9 Dusty Decks: Porting Existing Message-Passing Programs to MPI

It is a sign of the maturity of parallel computing that there already exist many explicitly parallel application programs written for the message-passing model. Old programs needing to be ported to new hardware and software platforms used to be called "dusty decks." Now they are more respectfully called "legacy applications." In this chapter we describe how to port such programs to MPI. In general, porting will be straightforward, because MPI is for the most part a functional superset of existing message-passing libraries. Therefore the changes to existing programs will be largely syntactical changes and, in some cases, can be done automatically.

This chapter emphasizes the need for the MPI Standard by illustrating how similar in semantics, yet different in syntax, the various message-passing systems are. It also illustrates the range of extra facilities that have been found useful by the users of these systems. Including the best of all of these facilities has contributed to the size of the MPI specification.

We will describe how to make minimal modifications to existing message-passing programs so that they will run on MPI-based systems. In most cases the most straightforward translations will provide improved performance on systems with well-tuned MPI implementations. Further performance gains may be possible by taking advantage of MPI features not found in the original programming system, but that will not be our main focus here. We nonetheless recommend that users consider such changes as an important second phase of using MPI.

We first consider message-passing libraries provided by the vendors: Intel NX, IBM EUI libraries, and Thinking Machines CMMD. These systems represent high-performance libraries that have been customized for particular platforms. We then look at public-domain message-passing libraries. These are precursors of MPI in that they were designed to be portable, but their design has in most cases been driven by particular research projects, and they have not had the benefit of the broad-based design process that produced MPI. They have, on the other hand, contributed many research ideas that have been incorporated into MPI, and they have provided an initial impetus to parallel computing in general by offering early mechanisms by which users could write portable parallel code. In particular, we describe how to port programs written for PVM, PARMACS, p4, TCGMSG, Chameleon, and Zipcode. Because PVM is so widely used (both versions 2.4.x and 3.2.x) on workstation networks, we devote special attention to moving both styles of PVM programs to MPI.

For each of the libraries we describe, we will organize the presentation as follows.

Basics Initialization and cleanup, identifying one's self and one's peers, and basic (blocking) send and receive

Miscellaneous Functions Nonblocking send and receive operations, wait, test, probe, status information, and error handling

Collective operations Synchronizing processes, collective data movement and computations, process subgroups

MPI Counterparts of Other Features Special features in the library that have MPI counterparts

Features not in MPI Special features in the library that do not have MPI counterparts

Process Startup Features outside MPI, but worth considering and comparing

Translation Tools programs to aid in conversion to MPI.

In the sections on process startup, we will describe the particular method used by the library at hand. We note that MPI provides no mechanism for process startup. It was decided during the MPI design process that

- On parallel machines, vendor-specific mechanisms were currently too diverse to coerce into a single style.
- On workstation networks, the problems of process startup and management were intertwined with the difficulties of workstation management and parallel job queueing, which were being addressed by a variety of both proprietary and public-domain systems. (DQS is one example of such a system.) These systems are being designed to interface to a number of different parallel programming systems being run on the networks, including MPI.

The systems summarized here, however, all did have to provide a startup mechanism, and we describe each in its place.

9.1 Intel NX

NX is Intel's message passing library. It is called NX/2 on Intel iPSC/2 and iPSC860 computers, and sometimes called NX/M on the Intel Delta and Paragon. We describe the common features of recent versions of the system.

The Basics

Table 9.1 shows the correspondences between basic NX functions and their MPI counterparts. The table shows Fortran bindings for both NX and MPI; the C translations are similar.

(most programs start in parallel)	`MPI_INIT(...,ierr)`
(no special exit code)	`MPI_FINALIZE(ierr)`
`me = mynode();`	`MPI_COMM_RANK(MCW,all_of_me,ierr)`
`mep = myptype();`	
`np = numnodes()`	`MPI_COMM_SIZE(MCW,np,ierr)`
`csend(tag,buf,count, dest,destp)`	`MPI_SEND(buf, count, MPI_BYTE, dest, tag, MCW)`
`crecv(tag,buf,count)`	`MPI_RECV(buf, count, MPI_BYTE, MPI_SRC_ANY, tag, MCW, status)`
`crecvx(tag,buf,count, src,srcp,status)`	`MPI_RECV(..., MPI_BYTE, src,...,status)`

Table 9.1
Translating NX to MPI: basic functions. "`MCW`" is short for `MPI_COMM_WORLD` in the table.

NX has both traditional and new-style calls (with `x` at the end of the names). The `x`-style calls are more general, support both tag- and source-matching, and universally include the secondary node name, the "ptype," which is a means of distinguishing multiple processes per processor. For our purposes, the source pairs `(src,srcp)` and destination pairs `(dest,destp)` correspond to ranks. For many applications, and for older NX codes, all the ptype's are zero.

Miscellaneous Functions

Good correspondence exists between nonblocking NX and MPI point-to-point communication, as we can see from Table 9.2. For error handling, NX provides access through prepended underscore versions of most routines that return success/failure; these are documented in the Intel manual. Furthermore, while MPI provides different communication modes with variations of send (e.g., `MPI_RSEND`), NX accomplishes "ready send" with "force types," which are the available tags 1,073,741,824–1,999,999,999. Also, NX reserves all message tags above 1,000,000,000, except for the force types. Force type messages don't match up against wildcard tag selection (`tag=-1` in Intel calls), to help provide some guarantee that messages don't end up in the wrong buffer.

`isend(...)`	`MPI_ISEND(.., MPI_BYTE,..)`
`irecv(...)`	`MPI_IRECV(.., MPI_BYTE,..)`
`iprobe(tag)`	`MPI_IPROBE(MPI_SRC_ANY,tag,comm,flag,status, ierr)`
`cprobe(tag)`	`MPI_PROBE(...)`
`cprobex(...)`	`MPI_PROBE(...)`
`msgwait(...)`	`MPI_WAIT(...)`
`msgcancel(...)`	`MPI_CANCEL(...)`

Table 9.2
Translating NX to MPI: miscellaneous functions

Collective Operations

Several but not all of the MPI collective operations are provided in NX but, crucially, without subgroups in NX. Table 9.3 shows a sampling of the correspondences. In Table 9.3, the tag used by the pseudo-broadcast **gsendx()** must be unique to

`gsync()`	`MPI_BARRIER(MCW,ierr)`
`gsendx(excltag,buf,count, root,group,groupsize)`	`MPI_BCAST(buf,count,datatype, comm_of_group,ierr)`
`gopf(sbuf,count,rbuf,op)`	`MPI_ALLREDUCE(sbuf,rbuf,count, datatype,op,MCW)`
`gd[high,low](sbuf,count,rbuf)`	`MPI_ALLREDUCE(sbuf,rbuf,count, MPI_DOUBLE,MPI_[MAX,MIN],MCW)`
`gd[sum,prod](sbuf,count,rbuf)`	`MPI_ALLREDUCE(sbuf,rbuf,count, MPI_DOUBLE,MPI_[SUM,PROD],...)`
`gcol(sbuf,slen,rbuf,rlen,count)`	`MPI_ALLGATHERV(sbuf,scount, datatype,rbuf,rcount, displs,recvtype,MCW)`
`gcolx(sbuf,all_lens,rbuf)`	`MPI_ALLGATHER(sbuf,scount, datatype,rbuf, rcount,datatype,MCW)`

Table 9.3
Translating NX to MPI: collective operations; `MCW` is short for `MPI_COMM_WORLD`. Only a sampling of the NX collective routines are shown.

prevent mixups with other messages; **gsendx()** is matched with **crecv()**, **irecv()**, or `hrecv()` at the receiver. The `x`-suffix to `gcolx()` refers to the fact that each process knows the size of all elements going to be concatenated, whereas for `gcol()` only each local node knows this information beforehand. The use of `sbuf` and `rbuf` shown in Table 9.3 is not precise; Intel calls require extra storage for received data for operations, but actually place the final result over `sbuf`, unlike MPI routines, which use both `sbuf` and `rbuf` but place the result in `rbuf`.

MPI Counterparts of Other Features

MPI provides several other calls that also appear in NX. Some of these are shown in Table 9.4.

`csendrecv(...)`	`MPI_SENDRECV(...)`
`isendrecv(...)`	`MPI_ISEND(...); MPI_IRECV(...);`
`host = myhost()`	`MPI_ATTR_GET(MPI_COMM_WORLD, MPI_HOST, &host);`

Table 9.4
Translating NX to MPI: mapping other features of NX.

