Message Passing Interface Tutorial (Introduction and Part II)

Alan Scheinine, IT Consultant

High Performance Computing,
Center for Computational Technology and
Information Technology Services
Louisiana State University

E-mail: scheinin@cct.lsu.edu

Contents

1 Introduction **1**

2 General Considerations **4**

2.1 Overall Program Structure 4

2.2 Predefined Types 5

3 Point-to-Point Communication **6**

3.1 Blocking Send 6

3.2 Blocking Receive 7

3.3 Buffered Communication 7

3.4 Non-blocking Send and Recv 8

3.5 Note on Dense Communication 9

4 Collective Communication **10**

4.1 Summary 10

4.2 Description of routines 11

4.3 Note on efficiency of all-to-all 15

4.4 Note on branching from float comparison 15

5 Groups and Communicators **16**

6 MPI-2 **18**

6.1 Capabilities of MPI-2 18

6.2 Intergroup Communication in MPI-2 18

6.3 Memory Allocation 19

6.4 Introduction to One-Sided Communications 19

6.5 Details of One-Sided Communications 20

7 Compiling Parallel Programs and Running MPI **25**

7.1 Combinatorial Explosion of Versions 25

List of Figures

1 Sparse versus Dense Communication 10

List of Tables

1 Sources of Documentation and Software 3

2 MPI Predefined Fortran Data Types 5

3 MPI Predefined C Data Types 6

4 Predefined Reduction Operations 14

1 Introduction

This write-up will provide more details than is possible in a set of slides. However, this is not at all sufficient for a reference to MPI-1 and MPI-2. The intention is to point-out the key ideas from the manuals and from experience.

Parallel programming can be grouped into two categories: shared memory and message passing. In shared memory programs, such as OpenMP and multithread, the memory addresses are consistent for all processes. For PC-like computers this limits the processes to one board (or node). As an example, for a dual-socket board with quad-core processors, eight shared-memory processes could run efficiently. (Note, OpenMP is different from Open-MPI, the latter is an implementation of MPI.) In contrast, the message-passing paradigm of MPI can be used for programs with hundreds to thousands of processes. This is called "distributed memory." Data is passed using function calls. For MPI-1 both sender and receiver must call the appropriate function. For MPI-2 there is an additional communication mode "one-sided communication" for which a `send()` does not need to be paired with a `recv()`. Message-passing and shared memory can be combined, which is motivated by the assumption that the shared-memory communication is faster. However, in a discussion in the Beowulf mailing list several people claimed that, in practice, the combination is too complex and does not result in faster execution. The discussion can be found in the thread named
"multi-threading vs. MPI" in
`http://www.beowulf.org/archive/2007-December/thread.html`

MPI has become a widely used standard, though not necessarily the best language for parallel programming. At a European supercomputer conference several years ago (most likely International Supercomputer Conference, Heidelberg, 2004) the speaker took a poll of the several hundred people in the audience. He said that many parallel languages have been developed but most are now rarely used. He asked for a show of hands for how many people use each language, naming languages such as Orca, Linda, Erlang, High Performance Fortran (HPF) and Occam. At most only a few people raised their hands for each language, except for MPI which was (and is) widely used.

By the way, HPF was once popular. In HPF the message passing is implicit, hidden from the user. In practice the implementations were not efficient, more message-passing was done than necessary. A more recent Fortran 95 feature that is similar to HPF is Co-Array Fortran `http://www.co-array.org/` .

Initially MPI launches copies of an executable. Each copy could branch to execute a different part of the code (different yet related algorithms) but it is much more common for

each process to execute the same code with differences in the data for the inner-most loops. For example, domain decomposition with one domain per process. Actually, each process can handle more than one domain, this is especially relevant when domains have a wide range of sizes for which the goal is load-balancing so that every process finishes an iteration at the same time.

MPI works better for *coarse-grained* parallelism, as opposed to *fine-grained* parallelism. Each copy of the executable runs at its own rate and is not synchronized with other copies unless the synchronization is done explicitly with a “barrier” or with a blocking data receive. As an aside, the LAM version of MPI permits heterogeneous executables.

The symmetry of the copies of the algorithm must be broken for communication. In particular, the following example will deadlock.

```
Process 0:
   send(message, proc1); // Wrong, for blocking send, will deadlock.
   recv(message, proc1);

Process 1:
   send(message, proc0);
   recv(message, proc0);
```

The send() and recv() in this example are more simple than the actual MPI routines. The point here is to emphasize that often the sequence of instructions of an MPI program are nearly identical for every process, but additional attention is needed to avoid communication deadlock.

The following are a few miscellaneous comments about I/O. MPI attempts to combine from all processes the output to *stdout* and *stderr* and have these output appear on one *stdout* and one *stderr*. This does not always work correctly or may result in output that is difficult to read. MPI has mechanisms built-in for combining *stdout* and *stderr* but in general **do not have more than one process write to a sequential file**. The confusion arises because each process needs to append to the end of the file but the end changes due to other processes writing. One solution is to have a single process collect the data and just that process open the file for writing. Another solution, for MPI-1, is to use a special-purpose parallel I/O library such as ROMIO. MPI-2 has an extensive I/O interface, though it will not be described in this tutorial.

You can install public-domain MPI on your home computer, though it is oriented towards Unix-like operating systems. You can run MPI on a single personal computer, even

off-line using "localhost" as the node name. So one can test parallel programs on a single laptop, afterall the laptop might have two or even four cores. The number of processes can exceed the number of cores, though efficiency will be reduced.

