An Introduction To Message Passing Paradigms

David W. Walker
Oak Ridge National Laboratory, Oak Ridge, U.S.A.

Abstract

This paper discusses some of the most important message passing approaches used on high performance computers, and describes their central features. Occam and Fortran M are coordination languages in which processes communicate through typed channels. Split-C and Linda provide support for shared memory programming. In Split-C all processes can access a global address space through global pointers. The central feature of Linda is an associative shared address space called "tuple space." PVM and MPI are examples of message passing libraries that are typically used with seqential languages such as C and Fortran. MPI provides support for modular application libraries through the communicator abstraction. PVM features extensive job and process control functionality which has resulted in it being widely used on networks of workstations.

Keywords: Parallel computing; message passing; occam; Fortran M; Linda; Split-C;PVM;MPI

1 Introduction

This paper gives an introduction to some of the more widely-used and interesting approaches to the message passing style of programming high performance computers, in particular distributed memory computers. It is not my intent to give a complete review of the message passing paradigm, nor to describe in detail the features of any particular approach. Rather, this paper is an eclectic look at a few of the languages, extensions, and message passing libraries that are widely-used and/or have been influential in advancing parallel programming and the programming of parallel computers. This paper should be useful to novices wanting to get a basic understanding of some of the options open to them for using parallel computers. Wherever possible references are given to more detailed information available in the literature or electronically should readers wish to delve more deeply.

The rest of this paper is organized as follows. Section 2 presents an overview of message passing, and places it in the broader context of parallel programming paradigms. In Section 3, coordination languages are discussed, and the main features of Occam and Fortran M are described. Linda and Split-C, described in Section 4, strictly speaking are based on a shared memory approach, but they are included here on the basis of their strong message passing flavor. Two message passing libraries, PVM and MPI, are discussed in Section 5.

2 The Message Passing Approach

In the past decade the concept of message passing has become closely associated with the efficient use of parallel multicomputers and with distributed computing. However, as Hoare relates in his very readable article on occam and the transputer [25], the concept of message passing was also strongly motivated by its use in the mid-1960s in the design of operating systems. The idea of communicating sequential processes as a model for parallel execution was developed by Hoare [24] in the 1970s, and is the basis of the message passing paradigm. This paradigm assumes a distributed process memory model, i.e., each process has its own local address space. Processes cooperate to perform a task by independently computing with their local data and communicating data between processes by explicitly exchanging messages. The message passing approach is particularly well-suited to computers with physically distributed memory since there is a good match between the distributed memory model and the distributed hardware. However, message passing can be used on shared memory and sequential computers, and, indeed, can be used as the basis for the development of portable and efficient programs on all these architectures. An alternative approach is based on the shared process memory model, in which all processes have access to the same global address space. As might be expected, this approach works well on shared memory architectures, but may also be supported in software on distributed memory computers. An example is Orca which is a language that supports shared objects [9]. Other alternatives to the message passing approach include data parallel extensions to sequential languages, such as High Performance Fortran [27], and implicit parallel languages employing functional parallelism, such as Sisal [16].

There are two main ways of writing a parallel message passing program. The first is through a coordination language such as Fortran M [20] or occam [28]. These are specialized languages for specifying concurrency, communication and and synchronization, and will be discussed in greater detail in Section 3. The second way of performing message passing is with calls to a message passing library from within a sequential program. This has proved to be a very popular way of writing concurrent applications since it is expressive, closely models the functionality of the parallel hardware, and permits explicit management of the memory hierarchy. A number of different message passing libraries have been developed over the past decade in addition to the hardware specific message passing libraries provided by different parallel computer vendors. Express is a commercial parallel computing environment produced by Parasoft Inc. [30], and is based on the CrOS III message passing library [21]. Other message passing libraries include PARMACS [11], which is based on `p4` [10], and Zipcode [32]. The special issue of *Parallel Computing* on message passing include more detailed articles on these, and other, message passing libraries [1].

3 Coordination Languages

Coordination languages are languages with explicit support for expressing parallelism, communication, and synchronization. In this section two examples, occam and Fortran M, will be considered.

3.1 Occam

Occam was designed specifically for programming transputers, indeed, it is often referred to as the assembly language of the transputer. Occam is based on Hoare's model of communicating sequential processes [24] which allows programs to be analyzed mathematically to check for program correctness. In occam, parallelism is expressed through the `PAR` construct. Thus, the code fragment

```
PAR
  process 1
  process 2
  process 3
```

indicates that the three processes may be executed in parallel. Similarly, the `SEQ` construct is used to specify the sequential execution of a set of processes. There are three primitive processes in occam, namely, input to a channel, output from a channel, and assignment. More complex processes can be built from these primitives by defining procedures. In occam, messages are communicated through uni-directional, typed channels, Each channel has one sender process and one receiver process. A channel is said to be *typed* if the datatypes of the variables that may be communicated by the channel are predefined. The following occam code illustrates a simple use of a typed channel.

```
CHAN OF INT chan :
PAR
  chan ! 2
  INT x, z :
  SEQ
    chan ? x
    z := x
```

The above program illustrates the use of the three primitive processes, as well as the `PAR` and `SEQ` constructs. The first line of code declares a channel, `chan`, that will be used to communicate an integer. The `PAR` construct refers to two processes. One process outputs the integer 2 on channel `chan` (`chan ! 2`). The second process is a compound process beginning with the `SEQ` and consists of two primitive subprocesses executed sequentially. The first inputs the data from channel `chan` into the variable `x` (`chan ? x`), and the second copies the value of `x` to variable `z` (`z := x`). Note that `x` and `z` are declared immediately before the process that uses them, and that their scope is limited to just that process.

In occam, communication is *synchronous*. This means that that the sending process must output to a channel and the receiving process must input from that channel in order for the communication to occur. If either the sending or receiving process is not yet ready to participate in the communication then the other process will wait. In incorrectly designed programs *deadlock* may occur. Deadlock arises in a parallel program if one or more processes are waiting for an event that will never occur, and is a common programming bug. The following program gives an example of deadlock.

```
CHAN OF INT chan1, chan2 :
PAR
  INT x :
  SEQ
    chan1 ! 2
    chan2 ? x
  INT z :
  SEQ
    chan2 ! 4
    chan1 ? z
```

Here, one process outputs the value 2 on channel `chan1`, and another outputs the value 4 on channel `chan2`. Both processes then wait indefinitely because neither can continue until the other inputs to their respective channels. Thus, deadlock occurs. In this case, deadlock can be avoided simply be reversing the order of the input and output processes in either (but not both!) of the parallel processes.

