CS 495, Spring 2002
MPI: A Short Introduction

This document is meant to be a short introduction to MPI, the Message Passing Interface. MPI is a library of C functions, data structures and macros that enable you to write portable and efficient message-passing programs with ease.

1 Learning MPI

This document is only meant to get you started on compiling and running your MPI programs on the NCSA Origin machines. To learn more about MPI, you should take the following online course:

Intoduction to MPI
http://webct.ncsa.uiuc.edu:8900/public/MPI/

The course is free; register online at the above URL, with any easy-to-remember login and password (these need not be the same as your login and password for the NCSA machines). Chapters 2-8 cover everything you will need for this assignment. You can log into your account any number of times, so the course material can also be used as an online reference. The material in this document has been derived from the material in the above course.

2 Compiling and running MPI programs

During compilation, MPI programs have to be linked with the libmpi.so library, as follows:

```
% gcc my_prog.c -lmpi
```

The program can be run using the `mpirun` utility:

```
% mpirun -np <no. of processors> <binary name> <command line input/options>
```

This runs the specified program on the desired number of processors. `mpirun` also lets you do other more sophisticated things, such as running different binaries on different processors which can communicate with each other. See the `mpirun` manpage for more details.

To submit MPI jobs to the `LSF` batch queueing system, do the following:

```
% bsub -n<no. of processors>  mpirun -np <no. of processors> <binary name>
  <command line input/options>
```

As before, the binary should be in your home directory on modi4.

2.1 Timing your MPI programs

In order to obtain speedup and timing information, you should use the `MPI_Wtime()` function, which returns a `double` containing the time in seconds since some reference point. You can use the difference in the values returned by two calls to this function, one at the beginning of your program and one at the end, to obtain total execution time. The resolution of the timer is 1 microsecond.

3 MPI basics

MPI provides a simple set of routines that let a number of processes running on different processors and/or machines communicate with one-another. As the name suggests, the MPI specification is just an *Interface*; different vendors are allowed to write different implementations that are optimized for specific machine architectures. This allows programs to be portable across different machines while still allowing for good performance. On the NCSA Origin 2000s, we will use the SGI-supplied MPI implementation.

MPI allows for the following types of communications:

1. Point-to-point: between 2 processors at a time, or one-one
2. Collective: one-many, many-one or many-many

We will deal with each of these in later sections.

4 MPI program structure

All MPI programs should include the header file `mpi.h`. All MPI functions, variables and data structures have the prefix "MPI_", and return an error code on completion. The returned value is MPI_SUCCESS for successful completion.

4.1 Starting and terminating an MPI program

Before any other MPI routines are called, the program should execute a call to `MPI_Init`:

```
MPI_Init(&argc, &argv);
```

Similarly, just before termination, all programs should execute `MPI_Finalize()`. This has to be done by all MPI processes; otherwise the program would appear to hang.

4.2 Types in MPI

Most MPI communication routines have an argument to specify what type of data is being communicated. This argument is of type MPI_Datatype, whose pre-defined values include correspondences for all of C's standard types, for example, `MPI_CHAR` for a `char`, `MPI_INT` for an `int` and so on. In addition, you can create your own custom MPI types for transferring data of many different types or from non-contiguous memory locations in one message (see Chapters 3.6 and 5 of the online course).

4.3 Communicators

Communication in MPI is based on the notion of a *communicator*: a group of processes that can communicate amongst themselves. Each process has a *rank* for every communicator it is a part of (note that a process can be part of many different communicators); a combination of a communicator and a rank uniquely identify a processor. MPI supplies a pre-built communicator: MPI_COMM_WORLD, that contains all the processes belonging to this program. You can also create your own communicators for more restricted communications (see Chapters 3.11, 7 and 8 of the online course).

5 Point to point communication

(see Chapter 4 of the online course for more details)

This type of communication allows two processes to communicate with one another. One process sends data using the `MPI_Send` call, which the other can receive by executing an `MPI_Recv`. The message contains sender identification, recepient, the communicator that they belong to (the "sender" and "recipient" are actually just ranks within this communicator), a tag (helps identify the data and has meaning to the application) the data itself, the type of the data (one of MPI's pre-built types, or a custom datatype), and the number of data elements of this type present in the data.

Note that the recepient can get the data only by executing an `MPI_Recv`: the operation terminates immediately if there is a message present to match the criteria specified by the recipient; otherwise, the termination is delayed till appropriate data is available. The recipient can specify which message (of potentially many that may be pending) it wants to receive by specifying a communicator, a sender, and a value for the tag. The operation completes if (or when) a message matching these parameters is available. Wildcards can be used for the communicator and/or the sender.

