PHYS:5905 Handout: Message Passing Interface (MPI)

The Message Passing Interface (MPI) libraries are the most widely used choice for parallel programming. Today we will learn some of the very basics about MPI and get to do some hands on practice with writing parallel code ourselves.

1 General Comments

1. The biggest hindrance to parallel computing for most people is just getting started. Once you get your first parallel code to work (before lunch today), you'll see that it's not really all that hard.

2. I will *not* cover all of the wide functionality of the MPI libraries. I will merely discuss how to implement basic point-to-point and collective communications. The many topics that I will not cover include:

   (a) Non-blocking and synchronous vs. asynchronous communication

   (b) Derived data types

   (c) Groups and communicators

   (d) Virtual topologies

   (e) Extended functions of the MPI-2 Standard

3. You can do almost anything you want to do with just 8 commands that handle environment management and point-to-point communications. Another 4 commands for collective communications are also very useful. Probably 95% of MPI users can get away with just these 12 commands, so I will focus on these commands here, and briefly mention a few others.

4. I will attempt to give the correct syntax for both C and Fortran

2 Concepts

During this presentation, we will learn about the following concepts. The most efficient way to present these concepts is to define and discuss them as they arise in the MPI implementation, so I only list them here for reference:

1. Communicators

2. Point-to-point vs. collective communications

3. Buffering of messages

4. Issues of Synchronization and Determinism

   (a) Deadlocks and Race Conditions

3 Environment Management Routines

This section describes the basic environment management routines that allow the initialization, management, and termination of parallel tasks.

1. First, every MPI program needs to include a header file at the beginning of the program:

C	`#include "mpi.h"`
Fortran	`include 'mpif.h'`

This file includes the definition of a number of MPI variables, all of which can be distinguished by starting with `MPI_` and begin in all capitals.

2. The general format of MPI calls differs slightly in C and Fortran, so here is an example of each:

C	`rc = MPI_Bsend(&buf,count,type,dest,tag,comm)`
Fortran	`CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)`

The error code of the call is returned in `rc` for C and `ierr` for Fortran. If it has a value equal to the variable `MPI_SUCCESS`, then the call was successful. We'll discuss the arguments of each call as we present them below. Because the error code is always returned for all MPI calls, it is left out of the argument lists below.

3. Initialize MPI Environment, starting parallel tasks:
`MPI_Init` call must come before any of the MPI calls in the code for each of the MPI tasks.

C	`MPI_Init (&argc,&argv)`
Fortran	`MPI_INIT (ierr)`

Here `argc` and `argv` are required only in C, where they are the main program's arguments.

4. Terminate MPI Environment, cleaning up and closing parallel tasks:
`MPI_Finalize` must be the last MPI call for every MPI program.

C	`MPI_Finalize ()`
Fortran	`MPI_FINALIZE (ierr)`

5. Note that we are only going to deal with the SPMD model, where each process executes the same program. This does not mean that each process executes exactly the same instruction at each step. Conditional logic can—and, in general, must—be used so that each process executes a different series of instructions.

6. Determine Number of MPI tasks:
`MPI_Comm_size` determines the number of MPI tasks associated with the communicator that is passed in as an argument.

C	`MPI_Comm_size (comm,&size)`
Fortran	`MPI_COMM_SIZE (comm,size,ierr)`

Arguments:

Intent	Argument	Type	Description
IN	`comm`	handle	Communicator
OUT	`size`	integer	Number of MPI tasks associated with `comm`

The first argument `comm` is a variable called the *communicator*. The communicator defines a group of processes. The standard pre-defined communicator that is most commonly used (and the only one we will use in this class, is `MPI_COMM_WORLD`, and it includes all of the MPI tasks for your run.

7. Determine the Rank of this MPI task:
`MPI_Comm_rank` determines the rank of this MPI task. Also referred to as the task ID.

