is received. The waiting on the receiving side is simply to suspend until the message physically arrives. The waiting on the sending side, however, requiring an acknowledgment from the recipient after the message is picked up. This two-way protocol enforces a barrier synchronization between the two processes at the send/receive statement, and hence we call the corresponding primitives synchronous message passing primitives.

Synchronous message passing is a very restricted way of writing parallel programs. All the communication points have to be planned carefully, otherwise deadlocks may occur. Asynchronous messages, on the other hand, allows more flexibility in the scheduling of communication. For example, when two processes X and Y wish to exchange two local arrays, the following program will break down if synchronous messages are used:

process X:	*process* Y:
`send` A `to` Y	`send` B `to` X
`receive` B `from` Y	`receive` A `from` X

Since the receive statements can not execute before the send statements are done, none of these two messages will be received. The above program, however, functions normally if asynchronous messages are used, because in that case the send returns immediately to allow the subsequent receive to proceed.

The above example illustrates the restriction synchronous messages put on the ordering of messages—although the exchange works regardless of the order in which A and B are received, the programmer must select an order a priori. The following program performs the exchange with synchronous messages:

process X:	*process* Y:
`send` A `to` Y	`receive` A `from` X
`receive` B `from` Y	`send` B `to` X

Restricting the ordering of messages requires the programmer to think more carefully. The result, however, can be a more reliable, data-race free program. Fixing the communication schedule means that multiple runs of the program yield the same result, which greatly simplifies debugging. Furthermore, even asynchronous messages can produce unexpected deadlocks, as explained later.

The main performance drawback of synchronous messages is its inability to overlap communication with other activities. A synchronous send statement always observe the full latency of a round-trip message. We can see that in the first program, the two asynchronous send statements can execute in parallel; while in the second program, the two send statements are forced to execute sequentially. Synchronous messages, however, do have some performance advantage. It is possible to implement synchronous messages in such a way that the receive statement *pulls* the message from the send statement. Using this protocol the message can be transferred directly between the two user buffers to avoid the system buffering overhead. The savings in buffering cost can be quite significant for large messages.

4.2 Blocking vs. non-blocking message passing

A process is said to block if its execution is suspended for some external event. In the context of message passing a process can block for three types of events: the arrival of a message, the synchronization between two processes, and the availability of buffer space. The first two types of events are evident in the synchronous send and receive primitives described above. The third type of event is explained in details below.

When performing an asynchronous send operation, the programmer specifies the message data via a pointer to a user buffer. The send operation can not return until the entire message content is moved out to a safe place, as the statements following the send may modify the user buffer. There are three possible places to which the message can be moved: the sending process, the network, and the receiving process. Buffer space must be negotiated for the latter two cases, and thus the sending process may block for an unpredictable amount of time. Even in the first case the sending process may block if the local buffer is full, although that rarely happens. Consider the exchange operation again. Suppose that local buffering is used and the run-time system delivers the message only when there is space in the network. One may write the following program to carry out the exchange of several large arrays, using the asynchronous send and receive primitives:

process X :	*process* Y :
`send` $A1$ `to` Y	`send` $B1$ `to` X
`send` $A2$ `to` Y	`send` $B2$ `to` X
`receive` $B1$ `from` Y	`receive` $A1$ `from` A
`receive` $B2$ `from` Y	`receive` $A2$ `from` A

Suppose the sizes of the arrays are 10K bytes each. The above program will deadlock if the amount of space available for buffering outgoing and incoming messages, plus the buffering space in the network, is less than 20K. In this case the two processes fill up each others buffer space and neither can proceed to the receive statements. The danger of using asynchronous blocking primitives is then the unexpected deadlocks due to system-dependent limitations, which is hidden beneath the communication abstraction. Another disadvantage is the buffering cost required to temporarily save the message content at the sending and the receiving sites. A worst case scenario for asynchronous blocking primitives is *quadruple buffering*, in which the message is copied four times, two of them are the scatter/gather type of operations at the user level, and the other two are between the user buffer and the system buffer.