Features not in MPI

MPI does not support interrupt-driven messages, such as those that appear in both NX and CMMD. The identifying feature of these interrupt messages is that a user handler is provided, and this handler is called when such a message is received. This makes interrupt messages essentially equivalent to coarse-grained "active messages." These calls are extremely useful for creating virtual shared memory in the absence of a true multithreaded capability on a compute node.

In addition, NX provides access to basic parallel I/O capability. It has a number of synchronous and asynchronous calls related to file I/O, which are sandwiched into the NX library. MPI does not currently define any parallel I/O calls.

Process Startup

Earlier versions of NX supported dynamic process management, particularly the version for iPSC/860 and earlier; the `load()` call exemplifies this support. With the Paragon came the re-introduction of dynamic process support. NX on the Paragon also has extensive calls for dynamic partition management.

9.2 IBM EUI

EUI, also known as MPL, is IBM's product for message passing on the SP1 and SP2 parallel machines.

The Basics

Table 9.5 shows the correspondences between basic EUI functions and their MPI counterparts. The table shows Fortran bindings for both EUI and MPI; the C translations are similar; EUI C calls are all lower case and replace `MP_` with `mpc_`. We omit most arguments for this conversion, because they can be easily inferred from the way in which the EUI (and respective MPI) calls are used.

(programs start in parallel)	`MPI_INIT(...,ierr)`
(no special exit code)	`MPI_FINALIZE(ierr)`
`MP_GETRANK(...)`	`MPI_COMM_RANK(...)`
`MP_GETSIZE()`	`MPI_COMM_SIZE(...)`
`MP_BSEND(...)`	`MPI_SEND(...)`
`MP_BRECV(...)`	`MPI_RECV(...)`

Table 9.5
Translating EUI to MPI: basic functions

Miscellaneous Functions

There is also substantial compatibility between nonblocking EUI and MPI point-to-point communication, as can be seen in Table 9.6

`MP_SEND(...)`	`MPI_ISEND(.., MPI_BYTE,..)`
`MP_RECV(...)`	`MPI_IRECV(.., MPI_BYTE,..)`
`MP_WAIT(...)`	`MPI_WAIT(...)`
`MP_STATUS(...)`	`MPI_TEST(...)`

Table 9.6
Translating EUI to MPI: miscellaneous functions

Collective Operations

EUI has a relatively complete set of collective operations, defined on groups and subgroups, all working from the `allgrp`, which is directly analogous to

MPI_COMM_WORLD's group. In Table 9.7, each EUI call with a group argument is to

MP_SYNC()	MPI_BARRIER(...)
MP_BCAST()	MPI_BCAST(...)
MP_COMBINE(...)	MPI_ALLREDUCE(...)
MP_SCATTER(...)	MPI_SCATTER(...)
MP_GATHER(...)	MPI_GATHER(...)
MP_PREFIX(...)	MPI_SCAN(...)
MP_CONCAT(...)	MPI_ALLGATHER(...)
MP_INDEX(...)	MPI_ALLTOALLV(...)

Table 9.7
Translating EUI to MPI: collective operations

be replaced with a corresponding communicator in MPI, so conversion is obvious.

MPI Counterparts of Other Features

MP_GROUP(...)	(MPI Group Setlike Operations)
MP_PARTITION(...)	MPI_COMM_SPLIT(...)
MP_PACK(...)	MPI_PACK(...)
MP_UNPACK(...)	MPI_UNPACK(...)
MP_STOPALL(...)	MPI_ABORT(...)

Table 9.8
Translating EUI to MPI: mapping other features of EUI

EUI's collective operations can operate over subgroups.

Features Not in MPI

EUI provides access not only to the ranks of processes, but to their "task ID" for providing low-level translations. This provision is specifically disallowed in MPI. Furthermore, EUI provides a number of task inquiry operations not found in MPI.

Process Startup

The IBM SP1 manages process creation statically, outside the scope of its message-passing system. Thus, like MPI, EUI has no process management functionality.

9.3 TMC CMMD

CMMD is the message-passing system supported on the Thinking Machines CM-5. It in turn is implemented on top of the CMAM active message library. CMMD has gone through a number of major revisions; to date, the major versions have all had incompatibilities. We discuss here version 3.0.

The Basics

Table 9.9 shows the correspondences between the basic MPI routines and CMMD. CMMD provides a number of different send and receive modes; we have made one of several possible correspondences in this table.

(programs start in parallel)	`MPI_INIT(...)`
(no special exit code)	`MPI_FINALIZE(ierr)`
`me=CMMD_self_address()`	`MPI_COMM_RANK(MPI_COMM_WORLD,me,ierr)`
`np=CMMD_partition_size()`	`MPI_COMM_SIZE(MPI_COMM_WORLD,np,ierr)`
`CMMD_send_block(...)`	`MPI_SEND(.., MPI_BYTE,...)`
`CMMD_receive_block(...)`	`MPI_RECV(.., MPI_BYTE,...)`

Table 9.9
Translating to MPI: basic functions

Miscellaneous Functions

`CMMD_send_and_receive(...)`	`MPI_Sendrecv(...)`
`CMMD_send_block_v(...)`	`MPI_Send(...,vector-datatype,...)`
`CMMD_msg_pending(...)`	`MPI_Iprobe(..)`
`CMMD_msg_sender()`	`status.MPI_SOURCE`
`CMMD_msg_tag()`	`status.MPI_TAG`
`CMMD_bytes_received()`	`MPI_Get_count(&status,MPI_BYTE,&len)`
`CMMD_send_async(...)`	`MPI_Isend(...)`
`CMMD_receive_async(...)`	`MPI_Irecv(...)`
`CMMD_send_noblock(...)`	`MPI_Send(...)`
`CMMD_msg_done(mcb)`	`MPI_Test(&request,&status)`
`CMMD_msg_wait(mcb)`	`MPI_Wait(&request,&status)`

Table 9.10
Translating to MPI: Miscellaneous functions

CMMD provides both blocking and nonblocking operations. The nonblocking communication ("async") routines provide a handler that can be called when the message arrives; the MPI idiom for this is a blocking operation in a separate thread. CMMD `mcb`s are somewhat like a combination of MPI's `MPI_Request` objects and `MPI_Status` objects. CMMD has a `CMMD_all_msgs_wait()`; but it takes no arguments and refers to all messages globally, and hence can not be used in libraries (unless they clearly restrict the user's ability to use CMMD).

CMMD provides for "virtual channels." MPI has no direct counterpart, but for many situations, MPI's persistent requests (created with, e.g., `MPI_Send_init`) and "ready send" mode (`MPI_Rsend`) provide similar functionality.

Another view of the buffering choices made by CMMD is that `MPI_Rsend` can be thought of as the MPI counterpart of `CMMD_send_block`; users porting *to* CMMD may wish to bear this in mind.

Collective Operations

The CM-5 has special hardware to perform some collective operations; the design of the CMMD collective operations reflects this.

`CMMD_bc_to_nodes(...)`	`MPI_BCAST(...,MCW)`
`CMMD_receive_bc_from_node(...)`	
`CMMD_scan_int(...)`	`MPI_SCAN(...,1,MPI_INT,...,MCW)`
`CMMD_reduce_double(...)`	`MPI_ALLREDUCE(...,1,` `MPI_DOUBLE,...,MCW )`
`CMMD_reduce_v(...)`	`MPI_ALLREDUCE(...)`
`CMMD_concat_with_nodes(...)`	`MPI_ALLGATHER(...)`
`CMMD_sync_with_nodes()`	`MPI_BARRIER(MCW)`

Table 9.11
Translating CMMD to MPI: collective functions; `MCW` is short for `MPI_COMM_WORLD`.

The CMMD scan routines can be *segmented*; MPI can accomplish the same effect by using multiple contexts.

CMMD collective routines can be applied only to all processes; no subsets are possible.[1]

`CMMD_error(format,args...)`	`printf(...);` `MPI_Abort(0,MPI_COMM_WORLD);`

Table 9.12
Translating to MPI: special functions

MPI Counterparts of Other Features

CMMD provides some timers that measure the time taken by a user program; the *elapsed time* field of these timers is roughly equivalent to the time given by `MPI_Wtime`. It is difficult for libraries to use these timers because they are identified by a single integer and any part of a program may reset them. There is no "registry" mechanism or other way for a routine to discover which timers are already being used.

Features Not in MPI

CMMD contains an active messages interface (it is in fact implemented on top of it) that is available to sophisticated users. The interface includes routines to copy data (`CMAML_scopy` and `CMAML_pcopy`) and routines such as `CMAML_rpc` to send remote procedure calls.

CMMD includes some I/O features. In particular, CMMD I/O operations can be collective as well as independent; the CMMD mode `CMMD_local` matches the requirements of MPI for non-message-passing operations.