C	`MPI_Comm_rank (comm,&rank)`
Fortran	`MPI_COMM_RANK (comm,rank,ierr)`

Arguments:

Intent	Argument	Type	Description
IN	`comm`	handle	Communicator
OUT	`rank`	integer	Rank, or task ID of this MPI task.

For a `size` of N tasks, the ranks, or task IDs, of each of the processes run from 0 to $N-1$.

8. EXAMPLE: SERIAL HELLO WORLD PROGRAM:

```
!------------------------------------------------------------------------------
!  HELLO WORLD
!------------------------------------------------------------------------------
program helloworld_serial
implicit none

!Write out message to screen
write(*,'(a)')'Hello World.'

end program helloworld_serial
```

9. EXAMPLE: PARALLEL HELLO WORLD PROGRAM:
Here we must include the MPI header file `mpif.h`, add declarations for the variables `nproc`, `iproc`, and `ierror`, and make the calls to `mpi_init`, `call mpi_comm_size`,`mpi_comm_rank`,and `mpi_finalize`.

```
!------------------------------------------------------------------------------
!  HELLO WORLD
!------------------------------------------------------------------------------
program helloworld
implicit none
include 'mpif.h'
integer ::  nproc !Number of Processors (1 to nproc)
integer ::  iproc !Number of local processors (0 to nproc-1)
integer ::  ierror !Integer error flag

!Initialize MPI message passing-------------------------------------------
call mpi_init (ierror)
call mpi_comm_size (mpi_comm_world, nproc, ierror)
call mpi_comm_rank (mpi_comm_world, iproc, ierror)

!Write out message to screen
write(*,'(a,i4,a,i4)')'Hello World.  I am processor ',iproc,' of ',nproc

!Finalize MPI message passing
call mpi_finalize (ierror)

end program helloworld
```

10. When the parallel `helloworld` is run on 16 processors using the command
`srun -p scx-event -n 16 ./helloworld.e`, the output appears like this:

```
Hello World.  I am processor 11 of 16
Hello World.  I am processor 5 of 16
Hello World.  I am processor 9 of 16
Hello World.  I am processor 15 of 16
Hello World.  I am processor 8 of 16
Hello World.  I am processor 10 of 16
Hello World.  I am processor 12 of 16
Hello World.  I am processor 14 of 16
```

```
Hello World.  I am processor 6 of 16
Hello World.  I am processor 7 of 16
Hello World.  I am processor 13 of 16
Hello World.  I am processor 0 of 16
Hello World.  I am processor 1 of 16
Hello World.  I am processor 3 of 16
Hello World.  I am processor 4 of 16
Hello World.  I am processor 2 of 16
```

This raises the issue of determinism, a subtle but critically important issue in parallel programming. Why does the output occur in an apparently random order, rather than in rank order from 0 to 15? How can we modify the code to produce output in rank order? We will be tackle this issue as part of In-Class Exercises #1 this afternoon.

4 Point-to-Point Communication Routines

Point-to-point communication involve the sending of a message from one task (A) and the receiving of that message by one task (B). This will involve a matched pair of commands, one send and one receive.

1. Send a message to another MPI task:
   `MPI_Send` initiates a blocking send operation, returning from the call only when the sending task is free for re-use (this does not *necessarily* mean that the message has been received—see the discussion on message buffering below).

C	`MPI_Send (&buf,count,datatype,dest,tag,comm)`
Fortran	`MPI_SEND (buf,count,datatype,dest,tag,comm,ierr)`

   Arguments:

Intent	Argument	Type	Description
IN	`buf`	choice	Address of data array to send
IN	`count`	integer(≥ 0)	Number of array elements to send
IN	`datatype`	handle	Type of data to send
IN	`dest`	integer	Rank (task ID) of destination task
IN	`tag`	integer	Message tag
IN	`comm`	handle	Communicator