A message passing primitive can be made truly non-blocking only if the programmer helps managing the buffers. A non-blocking send returns as soon as the address of the user buffer is recorded. To ensure the integrity of the message the user must not modify the user buffer before the message is sent. The completion of a send operation is checked by another primitive. Since no buffering is required the send statement always have a bounded latency, and thus no deadlock can occur due to sends blocking later receives. Messages may still need to be buffered at the receiving site if non-blocking receives are used. User level copying may also be required if the user buffer is needed for continuing computation, or if the message is not laid out contiguously (e.g., a column in a two dimensional array with row-major layout).

A different interface of non-blocking messages is found in the CE/RK implementation. Instead of having the programmer check before re-using buffers, CE/RK forbids reuse at all – a send automatically frees the user message buffer. Whatever the interface is, using non-blocking messages gains performance and flexibility, at the cost of higher programming complexity.

4.3 Active vs. Cooperative Message Passing

So far we have discuss various options of sending and receiving messages, assuming that the messages to be sent and received by all processes are known a priori. There are many applications where the communication pattern is irregular and difficult to predict. Cooperative message passing is clearly not suitable for these applications.

Active message passing allows a message to trigger a procedure at the receiving site. This procedure can be used to performed application specific message handling or to schedule new computation. For example, the following program exchanges two arrays (with 10 integers each) between process X and Y:

```
int f = 0;
put(int *a, int i, int x)
     a[i] = x;
     f = f + 1;
```

process X:	*process Y*:
`for (i=0;i<10;++i)`	`for (i=0;i<10;++i)`
`call put(A,i,A[i]) at Y`	`call put(B,i,B[i]) at X`
`wait until f equals 10`	`wait until f equals 10`

We assume the addresses of A (and B) are the same for both processes.

The purpose of active messages is to retrieve messages from the network as soon as possible, and to trigger the proper operation upon receiving the message. scheduling new computation. In general a process can be interrupted to handle active messages at any time. Therefore, the execution of the handler must have bounded latency to avoid unexpected problems. For example, the above program can be rewritten so that each process requests for the data it needs:

```
get(int *a, int i, int P)
     call put(a,i,a[i]) at P
```

process X:	*process Y*:
`for (i=0;i<10;++i)`	`for (i=0;i<10;++i)`
`call get(B,i,X) at Y`	`call get(A,i,Y) at X`
`wait until f equals 10`	`wait until f equals 10`

Assuming that a process does not accept incoming active messages when executing the handler. The above program may deadlock because the put statement may block due to limited buffering in the network. On the other hand, if one accepts incoming active messages when executing a handler (as in CMAM), the nesting of handler functions may be arbitrarily deep, possibly causing the stack to overflow. The CMAM implementation solves this problem for the client/server type of communication by providing two types of active messages, each using a separate network on the CM-5 (the request and the reply network). However, it is not clear if a general solution exists for other types of communication network architectures. Thus the user is warned to leave communication out of the handler.

The advantage of active messages is the ability to perform asynchronous communication with high efficiency. In the above example all messages contain sufficient information to specify their destination addresses, and thus no buffering is needed for receiving these messages. of the destination. The disadvantage of active messages is the complexity of programming, since the programmer must now consider unpredictable external events that may change the local state of a process.

4.4 Determinism

Assume that the behavior of a process is dependent only on its input and the messages it receives. This assumption is valid for ordinary message passing programs without shared mutable states. Then a message passing program is deterministic if, given the same input, each receive statement always picks up the same message. Under this definition nondeterminism can only be introduced by the reordering of messages in the network and at the sending and receiving sites.

We now investigate how different message passing semantics affect the ordering of messages. The most restrictive ordering is imposed by synchronous primitives without wildcards. This combination always yields a deterministic order, since the messages sent by the same process follows the program order of the send statements, and the messages sent by different processes follows the program order of the receive statements.

A less restrictive scheme is to use synchronous sends and synchronous receives, but allow the receive statements to use wildcard sender ids or message tags. The process now sees its sends handled in the program order, but the ordering between messages sent by different processes is not deterministic. This scheme is well suited for the client/server type of communication [21].