Communication is said to be *non-deterministic* if the receive on the receiving process may be matched by any one of several potential send operations on other processes. Non-deterministic communication is often used to improve the performance of a parallel program by ensuring that the receiving process is supplied with enough data from other processes to keep it busy. However, non-determinism may be introduced unintentionally into a program and is the second most common type of bug in parallel programming. A program with this sort of bug may fail intermittently since its behavior depends on the synchronization between processes. Non-determinism may be introduced into an occam program through the ALT construct. An ALT construct monitors several channels, each of which is associated with a process. Only one of these processes is executed, and that is the one associated with the channel that first produces an input. Consider the following code fragment.

```
CHAN OF INT chan1, chan2 :
INT x, y :
ALT
  chan1 ? x
    SEQ
      y := 2*x + 1
      chan3 ! y
  chan2 ? x
    SEQ
      y := 2*x
      chan3 ! y
```

Here a process waits for input on either chan1 or chan2. When it receives input on either channel it performs a simple computation and outputs the result on chan3. This code fragment does not show the processes that produce the input or consume the output. If the process receives input x first on chan1 it computes $2x + 1$ and outputs the results on chan3. If it receives input first on chan2 it will output $2x$ on chan3. Thus, there are two possible outcomes, and it is not possible to determine *a priori* which will actually occur.

In addition to the PAR, SEQ, and ALT constructs occam also provides IF and WHILE constructs for controlling program flow. All of these constructs, except WHILE, may be replicated. Replicated SEQ and PAR constructs create sets of similar processes that execute sequentially or in parallel. A replicated PAR may be used for the single-program, multiple-data (SPMD) style of programming. A replicated ALT can be used to monitor an array of channels, and a replicated IF can be used to create a conditional with similar clauses. Occam also provides a mechanism for placing processes on physical processors which may improve the performance of a program, but does not affect its correctness. Another important and useful feature is occam's use of protocols to allow channels to carry data of mixed types. Protocols can also be used for communicating fixed and variable length arrays, and for allowing a single channel to carry messages of differing format.

It is hoped that the above gives a good idea of the main features of the occam language and how they are used. For further information the interested reader is referred to the excellent (and short) introduction by Pountain and May [31], and to the more advanced text by Jones and Goldsmith [26] which also contains a good bibliography. The occam web page contains pointers to information on occam compilers, documentation, and projects [2].

3.2 Fortran M

Fortran M was developed by Ian Foster of Argonne National Laboratory and Mani Chandy of the California Institute of Technology [20], and consists of Fortran 77 plus a set of extensions for process creation and control, and communication. Fortran M is broadly similar to occam in that it provides constructs for creating parallel processes, communicates data through uni-directional, typed channels, and supports the mapping of processes to processors. However, there are important differences. One important difference is that communication in Fortran M is not synchronous because the send may complete before the matching receive operation is started at the other end of the channel. The second important difference is that channels are specified as links between *inports* and *outports*. A process sends data through an outport and receives data through an inport. Furthermore, ports may be communicated giving rise to dynamic channels. Fortran M does not have any equivalent of the `SEQ`, `WHILE`, and `IF` constructs as these are subsumed within the Fortran language. Fortran M introduces non-determinism through the `MERGER` and `PROBE` constructs, which replace occam's `ALT` construct.

The following program illustrates the basic message passing capabilities of Fortran M.

```
program swap1
INPORT  (integer) portin(2)
OUTPORT (integer) portout(2)
CHANNEL (in=portin(1), out=portout(2))
CHANNEL (in=portin(2), out=portout(1))
PROCESSES
  PROCESSCALL swap (portin(1), portout(1))
  PROCESSCALL swap (portin(2), portout(2))
ENDPROCESSES
end

PROCESS swap (in, out)
INPORT  (integer) in
OUTPORT (integer) out
integer p
SEND (out) p
RECEIVE (in) p
end
```

This program declares two inports and two outports capable of receiving and sending an integer. Two channels are then established, each connecting an inport to an outport. The section of code beginning with "`PROCESSES`" and ending with "`ENDPROCESSES`" constitutes a process block, and is similar to the `PAR` construct in occam. The two processes in the process block are executed concurrently, with each process calling `swap` with different inport and outport arguments. The `swap` processes exchange an integer value, `p`. Note that deadlock will not occur here because the send operation can complete before the matching receive operation is initiated.

In Fortran M the `MERGER` construct can be used to connect multiple outports to a single inport. This can be used to introduce non-determinism into a parallel program since data from only one of the outports will be received at the inport. The `MERGER` construct is equivalent to a replicated `ALT` in occam. The following Fortran M code fragment connects three producer processes to one consumer process.

```
INPORT  (integer) portin
OUTPORT (integer) portout(3)
MERGER  (in=portin, out=pout(:))
PROCESSES
  call consumer (pin)
  PROCESSDO i=1,3
    PROCESSCALL producer (pout(i))
  ENDPROCESSDO
```

```
ENDPROCESSES
```

In this example the `MERGER` statement connects the three elements of the array of outports `portout` to the inport `portin`. Within the process block the subroutine `consumer` is executed on the current process, and the `PROCESSDO` loop creates three `producer` processes. The `consumer` routine is passed the inport of the merger and each `producer` process is passed one of the outports.

Nondeterminacy may also be introduced into a Fortran M program through the `PROBE` construct which is used to check if a message is waiting to be received on an inport. `PROBE` returns a logical variable to indicate if a message is pending on a specified inport, and is typically used to allow a process to do useful work while intermittently checking if a message is ready for receipt, thereby avoiding idle time.

A useful feature of Fortran M is the ability to communicate an inport or outport from one process to another. By communicating a port the processes connected by a channel can be changed. Thus, instead of statically connecting two fixed processes channels may be dynamic. Dynamic channels are often used when a process wants another process to return data to it. In such cases the process creates a channel and then sends the channel's outport to another process. That process then has the ability to send data back to the original process. This approach is useful for dealing with "migrating computations" in which a task may move between processes before completing, and then sends back the results of the work to the process that originated the task.