Process Startup

Programs start automatically; the setup is handled essentially at link time by linking with special initialization code. This process is relatively transparent to the user. The number of nodes is chosen by selecting a partition; if an application needs fewer nodes than the smallest partition it can call `CMMD_reset_partition_size`.

9.4 Express

Express is a commercial portable set of tools and utilities designed for parallel processing, sold by ParaSoft. It provides not only message passing and collective operations, but also a rich tuning and debugging environment. There are tools for

[1] It is possible to use `CMMD_reset_partition_size` to permanently decrease the number of processes, but this procedure is not reversible and is effectively redefining the CMMD equivalent of `MPI_COMM_WORLD`.

managing I/O and parallel graphics. Here we focus just on the message-passing library, and, for variety, show the C bindings instead of Fortran. The routines described here come from Version 3.0 of the C User's Guide [5].

The Basics

`exparam(&env)`	`MPI_Init(...)`
`me = env.procnum`	`MPI_Comm_rank(MPI_COMM_WORLD,&me)`
`np = env.nprocs`	`MPI_Comm_size(MPI_COMM_WORLD,&np)`
`exread(...)`	`MPI_Send(.., MPI_BYTE,...)`
`exwrite(...)`	`MPI_Recv(.., MPI_BYTE,...)`

Table 9.13
Translating Express to MPI: basic functions

Miscellaneous Functions

The nonblocking operations in Express have an extra integer argument, shown as `status` in Table 9.14, which is set to -1 at the initiation of the operation, which can be tested to determine whether the operation is complete. At completion, it is set to the number of bytes sent or received.

`exsend(...,&status)`	`MPI_Isend(.., MPI_BYTE,..)`
`exreceive(...,&status)`	`MPI_Irecv(.., MPI_BYTE,..)`
`(status < 0) ?`	`MPI_TEST(...)`

Table 9.14
Translating Express to MPI: miscellaneous functions

Collective Operations

Express has a simple set of collective operations, and supports subgroups. The single operation `excombine` is used for nearly all of them; the user supplies the specific function to be used in the operation. Arguments to `excombine` specify a list of processes that are to participate in the operation. The result is always distributed back to all the processes who participated. A reserved value (`ALLNODES`) specifies all the processes.

`exsync(...`	`MPI_Barrier(...)`
`exbroadcast(...)`	`MPI_Bcast(...)`
`excombine(...,op,...)`	`MPI_Allreduce(...,op,...)`

Table 9.15
Translating Express to MPI: Collective Operations

MPI Counterparts of Other Features

Express does have features for dealing with a virtual topology of processes. These are currently limited to grid topologies. Some of the capabilities of MPI datatypes are provided by the `exvread` and `exvwrite` functions for sending and receiving noncontiguous data.

Features not in MPI

Express is an entire environment for parallel programming, not just a message-passing system. Thus it contains many features that have not been addressed by MPI, including graphics, debugging, and some aspects of parallel I/O.

Process Startup

There are facilities for starting processes very simply (`cubix -n10` to start 10 processes, for example) or to load files explicitly under program control. There is complete support for the "host-node" system model.

9.5 PVM 2.4.x

PVM 2.4.x (normally 2.4.1 or 2.4.2) is an important intermediate version of a library of message-passing models developed at Oak Ridge National Laboratory, University of Tennessee, and Emory University by Vaidy Sunderam, Al Geist, Jack Dongarra, Robert Manchek, and others (see [9]). PVM stands for "Parallel Virtual Machine," heralding its support for process management, heterogeneous clusters, and message passing. PVM 2.4.2 and earlier versions run on heterogeneous networks of workstations and on front ends of parallel machines, whereas PVM 3.0 and later also support some parallel machines. Version 2.4.x's message-passing aspects are quite simple, and translation to MPI should consequently be straightforward. In the next section, we detail porting from PVM 3.2.x to MPI separately, since PVM 2.4.x and 3.2.x are different. For more information, see [8].

The Basics

Table 9.16 shows the correspondences between basic PVM 2.4.x functions and their MPI counterparts. The table shows Fortran bindings for both PVM 2.4.x and MPI; the C translations are similar. We show only certain of the arguments to each routine; the correspondences for the others can be readily inferred.

`enroll()`	`MPI_INIT(...)`
`leave()`	`MPI_FINALIZE(ierr)`
`whoami(...,me)`	`MPI_COMM_RANK(MPI_COMM_WORLD,me,ierr)`
`pstatus(np,...)`	`MPI_COMM_SIZE(MPI_COMM_WORLD,np,ierr)`
`snd(...), vsnd(...)`	`MPI_SEND(.., buf_type,...)`
`rcv(...), vrcv(...)`	`MPI_RECV(.., buf_type,...)`

Table 9.16
Translating PVM 2.4.x to MPI: basic functions

In PVM 2.4.x, the `enroll()` call may be omitted in favor of `whoami()` if the program was "initiated" by PVM; the corresponding call to `MPI_INIT()` is not optional. The call to `leave()` is optional as well (when processes have been started by a call to `initiate()` or `initiateM()`), but its counterpart, `MPI_FINALIZE()` is never optional. The call `pvm_mstat()` provides further information. (see [9]). (In Table 9.16, the "`buf_type`" argument in `MPI_SEND()` and `MPI_RECV()` refers either to a simple type like `MPI_BYTE` or `MPI_INTEGER` or to a more complex data type, covered below.)

The PVM design makes no claims about buffering, though the most widely distributed implementation provides significant (though not infinite) buffering. PVM users who depend on the buffering in the implementation can use the buffered sends (`MPI_Bsend`) of MPI.

The functions `rcvmulti` and `vrcvmulti`, which are capable of enumerating acceptable message tags, can be emulated in several ways with MPI. The preferred means is to use multiple contexts of communication if wild-carded source and/or tag (`MPI_ANY_SOURCE`, `MPI_ANY_TAG`) should prove insufficient to distinguish messages in a single context. Since MPI provides both source and tag selectivity, removing `rcvmulti()` and `vrcvmulti()` calls does not usually pose a practical problem of conversion.

Places where "components" are used in PVM 2.4.x correspond roughly to groups in MPI, and groups are encapsulated in communicators. Hence, functionality achieved with components in PVM 2.4.x can be emulated. Since not all PVM 2.4.x calls refer to the component consistently, however, there is no one-to-one mapping of calls. PVM 2.4.x "instance numbers" correspond roughly to MPI process ranks in a group.

We have simplified the discussion of sending operations so far by neglecting the fact that the `snd()` and `vsnd()` calls must always be preceded with an `initsend()` command and commands to pack the single send buffer of a process. There is an analog in MPI to this procedure, which we will discuss later in this section. For now it is sufficient to say that for sending contiguous data of any one type, MPI needs only a single send command to prepare, convert, and dispatch data.

Miscellaneous Functions

PVM 2.4.x does not have a nonblocking receive. It does have a "probe" function and a "probemulti" function. The former is directly replaceable by an MPI call, while the latter follows logic analogous to that described above for replacing `rcvmulti`. Note that the output role of `source` (represented by the value `MPI_ANY_SOURCE` in

`probe(tag)`	`MPI_IPROBE(MPI_ANY_SOURCE,tag,MPI_COMM_WORLD, flag,status)`
`pvm_error(str)`	`MPI_ABORT(comm, val)`

Table 9.17
Translating PVM 2.4.x to MPI: miscellaneous functions

Table 9.17) has no counterpart in PVM 2.4.x.

Collective Operations

PVM 2.4.x provides a few collective operations. A rough correspondence to the functions of MPI is shown in Table 9.18.

`barrier("name",n)`	`MPI_BARRIER(comm,ierr)`
`snd(...,-1,...)`	`MPI_BCAST(buf,count,datatype,root,comm,ierr)`

Table 9.18
Translating PVM 2.4.x to MPI: collective operations

PVM 2.4.x has no counterpart to the a wide variety of global data-movement routines in MPI. In addition, PVM 2.4.x does not have subgroups, but has more fuzzy components (a process in PVM 2.4.x belongs to one component, whereas an MPI process can belong to an arbitrary number of groups and hence to an arbitrary number of communication contexts). Note that the broadcast operation in PVM is really a multisend; whereas in MPI messages sent with `MPI_BCAST` are received with `MPI_BCAST`. The sender is specified by the `root` parameter; the other processes receive. The semantics of a PVM 2.4.x barrier are much different from those of an MPI barrier, in that MPI barriers are over a specific group and synchronize, whereas PVM 2.4.x barriers wait for any `n` processes to call the function, with possible race conditions. The PVM 2.4.x `waituntil()` and `ready()` event functions are not implementable strictly as MPI functions. MPI has no non-blocking collective communications; these must be implemented via thread extensions.