The next scheme is to use asynchronous messages. In this case the sending process will not know when and in what order its outgoing messages are received. The ordering of messages depends on the use of sender ids and message tags.

Active messages introduce the most nondeterminism. As in the asynchronous scheme the message order can not be known a priori. The additional confusion comes from the fact that there is no well defined communication point; the local state of a process can be mutated by an incoming message at any point of the program. The programmer must then carefully code the active message handlers to ensure consistency of the program.

We now use an example program to illustrate the message ordering enforced by different message passing semantics. Consider the Gaussian elimination operation which converts a matrix into the upper triangular form. For the moment it suffices to consider a 3-by-3 coefficient matrix whose columns are distributed over 3 processes. The sequential algorithm for the Gaussian elimination of a 3-by-3 matrix (no pivoting) is given below:

```
Gaussian elimination of a 3-by-3 matrix A:

for (i=0;i<3;++i)
     for (j=i+1;j<3;++j)
          subtract row i * A[j,i] / A[i,i] from row j
```

A first-cut implementation using asynchronous messages follows:

Process 1:

```
compute s1 = A[1,0]/A[0,0]
send s1 to process 2
send s1 to process 3
compute s2 = A[2,0]/A[0,0]
send s2 to process 2
send s2 to process 3
```

Process 2:

```
receive s1
subtract A[0,1]*s1 from A[1,1]
receive s2
subtract A[0,1]*s2 from A[2,1]
compute s3 = A[2,1]/A[1,1]
send s3 to process 3
```

Process 3:

```
receive s1
subtract A[0,2]*s1 from A[1,2]
receive s2
subtract A[0,2]*s2 from A[2,2]
receive s3
subtract A[1,2]*s3 from A[2,2]
```

Since a synchronous send always blocks until its message is received, s1, s2, and s3 will arrive in the desired order. Therefore there is no need for the receive statements to specify the sender id or the message tag. The main drawback of the above program is that the communication latency of each send operation is observed by the sender. It is clear that the send statements can be overlapped without affecting the semantics of the program. This can be done by replacing the synchronous sends with asynchronous sends. However, there may be a race condition between s1, s2, and s3, causing incorrect row updates. To enforce the correct ordering we add tags to the messages and obtain the following program:

Process 1:

```
compute s1 = A[1,0]/A[0,0]
send s1 tag 1 to process 2
send s1 tag 1 to process 3
compute s2 = A[2,0]/A[0,0]
send s2 tag 2 to process 2
send s2 tag 2 to process 3
```

Process 2:

```
receive s1 tag 1 from process 1
subtract A[0,1]*s1 from A[1,1]
receive s2 tag 2 from process 1
subtract A[0,1]*s2 from A[2,1]
compute s3 = A[2,1]/A[1,1]
send s3 to process 3
```

Process 3:

```
receive s1 tag 1 from process 1
subtract A[0,2]*s1 from A[1,2]
receive s2 tag 2 from process 1
subtract A[0,2]*s2 from A[2,2]
receive s3 from process 2
subtract A[1,2]*s3 from A[2,2]
```

Both of the above two programs are deterministic because the receive statements always return the same messages. If we look further into the code, we can see that although the ordering of messages sent from different processes are important (s1,s2 must be received before s3 in this case), the ordering of messages from the same process is irrelevant. Under this weaker constraint we may be able improve performance further by allowing the processes to pick up s1 or s2 in the order they arrived:

Process 2:

```
receive s from process 1 with tag t
subtract A[0,1]*s from A[t,1]
receive s from process 1 with tag t
subtract A[0,1]*s from A[t,1]
compute s3 = A[2,1]/A[1,1]
send s3 to process 3
```