Table 1: Sources of Documentation and Software

- `http://www.mpi-forum.org/docs/docs.html`
  Contains the specifications for MPI-1 and MPI-2.
- `http://www.redbooks.ibm.com/redbooks/pdfs/sg245380.pdf`
  RS/6000 SP: Practical MPI Programming
  Yukiya Aoyama and Jun Nakano
- `http://www-unix.mcs.anl.gov/mpi/`
  Links to tutorials, papers and books on MPI.
- `http://www-unix.mcs.anl.gov/mpi/tutorial/index.html`
  A list of tutorials.
- Books
  - Using MPI, by W. Gropp, E. Lusk and A. Skjellum [1]
  - Using MPI-2, by W. Gropp, E. Lusk and R. Thakur [2]
  - MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions,
    William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir and Marc Snir [3]
  - Parallel Programming With MPI, by Peter S. Pacheco [4]
- Software
  - MPICH2
    `http://www.mcs.anl.gov/research/projects/mpich2/`
  - OpenMPI
    `http://www.open-mpi.org/software/`

2 General Considerations

2.1 Overall Program Structure

The overall, outermost structure of an MPI program is shown below.

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
   int rank;
   MPI_Init( argc, argv );
// From here on, N processes are running, where N is
// specified on the command line.
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
// Can use the rank to determine algorithm behavior.
   if( rank == 0 ) fprintf(stdout, "Starting program\n");
   fprintf(stdout, "Process %d running\n", rank );
// At end of program, finalize.
   MPI_Finalize();
   return 0;
}
```

All code before `MPI_Init()` runs as a single executable but typically `MPI_Init()` is the first executable line. Since MPI is based on distributed memory, aside from MPI communications, the variables of each process are independent.

The C++ bindings of MPI will not be discussed. For other languages, the headers for each program unit are the following.

- For C,
  `#include <mpi.h>`
- For Fortran 77,
  `      include 'mpif.h'`
- For Fortran 95,
  `   use mpi`

The Fortran subroutines of MPI use the last argument to supply an error status whereas C functions of MPI use the return value for the error status. Success corresponds to the value of `MPI_SUCCESS`.

A subset of processes can be grouped together, assigned ranks within the group and a "communicator" can be assigned to the group. A process can be a member of more than one group. Further details concerning process groups will be explained in Sec. 5. My point here is to explain why there is always the argument `MPI_Comm comm` in all function calls. The choice of including all processes has a predefined macro `MPI_COMM_WORLD`.

The point-to-point routines will be described only briefly because details are available in the manual.

2.2 Predefined Types

The user can define a data type, but this will not be discussed in this tutorial. The predefined data types for Fortran and C are given in the following tables.

Table 2: MPI Predefined Fortran Data Types

MPI datatype	Fortran datatype
`MPI_INTEGER`	INTEGER
`MPI_REAL`	REAL
`MPI_DOUBLE_PRECISION`	DOUBLE PRECISION
`MPI_COMPLEX`	COMPLEX
`MPI_LOGICAL`	LOGICAL
`MPI_CHARACTER`	CHARACTER(1)

Table 3: MPI Predefined C Data Types

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Other data types may be predefined such as MPI_DOUBLE_COMPLEX for Fortran and MPI_LONG_LONG_INT for 64-bit C integers declared as `long long int`. I recommend reading the MPI C header file to be certain of the definition of MPI_LONG and MPI_LONG_LONG_INT in an actual implementation.

The language independent types are MPI_BYTE and MPI_PACKED.

3 Point-to-Point Communication

3.1 Blocking Send

```
int MPI_Send(void *buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm)
```

The program can change the contents of buf when the function returns. When the function returns either a local copy has been made or the message has already been sent to the destination, depending on implementation and message size. The count is the number of elements, not the number of bytes. The most common case is that “comm” is MPI_COMM_WORLD.

3.2 Blocking Receive

```
int MPI_Recv(void *buf, int count, MPI_Datatype type,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

The size of count must be at least as large as the message size. Array buf has the complete message when the function returns.

To discover the actual size of the message use

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype type,
                  int *count);
```

Rather than specifying a specific source process, any source can be specified by the argument `MPI_ANY_SOURCE`. Messages can be categorized by an integer-valued "tag." A `recv()` can be restricted to a specific tag, or to receive a message with any tag the argument `MPI_ANY_TAG` can be used. Messages are non-overtaking between matching `send()` and `recv()` for a single-threaded application. On the other hand, MPI does not guarantee fairness for wildcard receives, that is, all receives could come from the same source while ignoring messages from other sources.

Other send types are "buffered", "synchronous" and "ready." Only buffered send will be described.

3.3 Buffered Communication

A local buffer is used to hold a copy of the message. The function returns as soon as the message has been copied to the local buffer. Advantage: prevents deadlocks and quickly returns. Disadvantage: latency introduced by having an addition copy operation to the local buffer. The buffered send routine has the same arguments as blocking send and is called `MPI_Bsend`.

Buffer Allocation

The program code for allocating and releasing buffer space is shown below.