Fortran M provides mechanisms for process placement. As in occam, this affects the performance, but not the correctness, of programs. Process placement is based on the concept of a *virtual process array* which is declared in the Fortran M program. Fortran M provides annotations that can be used to specify where in the virtual process array a process is to run, and for subdividing the virtual process array so that different processes run on different "subcomputers."

For further details of the Fortran M language readers are referred to the paper by Foster and Chandy [20], and to the Fortran M manual [18]. Foster has also written a book that discusses Fortran M as well as other approaches to parallel programming, such as CC++ and High Performance Fortran, that are not covered in this paper [19]. All of these publications are available from the Fortran M web page at Argonne National Laboratory [3]. If you want to try out Fortran M for yourself the necessary software can also be obtained from the web page.

4 Shared Memory Paradigms

In this section we discuss two alternatives to the message passing paradigm, Linda and Split-C. These provide support for shared memory, and were designed with distributed memory implementations in mind.

4.1 Linda

Linda was developed by Gelertner and colleagues, and is a set of simple extensions to a conventional programming language such as C or Fortran. Linda is based on an associative shared memory known as "tuple space." A tuple is an ordered sequence of data. For example the tuple

```
('array x', 5, [1 4 2 7 9])
```

is a tuple consisting of a string, an integer, and an array of integers. In Linda processes only interact through tuple space, and routines are provided for placing tuples into, and extracting tuples from, tuple space. Tuples may be created and placed in tuple space using the `out` and `eval` operations. Tuples may be read or removed from tuple space using the `rd` and `in` operations. Linda's extensions consist of just these four operations making it very simple, but at the same time very expressive.

The `out` operation evaluates the tuple fields sequentially on the calling process, and places the resulting tuple into tuple space. For example

```
out ('function', i, f(i))
out ('common data', /params/)
out ('array section', i, j, x(i:j)).
```

For the first of these the calling process evaluates the function f for argument i, making this the third component of the tuple, and places the resulting tuple into tuple space. The second example places a tuple into tuple space consisting of the string `'common data'` and contents of the common block `params`. The third example shows how to place an array section into tuple space consisting of a string, two integers (with $i \geq j$), and a section of the array x starting with $x(i)$ and going up to $x(j)$. It should be noted that `out` is similar to a send operation, but is more general — instead of sending data from one process to another data are sent from a process to tuple space, where it can be sequently retrieved by another process. Also Linda supports self-describing messages in a very natural way.

The `eval` operation can be used as a general way of spawning processes. It evaluates the tuple fields in parallel and then places the resulting tuple in tuple space. For example the code fragment

```
do i=1,100
  eval('function', i, f(i))
end do
```

may cause 100 new processes to be created, each of which handles one of the tuples, placing it in tuple space after performing the function evaluation. The `eval` operation is nonblocking on the calling process so the calling process continues without waiting for the `eval` operation to complete. `eval` is typically called by a host process to spawn a set of worker processes which return their results to tuple space, allowing them to be retrieved by the host which can then process them further and/or output results.

There are two operations for retrieving tuples from tuple space. The `in` operation finds a tuple that associatively matches a given template and removes it from tuple space. The `rd` operation is similar to `in` but does not remove the tuple from tuple space. For example the tuple placed into tuple space with

```
out ('message', i, n, data(1:n))
```

can be retrieved with

```
in ('message', i, ?size, ?mydata(1:n))
```

This will match all tuples in tuple space with four fields whose datatypes match, and whose first two fields are `'message'` and `i`. The question marks in front of the last two fields indicate that the values of these two fields do not matter in establishing a match. When a matching tuple is retrieved the variable `size` takes the value of the third field, and the array `mydata` receives the values of the array section in the fourth field. If more than one matching tuple is found one is selected at random. If no matching tuple is found the process waits until one is available.

To illustrate how Linda is used consider the following example in Linda Fortran which integrates the function $\sin x$ from 0 to π.

```
program integrate
parameter (npts=100, pi=3.141592654)
deltax = pi/npts
do i=2,npts
  x = (i-0.5)*deltax
  eval(sin(x))
```

```
      end do
      sum = sin(0.5*deltax)
      do i=2,npts
        in(?result)
        sum = sum + result
      end do
      print *,'The integral is ',sum*deltax
      end
```

In this example the real axis is divided into `npts` equally-sized intervals, and the integral within each integral is approximated by the value of $\sin x$ evaluated at the mid-point of the interval multiplied by the width of the interval. At the start of the program a single process exists which we shall call the host process. The `eval` operation in the first loop causes `npts-1` worker processes to be created, each of which evaluates the integrand at the mid-point of one interval. The host process evaluates the integrand at the mid-point of the first interval. Having evaluated their respective integrands the worker processes place their results in tuple space. In the second loop the host process retrieves the results from tuple space, sums them together, and outputs the approximate integral. Of course, this is trivial example since the integrand is so simple. However, the integrand can be arbitrarily complicated and perhaps involve a lengthy computation. For such problems it makes sense to use a parallel computer.

It should be noted that Linda is capable of simulating message passing by libraries such as PVM and MPI, discussed in Section 5. For example, if a tuple is placed in tuple space as follows

```
      out('message', myrank, dest, tag, 'real', n, data(1:n))
```

and is retrieved thus

```
      in('message', source, myrank, tag, 'real', ?n, ?data(1:n))
```

this is similar in MPI to a standard blocking send matched by a blocking receive that communicates `n` values in a real array from the process with rank `source` to that with rank `dest`. `myrank` is the rank of the calling process in each case.

Linda is well-suited for handling dynamic load balancing by treating tuple space as a "bag of tasks." Tasks to be performed can be placed into tuple space with `eval`, and on completion of a task the results are put back into tuple space.

An important distinction between Linda and approaches using message passing is that there is only a loose coupling between processes — processes interact only with tuple space and not directly with each other. Thus, a process that consumes data in tuple space does not have to even exist at the time the data is placed into tuple space.

Linda is commercially supported by Scientific Computing Associates, Inc. Gelertner and colleagues have written several papers on Linda, for example [12, 13, 14]. These papers and lots more information about Linda are available from the Linda web page at Yale University [4]. There is also an active Linda group at York University in the United Kingdom [5].