   The argument `buf` supplies the address of the variable array to be sent. In Fortran this is just the variable name; in C, it is passed by reference and should be the variable name prepended by '&'. The `datatype` is the type of variable array, specified by a pre-defined handle, such as `MPI_REAL`, `MPI_INTEGER`, `MPI_CHARACTER`, or `MPI_COMPLEX` in Fortran. These handles are defined in the include file `mpi.h` or `mpif.h`, and a brief reference of the most common datatypes for C and Fortran can be found at `https://computing.llnl.gov/tutorials/mpi/`. The argument `tag` is a unique identifier to identify the message that can be assigned by the programmer. Good programming practices do not depend on this `tag` unless unavoidable.

2. Receive a message from another MPI task:
   `MPI_Recv` receives a message from another MPI task, blocking until the requested data has been received and is available.

C	`MPI_Recv (&buf,count,datatype,source,tag,comm,&status)`
Fortran	`MPI_RECV (buf,count,datatype,source,tag,comm,status,ierr)`

   Arguments:

Intent	Argument	Type	Description
OUT	`buf`	choice	Address of data array to receive
IN	`count`	integer(≥ 0)	Number of array elements to receive
IN	`datatype`	handle	Type of data to recv
IN	`source`	integer	Rank (task ID) of source task
IN	`tag`	integer	Message tag
IN	`comm`	handle	Communicator
OUT	`status`	status	Status object

The argument `status` can be used subsequently to inquire about the size, tag, and source of the received message.

3. Note that blocking communication is not necessarily synchronous. However, this is not specified in the MPI standard and varies depending on the implementation. A blocking send *can* be asychronous. In other words, the send can be completed, allowing the sending task move on to other work, before the message is received by the receiving task. In this case, the receiving process may have a buffer for incoming messages, that hold received messages, waiting for the receiving task to request the message, but allowing the sending task to move ahead (asynchronously) to other work.

4. Determinism:
   Message passing using point-to-point communications in MPI should be deterministic—meaning that sends and receives should always occur in a specified order—otherwise you can get into trouble. The following example demonstrates some of the subtleties associated with programming deterministic communications.

   Can you identify the potential problem in the following code?

```
!------------------------------------------------------------------------------
!  Example of Potential Deadlock in point-to-point communications
!------------------------------------------------------------------------------
program deadlock
implicit none
include 'mpif.h'
integer ::  nproc !Number of Processors (1 to nproc)
integer ::  iproc !Task ID of local processor (0 to nproc-1)
integer ::  ierror !Integer error flag
integer ::  sproc, rproc !Processor IDs to Send to and Receive from
real ::  sx,rx !Variable to send and receive
integer ::  tag=0 !message tag
integer, dimension (MPI_STATUS_SIZE) ::  status !Receive status variable

!Initialize MPI message passing-----------------------------------------------
call mpi_init (ierror)
call mpi_comm_size (mpi_comm_world, nproc, ierror)
call mpi_comm_rank (mpi_comm_world, iproc, ierror)

!Set sx equal to iproc
sx=real(iproc)
rx=-1.

!Ring communications to local neighbor:  Send message to sproc=iproc+1
!  Receive message from rproc=iproc-1
!NOTE: mod function enables periodic wrap-around
sproc=mod(iproc+1,nproc)
rproc=mod(iproc+nproc-1,nproc)

!Send Message to sproc and receive message from rproc
call mpi_send(sx,1,MPI_REAL,sproc,tag,MPI_COMM_WORLD,ierror)
call mpi_recv(rx,1,MPI_REAL,rproc,tag,MPI_COMM_WORLD,status,ierror)

!Write out message of send and received values
write(*,'(a,i4,a,f6.3,a,f6.3)')'I am processor ',iproc,' Sent:  ',sx,' Received:  ',rx

!Finalize MPI message passing
call mpi_finalize (ierror)

end program deadlock
```