```
// Allocate buffer space
int size = 10000;
int stat;
char* buf = (char*) malloc(size);
// Make buffer available to MPI.
stat = MPI_Buffer_attach(buf, size);
// When done release the buffer.
stat = MPI_Buffer_detach(buf, &size);
free(buf);
```

If the program has nearly constant behavior during a run, then the buffer allocation need only be done once at the beginning of the program (after calling `MPI_Init`).

3.4 Non-blocking Send and Recv

For non-blocking send and recv, when a function returns, the program should not modify the contents of the buffer because the call only initiates the communication. The functions `MPI_Wait()` or `MPI_Test()` must be used to know whether a buffer is available for reuse. Analogously two pending `MPI_Isend()`, `MPI_Ibsend()` or `MPI_Irecv()` cannot share a buffer.

Without user-defined buffer

```
int MPI_Isend(void* buf, int count, MPI_Datatype type, int dest,
              int tag, MPI_Comm comm, MPI_Request *request);
```

With user-defined buffer

```
int MPI_Ibsend(void* buf, int count, MPI_Datatype type, int dest,
               int tag, MPI_Comm comm, MPI_Request *request);
```

The receive command is

```
int MPI_Irecv(void* buf, int count, MPI_Datatype type, int source,
              int tag, MPI_Comm comm, MPI_Request *request);
```

The following function is used to test whether an operation is complete. Operation is complete if “flag” is true.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

The following can be called to wait until the operation associated with request has completed.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

Often the simplicity of blocking send and recv is preferred.

There are more types of point-to-point communication such as ”persistent communication requests”, send-receive combined, and derived data types including structures; all of these will not be covered in this tutorial.

3.5 Note on Dense Communication

There has been a paradigm shift in how the communication network is used. To simplify the discussion, suppose that we are only using point-to-point communication. The conventional approach to `MPI_Allreduce` is shown in the diagram labelled “Fewer messages” in Fig. 1. A faster method is shown in the second diagram labeled “Denser Communications” in the same figure.

Of course, those learning MPI for the first time will not notice a paradigm shift. At one time in the past, about a decade ago, a reduction in the traffic of the communication network — fewer messsages near the waist of the first diagram — was desirable. This was especially true for older Ethernet switches.

In contrast, higher quality Ethernet, Myrinet and Infiniband switches used on many institutional clusters can handle all ports being active simultaneously, at least for certain subsets of communication patterns. Of course one needs to avoid hot spots in which many processes try to send messages to the same port at the same time. For the sake of this discussion, a “port” is a compute node connection to the fabric.

I’m fairly certain I heard a talk about this idea at the Proceedings of the 2000 ACM Symposium on Applied Computing, held at Como, Italy, but now I’m unable to read online

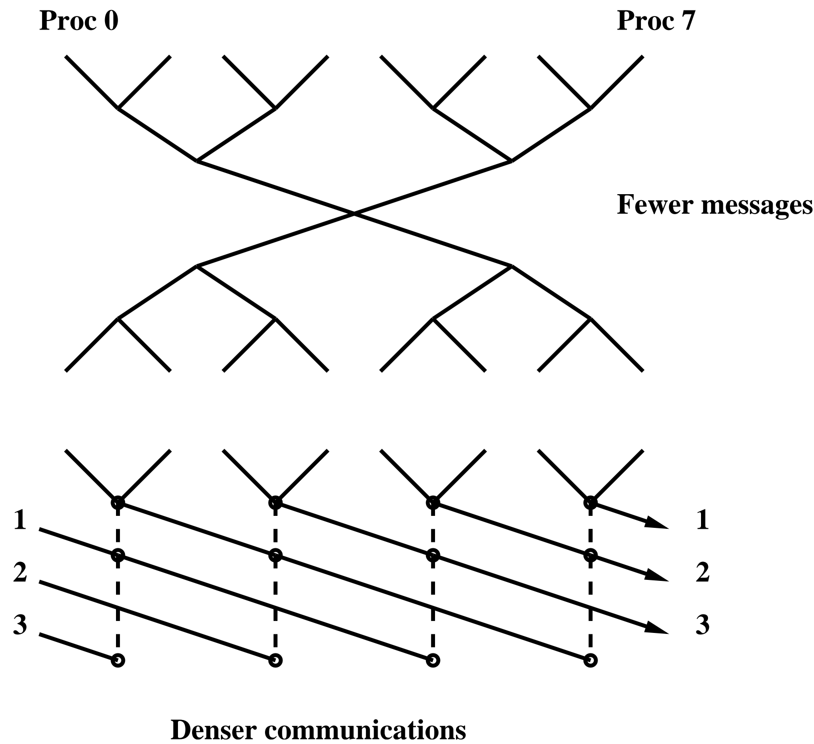

Figure 1: Full use of the connection fabric can reduce the execution time of algorithms.

the proceedings. The concept is not specific to `MPI_Allreduce`, far from it. Your specific application may have complex, non-symmetric communication with intermediate calculations. Nonetheless, if one possible solution can use nearly all the connections fairly uniformly, you may find that the solution scales well, without suffering from a network saturation effect.

4 Collective Communication

4.1 Summary

The various categories are the following.

- Barrier synchronization blocks the caller until all group members have called it.
- Broadcast from one member to all members of a group.
- Gather data from all group members to one member.
- Scatter data from one member to all members of a group.

- Allgather, a gather where all members of the group receive the result.
- Alltoall, every process in a group sends a message to every other process.
- Global reduction such as sum, max, min and others. The result can be returned to all group members. The reduction can be a user-defined function.
- Reduction and scatter where the result is a vector that is split and scattered.
- Scan, which is a reduction from all a each processes with all lower ranked processes. No further description of scan will be given in this tutorial.

For collective routines that have a single originating or receiving process, that special process is called the "root". MPI routines have the same set of arguments on all processes, though for some collective routines some arguments are ignored for all processses except the root. When a caller returns, the caller is free to access the communication buffers. But it is not necessarily the case that the collective communication is complete on other processes, so the call (aside from `MPI_Barrier`) does not synchronize the processes. The processes involved are the group associated with the `MPI_COMM` argument. MPI guarantees that messages generated on behalf of collective communication calls will not be confused with point-to-point communications. The amount of data sent must be equal to the amount of data received, pairwise between each process and the root.

4.2 Description of routines

The routines will be described briefly when the operation is obvious. Additional details will be given where clarification is needed.