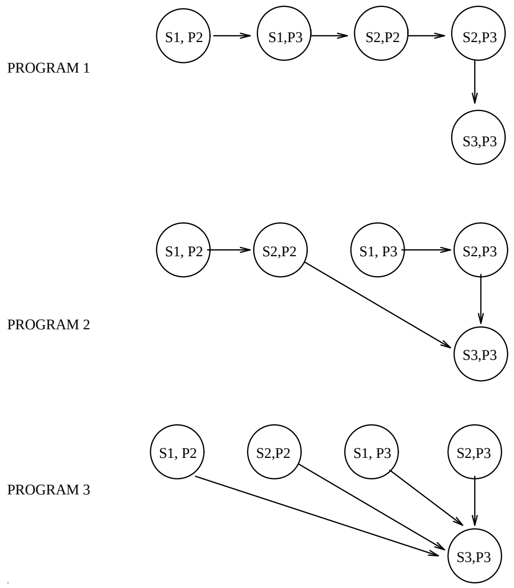

Figure 1: Ordering of messages. The messages are labeled with their destination processes.

Process 3:

```
receive s from process 1 with tag t
subtract A[0,2]*s from A[t,2]
receive s form process 1 with tag t
subtract A[0,2]*s from A[t,2]
receive s3 from process 2
subtract A[1,2]*s3 from A[2,2]
```

The resulting program is nondeterministic, because a receive statement may not return the same message for different runs when the network is reordering. The semantics of the program stays intact, however.

Figure 4.4 illustrates the message order enforced by these three programs. Figure 4.4 compares how time is spent in these programs.

5 Porting Message Passing Programs

The previous section shows that there are many subtle differences in the semantics of message passing primitives, although the basic interfaces appear to be the same (sends and receives). The programmer must be aware of these differences when he ports programs across libraries, so that he can preserve correctness while achieving good performance.

The focus of this section is the issues in porting message passing programs. We

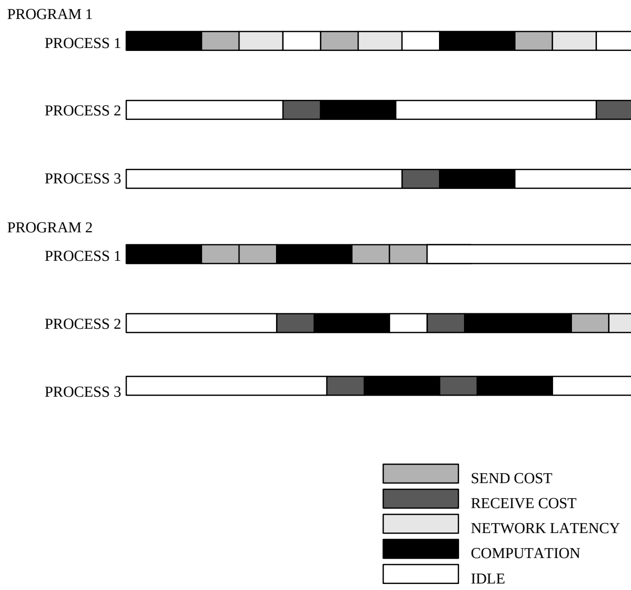

Figure 2: Analysis of running time.

start by categorizing the ten communication libraries according to the semantics of their basic message passing primitives. We then discuss two ways of porting programs – by emulating the original primitives with the new ones on the target machine, or by changing the program to use these new primitives.

5.1 Categorization of the Message Passing libraries

We present these models in order of increasing generality. When a library supports more than one model, we put it in the most general model applicable.

- **Synchronous send and receive, deterministic model.** This model uses synchronous sends and synchronous receives, and further requires that the sender id be fully specified for each receive statement. TCGMSG is in this category.
- **Synchronous send and receive, nondeterministic receive model.** Wildcards can be used to specify the senders in the receive statements, although each send or receive must block for synchronization. CMMD belongs to this category.
- **Asynchronous send and receive, blocking (buffered) model.** The library buffers messages for asynchronous delivery. P4, PICL, PVM, APPL, Zipcode belongs to this category.
- **Asynchronous send and receive, non-blocking (unbuffered) model.** This model allows the programmer to use non-blocking primitives and to handle the message buffers explicitly. EXPRESS belongs to this category. We put CMAM in

this category because it requires the user to manage buffers, although the current implementation of CMAM does block if the network is congested.