MPI Counterparts of Other Features

Communication among heterogeneous machines in PVM 2.4.x is provided by packing and unpacking functions that apply to the single send and receive buffers on each process. The first pack command is always preceded by `initsend()`. Only one pending send is permitted in PVM 2.4.x because of the single send buffer.

In MPI every *send* has a datatype argument, so we have the correspondences shown in Table 9.19. The table shows the case where single datatypes are sent in PVM 2.4.x. The corresponding receives are shown in Table 9.20.

`putnshort(data,len); snd(...)`	`MPI_SEND(...,MPI_SHORT,...)`
`putnint(data,len); snd(...)`	`MPI_SEND(...,MPI_INT,...)`
`putnlong(data,len); snd(...)`	`MPI_SEND(...,MPI_LONG,...)`
`putnfloat(data,len); snd(...)`	`MPI_SEND(...,MPI_FLOAT,...)`
`putndfloat(data,len); snd(...)`	`MPI_SEND(...,MPI_DOUBLE,...)`
`putncplx(data,len); snd(...)`	`MPI_SEND(...,MPI_COMPLEX,...)`
`putndcplx(data,len); snd(...)`	`MPI_SEND(...,` `MPI_DOUBLE_COMPLEX,...)`
`putbytes(data,len); snd(...)`	`MPI_SEND(...,MPI_BYTE,...)`
`putstring(data); snd(...)`	`MPI_SEND(...,MPI_CHARACTER,...)`

Table 9.19
Translating PVM 2.4.x to MPI: heterogeneous and synchronous communication (sends)

PVM 2.4.x has support for most, and MPI support for all, the basic datatypes supported by the host language, whether Fortran or C. In converting string transmissions from PVM to MPI, the correspondence between `putstring()` and the use of `MPI_CHARACTER` is not exact, in that the send must elect to send or not send the null termination when it sizes the character array. (The detailed issues of Fortran

`rcv(...); getnshort(data,len)`	`MPI_RECV(...,MPI_SHORT,...)`
`rcv(...); getnint(data,len)`	`MPI_RECV(...,MPI_INT,...)`
`rcv(...); getnlong(data,len)`	`MPI_RECV(...,MPI_LONG,...)`
`rcv(...); getnfloat(data,len)`	`MPI_RECV(...,MPI_FLOAT,...)`
`rcv(...); getndfloat(data,len)`	`MPI_RECV(...,MPI_DOUBLE,...)`
`rcv(...); getncplx(data,len)`	`MPI_RECV(...,MPI_COMPLEX,...)`
`rcv(...); getndcplx(data,len)`	`MPI_RECV(...,` `MPI_DOUBLE_COMPLEX,...)`
`rcv(...); getbytes(data,len)`	`MPI_RECV(...,MPI_BYTE,...)`
`rcv(...); getstring(data)`	`MPI_RECV(...,MPI_CHARACTER,...)`

Table 9.20
Translating PVM 2.4.x to MPI: heterogeneous and synchronous communication (receives)

character strings vs. character arrays and substrings are thorny, as described in the MPI Standard [6, Chapters 2 and 3].)

A number of `putxxx()` calls can be posed before the `snd()`; the corresponding `rcv()` is then followed by a number of `getxxx()`s. This can be emulated in MPI as well, but not without the use of a packing/unpacking mechanism described in Section ??.

Features Not in MPI

Key features in PVM 2.4.x that are not in MPI are as follows:

- Global events and barriers, as described earlier,
- Interactive process management as described in the following section,

Process Startup

Parallel programs written using PVM 2.4.x are started by running the first instance of the program (or the master program executable, if the slaves and master have different executables) in a normal Unix way, after the master PVM daemon has also been started as a regular Unix process. The first process then starts the other processes via an `initiate()` or `initiateM()` call, which specify locale by architecture and name, respectively. The virtual machine is specified to PVM 2.4.x through a "hostfile" with a specific format (see [9, Section 9]), and a default location in the user's directory structure. Note that the `enroll()` that appears in the master program does *not* start the PVM master daemon. The daemon must be started beforehand; it in turn subsequently starts slave daemons. The master daemon may be run interactively, particularly when passwords are needed to achieve startup on remote nodes, or the user wants to observe the behavior of processes (see Section 6.2 of [9]). MPI has not specified any analogous portable mechanism for either process startup or dynamic management.

9.6 PVM 3.2.x

PVM 3.2.x (the most recent as of this writing is 3.2.6) is the currently supported version of the PVM library described in the preceding section [8]. Unlike PVM 2.4.x, PVM 3.0 and higher have support for specific parallel machines as well as for heterogeneous networks of workstations. Furthermore, the Version 3 release constituted a complete redesign of the system. Yet, version 3.2.x's message-passing aspects remain quite simple, and translation to MPI should consequently be straightforward, just as for PVM 2.4.x and p4.

The Version 3 release uses a new, more systematic naming convention for functions and constants. In C, `pvm_` is prepended to all the names of routines to avoid the many potential conflicts in short, common names. In Fortran, this is modified to `pvmf`.

The Basics

Table 9.21 shows the correspondences between basic PVM 3.2.x functions and their MPI counterparts. The table shows Fortran bindings for both PVM 3.2.x and MPI; the full C translations are similar. We show only certain of the arguments to each routine; the correspondences for the others can be readily inferred. Group arguments for PVM 3.2.x correspond to the dynamic group concept added in this Version 3 release. In MPI, the communicator would hold a corresponding static group. Use of dynamic addition or subtraction of processes after an initial group formulation is not within the MPI model and requires a more complex porting strategy than what which is shown in Table 9.21. Intercommunicators provide one alternative means for handling complex group management in MPI. (The rank of a task and the number of tasks can also be determined without using the PVM group operations at the time the PVM processes are started. We will not discuss this approach here.)

PVM 3 has replaced the notion of "components" and "instances" with TIDs (task IDs) and dynamic process groups. This change makes porting to MPI somewhat simpler than with the earlier versions of PVM, unless dynamic groups are used extensively. Furthermore, PVM 3 has replaced the restriction of single buffers with

`pvmfmytid(mytid)`	`MPI_INIT(...)`
`pvmfexit(info)`	`MPI_FINALIZE(ierr)`
`pvmfgetinst(grp,mytid,me)`	`MPI_COMM_RANK(comm,me,ierr)`
`pvmfgsize(grp,np)`	`MPI_COMM_SIZE(comm,np,ierr)`
`pvmfsend(tid,tag,info)`	`MPI_SEND(.., buf_type,...)`
`pvmfrecv(tid,tag,bufid)`	`MPI_RECV(.., buf_type,...)`

Table 9.21
Translating PVM 3.2.x to MPI: basic functions

multiple buffers, each identified by its "bufid." PVM 3 buffers are created, deleted and selected; these operations are unneeded in MPI. As before, `buf_type` refers to a simple type like `MPI_DOUBLE` or `MPI_BYTE`, or to a more complex datatype as we discussed for the PVM 2.4.x conversion process above; we return to this again below.

Miscellaneous Functions

PVM 3.2.x has both "probe" and nonblocking receive. Note that PVM's nonblocking receive is not like MPI's `MPI_Irecv`; rather is combines the effects of `MPI_Iprobe` and `MPI_Recv`.

`pvmfnrecv(src,tag,bufid)`	`MPI_IPROBE(...);`
	`if(flag) MPI_RECV(...);`
`pvmfprobe(src,tag,bufid)`	`MPI_IPROBE(src,tag,`
	`MPI_COMM_WORLD,flag,status)`
`pvmfperror(str,info)`	`MPI_ABORT(comm, val)`

Table 9.22
Translating PVM 3.2.x to MPI: miscellaneous functions

Collective Operations

To get access to most of PVM 3.2.x's collective operations, one needs to use dynamic groups. The `pvmfmcast()` call (corresponding to `snd(...,-1,...)` in PVM 2.4.x) still maps to an MPI broadcast. The problem with dynamic groups, unlike MPI's static groups, is that there are race conditions associated with "joining" and "leaving" these groups (since a dynamic group is a distributed data structure, it may be out of date in some processes unless it is synchronized properly). Assuming one is careful to form a group and then barrier synchronize, Table 9.23 gives an accurate picture of how to map from PVM 3.2.x's groups to MPI communicators. Note that the tag argument of PVM 3.2.x's broadcast has no corresponding entry in MPI.