The problem with the program `deadlock` can be demonstrated simply by considering a run with just two MPI tasks, `nproc=2`. In this case, the variables `sproc` and `rproc` have the following values for ranks (task IDs) `iproc=0` and `iproc=1`:

	proc=0	iproc=1
sproc	1	0
rproc	1	0

Therefore, the MPI send and receive commands on `iproc=0` are

```
call mpi_send(sx,1,MPI_REAL,1,tag,MPI_COMM_WORLD,ierror)
call mpi_recv(rx,1,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierror)
```

and the MPI send and receive commands on `iproc=1` are

```
call mpi_send(sx,1,MPI_REAL,0,tag,MPI_COMM_WORLD,ierror)
call mpi_recv(rx,1,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierror)
```

The potential error here is that both tasks execute the instruction to send a message to the other task before executing the instruction to receive the message. *If* the send instruction on each task waits for the message to be received before moving on (this would be both a blocking and synchronous communication), the both tasks will hang up on the MPI send command and never move forward to the MPI receive command.

In practice, this non-deterministic coding will usually work and give you the result that you intended. The reason is that the implementation of `MPI_SEND` in most MPI libraries will complete the send instruction (blocking until any subsequent send operations will not interfere with the current send operation) and then allow the task to move on to the next instruction (without waiting for the message to be received, meaning that the communication is asynchronous). Therefore, each task moves on to the receive instruction to successfully receive the message (possibly waiting in its message buffer) that was just by the other task. However, the MPI and MPI-2 standards do not specify that this must be the action of the send instruction. It is possible for an implementation to require the message to be received (forcing synchronous communication) before completing the send instructions, in which case the code will hang up. This problem is generally called a *deadlock*, and is also referred to as a *race condition* (code execution depends on the order of operations, which is not deterministic, leading to success when one task reaches in instruction in the code first and failure when another task gets there first).

In addition to varying implementations of `MPI_SEND`, if the message to be sent is particularly long, then blocking by the send instruction may continue until the message is received, and the code will hang up.

The overall lesson here is that, although most of the time the code in program `deadlock` will execute as intended without a hitch, the communication sequence is not deterministic and potential for a crash exists. This leads to the possibility that the code will perform satisfactorily in 999 out of 1000 cases, so trying to find the bug causing the problem in the 1000th case becomes very difficult. The root cause is the non-deterministic communication, but many new to the practice of parallel programming may be unaware of the problem because, in practice, the code works nearly all of the time.

Beginning on the next page is a potential solution to this problem of non-deterministic communications.

```
!------------------------------------------------------------------------------
!  Example of Fixing Deadlock in point-to-point communications
!------------------------------------------------------------------------------
program deadlock_fix
implicit none
include 'mpif.h'
integer ::  nproc !Number of Processors (1 to nproc)
integer ::  iproc !Task ID of local processor (0 to nproc-1)
integer ::  ierror !Integer error flag
integer ::  sproc, rproc !Processor IDs to Send to and Receive from
real ::  sx,rx !Variable to send and receive
integer ::  tag=0 !message tag
integer, dimension (MPI_STATUS_SIZE) ::  status !Receive status variable

!Initialize MPI message passing-----------------------------------------------
call mpi_init (ierror)
call mpi_comm_size (mpi_comm_world, nproc, ierror)
call mpi_comm_rank (mpi_comm_world, iproc, ierror)

!Check for possible error:  Algorithm fails if nproc is odd
if (mod(nproc,2) .ne.  0) then
if (iproc .eq.  0) write(*,'(a)')'ERROR: Algorithm requires even value of nproc'
!Finalize MPI message passing
call mpi_finalize (ierror)
stop
endif

!Set sx equal to iproc
sx=real(iproc)
rx=-1.

!Ring communications to local neighbor:  Send message to sproc=iproc+1
!  Receive message from rproc=iproc-1
!NOTE: mod function enables periodic wrap-around
sproc=mod(iproc+1,nproc)
rproc=mod(iproc+nproc-1,nproc)

!Send Message to sproc and receive message from rproc
!NOTE: Avoid deadlock by having even processor always send first
if (mod(iproc,2) .eq.  0) then
call mpi_send(sx,1,MPI_REAL,sproc,tag,MPI_COMM_WORLD,ierror)
call mpi_recv(rx,1,MPI_REAL,rproc,tag,MPI_COMM_WORLD,status,ierror)
else
call mpi_recv(rx,1,MPI_REAL,rproc,tag,MPI_COMM_WORLD,status,ierror)
call mpi_send(sx,1,MPI_REAL,sproc,tag,MPI_COMM_WORLD,ierror)
endif

!Write out message of send and received values
write(*,'(a,i4,a,f6.3,a,f6.3)')'I am processor ',iproc,' Sent:  ',sx,' Received:  ',rx
```

```
!Finalize MPI message passing
call mpi_finalize (ierror)

end program deadlock_fix
```