4.2 Split-C

Split-C is being developed by David Culler and colleagues at the University of California, Berkeley. Split-C is a set of parallel extensions to the C programming language that provide mechanisms for supporting shared memory. Split-C is based on the single-program, multiple-data (SPMD) model, i.e., a fixed number of processes all execute the same executable code. In Split-C, processes can access a global address through global pointers. We may regard this address space as two-dimensional with processes running in one direction and local memory in the other.

In addition each process can access its own local address space using standard pointers. Dereferencing a global pointer may involve communication and be more expensive than dereferencing a global pointer. Hence, in the interests of performance, global pointers should be used sparingly. The following lines of Split-C illustrate some of the uses of global pointers.

```
int *global gptr;
int xval, proc, *lptr;
xval = &gptr;
gptr = toglobal(MYPROC, lptr);
lptr = tolocal(gptr);
proc = toproc(gptr);
```

The first of the above lines shows how to declare a global pointer. The second line dereferences a global pointer into a local variable. The next three lines make use of some Split-C functions. `toglobal` converts a local pointer to a global pointer. `MYPROC` is a special integer that uniquely numbers a process in the range 0 to `PROCS-1`, where `PROCS` is another special integer giving the number of processes. The functions `tolocal` and `toproc` destructure a global pointer into its local pointer and process number parts. The following routines shows how a global pointer could be used to broadcast `n` integers from a root process to all processes.

```
all_bcast_int(int root, int *val, int n)
{
  int i;
  int *global = toglobal(root,val);
  barrier();
  for(i=0;i<n;i++) *(val++) = *(gptr++);
  barrier();
}
```

The `toglobal` function is used to create a global pointer that references the start of the data on the root process. A barrier is then performed to make sure that the data to be broadcast from the root process is ready. Then each process dereferences the global pointer so the value it points to is stored at the address pointed to by the local pointer, `val`. By incrementing the local and global pointers each of the `n` elements are broadcast. A second barrier then has to be performed to make sure that the root process does not overwrite the broadcast data before all the other processes have received it. This is almost certainly not a very efficient way to perform a broadcast because each time the global pointer is dereferenced data must be communicated from the root to all other processes. It should be noted that when arithmetic is performed on a global pointer another object on the same process is accessed. It is an error to do arithmetic on a global pointer outside of the local address space.

Spread pointers are global pointers that traverse the two-dimensional global address space in the process direction, so that successive objects are on different processes. Spread pointers wrap around in the process direction so if `sptr` is a spread pointer, `sptr+PROCS` points to the next object on the same process, and `sptr+1` points to the object on the next process. The following example shows how to use a spread pointer to sum a set of numbers distributed one to a process.

```
/* include files should go here */
splitc_main(){
  int *spread sptr;
  int *lptr, i, sum=0;
  sptr  = all_spread_malloc(PROCS,sizeof(int));
  lptr  = tolocal(sptr);
```

```
  *lptr = MYPROC;
  barrier();
  on_one{
    for(i=0;i<PROCS;++i) sum += *(sptr++);
    printf("\nThe sum is %i\n",sum);
  }
  barrier();
  all_spread_free(sptr);
  exit(0);
}
```

In this example the routine `all_spread_malloc` is called to create a spread pointer to an integer over the processes. Each process extracts the local part of the spread pointer and stores its process number at that location. Then the macro `on_one` is used to sum the values stored at the spread pointer locations. Note the use of barriers to ensure proper synchronization.

Split-C also provides spread arrays in which one or more dimensions are spread over the processes. By convention, the array dimensions that are distributed lie to the left of the spreader "`::`". For example,

```
int x[10*PROCS]::[10];
```

declares an array `x` whose first dimension is cyclicly distributed over the processes, so `x[i][j]` refers to the jth element on process $i \bmod PROCS$. This is equivalent to a vector of `100*PROCS` elements block cyclicly distributed with a block size of 10. The summation example shown above can be rewritten to use a spread array as follows.

```
static int sarray[PROCS]::;
/* include files should go here */
splitc_main(){
  int i, sum=0;
  sarray[MYPROC] = MYPROC;
  barrier();
  on_one{
    for(i=0;i<PROCS;++i) sum += sarray[i];
    printf("\nThe sum is %i\n",sum);
  }
  barrier();
  exit(0);
}
```

In the preceding Split-C examples global objects have been accessed one at a time. This is likely to be inefficient because the latency associated with the communication is paid for each access. Split-C therefore provides bulk assignment routines for copying to or from arrays of global objects. Thus, the broadcast example above could be implemented by having each process call the routine `bulk_read` to read the data from the root process into local memory in one operation.

In addition to support for a global address space, the second important feature of Split-C is split phase assignment. Split phase assignment allows access to a global object to be overlapped with other useful work. The assignment initiates access to the global object, but the access is guaranteed to have completed only after a subsequent call to the routine `sync()`. The "get" form of split phase assignment places the content of a global reference into a local one, and the "put" form places the contents of a local reference into a global one. The split phase operator is "`:=`".

The get form of the broadcast example may be written as follows,

```
all_bcast_int(int root, int *val, int n)
{
  int i;
  int *global gptr = toglobal(root,val);
  barrier();
  for(i=0;i<n;i++) *(val++) := *(gptr++);
  sync();
  barrier();
}
```

In this version of the example we do not have to wait for one assignment to finish before starting the next, so the use of split phase assignment should result in more efficient code. There are also routines for performing bulk split phase assignment.

When storing into a global location using a standard assignment an acknowledgement is returned to the initiating process when the store has completed. Split-C provides another form of assignment, called *store assignment*, that does not perform this acknowledgement, and is hence faster. A subsequent call to the routine `all_store_sync` synchronizes processes and waits for all the stores to complete. The store assignment operator is "`:-`". The following example shows how to use store assignment in the put form of a broadcast in which the root stores the data on each process.

```
all_bcast_int(int root, int *val, int n)
{
  int *global gptr;
  int i, ip;
  barrier();
  on_proc(root){
     for(ip=0;ip<PROCS;ip++) {
        gptr = toglobal(ip,val);
        for (i=0;i<n;i++) *(gptr++) :- *(val++);
     }
  }
  all_store_sync();
}
```

In this example the root loops over all processes, creates a global pointer into each process' memory, and then does a store assignment to that location. There is also a bulk store assignment routine for the store assignment of arrays.

Only a brief introduction into Split-C has been given here. Split-C also contains a number of macros and auxiliary library routines for performing tasks such as broadcast and reduction. For further details the reader is referred to the tutorial introduction by Culler et al. [15]. The software for implementing Split-C, as well as other information, is available from the world wide web [6].