5.2 Porting Message Passing Primitives

It is clear that asynchronous primitives can not be emulated by synchronous primitives, and non-blocking primitives can not emulated by blocking primitive. Active messages, however, can be emulated by cooperative messages with non-blocking receives. This is analogous to detecting external events via polling instead of via interrupts. The current implementation of CMAM is actually built using this method, and the programmer must insert sufficient amount of polls to receive incoming active messages. The main motivation for polling is the lack of light-weight, user-level interrupts on the CM-5.

Emulation in the opposite direction is straightforward. A synchronous message can be emulated by a pair of asynchronous messages, one of which serves as the acknowledgment; a buffered (or blocking) message can be replaced by a unbuffered (or non-blocking) message and some copying to avoid clobbering; cooperative message passing can be emulated by active message handlers that simply buffer the messages for later retrieval.

5.3 Porting Message Passing Applications

Sometimes a programmer has to modify his application to run on a new system, either because the original primitives can not be emulated by the new, and possibly more restrictive primitives, or because a straightforward emulation can not take advantage of the features in the new system.

It would be hard to describe a porting scheme that applies to all message passing programs. We can, however, observe some common sources of errors when porting such programs. For example, when one goes from asynchronous messages to synchronous messages, the main problem is to avoid deadlocks due to unnecessary synchronization. We can imagine each send/receive pair as a solid line between the two processes (implying a barrier synchronization), and we must re-arrange the pairs in such a way that no two solid lines cross. Sometimes porting applications to a more restrictive message passing model may require redesigning the underlying algorithm. For example, an data-flow style algorithm may have to be replaced by a synchronous algorithm to run on top of synchronous message passing library, because the new system requires predictable communication patterns.

6 High-level Communication Primitives

In this section we summarize the communication primitives other than the simple sends and receives. Functionally they can be implemented by a few send and receive primitives, but they have two important advantages. First, these communication primitives provide the programmer with high-level programming abstractions, making it easier to develop, debug, and maintain message passing programs. Second, they help hiding the architecture-dependent optimizations from the user. The resulting code is thus more portable.

Global communication primitives

- **broadcast**: distribute a copy of the message to all processes. A synchronous broadcast blocks until everyone has received its copy. This operation is carried out cooperatively by a broadcast and the corresponding *receive broadcast* calls, thus a barrier synchronization among the participants is implied. Asynchronous broadcast is functionally equivalent to a set of asynchronous sends. A **multicast** operation is a special form of broadcast that delivers a message to a subset of all processes.
- **synchronous exchange**: exchange two messages between a pair of processes and block until both are received. The exchange primitive is particularly useful for grid-type communication.
- **scatter**: distribute elements in the sender's array to the recipients according to their indices. For example, the ith element of the array is sent to process i in a global scatter operation involving all processes in the system.
- **gather**: collect elements sent by multiple processes into a local array. For example, the element sent by the ith process is placed at the ith position of the recipient's array.
- **scan**: combine and distribute the results computed by different processes. The programmer can use one of the standard associative operators (e.g., min, max, and sum) or supply his own to perform the combination. A special case of the scan primitive is the **prefix** operation, where a process combines all the elements computed by the processes ordered before it. For example, a prefix operation using the sum operator computes the running sum over a distributed array. Another special case is the **reduce** operation which combines elements from all processes and ships the result to one or all of them. For example, a reduce operation using the min operator finds the minimum value in a distributed array.
- **crystal router**: the crystal router [5] defers individual message transfers until the processes synchronize, at which point the accumulated messages are sent *en masse*. The operation exploits the regular communication pattern in *loosely synchronous* applications (such as meshes in DIME[17]). These applications decompose the problem into cycles of computations, and the results produced (and distributed) by a certain compute phase are not acted upon until the next cycle.

The latency of broadcasts and scans can be made logarithmic with the number of processes by applying techniques such as *recursive doubling* [12, 15] . On some machines (such as the CM-5 or the shared-memory multiprocessors) broadcasts can be performed efficiently by the hardware.