The program `deadlock_fix` handles fix the potential problem of non-deterministic communications in program `deadlock` by specify unique which task sends first and which receives first in point-to-point communications between two tasks. This is done choosing that the processor and and even value of the rank (task ID) sends first. This coding of the communication will always work, independent of possibly varying implementations of the MPI routines.

Note, however, that the decision to use whether the rank (task ID) is even or odd restricts the validity of this algorithm to parallel runs with only an even number of tasks. This is typical of solutions to ensure deterministic communications, that it may restrict the flexibility of your parallelization scheme. To prevent the code from being run with an odd number of processors (which would again lead to non-deterministic communications, but would probably work most of the time, just as program `deadlock` worked most of the time), good programming practice requires that a test be inserted to prevent this possibility.

5 Collective Communication Routines

Although you could really get away with only using point-to-point send and receive commands to do all necessary communication in a parallel code, there are some very useful collective communication routines the provide optimized methods for one-to-all, all-to-one, and all-to-all communications. For these routines, all tasks associated with the communicator (and for `MPI_COMM_WORLD` this is all MPI tasks) must participate. In addition, these collective communication routines are all blocking.

One of the most useful collective operations is called reduction: this operation takes the data in the same variable on each processor, or array of variables, and performs an operation on all of the variables, for example computing the sum or finding the maximum value. The result is either collected at the root process (for `MPI_Reduce`) or distributed to all processes (for `MPI_Allreduce`). Thus, Reduce is an all-to-one operation, and Allreduce is an all-to-all operation.

1. Broadcast a message to all MPI tasks:
   `MPI_Bast` broadcasts (sends) a message from the process with rank "root" to all other processes.

C	`MPI_Bcast (&buffer,count,datatype,root,comm)`
Fortran	`MPI_BCAST (buffer,count,datatype,root,comm,ierr)`

   Arguments:

Intent	Argument	Type	Description
INOUT	`buffer`	choice	Address of input data array, or output data array at root
IN	`count`	integer(≥ 0)	Number of array elements to receive
IN	`datatype`	handle	Type of data to recv
IN	`root`	integer	Rank (task ID) of root task
IN	`comm`	handle	Communicator

   The argument `root` defines the rank (task ID) of the tasks that sends the broadcast; all other tasks receive the broadcast

2. Perform a reduction operation on data from all MPI tasks and send result to one task:
   `MPI_Reduce` applies a reduction operation on all tasks in the group and places the result in the one task with rank specified by "root".

C	`MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)`
Fortran	`MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)`

   Arguments:

Intent	Argument	Type	Description
IN	`sendbuf`	choice	Address of input data array to send
OUT	`recvbuf`	choice	Address of output data array for result at root
IN	`count`	integer(≥ 0)	Number of array elements to receive
IN	`datatype`	handle	Type of data to recv
IN	`op`	handle	Operation to perform
IN	`root`	integer	Rank (task ID) of root task
IN	`comm`	handle	Communicator

The argument `op` accepts a pre-defined handle that specifies the operation to perform on the reduced data, for example `MPI_MAX`, `MPI_MIN`, or `MPI_SUM`. These handles are defined in the include file `mpi.h` or `mpif.h`, and a brief reference of the most common reduction operations for C and Fortran can be found at `https://computing.llnl.gov/tutorials/mpi/`. The result is sent to the task with the rank (task ID) specified by the argument `root`.