5 Message Passing Libraries

This section describes two message passing libraries that have emerged as potential standards: PVM and MPI. These libraries are designed for use with a sequential programming language such as C or Fortran. Central to each library is a set of routines for performing point-to-point communication between pairs of processes. Additional routines perform collective communication operations, such as broadcasts and reductions, and process and job management. A recent special issue of the journal Parallel Computing on the topic of message passing contains articles on several widely-used message passing environments [1].

5.1 Parallel Virtual Machine (PVM)

The earliest version of PVM was developed by Vaidy Sunderam in 1989 and 1990 while visiting Oak Ridge National Laboratory (ORNL) [33]. Later versions of PVM were developed by a team of researchers at ORNL, the University of Tennessee, Knoxville, Emory University and Carnegie Mellon University. PVM enables a collection of different computing systems to be viewed as a single parallel machine. In particular, it has been very widely used to connect networks of workstations (NOWs) together to execute parallel programs. The component computers in the parallel virtual machine need not be all of the same type, and PVM will transparently handle any necessary data conversions when messages are passed between machines with different internal data representations. Currently C and Fortran interfaces to PVM exist.

PVM is more than just a message passing library — it is a complete environment for heterogeneous parallel computing consisting of four main parts.

1. The PVM daemon. This runs on every host computer in the parallel virtual machine. It controls PVM resources on each host and mediates interactions with other hosts.
2. The PVM console. This is a user interface that allows the user to interactively configure the parallel virtual machine.
3. The PVM group server. This manages process groups.
4. The PVM library. The set of routines for performing message passing, and managing tasks and process groups.

The remainder of this subsection will focus on the PVM library.

The most natural way to program with PVM is to use a host-worker model of computation in which a single host process is responsible for doing I/O and for spawning a set of worker processes that concurrently perform most of the computation. The host and worker processes are each uniquely identified by an integer known as the task ID.

In PVM, point-to-point messages are typed, and tagged. A typed message is one in which the message passing system is aware of the datatypes of the components making up a message. PVM is capable of sending messages of mixed datatypes. A tagged message is one that has an integer "tag" associated with it. When a message is sent the destination is specified by the task ID of the receiving process. Similarly, when a message is received the source of the message is identified by its task ID. Message selectivity is by source task ID and tag. Thus, when a message is received the source task ID and tag associated with the message must match those specified in the argument list of the receive routine, unless either is wildcarded. If the receive routine is passed a value of -1 for the source task ID then this criterion will be ignored in message selection and we say that it is wildcarded. The tag may be wildcarded in a similar way.

Message passing is done by calls to routines in the message passing library and makes use of system controlled buffers. Typically, sending a message takes place in three phases. First the system send buffer is cleared and prepared for use. Next the message is packed into the message buffer. Lastly the message is sent. Receiving a message usually takes place in two phases. First the message is received into the system receive buffer and then it is unpacked into the application space. The following code fragment shows how one processes sends a message consisting of one integer followed by two floats to another process.

```
tid = pvm_mytid();
if (tid == source){
   bufid = pvm_initsend(PvmDataDefault);
   info  = pvm_packint(&i1, 1, 1);
   info  = pvm_packfloat(vec1, 2, 1);
   info  = pvm_send (dest, tag);
}
elseif (tid == dest){
   bufid = pvm_recv(source, tag);
   info  = pvm_upkint(&i2, 1, 1);
   info  = pvm_upkfloat(vec2, 2, 1);
}
```

In this example each process calls the routine `pvm_mytid()` to determine its task ID. It is assumed that the values of `source` and `dest` are distinct valid task IDs for the set of processes executing the code. The source process initializes the send buffer, and packs an integer (`i1`) and two floats (the first two elements of `vec1`) into it. Finally the source process sends the data to the destination process, `dest`. The destination process waits to receive the message with the specified tag value from the source process. Once it has been received it unpacks the integer into variable `i2` and the two floats into the first two elements of `vec2`.

The receive routine `pvm_recv` is blocking, i.e., if no message satisfying the selection criteria is available it will wait until one is available. The send routine `pvm_send` is nonblocking in the sense that a call to `pvm_send` initiates the send operation and then returns. PVM provides a receive routine `pvm_nrecv` that is nonblocking in the sense that if a message satisfying the selection criteria has not arrived when it is called it returns a value of zero, otherwise it receives the message into a new receive buffer and returns the strictly positive buffer ID. PVM also provides a receive routine that times out if a suitable message is not received within a specified time. The packing and unpacking of data is useful when messages of mixed datatype need to be communicated, but is rather inconvenient for messages of a single datatype. In such case the routines `pvm_psend` and `pvm_precv` may be used. For these routines the data buffer to send, or receive into, is specified in the argument list, together with the datatype. The routine `pvm_psend` is blocking, i.e., it will not return until it is safe to reuse the data buffer. The routine `pvm_precv` is also blocking, just like `pvm_recv`. Finally, the routine `pvm_probe` can be used to check if a message with a specified tag from a specified source is ready for receipt without actually receiving it.

In addition, to point-to-point communication and related routines PVM also includes a small set of collective communication routines. These routines involve coordinated communication within groups of processes. PVM provides mechanisms that allow processes to asynchronously join or leave a process group, and groups are identified by a user-supplied string. The three collective communication operations are barrier synchronization, broadcast, and reduction. The barrier routine ensures that no process in the group exits the routine until they all have entered it. The broadcast routine sends data from one process in the process group to all processes in the group. The reduction routine combines the values provided in the input buffer of each process in the group using a specified function. Thus, if D_i is the data in the ith process in the group, and $\oplus$ is the combining function, then the following quantity is evaluated,

$$\mathcal{D} = D_0 \oplus D_1 \oplus D_2 \oplus \cdots \oplus D_{n-1} \qquad (1)$$

where n is the size of the process group. The result $\mathcal{D}$ is returned to a specified root process. The combining function is supplied by the user, although PVM provides some pre-defined functions, such as summation.

Further details of PVM can be best obtained from the PVM users' guide and tutorial which is available as a book [22]. The PVM web page provides further documentation, recent news on PVM, research projects involving PVM, and pointers to other PVM resources [7]. The software for implementing PVM in a variety of computing environments is also available from the web page.