```
int MPI_Barrier(MPI_Comm comm)
```

Barrier synchronization blocks the caller until all group members have called it.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

Broadcast from one member to all members of a group.

```
int MPI_Gather(void *sendbuffer, int sendcount, MPI_Datatype sendtype,
               void *recvbuffer, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

Gather data from all group members to one member. The arguments recvbuffer and recvcount are only significant on the root process. Data from process i starts at `recvbuf[`$i \times recvcount$`]`

There is a more flexible version as shown below.

```
int MPI_Gatherv(void *sendbuffer, int sendcount, MPI_Datatype sendtype,
                void *recvbuffer, int *recvcounts,
                MPI_Datatype recvtype, int *displacements,
                int root, MPI_Comm comm)
```

Data from process i starts at `recvbuffer[displacements[i]]` and there is different recvcounts for each process. The recvbuffer is ignored for all non-root processes. The recvcounts and displacements should not cause any location on the root to be written more than once.

There is an example in the MPI-1 Standard book of where the size of the dataset from each process is first sent from each process with `MPI_Gather`, then `MPI_Gatherv` is used to send the differently sized datasets.

```
int MPI_Scatter(void *sendbuffer, int sendcount, MPI_Datatype sendtype,
                void *recvbuffer, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

Segments of sendbuffer are sent to different processes, for process i data starts at `sendbuffer[`$i \times sendcount$`]`. The sendbuffer is ignored for all non-root processes. The root process also receives a portion.

```
int MPI_Scatterv(void *sendbuffer, int *sendcounts,
                 int *displacements, MPI_Datatype sendtype,
                 void *recvbuffer, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

`MPI_Scatterv` is the inverse operation to `MPI_Gatherv`. For process i the data starts at `sendbuffer[displacements[i]]`. The specification of counts and types should not cause any location on root to be read more than once.

There are routines `MPI_Allgather` and `MPI_Allgatherv` which have the same arguments as `MPI_Gather` and `MPI_Gatherv` except the argument "root." All processes receive the same result.

There are two versions of all-to-all as shown below.

```
int MPI_Alltoall(void *sendbuffer, int sendcount, MPI_Datatype sendtype,
                 void *recvbuffer, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```

```
int MPI_Alltoallv(void *sendbuffer, int *sendcounts,
                  int *send_displacements, MPI_Datatype sendtype,
                  void *recvbuffer, int *recvcounts,
                  int *recv_displacements, MPI_Datatype recvtype,
                  MPI_Comm comm)
```

No further description will be given because the operation is easily deduced from the routines already described.

Reduction across all processes can be done using the following.

```
int MPI_Reduce(void *sendbuffer, void *recvbuffer, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Allreduce(void *sendbuffer, void *recvbuffer, int count,
                 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

The reduction can be done on more than on element of **sendbuffer** (in which case **count** > 1) but each element is treated independently, that is, the reduction is done over the processes. `MPI_Allreduce` is the same as `MPI_Reduce` except that the result appears in the receive buff of all the group members. The MPI-1 standard advice to implementors is

> It is strongly recommended that `MPI_Reduce` be implemented so that the same result be obtained whenever the function is applied on the same arguments,

appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors.

Nevertheless, I'd recommend not assuming that the advice is followed, as I've described in Sec. 4.4.

The reduction operation can be user-defined, which will not be discussed in this tutorial. The predefined operations are described in Table 4.2

Table 4: Predefined Reduction Operations

Name	Meaning
`MPI_MAX`	maximum
`MPI_MIN`	minimum
`MPI_SUM`	sum
`MPI_PROD`	product
`MPI_IAND`	logical and
`MPI_BAND`	bit-wise and
`MPI_IOR`	logical or
`MPI_BOR`	bit-wise or
`MPI_IXOR`	logical xor
`MPI_BXOR`	bit-wise xor
`MPI_MAXLOC`	max value and location
`MPI_MINLOC`	min value and location

4.3 Note on efficiency of all-to-all

Several times there has been a debate on the Beowulf mailing list about the efficiency of `MPI_Alltoall`. One would expect that MPI developers (MPICH, OpenMPI and Cray, for example) would spend time optimizing this function. Mailing list contributors who have worked on MPI professionally have claimed that `MPI_Alltoall` is often slow because of the huge number of combinations that must be covered. The number of processors can vary from tens to thousands. Short messages often need to be handled differently from long messages. Moreover, some processes may be on the same node. Even if `MPI_Alltoall` is optimized at one point in time, the attributes of the communication fabric will change over time. Though I recall this being a perennial subject, debated on the mail list every few years, I am only able to find a 2000 thread named
"Beowulf and FFT" in
`http://www.beowulf.org/archive/2000-July/thread.html`

The claim made by some, for which there is not universal agreement, is that the number of possible cases exceeds the capacity of even the brightest programming team. So if time is available for experimentation, in addition to `MPI_Alltoall`, also try collections of point-to-point MPI functions because the latter may prove faster.

4.4 Note on branching from float comparison

Do not have a program branch globally based on independent comparisons of floating-point numbers. "Independent" in the sense of the same comparison in each process of many processes. For the same vector, the magnitude (or any other floating-point reduction) depends on the order with which the squares of elements are summed. For global branching, one process should decide and send the decision to the other processes.