`pvmfmcast(ntask,tids,tag,info)`	`MPI_BCAST(buf,count,datatype, root,comm,ierr)`
`pvmfbcast(grp,tag,info)`	`MPI_BCAST(buf,count,datatype, root,comm,ierr)`
`pvmfbarrier(grp,count,info)`	`MPI_BARRIER(comm,ierr)`

Table 9.23
Translating PVM 3.2.x to MPI: collective operations

MPI Counterparts of Other Features

Communication among heterogeneous machines in PVM 3.2.x is provided by packing and unpacking functions that apply to the current-send and specified-receive buffers on each process. The first pack command is normally preceded by **pvmfinitsend()** (when **pvmfmkbuf()**/**pvmffreebuf** are used, **pvmfinitsend()** may not be necessary). Multiple pending sends are possible.

In MPI every *send* has a datatype argument; this matches PVM's use of different routines for different types of data.

PVM 3.2.x has support for most of the the basic datatypes supported by the host language, whether Fortran or C. In converting string transmissions from PVM to MPI, it should be noted that the correspondence of PVM's `STRING` and the use of `MPI_CHARACTER` is not exact, as stated above under PVM 2.4.x's section.

Features Not in MPI

Key features in PVM 3.2.x that are not in MPI are as follows:

- Interactive process management as described in the following section
- Routines for adding and deleting hosts from the virtual machine dynamically (in addition to dynamic process management)

Process Startup

Parallel programs written using PVM 3.2.x are started by running the first instance of the program (or the master program executable, if the slaves and master have different executables) from the Unix shell, after the master PVM daemon has also been started as a regular Unix process. The first process then starts the other processes via a `pvmfspawn()` call, which can specify location either by architecture or by machine name. The virtual machine is specified to PVM 3.2.x through a "hostfile" with a specific format (see [1]), and a default location in the user's directory structure. Note that the `pvmfmytid()` that appears in the master program does *not* start the PVM master daemon. The daemon must be started beforehand (perhaps as a separate program, running interactively in a separate window); it in turn subsequently starts slave daemons. The PVM console may be run interactively and interacts with the daemons; this feature is particularly useful when passwords are needed to achieve startup on remote nodes or when the user wishes to observe the behavior of processes (see [1]). There is also a group server for dynamic group management, which gets started automatically if needed. In contrast, MPI has not specified any analogous portable mechanism for process startup or dynamic management.

Tools

A recent master's project report by Fei Cheng of Mississippi State University illustrates how one can reasonably unify the existing PVM 3.2.x application programmer interface (API) with the MPI API. An internal upgrade of PVM is demonstrated (using two complementary strategies) in order to support contexts and static groups [4]. In about a person-month's work, Cheng implemented a large subset of the November, 1993 MPI specification[7], while allowing most existing PVM 3.2.x calls to work alongside MPI calls in user applications. There are plans to make this software available by anonymous ftp once it is made fully MPI standard compliant (most likely by leveraging the MPI model implementation and by providing a "patch" utility for existing PVM source code). This software technology differs a bit from the abstract device interface approach of Gropp and Lusk (that allows the model implementation to run on top of PVM 3.2.x) in that a dual API results with Cheng's strategy. Both PVM 3.2.x and MPI will show through to the application; in the long run, PVM calls would be changed to MPI calls to evolve a legacy application to be fully MPI compliant.

9.7 p4

p4 is a library for both shared-memory and message-passing models, developed at Argonne National Laboratory [2] by Ralph Butler and Ewing Lusk. Its message-passing part is quite simple, and translation to MPI should consequently be straightforward.

The Basics

Table 9.24 shows the correspondences between basic p4 functions and their MPI counterparts. The table shows Fortran bindings for both p4 and MPI; the C translations are similar. We show only certain of the arguments to each routine; the correspondences for the others should be easy to infer.

`p4init()`	`MPI_INIT(...)`
`p4cleanup()`	`MPI_FINALIZE(ierr)`
`me = p4myid()`	`MPI_COMM_RANK(MPI_COMM_WORLD,me,ierr)`
`np = p4ntotids()`	`MPI_COMM_SIZE(MPI_COMM_WORLD,np,ierr)`
`p4send(...)`	`MPI_SEND(.., MPI_BYTE,...)`
`p4recv(...)`	`MPI_RECV(.., MPI_BYTE,...)`

Table 9.24
Translating to MPI: basic functions

The p4 library uses in/out arguments for source and tag. In MPI, the output part of these is replaced by the single argument `status`, from which the tag, sender, and length of the received message can be retrieved.

The C version of `p4recv` can be passed a null buffer pointer to indicate that no buffer is being provided and that the system should acquire one that fits the message that is received. This is useful in situations where the length of a message cannot be inferred from its tag. In MPI, `MPI_Iprobe` can be used to provide this functionality, since the length of the message waiting to be received can be extracted from the `status` argument. In place of the buffer pointer, the p4 user can specifically manage buffers with `p4_msg_alloc` and `p4_msg_free`.

Miscellaneous Functions

P4 does not have a nonblocking send or receive. It does have a "probe" function.

`p4probe(tag,src,flag)`	`MPI_IPROBE(src,tag,comm,flag,status,ierr)`
`p4error(str,val)`	`MPI_ABORT(comm,val,ierr)`
`p4softerrs(new,old)`	`MPI_ERRHANDLER_SET(ierr)`
`p4clock(),p4ustimer()`	`MPI_WTIME(ierr)`

Table 9.25
Translating p4 to MPI: miscellaneous functions

The function `MPI_Probe` is a blocking call; therefore the MPI call that matches the semantics of `p4probe` is `MPI_Iprobe` instead. Note also that the output roles of `tag` and `source` have been replaced by the `status` argument in `MPI_Iprobe`. For error handling, p4 provides only the options provided by MPI's `MPI_ERRORS_FATAL` and `MPI_ERRORS_RETURN`. MPI provides also the capability for a library to provide its own error handler. Finally, the routine `MPI_Wtime` provides a high-resolution timing facility and substitutes for both the low-resolution `p4clock` and the high-resolution `p4ustimer`.

Collective Operations

Many but not all of the MPI collective operations are provided in p4. Table 9.26 shows the correspondences.

`p4globarr(tag)`	`MPI_BARRIER(MPI_COMM_WORLD,ierr)`
`p4brdcast(type,data,count,..)`	`MPI_BCAST(buf,count,datatype,)` `root,comm,ierr)`
`p4globop(tag,buf,count,` `size,op,datatype,count,)`	`MPI_REDUCE(sendbuf,recvbuf,` `datatype,op,root,comm,ierr)`

Table 9.26
Translating p4 to MPI: collective operations

MPI has a wide variety of global data-movement routines, which have no counterpart in p4. In addition, p4 does not have subgroups. Note that p4's broadcast operation is really a multisend; messages sent with `p4brdcast` are received with `p4recv`, whereas in MPI messages sent with `MPI_BCAST` are received with `MPI_BCAST`. The sender is specified by the `root` parameter; the other processes receive.

MPI Counterparts of Other Features

Communication among heterogeneous machines in p4 is provided by `p4_sendx`, which has an extra, single-datatype argument. In MPI every *send* has a datatype argument, so we have the correspondences shown in Table 9.27. None of the MPI derived datatypes are necessary to translate p4 programs, since p4 programs support transmission only of contiguous buffers.

In p4 one can request an acknowledgment of a message by using `p4_sendr` or `p4_sendrx`. Although there is not an exact correspondence, much of the same functionality can be obtained in MPI by using `MPI_SSEND` and its relations; in both cases execution of the sender is halted until it is certain that a matching receive

p4sendx(...,P4INT,...)	MPI_SEND(...,MPI_INT,...)
p4sendx(...,P4LNG,...)	MPI_SEND(...,MPI_LONG,...)
p4sendx(...,P4FLT,...)	MPI_SEND(...,MPI_FLOAT,...)
p4sendx(...,P4DBL,...)	MPI_SEND(...,MPI_DOUBLE,...)
p4sendx(...,P4NOX,...)	MPI_SEND(...,MPI_BYTE,...)
p4brdcastx(....)	MPI_BCAST(...)
p4sendr(...)	MPI_SSEND(...)

Table 9.27
Translating p4 to MPI: heterogeneous and synchronous communication

has taken place. MPI has support for all of the datatypes supported by the host language, in either Fortran or C.

Features Not in MPI

MPI has no support for the shared-memory mode supported by p4 with its monitor datatype and library of predefined monitors. We anticipate that the need for a shared-memory programming model will be at least partly provided by threads and MPI's thread safety. The cluster model, in which processes that share memory communicate via message passing, is part of p4 but not of MPI. In the future, MPI together with a thread library may replace the p4 cluster model.