5.2 Message Passing Interface (MPI)

MPI is a proposed standard message passing interface. The design of MPI was a collective effort involving researchers in the United States and Europe from many organizations and institutions. MPI includes point-to-point and collective communication routines, as well as support for process groups, and application topologies.

In MPI there is currently no mechanism for creating processes, and an MPI program is parallel *ab initio*, i.e., there is a fixed number of processes from the start to the end of an application program. All processes are members of at least one process group. Initially all processes are members of the same group, and a number of routines are provided that allow the user to create (and destroy) new subgroups. Within a group each process is assigned a unique rank in the range 0 to $n - 1$, where n is the number of processes in the group. This rank is used to identify a process, and, in particular, is used to specify the source and destination processes in a point-to-point communication operation, and the root process in certain collective communication operations. As in PVM, message selectivity in point-to-point communication as by source process and message tag, each of which may be wildcarded to indicate that any valid value is acceptable.

The key innovative ideas in MPI are the communicator abstraction and general, or derived, datatypes. These will be discussed before describing the communication routines in more detail. Communicators provide support for the design of safe, modular software libraries. Here "safe" means that messages intended for a particular receive routine in an application will not be incorrectly intercepted by another receive routine. Thus, communicators are a powerful mechanism for avoiding unintentional non-determinism in message passing. This is a particular problem when using third-party software libraries that perform message passing. The point here is that the application developer has no way of knowing if the tag, group, and rank completely disambiguate the message traffic of different libraries and the rest of the application. Communicator arguments are passed to all MPI message passing routines, and a message can be communicated only if the communicator arguments passed to the send and receive routines match. Thus, in effect communicators provide an additional criterion for message selection, and hence permit the construction of independent tag spaces.

If communicators are not used to disambiguate message traffic there are two ways in which a call to a library routine can lead to unintended behavior. In the first case the processes enter a library routine synchronously when a send has been initiated for which the matching receive is not posted until after the library call. In this case the message may be incorrectly received in the library routine. The second possibility arises when different processes enter a library routine asynchronously, as shown in the example in Figure 1, resulting in nondeterministic behavior. If the program behaves correctly processes 0 and 1 each receive a message from process 2, using a wildcarded selection criterion to indicate that they are prepared to receive a message from any process. The three processes then pass data around in a ring within the library routine. If separate communicators are not used for the communication inside and outside of the library routine this program may intermittently fail. Suppose we delay the sending of the second message sent by process 2, for example, by inserting some computation, as shown in Figure 2. In this case the wildcarded receive in process 0 is satisfied by a message sent from process 1, rather than from process 2, and deadlock results. By supplying a different communicator to the library routine we can ensure that the program is executed correctly, regardless of when the processes enter the library routine.

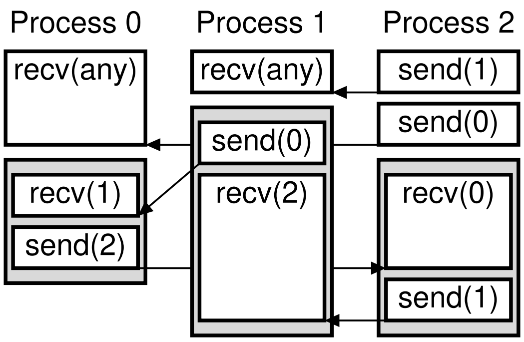

Figure 1: Use of communicators. Time increases down the page. Numbers in parentheses indicate the process to which data are being sent or received. The gray shaded area represents the library routine call. In this case the program behaves as intended. Note that the second message sent by process 2 is received by process 0, and that the message sent by process 0 is received by process 2.

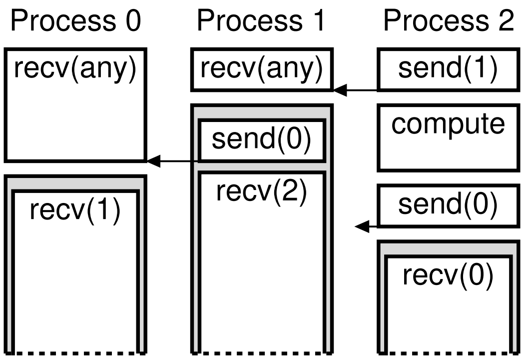

Figure 2: Unintended behavior of program. In this case the message from process 2 to process 0 is never received, and deadlock results.

Communicators are opaque objects, which means they can only be manipulated using MPI routines. The key point about communicators is that when a communicator is created by an MPI routine it is guaranteed to be unique. Thus it is possible to create a communicator and pass it to a software library for use in all that library's message passing. Provided that communicator is not used for any message passing outside of the library, the library's messages and those of the rest of the application cannot be confused.

Communicators have a number of attributes. The group attribute identifies the process group relative to which process ranks are interpreted, and/or which identifies the process group involved in a collective communication operation. Communicators also have a topology attribute which gives the topology of the process group. Topologies are discussed below. In addition, users may associate attributes with communicators through a mechanism known as caching.

All point-to-point message passing routines in MPI take as an argument the datatype of the data communicated. In the simplest case this will be a primitive datatype, such as an integer or floating point number. However, MPI provides a number of routines for creating more general datatypes, and thereby supports the communication of array sections and structures involving combinations of primitive datatypes.

In many applications the processes are arranged with a particular topology, such as a two- or three-dimensional grid. MPI provides support for general application topologies that are specified by a graph in which processes that communicate a significant amount are connected by an arc. If the application topology is an n-dimensional Cartesian grid then this generality is not needed, so as a convenience MPI provides explicit support for such topologies. For a Cartesian grid periodic or nonperiodic boundary conditions may apply in any specified grid dimension. In MPI, a group either has a Cartesian or graph topology, or no topology. In addition to providing routines for translating between process rank and location in the topology, MPI also:

1. allows knowledge of the application topology to be exploited in order to efficiently assign processes to physical processors,
2. provides a routine for partitioning a Cartesian grid into hyperplane groups by removing a specified set of dimensions,
3. provides support for shifting data along a specified dimension of a Cartesian grid.

By dividing a Cartesian grid into hyperplane groups it is possible to perform collective communication operations within these groups. In particular, if all but one dimension is removed a set of one-dimensional subgroups is formed, and it is possible, for example, to perform a multicast in the corresponding direction.