As an example, consider two outer loops. The outermost is a time iteration in which external boundary conditions are changed. The next level of looping consists of iterations of conjugate gradient (C.G.). Domain decomposition is used with one process for each domain. In effect, every vector operation is split between the domains. Several times during each C.G. iteration data for the borders between domains is passed with MPI. After each C.G. iteration the magnitude squared of the residual vector is compared with a number that determines the stopping criterion. Partial sums of squares can be calculated within each process then `MPI_Allreduce` can be used to combine the partial sums and have the result available to all processes. That each process might receive a slightly different value may seem like an error in MPI, but in fact the sum could be done in different orders for different destinations. The program would fail if some processes expect to exchange data at their mutual borders while

other processes are waiting for new external values for the next iteration of the outermost loop.

This concept must be generalized. The output of `MPI_Allreduce` will likely be that same for all processes, but let us consider a user-written reduction. Suppose a program collects the values of a certain floating-point variable from all other processes, sums the values and then adds the local value to obtain a "global sum." In this case infinite precision math would give the same values at all processes whereas actual computer precision would result in a different "global sum" on each process because the order of the same would be different at each process.

5 Groups and Communicators

Up until this point in this tutorial the "communicator" has been `MPI_COMM_WORLD`. The chapter on "Groups, Contexts and Communicators" in the 1995 MPI Standard describes this subject primarily for libraries of parallel subroutines. A separate communicator for a library "allows the invocation of the library even if there are pending communications on 'other' communicators and avoids the need to synchronize entry or exit into the library code." Since user-defined communicators can be used in any MPI program, not just in libraries, the basic concepts will be described.

A "process group" is represented by an opaque group object. The processes in a group are ranked, moreover, the ranks are contiguous and start from zero. The group associated with a communicator can be found with

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

A new group can be created with

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
group     Input
n         Input, size of ranks
ranks     Input array
newgroup  Output
```

The process of `rank[i]` in the already existing group is process of rank `i` in `newgroup`.

New groups can be created by operations on pairs of existing groups. Without going into the details, the functions include MPI_Group_union, MPI_Group_intersection and MPI_Group_difference. Members and size of a group can be found using

```
int MPI_Group_size(MPI_Group group, int *size)
int MPI_Group_rank(MPI_Group group, int *rank)
```

The second function above gives the rank of the calling process or MPI_UNDEFINE if the process is not a member of the group. A group can be eliminated using

```
int MPI_Group_free(MPI_Group *group)
```

A new communicator can be created with

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

where the input argument "group" is a subset of the group associated with the input communicator "comm". Of course, one can use

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

on the communicator you created.

MPI-2 has extended the functionality of collective communication between groups, briefly mentioned in Sec. 6.2 of this tutorial.

Point-to-point operations and collective operations, such as scatter, gather and reduce, can be applied to a user-defined communicator without interfering with any other communicator such as MPI_COMM_WORLD. On the other hand, recall that calls using the same communicator can interfere. For example, if a routine does both send and receive and the receive is for any tag, then messages with the same tag will be ordered but messages with different tags can be out of order.

6 MPI-2

6.1 Capabilities of MPI-2

The following capabilities of MPI-2 will not be discussed.

External Interfaces provides facilities for extending MPI in as yet unforeseen ways.

I/O provides cooperative I/O in which many processes concurrently access a single file. The standard is very detailed, too detailed for this tutorial. Apart from MPI, there are several parallel file systems which provide very high bandwidth to a collection of files, though I/O to each individual file is usually slower than a local HDD. For parallel programs one technique for increasing I/O bandwidth is to have different processors write to different files, utilizing the strong point of a typical parallel file system. In contrast, the MPI I/O provides fine-grained control for concurrent access to a file by several processors.

C++ Language Binding requires learning a new set of MPI classes and function names. It would probably be easier to use the C language binding within a C++ program.

Fortran 90 Language Binding is not discussed in this tutorial but if someone is using Fortran 90 or 95 then I recommend reading section 8.2 of [3] or section 10.2 of [5].

Process Creation and Management permits processes to be started during a run, not necessarily at the start. Integration with a batch queue system is non-trivial, though such integration is anticipated by the MPI language specs.

The focus of this section will be one-sided communications, but first a brief word on intergroup communication and memory allocation.

6.2 Intergroup Communication in MPI-2

MPI-2 specifies a method of collective communication between different groups. Briefly, the routine argument “root” is set to `MPI_ROOT` on the root process, argument “root” is set to `MPI_PROC_NULL` on all other processes in the root group and for all processes not in the root group the argument “root” is set to the rank of the root process in the root group.

6.3 Memory Allocation

On some implementations, message-passing and remote-memory-access (one-sided communications) may be faster when the memory used is allocated with

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
```

The "info" argument can be used to provide directives that control the location of the allocated memory. Valid "info" values are implementation dependent. The null directive `info = MPI_INFO_NULL` is always valid.

Memory can be freed with

```
int MPI_Free_mem(void *base)
```

Note that `*baseptr` is the address of `*base`.

6.4 Introduction to One-Sided Communications

Here I paraphrase the important points from the introduction of one-sided communications in [3]. Standard send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This may require all processes to participate in a time-consuming global computation, or to periodically poll for potential communication requests to receive and act upon. The use of Remote Memory Access (RMA) communication mechanisms avoid the need for global computations or explicit polling.