Process Startup

Parallel programs written using p4 and running on networks of machines are started by running the first instance of the program (or the master program executable, if the slaves and master have different executables) in a normal Unix way. The first process then starts the others, using the function `p4crpg` to read the contents of a file (the "procgroup file") and determine which executables should be started and where. On various parallel machines vendor-specific mechanisms are required, and the procgroup file must match them. MPI has not specified any portable mechanism for process startup.

9.8 PARMACS

PARMACS [3, 11] is a portable message-passing system widely used in Europe, originally developed by Rolf Hempel at GMD. It is now distributed commercially by Pallas GmbH. Originally a set of `m4` macro definitions for adding message-passing statements to Fortran, it is now (as of Version 6.0) a subroutine library for both C and Fortran. PARMACS made particular contributions to MPI in the area of process topologies.

The Basics

In Figure 9.28 we show only the Fortran bindings; the C bindings are similar. PARMACS provides an explicit host/node model of computation, realized, for example, on the Intel IPSC/860. MPI programs do not need to have explicit "host" code, but an inquiry function can be used to determine which process is the "host" if there is one. Table 9.28 gives the MPI routines that correspond to the basic PARMACS primitives.

`PMINIT(...)`	`MPI_INIT(...)`
`PMEND()`	`MPI_FINALIZE()`
`me = PMYPID()`	`MPI_COMM_RANK(MPI_COMM_WORLD,&me)`
`np = PMNRPS()`	`MPI_COMM_SIZE(MPI_COMM_WORLD,&np)`
`ho = PMHPID()`	`MPI_GET_HOST()`
`PMSND(...)`	`MPI_SEND(.., MPI_BYTE,..)`
`PMRCV(...)`	`MPI_RECV(.., MPI_BYTE,..)`

Table 9.28
Translating PARMACS to MPI: basic functions

Miscellaneous Functions

PARMACS does not have a nonblocking send or receive. It does have a "probe" function. It and some of the other functions are shown in Table 9.29.

`PMWAIT(...)`	`MPI_WAITANY(...)`
`PMPROB(...)`	`MPI_IPROBE(src,tag,comm,flag,status)`
`PMSERR(code,str)`	`MPI_ERROR_STRING(code,str,rlen)`
`PMTIME()`	`MPI_WTIME()`

Table 9.29
Translating PARMACS to MPI: miscellaneous functions

Collective Operations

PARMACS does not have global data-movement or collective-computation routines. It does provide a global process tree that sets up the communication structure for user-defined collective operations. PARMACS does have a global synchronization routine. Table 9.30 shows the correspondence with MPI. Note that one supplies

`PMBARR(barr_id)`	`MPI_BARRIER(comm)`

Table 9.30
Translating PARMACS to MPI: global synchronization

a tag on the `PMBARR` call. This corresponds to the use of a (possibly nondefault) communicator in the MPI version, although it would be one whose associated group included all processes.

MPI Counterparts of Other Features

PARMACS has a number of features in common with MPI, shown in Table 9.31. PARMACS supports communication among heterogeneous machines, by means of the `PMFORM` function. With this function the user specifies the contents of the message being sent with the next `PMSND`. If the data representations differ between sender and receiver, the information specified by `PMFORM` is used to perform the translation. Thus, data is copied or translated only if necessary. The source data to be sent may have mixed type. The components must be made up of a set of contiguous chunks, however. The corresponding functionality in MPI is proveded by `MPI_TYPE_STRUCT`, where the components are built using `MPI_TYPE_CONTIG`. Note that the concept of *next message* is not thread safe and has thus been avoided in MPI whereas thread safety has not been a design goal of PARMACS.

The best way to mimic the PARMACS behavior in MPI is to save the message format information, as with the `PMFORM` call, and to send this information together with the message instead of the next `PMSND`. The message is received with data type `MPI_PACKED`, and `MPI_UNPACK` is used to unpack the message.

`PMFORM(...)`	`MPI_UNPACK(...)`
`PMSNDR(...)`	`MPI_SSEND(...)`
`PMRCVR(...)`	`MPI_RECV(...)`

Table 9.31
Translating PARMACS to MPI: heterogeneous and synchronous communication

In PARMACS one can request an acknowledgment of a message by using `PMSNDR`. Although there is not an exact correspondence, much of the same functionality can be obtained in MPI by using `MPI_SSEND` and its relations; in both cases execution of the sender is halted until it is certain that a matching receive has taken place. In PARMACS, such a message must be received with `PMRCVR`; in MPI, messages send with `MPI_SSEND` are received with a normal `MPI_RECV`.

The most outstanding feature of PARMACS is its support for process topologies. Most of the topology features in MPI are inherited from PARMACS. What are called "torus" topologies in PARMACS are called "cart" (for "Cartesian") topologies in MPI. Table 9.32 shows the correspondences.

`PMPGRP(...)`	`MPI_GRAPH_CREATE(...)`
`PMPTOR(...)`	`MPI_CART_CREATE(...)`
`PMQMAP(...)`	`MPI_TOPO_TEST`
`PMQGRP(...)`	`MPI_GRAPH_GET`
`PMQTOR(...)`	`MPI_CART_GET`
`PMLKUP(...)`	`MPI_CART_RANK`
`PMGLTR(...)`	collective functions

Table 9.32
Translating PARMACS to MPI: process topologies

Features Not in MPI

Some of the PARMACS topology management functions do not have MPI counterparts. The agglomerate-torus and de-agglomerate-torus functions are examples. Their main purpose is the efficient support of multigrid solvers in PDE applications, whereas MPI has no special features to support any particular application area.

Process Startup

PARMACS postulates a host-node model, and so its process startup mechanisms involve separate initialization of the host process and the node processes. MPI does not make this distinction, and in fact does not specify how processes come into existence.

Tools

The distributors of PARMACS are planning to develop PARMACS-to-MPI software that would largely automate the process of translating their application base of programs that use PARMACS into MPI programs.

9.9 TCGMSG

The "TCG" in TCGMSG [10] stands for the Theoretical Chemistry Group at Argonne National Laboratory, where Robert Harrison first began developing the system. It is now widely used in the international computational chemistry community. TCGMSG has a slightly restricted semantics for its otherwise familiar message-passing primitives. These restrictions allow it to achieve extreme efficiency on high-performance parallel computers, while still enabling execution on workstation networks.

The Basics

Table 9.33 shows the MPI routines corresponding to the basic functions in TCGMSG.

`PBEGINF()`	`MPI_INIT(...)`
`PEND()`	`MPI_FINALIZE()`
`me = NODEID()`	`MPI_COMM_RANK(MPI_COMM_WORLD,me,ierr)`
`np = NNODES()`	`MPI_COMM_SIZE(MPI_COMM_WORLD,np,ierr)`
`SND(...,1)`	`MPI_SSEND(.., MPI_BYTE,..)`
`RCV(...,1)`	`MPI_RECV(.., MPI_BYTE,..)`

Table 9.33
Translating TCGMSG to MPI: basic functions

The final argument in `SND` and `RCV` determines whether the communication is synchronous or not. The TCGMSG model supports only synchronous or asynchronous (nonblocking) sends, although the behavior is for the synchronous send to be buffered into an ordinary blocking send if buffering is available, in which case it behaves like an `MPI_SEND` instead. Wild cards in receive operations are allowed for process identifiers, but not on tag values. This situation is true also for `PROBE`.

Miscellaneous Functions

TCGMSG also provides nonblocking communication routines and timing functions, and some error-handling mechanisms. Table 9.34 shows the corresponding MPI routines.

`SND(...,0)`	`MPI_ISEND(.., MPI_BYTE,..)`
`RCV(...,0)`	`MPI_IRECV(.., MPI_BYTE,..)`
`WAITCOM(...)`	`MPI_WAIT(...)`
`PROBE(...)`	`MPI_IPROBE(src,tag,comm,flag,status)`
`PARERR(code,str)`	`MPI_ERROR_STRING(code,str,rlen)`
`TCGTIME()`	`MPI_WTIME()`
`MTIME()`	`MPI_WTIME()`

Table 9.34
Translating TCGMSG to MPI: miscellaneous functions

Collective Operations

TCGMSG provides basic collective operations. The corresponding MPI routines are shown in Table 9.35.