A set of routines that supports point-to-point communication between pairs of processes forms the core of MPI routines for sending and receiving blocking and nonblocking messages are provided. A blocking send does not return until it is safe for the application to alter the message buffer on the sending process without corrupting or changing the message sent. A nonblocking send may return while the message buffer on the sending process is still volatile, and it should not be changed until it is guaranteed that this will not corrupt the message. This may be done by either calling a routine that blocks until the message buffer may be safely reused, or by calling a routine that performs a nonblocking check on the message status. A blocking receive suspends execution on the receiving process until the incoming message has been placed in the specified application buffer. A nonblocking receive may return before the message has been received into the specified application buffer, and a subsequent call must be made to ensure that this has occurred before the application uses the data in the message.

In MPI a message may be sent in one of four communication modes, which approximately correspond to the most common protocols used for point-to-point communication. In *ready* mode a message may be sent only if a corresponding receive has been initiated. In *standard* mode a message may be sent regardless of whether a corresponding receive has been initiated. MPI includes a *synchronous* mode which is the same as the standard mode, except that the send operation will not complete until a corresponding receive has been initiated on the destination process. Finally, there is a *buffered* mode. To use buffered mode the user must first supply a buffer and associate it with a communicator. When a subsequent send is performed using that communicator MPI may use the associated buffer to buffer the message. A buffered send may be performed regardless of whether a corresponding receive has been initiated. In PVM message buffering is provided by the system, but MPI does not mandate that an implementation provide message buffering. Buffered mode provides a way of making MPI buffer messages, and is useful when converting a program from PVM to MPI.

In addition, MPI provides routines that send to one process while receiving from another. Different versions are provided for when the send and receive buffers are distinct, and for when they are the same. The send/receive operation is blocking, so does not return until the send buffer is ready for reuse, and the incoming message has been received.

MPI includes a rich set of collective communication routines that perform coordinated communication among a group of processes. The process group is that associated with the communicator that is passed into the routine. MPI's collective communication routines can be divided into two groups: data movement routines and global computation routines. There are five types of data movement routine: broadcast, scatter, gather, all-gather, and all-to-all. These are illustrated in Fig. 3.

There are two global computation routines in MPI: reduce and scan. The MPI reduction operation is similar in functionality to that provided by PVM. Different versions of the reduction routine are provided depending on whether the results are made available to all processes in the group, just one process, or are scattered cyclicly across the group. The scan routines perform a parallel prefix with respect to a user-specified operation on data distributed across a specified group. If D_i is the data item on the process with rank i, then on completion the output buffer of this process contains the result of combining the values from the processes with rank $0, 1, \ldots, i$, i.e.,

$$\mathcal{D}_i = D_0 \oplus D_1 \oplus D_2 \oplus \cdots \oplus D_i \tag{2}$$

Two versions of the MPI specification exist. One is dated May 5, 1994 (version 1.0), and the other June 12, 1995 (version 1.1). The latter document incorporates corrections and clarifications to the former, but the two do not differ in any substantial way. At the time of writing version 1.1 is only available electronically [17]. The book on using MPI by Gropp, Lusk and Skjellum, who played an active role in MPI's design, gives a good introduction to application programming with MPI [23]. An annotated reference manual based on version 1.1 of MPI will be available by the end of 1995 [29]. A large amount of information about MPI is available via the web, including portable implementations of MPI, information about efforts to extend MPI, and publications related to MPI [8].

6 Summary

This paper has given an overview of the main features of some of the most interesting and/or widely-used approaches to programming parallel computers based on message passing. While it is not intended as a complete survey it is hoped that readers will have gained a good idea of the options open to them when using such machines. An obvious question to ask is "What is the best approach to parallel programming?" Of course, this is not a trivial question to answer, and often the depends on complicated trade-offs between performance, expressivity, ease-of-programming, maintainability, and portability. Languages such as occam and Fortran M are very good at expressing parallelism in a modular way, and offer guarantees of program correctness which may be particularly important in realtime programming. Split-C aims to give good performance through the use of bulk transfer operations while providing the convenience and ease-of-use of a global address space. Linda's main advantages are its simplicity and expressivity, and the fact that it is a commercially-supported product. Message passing libraries are perhaps the most widely-used way of programming parallel computers. Although it is often argued that the popularity of message passing libraries rests on their flexibility and the fact that they permit the memory hierarchy to be managed to give good performance, the root cause may actually be economic and historical. When commercial parallel computers first become popular