3. Perform a reduction operation on data from all MPI tasks and send result to all tasks:
`MPI_Allreduce` applies a reduction operation on data from all tasks in the group and places the result in all tasks.

C	`MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)`
Fortran	`MPI_ALLREDUCE (sendbuf,recvbuf,count,datatype,op,comm,ierr)`

Arguments:

Intent	Argument	Type	Description
IN	`sendbuf`	choice	Address of input data array to send
OUT	`recvbuf`	choice	Address of output data array for result at root
IN	`count`	integer(≥ 0)	Number of array elements to receive
IN	`datatype`	handle	Type of data to recv
IN	`op`	handle	Operation to perform
IN	`comm`	handle	Communicator

4. Synchronize all MPI tasks:
`MPI_Barrier` creates a barrier synchronization. Each task, when reaching the `MPI_Barrier` call, blocks until all tasks in the group reach the same `MPI_Barrier` call.

C	`MPI_Barrier (comm)`
Fortran	`MPI_BARRIER (comm,ierr)`

5. Note that although the barrier call is useful for ensuring a deterministic flow of the parallel code, it also can reduce the computational efficiency of a code significant by forcing many tasks to sit idle until the last task reaches the barrier. Careful attention to the details of the MPI communication often eliminates the need for barrier calls, avoiding unnecessary idling of tasks. The barrier call can be very useful in the debugging process.

6. In addition to the collective calls above, there are also `MPI_Scatter` and `MPI_Gather` commands that are useful for distributing data across tasks and collecting data on a single task. Note that this functionality is different from braodcast and reduce operations. For example, if you have an array of 16 elements and want to distribute subsections of the array to four different MPI tasks, `MPI_Scatter` will take the input array from the root task and output elements 0–3 on task 0, elements 4–7 on task 1, elements 8–11 on task 2, and elements 12–15 on task 3. Conversely, `MPI_Gather` performs the reverse operation, collecting the array subsections from each task and placing the full array on the root task.

6 Asynchronous Communication

Although the examples in point-to-point communications above has discussed *possible* asynchronous communication (dependent upon the implementation of the MPI routines), one can use commands that will guarantee asynchronous communication.

Here is an example of the benefit of using asynchronous communications. Consider a single timestep in a simulation where data needs to be exchanged between two tasks to complete the computation for that timestep. For most of the computational operations that need to performed by one task, the data is local, and only a small fraction of those operations are dependent upon the data from the other task. Therefore, each task can send a message with the required data to the other task (asynchronously), and then move on to perform all of the computational operations that depend only on local data. It can then poll to see if the data from the other task has been received. If so, it can receive the message and complete the remaining computational operations for that timestep.

The MPI calls to perform asynchronous communications are `MPI_ISEND`, `MPI_IPROBE`, and `MPI_IRECV`.

7 References for MPI

1. Message Passing Interface (MPI), by Blaise Barney
   `https://computing.llnl.gov/tutorials/mpi/`
   An excellent tutorial on the use of MPI, with both Fortran and C example code.

2. Designing and Building Parallel Programs, by Ian Foster
   `http://www.mcs.anl.gov/~itf/dbpp/`

3. Introduction to Parallel Computing textbook. Information on this book (but not the book) can be found at
   `http://www-users.cs.umn.edu/~karypis/parbook/`

4. Message Passing Interface Forum,
   `http://www.mpi-forum.org/`
   This site has links to various version of the MPI-1 and MPI-2 Standard documents.