MPI one-sided operations do *not* provide a shared-memory programming model. Moreover, correct ordering of memory access has to be imposed by the user, using synchronization calls. All RMA communication routines are nonblocking. We shall denote by **origin** the process that performs the RMA call, and by **target** the process in which the memory is accessed. Thus, in a `put()` operation, source $\equiv$ origin and destination $\equiv$ target; in a `get()` operation, source $\equiv$ target and destination $\equiv$ origin.

RMA allows one process to specify all communication parameters for origin and target, however, the target is involved in synchronization for "active target" communication but not for "passive target" communication.

Using shortened routine names to simplify this summary, already allocated buffer space is assigned to a "window" set for one-sided communications with `create` routine. One-side communication calls are `put`, `get` and `accumulate`. Synchronization for active target consists of the collective routine `fence` and the pairwise routines `start`, `complete`, `post` and `wait`. Passive target communications use `lock` and `unlock`.

6.5 Details of One-Sided Communications

Initialization of Windows

A window is created and freed with the following:

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
                   MPI_Info info, MPI_Comm comm, MPI_Win *win)
int MPI_Win_free(MPI_Win *win)
```

All processes in the group of `comm` must call these. It returns a window object `win`. Each process specifies a region of existing memory that it exposes to RMA access by processes in the group of `comm`. Note that `win` is a collection of windows, one on each process. A process may elect to expose no memory by specifying zero for the size. The only `info` predefined is the key `no_locks` if locks are never used for the window. Curiously, the macro for that value of `MPI_INFO` is not described in the reference manual, moreover, it is not declared in the include files for MPI-2. The predefined macro `MPI_INFO_NULL` seems to me to be the best choice due to the lack of information. Each process can have different values for `base`, `size` and `disp_unit`. The same memory area can be associated with different windows. Typical values of `disp_unit` are 1 for no scaling and `sizeof(type)`, the latter allows the use of array indices in RMA calls.

The attributes of a window can be found using the following:

```
MPI_Win_get_attr(MPI_Win win, MPI_WIN_BASE, void *base, int *flag);
MPI_Win_get_attr(MPI_Win win, MPI_WIN_SIZE, MPI_Aint *size, int *flag);
MPI_Win_get_attr(MPI_Win win, MPI_WIN_DISP_UNIT, int *disp_unit, int *flag);
MPI_Win_get_group(MPI_Win win, MPI_Group *group)
```

The flag is true if the value requested has been set.

Communication Routines

MPI_Put transfers data from the caller's memory (origin) to the target. MPI_Get transfers data from the target memory to the caller's memory. MPI_Accumulate modifies memory on the target using data on the caller using the MPI_Reduce operators. The buffers involved should not be changed until a synchronization routine is called for the window. It is erroneous to have concurrent conflicting access to the same memory location in a window. However, send buffers can overlap (unlike message passing), moreover, the same location can be updated by several concurrent MPI_Accumulate calls.

The routines have the following calling arguments.

```
int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_type,
            int target_rank, MPI_Aint target_disp, int target_count,
            MPI_Datatype target_type, MPI_Win win)

int MPI_get(void *origin_addr, int origin_count, MPI_Datatype origin_type,
            int target_rank, MPI_Aint target_disp, int target_count,
            MPI_Datatype target_type, MPI_Win win)

int MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_type,
            int target_rank, MPI_Aint target_disp, int target_count,
            MPI_Datatype target_type, MPI_Op op, MPI_Win win)
```

We see that both **put** and **get** have the same calling arguments and **accumulate** only differs in that an operation is specified. No routine call on the target is necessary, except that before and after a series of puts and gets the target will need to call a synchronization routine (for the case of active target communication). For a **put** data is transfered from the origin to the target. For a **get** data is transfered from the target to the origin. The buffer at the origin need not be a previously declared window, origin_addr is just the start address of contiguous memory. The origin_count is the number of elements, where each element has the size corresponding to origin_type. The location on the target is the location specified by the pair win and target_rank. Moreover, just as for the origin, at the target the number of bytes transfered depends on the element count target_count and the type target_type. The disp_unit declared when the window was created has a role in determining the offset. The target starting address (target_addr) for the data transfer is given by
target_addr = window_base + target_disp × disp_unit
where window_base is the base used in the **create** routine and target_disp is an argument of the **put**, **get** or **accumulate** routine.

The **accumulate** routine modifies the target and can use any of the predefined operations available for `MPI_Reduce`.

Synchronization Routines

Communication calls at the origin occur during an *access epoch* for `win`. For active communication, a target window can only be accessed within an *exposure epoch*. There is a one-to-one correspondence between access epochs at the origin and exposure epochs at the target. For different windows these epochs are independent.

The most general purpose synchronization is the **fence**,

```
int MPI_Win_fence(int assert, MPI_Win win)
```

An access epoch begins and ends with a **fence** call and an exposure epoch begins and ends with a **fence** call. An implementation of MPI could collect various communication commands and execute them all when a **fence** is called. A **fence** call both ends and starts an epoch, either access or exposure. A **fence** call should precede and follow calls to **put**, **get** or **accumulate** when these are being synchronized with **fence** calls. A process completes a call to **fence** after all other processes in the group entered their matching call, but it might not act as a barrier in the case that it does not end an epoch – a **fence** does not end an epoch if there has been no intervening communication.

The fence effect can be understood in terms of the following rules. All RMA operations on `win` originating at a given process and started before the **fence** call will complete at that process before the **fence** call returns. All RMA operations will be completed at their target before the **fence** call returns at the target. RMA operations on `win` started by a process after the fence call returns will access their target window only after **fence** has been called by the target process.

For finer control of active target communications there are the routines

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
int MPI_Win_test(MPI_Win win, int *flag)
```

Routine **start** starts an RMA access epoch for `win`. RMA calls issued on `win` during this epoch must access only windows at processes in `group`. Let me emphasize that the group can be a proper subset of the group of the communicator used when `win` was created.