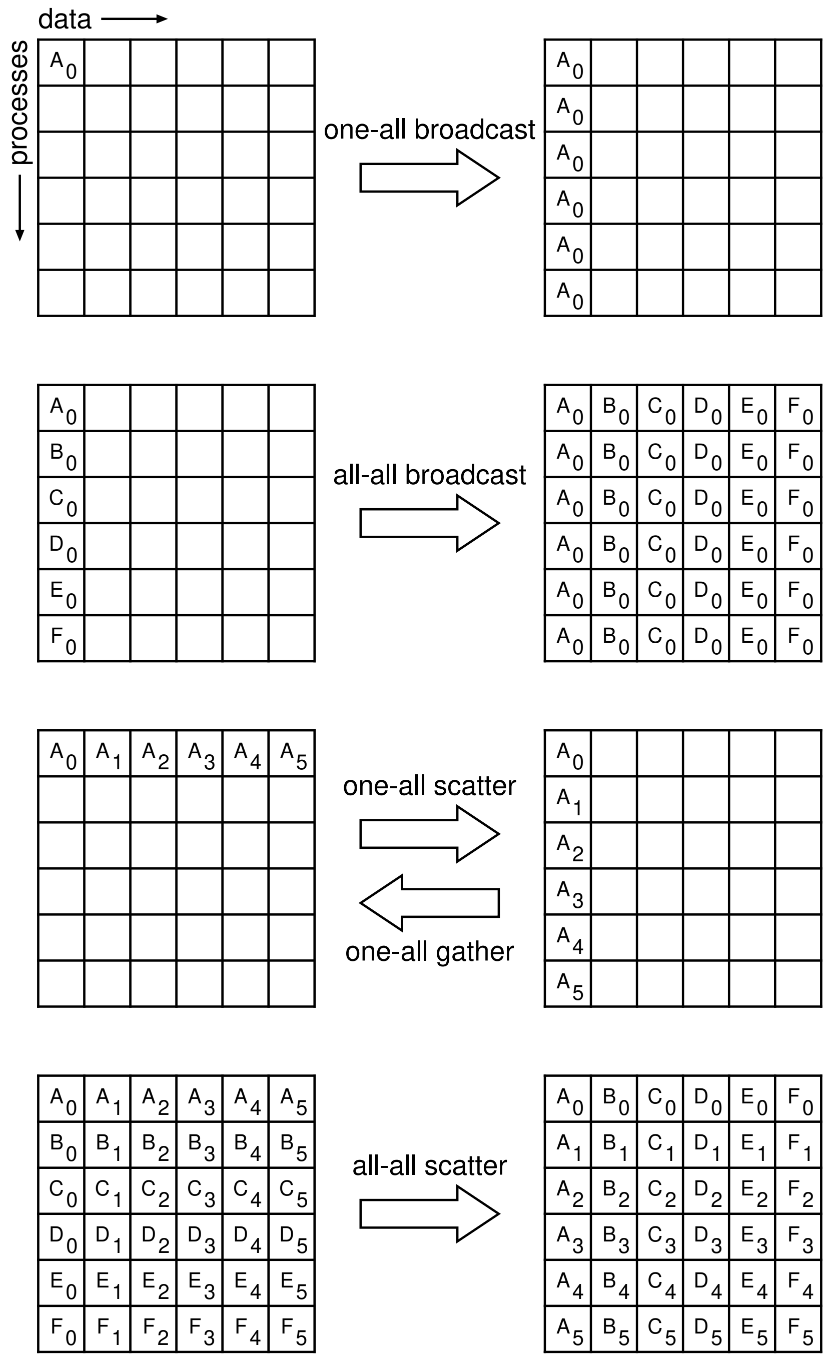

Figure 3: One-all and all-all versions of the broadcast, scatter, and gather routines for a group of six processes. In each case, each row of boxes represents contiguous data locations in one process. Thus, in the one-all broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

a decade ago the vendors supplied them with message passing libraries because that was (and still is) the cheapest type of parallel programming software. Users quickly developed portable message passing libraries on top of the vendor libraries, a development that ultimately led to PVM and MPI. PVM and MPI currently owe much of their popularity to the fact that they are supported on a large number of platforms, allowing software based on them to be easily ported, and they are freely available. Unfortunately no detailed study comparing different parallel programming paradigms ever seems to have been conducted. In the absence of such studies any attempt to rank them is largely subjective.

References

[1] Special issue on message passing of *Parallel Computing*, 20(4), April 1994.

[2] The URL of the occam world-wide web page is `http://www.hensa.ac.uk/parallel/languages/occam/`

[3] The URL of the Fortran M world-wide web page is `http://www.mcs.anl.gov/fortran-m/FM.html`.

[4] The URL of the Linda world-wide web page at Yale University is `http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html`.

[5] The URL of the Linda web page at York University, U.K., is `http://indy200.cs.york.ac.uk:8080/linda/linda.html`.

[6] The URL of the Split-C web page at UC Berkeley is `http://http.cs.berkley.edu/projects/parallel/castle/split-c/`.

[7] The URL of the PVM web page is `http://www.epm.ornl.gov/pvm/`.

[8] The URL of a good MPI web page is `http://www.mcs.anl.gov/mpi/`. This has links to other extensive MPI pages at Mississippi State Engineering Center and Oak Ridge National Laboratory.

[9] H. E. Bal. Orca: a language for parallel programming of distributed systems. *IEEE Trans. Software Engineering*, 18(3):190–205, 1992. See also the web page at `http://www.cs.vu.nl/vakgroepen/cs/orca.html`.

[10] R. M. Bulter and E. L. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.

[11] R. Calkin, R. Hempel, H.-C. Hoppe, and P. Wypior. Portable programming with the PARMACS message passing library. *Parallel Computing*, 20(4):615–632, April 1994.

[12] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[13] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[14] N. Carriero, D. Gelernter, T. Mattson, and A. Sherman. The Linda alternative to message passing systems. *Parallel Computing*, 20(4):633–655, April 1994.

[15] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T von Eicken, and K. Yelick. Introduction to Split-C, version 1.0. Technical report, Department of Computer Science, University of California, Berkeley, April 1995.

[16] J. Feo, D. Cann, and R. Oldehoeft. A report on the SISAL language project. *J. Parallel and Distributed Comput.*, 12(10):349–366, 1990. See also the web page at `http://www-atp.llnl.gov/sisal`.

[17] The MPI forum. MPI: A message passing interface standard (version 1.1). Available electronically from `http://www.mcs.anl.gov/mpi/`.

[18] I. Foster, R. Olson, and S. Tuecke. *Programming in Fortran M, Version 2.0*, August 1994. Available from [3].

[19] I. T. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1994. Also available from the world-wide web at `http://www.mcs.anl.gov/dbpp`.

[20] I. T. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *J. Parallel and Distributed Comput.*, 26(1):24–35, April 1995.

[21] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Current Processors, Volume 1*. Prentice Hall, Englewood Cliffs, New Jersey, 1988. This book is based on Cros III, and gives man pages in the appendix.

[22] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, 1994. Also available on the web at `http://www.netlib.org/pvm3/book/pvm-book.html`.

[23] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.

[24] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[25] C. A. R. Hoare. The transputer and occam: a personal story. *Concurreny: Practice and Experience*, 3(4):249–264, August 1991.

[26] G. Jones and M. Goldsmith. *Programming in Occam 2*. Prentice Hall International, 1988.

[27] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994. See also the HPF forum web page at `http://www.erc.msstate.edu/hpff/home.html`.

[28] INMOS Limited. *Occam2 Reference Manual*. Prentice Hall International, Hemel Hempstead, U.K., 1988.

[29] S. W. Otto, M. Snir, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1995. Should be available by the end of 1995.

[30] Parasoft Corporation. *A Tutorial Introduction to Express, Version 3.2*, 1992. See also the Express world-wide web page at `http://www.parasoft.com/express.html`.

[31] D. Pountain and D. May. *A Tutorial Introduction to Occam Programming*. McGraw-Hill, 1987.

[32] A. Skjellum, S. G. Smith, N. E. Doss, A. P. Leung, and M. Morari. The design and evolution of zipcode. *Parallel Computing*, 20(4):565–596, April 1994.

[33] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurreny: Practice and Experience*, 2(4):315–339, December 1990.