`SYNCH(tag)`	`MPI_BARRIER(comm)`
`BRDCAST(type,data,count,root)`	`MPI_BCAST(buf,count,datatype,` `  root,comm,ierr)`
`IGOP(tag,buf,count,size,` `  op,datatype)`	`MPI_REDUCE(sendbuf,recvbuf,count,` `  datatype,op,root,comm,ierr)`
`IDOP(tag,buf,count,size,` `  op,datatype)`	`MPI_REDUCE(sendbuf,recvbuf,count,` `  datatype,op,root,comm,ierr)`

Table 9.35
Translating TCGMSG to MPI: collective operations

There are two collective computation routines, one for integer operations and one for double-precision floating-point operations. The operations are `+`, `*`, `max`, `min`, `absmax`, and `absmin`. These have MPI counterparts except for `absmin` and `absmax`, which can be added to MPI through user-supplied operations. The integer operations include bitwise `or` and bitwise `and`. MPI has these operations, and bitwise `xor` as well.

MPI Counterparts of Other Features

In order to send and receive messages in a heterogeneous environment, one or's the tag field with a constant to indicate the data type.

`SND(tag | type,...)`	`MPI_SEND(..,datatype,..)`
`RCV(tag | type,...)`	`MPI_RECV(..,datatype,..)`
`EVON(), EVOFF()`	`MPI_PCONTROL`

Table 9.36
Translating TCGMSG to MPI: heterogeneous communication and event logging

TCGMSG provides an event-logging capability for producing files that performance tools can read to create statistical or graphical representations of the events that occurr during the execution of the parallel program. MPI does not provide a specific interface of this type, but rather a generic mechanism for intercepting MPI calls, by which anyone can attach any profiling tools to any conforming MPI implementation.

Features not in MPI

One unusual feature of TCGMSG is its provision of an absolutely minimal shared memory item (one integer). The TCGMSG function `NXTVAL` returns the next value of a counter. It is guaranteed that no two calls will return the same value and that all values will be eventually returned, in order. On machines such as Intel or IBM SP1 (EUIH) with interrupt-driven receives, the highest numbered process registers an interrupt handler to provide the necessary services. On other machines the process that runs the parallel command is actually a TCGMSG process numbered higher than any application process. It provides NXTVAL service.

This feature cannot be implemented strictly within the message-passing model. The MPE library implements it on some machines where a version of the second method is available. On other machines, threads can be used to implement this feature.

TCGMSG also has a parallel file copying command, called `PCOPY`, that is useful for broadcasting a file from the local disk of one process to the local disks of other processes. This command is `PCOPY`. No I/O commands are part of MPI.

Process Startup

TCGMSG provides a simple, portable mechanism for starting a set of parallel processes. The `parallel` command (a C program) invokes whatever command is required to start a fixed number of processes on a parallel computer, and invokes `rsh` to start processes on networks of workstations.

Tools

Translation of TCGMSG programs into MPI programs is straightforward, with the exception of `NXTVAL`, which is heavily used by the TCGMSG community. To supply the functionality of `NXTVAL`, an additional function must be supplied, such as the one in the MPE library (See Section ??).

9.10 Chameleon

Chameleon is a lightweight message-passing system that provides a common interface to a variety of popular message-passing systems, including both proprietary vendor-specific systems and portability systems. Chameleon is designed to provide the maximum performance; many of the functions, particularly the point-to-point operations, are implemented with compile-time macros. In fact, it is often possible to use Chameleon to generate "native" code by taking the output of the preprocessor. Chameleon made particular contributions to MPI in the area of point-to-point communication modes and datatypes. Chameleon provides support for both C and Fortran applications.

The Basics

Table 9.37 shows the MPI routines corresponding to the basic functions in Chameleon.

`PIiInit(...)`	`MPI_Init(...)`
`PIiFinish()`	`MPI_Finalize()`
`me = PImytid`	`MPI_Comm_rank(MPI_COMM_WORLD,&me)`
`np = PInumtids`	`MPI_Comm_size(MPI_COMM_WORLD,&np)`
`PIbsend(...,datatype)`	`MPI_Send(.., datatype,..)`
`PIbrecv(...,datatype)`	`MPI_Recv(.., datatype,..)`

Table 9.37
Translating Chameleon to MPI: basic functions

Chameleon supports only contiguous messages of a single datatype but does support heterogeneous systems with a datatype argument. Table 9.38 shows the correspondence between Chameleon and MPI datatypes.

MSG_OTHER	MPI_BYTE
MSG_INT	MPI_INT
MSG_LNG	MPI_LONG
MSG_SHRT	MPI_SHORT
MSG_FLT	MPI_FLOAT
MSG_DBL	MPI_DOUBLE

Table 9.38
Translating Chameleon to MPI datatype

Miscellaneous Functions

Chameleon supports nonblocking operations and an operation similar to the MPI "ready send." One difference is that when the ready-send mode is used, Chameleon requires that both the send and receive modes match; MPI allows any receive to match any mode of send.

PInsend(...)	MPI_Isend(...)
PIwsend(...)	MPI_Wait(...)
PInrecv(...)	MPI_Irecv(...)
PIwrecv(...)	MPI_Wait(...)
PInsendrr(...)	MPI_Irsend(...)
PIbprobe(...)	MPI_Probe(...)
PInprobe(...)	MPI_Iprobe(...)
PIcrecv(...)	MPI_Cancel(...)
PIcsend(...)	MPI_Cancel(...)
PInstatus(...)	MPI_Test(...)
SYGetElapsedTime()	MPI_Wtime()
PINodeName(...)	MPI_Get_processor_name(...)

Table 9.39
Translating Chameleon to MPI: miscellaneous functions

Collective Operations

Chameleon supports a rich set of collective operations, including operations on arbitrary groups of processes. Some of these, including the operations to form groups of processes, are shown in Table 9.40.

`newps=PSPartition(key,oldps)`	`MPI_Comm_split(oldcomm,color,key, newcomm)`
`ps=PSCreate(name)` `PSAddMember(ps,list,n)` `PSCompile(ps)`	`MPI_Group_incl(old,n,list,group)` `MPI_Comm_create(old,group,new)`
`PSDestroy(ps)`	`MPI_Comm_free(&comm)`
`size = PSnumtids(ps)`	`MPI_Comm_size(ps,&size)`
`rank = PSmytid(ps)`	`MPI_Comm_rank(ps,&rank)`
`PIgsync(procset)`	`MPI_Barrier(comm)`
`PIbcastSrc(buf,count,root, procset,datatype)`	`MPI_Bcast(buf,count,datatype, root,comm)`
`PIg`*xop*`(buf,len,work,procset)`	`MPI_Allreduce(sendbuf,recvbuf, size,MPI_`*op*`,datatype)`

Table 9.40
Translating Chameleon to MPI: operations on process sets

MPI Counterparts of Other Features

Chameleon provides a message tag request system to ensure that libraries are using different message tags. This works only as long as all parts of the code use this mechanism. MPI provides a better way to do this through the use of communicators. The Chameleon routines that provide the tag requests are shown in Table 9.41.

Chameleon also provides extensive correctness and performance debugging features when used with the profiling version of the Chameleon libraries. These provide built-in features that MPI can provide through the profiling interface. The preferred method for controlling these features in Chameleon is through a command-line interface.

Chameleon contains a few routines for manipulating virtual topologies for both 2-D Cartesian meshes and binary trees. MPI has corresponding routines for meshes, but for the binary trees MPI would have to use the general graph topologies.

`ltag=PIGetTags(ps,n)`	Create a new communicator
`PIReturnTags(ps,n,ltag)`	Free that communicator

Table 9.41
Translating Chameleon to MPI: counterparts of other functions

Features Not in MPI

Chameleon includes some routines for parallel I/O. Chameleon also allows a knowledgeable user to change the algorithms used for the collective operations.

Process Startup

Chameleon provides a simple and consistent process startup mechanism. Chameleon programs are started (on most systems) just as if they were regular uniprocessor programs. Command-line arguments are used to specify the number of processors, particular processors, and resource limits on the program (such as the number of CPU seconds or page faults). On many systems, running a Chameleon program appears no different from running any uniprocessor program. This interface is inherited by the model MPI implementation described in Appendix B.

Tools

Chameleon provides some tools for translating to MPI that are based on the `m4` macro processor. These tools have been used to help generate the Intel version of the model MPI implementation. To convert a file from Chameleon to MPI, the user simply uses the command

```
inlinecomm -mpi filename
```

this changes `filename` to use MPI calls instead of Chameleon calls. It is only an approximation, however; the output should be checked by hand.

9.11 Zipcode

Zipcode [14, 15, 16] is a portable message-passing system originally developed by Anthony Skjellum at Caltech. Zipcode made particular contributions to MPI in the area of process groups, contexts, and intracommunicators (though they have different names and some detailed differences in Zipcode). Zipcode also postulated a form of virtual topologies, but the form of PARMACS topologies was adopted by MPI. The support of diverse collective operations with group scope is another aspect of Zipcode that has also moved into MPI. Like PARMACS, Zipcode usually operates under the assumption of a host-node model of computation.