Synchronization primitives

Synchronization plays an important role in parallel programs. Explicit synchronization primitives coordinate processes through the exchange of *control* information instead of data (or messages). Some libraries do not have provisions for synchronization at all. The disadvantage is two-fold: the programmer must either use machine dependent synchronization constructs in the program, or embed the synchronization primitives within messages. Such embedding obscures the intent of the programmer and prevents optimizations that make use of machine-specific features. Some libraries treat zero-length messages as synchronization messages and handle them using specialized routines to get around the latter problem.

The following is a list of useful synchronization primitives:

- **barrier synchronization**: block until all processes have reached the synchronization point.
- **assert event and wait on event**: wait for a certain event to be signaled by some process. The event mechanism can be used to implement pair-wise synchronization or fuzzy barriers among a subset of processes.

Some machines (such as CM-5) have direct hardware implementation of these synchronization primitives. On other machines, synchronization can be implemented as light weight messages requiring no buffering and flow control overhead.

Distributed data structures

Some libraries provide sophisticated message formats and communication abstractions to accommodate certain applications. These high-level formats are conveyed as part of the message to optimize the performance of sends and receives. The high level abstractions also ease the programming task. Listed below are the distributed data structures found in the libraries:

- **grids**. Many scientific applications can be parallelized by decomposing the problem domain into fixed-size grids. A grid can be fully specified by its dimensionality. Communication on grids is usually between nearest neighbors, possibly mixed with infrequent global communications such as broadcast and reduce.
- **vectors and matrices**. Vectors and matrices are the core of most numerical applications. Standard operations on these data structures are in general very well defined (the three levels of BLAS[1]). Related communication operators have

[1]BLAS stands for Basic Linear Algebra Subroutines

also been proposed (LACS [2]).

- **meshes.** Meshes are generalization of grids to deal with decompositions of irregular shape, resolution, and connectivity. Partitioning the mesh elements among processes can be a difficult task for the programmer. It is even more complex with *adaptive meshes* where the mesh structure is refined over time and dynamic load balancing must be performed. Implementing the mesh operators in the communication library enables the programmer to concentrate on application specific computations.

Some libraries such as Zipcode provide abstractions to allow the programmer compose message passing programs with little modification. The concept of *mailing context* and its *inheritance* in Zipcode let the user build up larger data structures using existing modules.

7 Conclusion

Programming with the communication libraries has the following advantages. First, the programming task is simplified by the use of high level communication primitives such as broadcast and reduce. Secondly, the libraries support the message passing programming paradigm on machines where it is otherwise unavailable. Finally, the cost of porting applications to different machines is greatly reduced if the applications are developed using a portable communication library. Using these libraries does induce some performance overhead, but the cost is tolerable considering the advantages they offer.

The interfaces provided by the libraries vary in expressiveness and portability. However, there does exist a set of primitives that are supported by most (seven) of the ten libraries. These primitives include simple send and receive, broadcast, reduce, and barrier synchronization.

However, even for these simple primitives there can be variations in semantics. Examples include the use of message buffers (visible vs. invisible to the user) and the way communication is performed (synchronous vs. asynchronous, blocking vs. non-blocking). The interpretation of these primitives varies for different libraries, and sometimes even for different machine configurations supported by the same library (see table 4 for examples). To settle on a *universal* message passing interface these differences need to be resolved – we think it is now high time for standardization.

References

[1] *Express C: User's Guide – 3.0*, Parasoft Corporation.

[2]LACS stands for Linear Algebra Communication Subroutines

Table 2: Summary of communication primitives

Primitive	P4	PICL	PVM	TCGMSG	APPL	iPSC/2
low-level communication primitives						
Synchronous sends	Y			Y	Y	
Blocking sends	Y	Y	Y		Y	Y
Non-blocking sends						Y
Synchronous receive	Y			Y	Y	
Blocking receive	Y	Y	Y		Y	Y
Non-blocking receive	Y	Y	Y		Y	Y
Exchange						
Vector messages						
Active messages						Y
Global communication primitives						
Synchronous broadcast		Y		Y		Y
Asynchronous broadcast	Y		Y		Y	
reduce	Y	Y		Y	Y	Y
Scan						
Scatter-Gather						Y
Crystal router						
Synchronization primitives						
barrier	Y	Y	Y	Y		Y
Signal-Wait	Y		Y			
Distributed data structures						
Grids						
Vector and matrices						
Meshes						