Each process in `group` must issue a matching call to **post**. RMA communication to each target window will be delayed, if necessary, until the target process has executed a matching **post**. (The routine **start** may block until corresponding **post** calls are executed, depending on the implementation.) Routine **complete** completes the access epoch on `win` initiated by the call to **start**. All RMA communication calls issued on `win` during this epoch will have completed at the *origin* when the call returns, but at that point the RMA at the target may not be complete.

The routine **post** starts an RMA exposure epoch for the local window associated with `win`, it does not block. The routine **wait** completes an RMA exposure epoch started by the call to **post** on `win`. The call to **wait** will block until all matching calls to **complete** have occured (to assure that originators have finished communication), moreover, when the call returns, all associated RMA accesses will have completed at the target window.

Various implementations could have either: **start** blocks until the matching **post** call occurs on all target processes; or **start** can be nonblocking but **put** blocks until a matching call to **post**; or the first two calls do not block but **complete** blocks until the call to **post**; or **complete** is nonblocking before any target process calls **post**. In this last case, the implementation would need to buffer that data.

Note that each processes can call **post** or **start** with a group argument that has *different* members. Thereby maximizing the level of confusion . . . just joking — about the confusion. The requirements are easy to understand in terms of a directed graph where each node is a processes and each directed edge corresponds to an origin-target relationship of RMA. Every node with outgoing edges must call **start** with a `group` that corresponds to the destinations of the outgoing edges. In addition, every node with incoming edges must call **post** with a `group` that corresponds to the sources of the incoming edges. After RMA communication, each process that issued a **start** will issue a **complete** and each process that issued a **post** will issue a **wait**.

The **test** routine is a nonblocking version of **wait** and `flag` returns a value of `true` if **wait** would have returned. If `flag` is `true` then the effect is the same as a call to **wait**, if `false` then there is not effect.

Passive communication involves locking and unlocking a window on a target without any calls made by the target process. Two other processes can exchange data using the window on the target.

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_win win)
int MPI_Win_unlock(int rank, MPI_Win win)
```

The value of `lock_type` can be either `MPI_LOCK_EXCLUSIVE` or `MPI_LOCK_SHARED`. The **lock** command starts an access epoch. Only the window at the process `rank` can be accessed by operations on `win` during the epoch. When the call to **unlock** returns, RMA operations issued during the epoch will have completed both at the origin and at the target. It is erroneous to have a window locked and exposed concurrently, that is, the process with a locked window cannot call **post** on the window.

Assertions

Assertions provide information about the context of a call that aid in optimization of performance. Many implementations may not take advantage of the information in `assert`. The possible values (a bit-vector OR of options that depend on the routine) will not be described in this tutorial because no synthesis would be useful. The user will need to read the documentation. A value of zero can be used to indicate that no guarantees are made.

Fortran Memory

The book [3] that describes the MPI-2 standard warns that array slices of Fortran 90 and Fortran 95 involve a copy upon subroutine call and again upon subroutine return, and so are not appropriate for non-blocking communication (including RMA) where the buffer argument describes the location of I/O to be performed after the call. In addition the book describes Fortran's use of registers that results in a lack of cache coherence. To avoid the problem, the book recommends that for Fortran the buffers be in `COMMON` or declared as `VOLATILE`. On the other hand, in the same book in the part concerning locks, the authors write

> Implementations may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by `MPI_ALLOC_MEM`.

[...]

> Also, *passive* [emphasis mine] target communication cannot be portably targeted to `COMMON` blocks, or other statically declared Fortran arrays.

General Advice for RMA

The book [3] has several pages that describe the synchronization rules and further synthesis in this tutorial is not practical. However, one set of "Advice to users" is worth repeating here.

fence: During each period between fence calls, each window is either updated by put or accumulate calls, or updated by local stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.

post-start-complete-wait: A window should not be updated locally while being posted, if it is being updated by put or accumulate calls. Locations updated by put or accumulate call should not be accessed while the window is posted (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window is posted. With the post-start synchronization, the target process can tell the origin process that its window is now ready for RMA access; with the complete-wait synchronization, the origin process can tell the target process that is has finished its RMA accesses to the window.

lock: Updates to the window are protected by exclusive locks if they may conflict. Nonconflicting accesses (such as read-only accesses or accumulate accesses) are protected by shared locks, both for local accesses and for RMA accesses.

7 Compiling Parallel Programs and Running MPI

7.1 Combinatorial Explosion of Versions

More information about using MPI will be given in exercises. But one point is so fundamental that I would like to mention it in the introduction. Each base compiler and each type of connection will have a different MPI compiler and different MPI library files. By "base compiler" I mean the compilers from Gnu, Intel, PGI, Pathscale or others. Compilation and linking is done by scripts such as `mpicc` that pull-in the MPI communication libraries compiled with specific base compilers. In addition, the type of connection (such as Ethernet, Infiniband or Myrinet) used at runtime is determined by the libraries linked-into the program by the appropriate choice of `mpicc` (or `mpif90`, etc.). The choice of the command `mpirun`

to start execution must be consistent with how the executable was compiled.

Simply stated, on the cluster tezpur the command
`soft add +mvapich2-0.98-gcc`
will change PATH so the commands such as `mpicc` `mpif90` and `mpirun` will be found in directory `/usr/local/packages/mvapich2-0.98-gcc/bin/` ;
which is for using the MVAPICH2 distribution for Infiniband, compiled with gcc. (Also `LD_LIBRARY_PATH` is changed.) Whereas the command
`soft add +openmpi-1.2.4-intel10.1`
will cause executables in `/usr/local/packages/openmpi-1.2.4-intel10.1/bin/`
and libraries in `/usr/local/packages/openmpi-1.2.4-intel10.1/lib/`
to be used, which uses the OpenMPI implementation of MPI compiled with an Intel compiler. (By the way, OpenMPI is available for both Ethernet and Infiniband, looking at the contents of `openmpi-1.2.4-intel10.1/lib/` this version seems to be compiled for Infiniband.)

A very useful guide to using the HPC clusters can be found in
http://www.hpc.lsu.edu/help/linuxguide.php

A simple command for running MPI is
`mpirun -np` *number-of-processes* `-machinefile` *hosts.txt /absolute/path/myprog.exe* But the above is just a generic example. Specific examples appropriate for our batch queues are provided in the Exercises directory.

When using the batch queue, the scheduler creates the list of hosts. The command line (and a preparatory line to calculate NPROCS) in the batch script is shown below for MPI-1.