Porting between Zipcode and MPI is easy because they both have contexts, groups, and communicators, though with some differences.

The Basics

Zipcode as currently released has only a C binding, because of its use of dynamic message management for point-to-point messaging, following the Reactive Kernel/Cosmic Environment style of message management [12, 13]. Zipcode follows the CrOS III model of collective operations, in a generalized form [15]. Zipcode provides an explicit host/node model of computation available, for example, on the Intel Delta and Paragon, and BBN TC-2000 Butterfly. In contrast, MPI programs do not need to have explicit "host" code, but an inquiry function can be used to determine which process is the "host" if there is one. Table 9.42 gives the MPI routines that correspond to the basic Zipcode primitives. In the most general model, a Zipcode program includes a "Postmaster General" process that provides safe contexts of communication when communicating groups start "mailers" (called intracommunicators in MPI). MPI achieves safe contexts through a cooperative synchronization that does not require a "context server" of the form used by Zipcode.

Zipcode is based on "classes" of mail, each associated with a different naming strategy for processes (often a virtual topology). Consequently, the receipt selectivity of each class of Zipcode mail is different, because different amounts of information are available to inform the recipient of what the message contains. Most Zipcode classes have source selectivity, but not tag selectivity, since this proves quite sufficient for many applications. Other classes have both forms of selectivity. For simplicity, throughout we show the messaging primitives for the 3D grid abstraction class, called G3. A G3 class (unlike MPI's topologies) automatically contains recursively defined topologies for the plane children, and the plane children in turn have row/column children (G1).

Semantically, a Zipcode *receive* allocates a message like a "malloc," while a Zipcode *send* is analogous to a "free" with the side-effect that the message appears at its destination. A Zipcode program never needs to know the length of a message prior to accepting it, something that is not provided by any MPI receive mode.

Miscellaneous Functions

Zipcode has a nonblocking send and receive. It does not have a non-blocking "probe" function because its receive returns NULL if no message has arrived. It and some of the other functions are shown in Table 9.43.

`Zip_Init(...)`	`MPI_Init(...)`
`Zip_exit()`	`MPI_Finalize()`
`g3_pqr(comm,p,q,r)`	`MPI_Cart_coords(comm,rank,3,pqr)`
`g3_PQR(comm,P,Q,R)`	`MPI_Cart_get(comm,3,dims,...)`
`g3_send(comm,ltr,p,q,r)`	`MPI_Cart_rank(comm,pqr,rank);`
	`MPI_Send(.., MPI_BYTE,..,comm,ierr)`
	`free((void *)ltr)`
`ltr =`	
`g3_recvb(comm,p,q,r)`	
	`MPI_Cart_rank(comm,pqr,rank);`
	`MPI_Probe(.., MPI_BYTE,...);`
	`ltr = (char *)malloc(len*sizeof(char));`
	`MPI_Recv(.., MPI_BYTE,...,comm,ierr)`

Table 9.42
Translating Zipcode to MPI: basic functions

`g3_recv(comm,ltr,p,q,r)`	`MPI_Cart_rank(comm,pqr,rank);`
	`MPI_Iprobe(..,MPI_BYTE,...,flag,..);`
	`if(flag) ltr=(char *)malloc(...);`
	`if(flag) MPI_Recv(...,MPI_BYTE,...);`

Table 9.43
Translating Zipcode to MPI: miscellaneous functions

Collective Operations

Zipcode has a number of collective-computation routines, but not as extensive as MPI. Table 9.44 shows the correspondence.

`g3_fanin(comm,...,rootp,rootq,...)`	`MPI_Reduce(...,comm)`
`g3_fanout(comm,...,rootp,rootq,...)`	`MPI_Bcast(...,comm)`
`g3_combine(comm,...)`	`MPI_Allreduce(...,comm)`
`g3_prefix(comm,...)`	`MPI_Scan(...,comm)`
`zip_sync(comm)`	`MPI_Barrier(comm)`

Table 9.44
Translating Zipcode to MPI: selected collective operations

MPI Counterparts of Other Features

Zipcode supports packing and unpacking of data through "invoices," which are a form of datatype, also supported by other systems. Zipcode has selected a printf-style mechanism for creating such datatypes; it permits variable-sized objects and user-defined types, but currently does not support recursive definitions. The use of invoices with Zipcode's "pack" point-to-point and collective operations provides the user with insulation from heterogeneous conversion worries. Table 9.45 shows the correlations with MPI, in brief.

`zip_pack(...)`	`MPI_Pack(...)`
`zip_unpack(...)`	`MPI_Unpack(...)`
`zip_pack_send(...)`	`MPI_Send(...datatype,)`
`zip_pack_recv(...)`	`MPI_Recv(...datatype,)`
`g3_pack_combine(...)`	`MPI_Allreduce(...)`

Table 9.45
Translating Zipcode to MPI: heterogeneous communication

Zipcode also supports a mechanism for Cartesian subgridding, analogous to `MPI_Cart_sub` and `MPI_Comm_split` (see [15]).

Features Not in MPI

Zipcode topologies are more closely tied to message passing than in either MPI or PARMACS. Furthermore, Zipcode is tied to the model of a topology as a hierarchical family of intracommunicators. This is done automatically to support the communication over subgroups that is likely for a Cartesian topology.

As pointed out above, Zipcode supports dynamic memory allocation for point-to-point messages, while using user-specified buffers (as MPI does) for collective communication. In the "pack" modes, the operations for Zipcode use invoices and encapsulate heterogeneous conversion.

Process Startup

Zipcode provides the process management support for which there is no analog in MPI.

Tools

The distributors of Zipcode are planning to develop a Zipcode-to-MPI port that will allow Zipcode programs to operate on top of MPI, while defining versions of Zipcode invoices for MPI to simplify the use of MPI datatypes.

9.12 Where to Learn More

In this chapter, we have outlined porting strategies for a number of existing vendor and portability systems. Additional information about porting and about detailed features of the other systems is available. The Internet newsgroups *comp.parallel*, *comp.parallel.pvm*, and *comp.parallel.mpi* provide access to up-to-the-minute discussions on message-passing systems, including details of real experience with porting. See also the discussion in Appendix D for other ways to access information on MPI as well as many of the systems described in this chapter on the Internet.

.

This chapter was processed by LaTeX on September 5, 2000.

Bibliography

[1] A. Beguelin, G. A. Geist, W. Jiang, R. Manchek, K. Moore, and V. Sunderam. The PVM project. Technical report, Oak Ridge National Laboratory, February 1993.

[2] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994. (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493).

[3] Robert Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20, April 1994.

[4] Fei-Chen Cheng. Unifying the MPI and PVM 3 systems. Technical report, Mississippi State University — Dept. of Computer Science, May 1994. Master's Report. (Available by anonymous ftp: *cs.msstate.edu*, `pub/reports/feicheng.ps`).

[5] ParaSoft Corporation. *Express C User's Guide, Version 3.0*, 1990.

[6] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994. (To appear in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994).

[8] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[9] Brian K. Grant and Anthony Skjellum. The PVM Systems: An in-depth analysis and documenting study: Concise edition. Technical Report UCRL-JC-112016, Lawrence Livermore National Laboratory, August 1992. (Revised).

[10] R. J. Harrison. Portable tools and applications for parallel computers. *Intern. J. Quantum Chem.*, 40(847), 1991.

[11] R. Hempel, H.-C. Hoppe, and A. Supalov. *PARMACS 6.0 Labrary Interface Specification*. German National Research Center for Computer Science, December 1992.

[12] Charles L. Seitz et al. The C programmer's abbreviated guide to Multicomputer programming. Technical Report Caltech-CS-TR-88-1, California Institute of Technology, January 1988.

[13] Charles L. Seitz, Sven Mattisson, William C. Athas, Charles M. Flaig, Alain J. Martin, Jakov Seizovic, Craig M. Steele, and Wen-King Su. The architecture and programming of the Ametek series 2010 multicomputer. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (HCCA3)*, pages 33–36. ACM Press, January 1988. (Symult s2010 Machine).

[14] Anthony Skjellum and Alvin P. Leung. *Zipcode:* A portable multicomputer communication library atop the Reactive Kernel. In *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pages 767–776. IEEE Press, April 1990.

[15] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The design and evolution of zipcode. *Parallel Computing*, 20, April 1994.

[16] Anthony Skjellum, Steven G. Smith, Charles H. Still, Alvin P. Leung, and Manfred Morari. The Zipcode message-passing system. In Geoffrey C. Fox, editor, *Parallel Computing Works!* Morgan Kaufman, 1994. (book chapter).