** only for short messages

Primitive	EXPRESS	CMMD	Zipcode	CMAM	PARMACS
low-level communication primitives					
Synchronous sends		Y			Y
Blocking sends	Y		Y		Y
Non-blocking sends	Y				
Synchronous receive		Y			Y
Blocking receive	Y		Y		Y
Non-blocking receive	Y	Y			
Exchange	Y	Y			
Vector messages	Y	Y			
Active messages	Y			Y	
Global communication primitives					
Synchronous broadcast	Y?	Y			
Asynchronous broadcast			Y		
reduce	Y	Y	Y		
Scan		Y	Y		
Scatter-Gather	Y	Y	Y		
Crystal router	Y				
Synchronization primitives					
barrier	Y	Y	Y	Y	Y
Signal-Wait	Y	Y			
Distributed data structures					
Grids	Y		Y		
Vector and matrices			Y		
Meshes	Y				

[2] A. Skjellum, *Zipcode: A Portable Communication Layer for High Performance Multicomputing – Practice and Experience*, March 1991.

[3] *Application Portable Parallel Library version 2.0*, NASA Lewis Research Center, January 1992. Source code available from `fsang@lerc.nasa.gov`.

[4] *CMMD User's guide*, Thinking Machine Corporation.

[5] J. Flower and A. Kolawa, *A "packet" History of Message Passing Systems,* Parasoft Corporation.

[6] R. Lusk, *P4 Version 0.2 Documentation.* Source code is available from `info.mcs.anl.gov`.

[7] G. Geist et al., *A User's Guide to PICL: A Portable Instrumented Communication Library, Oak Ridge National Lab. Report No. ORNL/TM-11616*, February 1991.

[8] P. Worley and M. Heath, *Performance Characterization Research at Oak Ridge National Laboratory.*

[9] V. Sunderam, *PVM: A Framework for Parallel Distributed Computing*, PVM 2.3 documentation. Source code available from netlib.

[10] A. Beguelin et al., *A User's Guide to PVM Parallel Virtual Machine, Oak Ridge National Lab. Report No. ORNL/TM-11826*, July 1991.

[11] R. Harrison, *TCGMSG Version 4.0*, Theoretical Chemistry Group, Argonne National Laboratory, December 1991. Source code available from `harrison.tcg.anl.gov`.

[12] A. Skjellum and C. Baldwin, *The multicomputer toolbox: Scalable Parallel Libraries for Large-Scale Concurrent Applications, LLNL Numerical Mathematics Group Report No. UCRL-JC-109251*, December 1991.

[13] *PARMAC documentation.* Source code available from netlib.

[14] C. Seitz et al., *The C Programmer's Abbreviated Guide to Multicomputer Programming*, it Caltech Computer Science Technical Report No. Caltech-CS-TR-88-1, January 1988.

[15] J. Gistafson et al., *Development of Parallel Methods For a 1024-Processor Hypercube, SIAM Journal on Scientific and Statistical Computing*, Vol.9 No.4, July 1988.

[16] L. Bomans and D. Roose, *Benchmarking the iPSC hypercube multiprocessor, Concurrency: Practice and Experience*, Vol.1, pp.3-18, September 1989.

[17] R. Williams, *DIME: A Programming Environment for Unstructured Triangular Meshes on a Distributed-Memory Parallel Processor*, C3P Report 502, 1988.

[18] G. Wilson, *Design Principle for Message-Passing System*, Draft paper, January 1991.

[19] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, *Active messages: a mechanism for integrated communication and computation*, *19th Annual International Symposium on Computer Architecture*, pp.256-66, May 1992.

[20] E. Lusk et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc., 1987.

[21] W. Gentleman, *Administrators and multiprocessor rendezvous mechanisms.*, *Software - Practice and Experience* Jan. 1992, vol.22, no.1, pp1-39.