```
export NPROCS=‘wc -l $PBS_NODEFILE |gawk ’//{print $1}’‘
mpirun -machinefile $PBS_NODEFILE -np $NPROCS $WORK_DIR/test
```

For MPI-2 the scripts are slightly different than MPI-1 and depends on the version of MPI-2.

By the way, a useful routine provided by MPI is

```
double MPI_Wtime(void)
```

where `MPI_WTIME` returns a floating-point number of seconds, representing elapsed wallclock time since some time in the past. The “time in the past” is guaranteed not to change during the life of the process.

Two HPC clusters that can be used for exercises are

tezpur.hpc.lsu.edu

A 15.322 TFlops Peak Performance, 360 node, 2 Dual-Core processor Red Hat Enterprise Linux (RHEL) cluster from Dell with 2.66 GHz Intel Xeon 64-bit processors and 4 GB RAM per node.

pelican.hpc.lsu.edu

A 2.797 TFlops Peak Performance, 30 node, 16 processor AIX v5.3 constellation from IBM with 1.9 GHz IBM POWER5+ processors and 32 GB RAM per node.
Other clusters are available for those who have a LONI allocation.

The two clusters mentioned above are for those with only an HPC account, for those with a LONI allocation additional clusters are available.

The IBM cluster uses LoadLeveler for the batch queue. Scripts have not been prepared in this tutorial for the syntax of LoadLeveler.

Make sure you have a dsa key in `/home/$USER/.ssh` If a job does a large amount of I/O then use directory `/scratch/$USER/`

On tezpur in order to compile for MPI,

```
soft add +intel-cc-10.1
soft add +intel-fc-10.1
soft delete +mvapich-0.98-intel10.1
soft add +mvapich-1.01-intel10.1

or in file .soft
+intel-cc-10.1
+intel-fc-10.1
+mvapich-1.01-intel10.1
@default

# followed by the command
resoft
# followed by the command
soft delete +mvapich-0.98-intel10.1
```

Rather than `+mvapich-1.01-intel10.1`, for MPI-2 can choose `+mvapich2-1.02p1-intel10.1` or `+mvapich2-1.2rc2-intel10.1` .

For a parallel job,

- For MPI-1 using MVAPICH, in batch file use **mpirun**.
- For MPI-2 using MVAPICH2 less than version 1.2, use **mpirun** and need to have a file `$HOME/.mpd.conf`
- For MPI-2 using MVAPICH2 version 1.2 or greater, use **mpirun_rsh** and file .mpd.conf is not necessary.

For MVAPICH2 before version 1.2 you will need to create a file `$HOME/.mpd.conf` that no other user can read chmod 600 `$HOME/.mpd.conf` The content of .mpd.conf is one line
`MPD_SECRETWORD=xxx`
where "xxx" is a word that you choose. In the job script, the mpdboot executable is started before the parallel executable starts and the mpd daemon is stopped when the parallel job finishes using mpdallexit.

Various cases for one particular program have been prepared in the directory Exercises/PI3. Further details can be found in `Exercises/PI3/00README_pi3`. For MVAPICH2 before version 1.2, starting and stopping the mpd daemon can be found in the scripts
`pi3c-mpi2-v1.02p1.sh, pi3f90-mpi2-v1.02p1.sh` and `pi3f-mpi2-v1.02p1.sh`. The program in PI3 is very simple, the point is to demonstrate how to run programs on LSU and LONI systems. Simple programs for every feature of MPI can be found in
`/usr/local/packages/sources/mvapich2-1.0.2/test/mpi/`
and examples can be found in the directories
`MPI_Tutorial/UsingMPI` and `MPI_Tutorial/UsingMPI2`

For MPI-2 one example can be found in `MPI_Tutorial/Exercises/Jacobi/` with a write-up
`MPI_Tutorial/Exercises/Jacobi/Poisson.pdf`
The `Makefile` and the batch scripts only use mpi2-v1.2rc2.

References

[1] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[2] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, Cambridge, MA, 1999.

[3] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions.* MIT Press, Cambridge, MA, USA, 1998.

[4] Peter S. Pacheco. *Parallel Programming With MPI.* Morgan Kaufmann Publishers, 1998.

[5] MPI Forum. *MPI-2: Extensions to the Message-Passing Interface Version 2.0.* University of Tennessee, Knoxville, TN, USA, 1997.