All sends and receives have a notion of *completion*: when the operation has completed, it is safe to read or write the memory whose contents were being sent or into which the incoming data was being stored. This notion of completion is not the same as function call return, as we shall see.

Sends have a number of different *modes*:

1. Synchronous: the send operation is complete when the receiver has acknowledged it. It results in the sender and receiver getting synchronized.
2. Buffered: The data is copied into an MPI internal buffer. The operation completes when this copying is done. There is no guarantee that the sender has or has not received the message.
3. Ready: requires that the recepient has already posted a matching receive; otherwise, the result is undefined.
4. Standard: this is the most general-purpose send mode, and the implementor of the MPI library is free to chose the implementation. Depending on available resources and message size, it may behave as synchronous mode or buffered mode.

The receive (thankfully) has just one mode: the operation completes if there is an available message that matches the criteria; if not, completion is delayed till such a message is available.

To complicate things further, each of the send and receive calls have both *blocking* and *non-blocking* versions. As was mentioned earlier, the notion of an operation completing is independent of when the function call returns. If the call is blocking, it returns only when the operation completes. If the call is non-blocking, it returns immediately: the corresponding communication operation and any other computation can then proceed in parallel. MPI provides the functions `MPI_Wait` and `MPI_Test` to check when a non-blocking communication operation has completed. Note that it is unsafe to use the memory buffer that contains the departing or arriving data till the operation completes; if a non-blocking call is used, the termination of the operation should always be checked before this memory is touched.

6 Collective communications

(See Chapter 6 of the online course)

MPI's collective communication primitives allow one-many, many-one and many-many communication. The following types of communication are allowed:

1. Barriers (`MPI_Barrier`): no actual data is transferred; all calling processes are blocked until all the processes in the specified communicator have called the `MPI_Barrier` routine.
2. Broadcast (`MPI_Bcast`): allows one process to send the data to all the processes in the specified communicator; both the sender and receivers call `MPI_Bcast`.
3. Reduction (`MPI_Reduce`): allows data spread across numerous processes to be *reduced* to a single process' memory. Reduction involves application of a built-in reduction operation such as `MPI_MAX` (find the maximum of all the data and pass it to the destination), `MPI_SUM` etc., or some user-specified reduction operation.
4. Scatter (`MPI_Scatter`): break up a contiguous piece of data in the sender's memory into numerous small chunks, and pass each chunk to a differerent process in the communicator.
5. Gather (`MPI_Gather`): the opposite of scatter; collect data from all the processes and store in a contiguous chunk in the target.

7 Stuff you will need for this assignment

Of all the different communication modes and paradigms described above, the following is the syntax and semantics of the ones you are likely to find useful. If you find that you need something not described here, look it up in the NCSA course material.

- `int MPI_Init(int *argc, char ***argv)`

argc	in	address of the argc argument to `main`
argv	in	address of the argv argument to `main`
returns		error code

  This function has to be called before any other MPI routine is called.

- `int MPI_Finalize()`

returns	error code

  Cleanup routine; all processes must call it before the application can terminate.

- `double MPI_Wtime()`

returns	time in seconds since some reference point

- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`

comm	in	communicator in which this process's rank is sought
rank	out	rank of this process
returns		error code

  This allows a process to know it's rank; this can be used, for example, to distinguish sender and recipients.

- `MPI_Comm_size(MPI_Comm comm, int *size)`

comm	in	communicator whose size is sought
size	out	size of this communicator
returns		error code

  This function can be used to obtain the number of processes in a given communicator.

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

buf	in	initial address of send buffer
count	in	number of elements in send buffer
dtype	in	datatype of each element
dest	in	rank of the destination process
tag	in	message tag or identifier
comm	in	communicator which the sender and receiver are a part of
returns		error code

  This is the blocking, standard-mode send. Since you will only be sending small pieces of data in this assignment, this should be sufficient.

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

buf	out	initial address of receive buffer
count	in	maximum number of elements in receive buffer
datatype	in	datatype of each receive buffer element
source	in	rank of source
tag	in	message tag
comm	in	communicator
status	out	status object
returns		error code

  Blocking receive. The status variable has information about the identity of the sender, tag etc. if wildcards are used.

- `int MPI_Barrier (MPI_Comm comm)`

comm	in	communicator
returns		error code

  Blocks until all the processes in the communicator have calles it.

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

buffer	in/out	starting address of buffer containing sent/received data
count	in	number of entries in buffer
datatype	in	data type of buffer
root	in	rank of broadcast root
comm	in	communicator
returns		error code

  The data in the buffer is sent to all processes in the communicator.