Introduction to MPI

John Urbanic

Parallel Computing Scientist
Pittsburgh Supercomputing Center

Distinguished Service Professor
Carnegie Mellon University

Copyright 2026

Pre-Introduction: Why Use MPI?

- Dominant
- Has been around a long time (30+ years)
- Will be around a long time (on all new platforms/roadmaps)
- Lots of libraries
- Lots of algorithms
- Very scalable (10,000,000+ cores right now)
- Very portable
- Works with hybrid models
- Explicit parallel routines force the programmer to address parallelization from the beginning, not as an afterthought. This *can* enable both debugging and memory placement optimization.

- Therefore:
  - A good long term learning investment
  - Useful and possible to understand whether you are coder or a user

Introduction

What is MPI? The Message-Passing Interface Standard(MPI) is a ***library*** that allows you to do problems in parallel using ***message-passing*** to ***communicate*** between ***processes***.

•Library
It is not a language (like X10 or UPC), or even an extension to a language. Instead, it is a library that your native, standard, serial compiler (f77, f90, cc, CC, python, etc.) uses.

•Message Passing
Message passing is sometimes referred “paradigm”. But it is really just any method of explicitly passing data between processes and it is flexible enough to implement most other paradigms (Data Parallel, Work Sharing, etc.). It can be defined as having independent processors, with no shared data, communicate through subroutine calls.

•Communicate
This communication may be via a dedicated MPP dragonfly network, or merely an office LAN. To the MPI programmer, it looks much the same.

•Processes
These can be 1,000,000 cores on Frontier, or 40 processes on your laptop. Think “Linux process” and you won’t be far off, although we usually want only 1 process per core so that we actually get a speed up.

Basic MPI

In order to do parallel programming, you require some basic functionality, namely, the ability to:

- Start Processes
- Send Messages
- Receive Messages
- Synchronize

With these four capabilities, you can construct any program. We will look at the basic versions of the MPI routines that implement this capability. MPI-5 offers over 450 functions. These are largely just more convenient and efficient for certain tasks. However, with the handful that we are about to learn, we will be able to implement just about any algorithm, and you will be well-prepared for when we cover much of the remaining routines in the Advanced talk.

First Example (Starting Processes): Hello World

The easiest way to see exactly how a parallel code is put together and run is to write the classic "Hello World" program in parallel. In this case it simply means that every PE will say hello to us. Something like this:

```
mpirun –n 8 a.out
Hello from 0.
Hello from 1.
Hello from 2.
Hello from 3.
Hello from 4.
Hello from 5.
Hello from 6.
Hello from 7.
```

Hello World: C Code

How complicated is the code to do this? Not very:

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

Hello World: Fortran Code

Here is the Fortran version:

```
program hello
include 'mpif.h'

integer my_pe_num, errcode
call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)
print *, 'Hello from ', my_pe_num,'.'
call MPI_FINALIZE(errcode)
end program hello
```

We will make an effort to present both languages here, but they are really trivially similar in these simple examples, so try to play along on both.

MPI Routine Guidelines

Let's make a few general observations before we go into what is actually happening here:

- We have to include the header file, either mpif.h or mpi.h. MPI-3+ can use the (much better) "USE mpi_f08" for Fortran.
- The MPI calls are easy to spot, they always start with MPI_. The MPI calls themselves are the same for both languages except that the Fortran routines have an added argument on the end to return the error condition (optional in MPI-3), whereas the C ones return it as the function value.
- We should check these (for MPI_SUCCESS) in both cases as it can be very useful for debugging. We don't in these examples for clarity. You probably won't because of laziness.

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

```fortran
   program hello
   include 'mpif.h'

   integer my_pe_num, errcode
   call MPI_INIT(errcode)
   call MPI_COMM_RANK(MPI_COMM_WORLD,
                      my_pe_num, errcode)
   print *, 'Hello from ', my_pe_num,'.'
   call MPI_FINALIZE(errcode)
   end
```

MPI_INIT, MPI_FINALIZE and MPI_COMM_RANK

OK, lets look at the actual MPI routines. All three of the ones we have here are very basic and will appear in any MPI code.

MPI_INIT

This routine must be the first MPI routine you call (it does *not* have to be the first statement). It sets things up and might do a lot of behind-the-scenes work on some cluster-type systems (like start daemons and such). On most dedicated MPPs, it won't do much. We just have to have it. In C, it is common to to pass along the command line arguments. These are very standard C variables that contain anything entered on the command line when the executable was run. You may have used them before in normal serial codes. You can usually use NULL for both of these arguments, but we will stick with the normal convention.

MPI_FINALIZE

This is the companion to MPI_Init. It must be the last MPI_Call. It may do a lot of housekeeping, or it may not. Your code won't know or care.

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

MPI_COMM_RANK

Now we get a little more interesting. This routine returns to every PE its rank, or unique address from 0 to PEs-1. This is the only thing that sets each PE apart from its companions. In this case, the number is merely used to have each PE print a slightly different message. In general, though, the PE number will be used to load different data files or take different branches in the code. It has another argument, the communicator, that we will ignore for a few slides.

What's Happening?

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

```
Hello from 5.
Hello from 3.
Hello from 1.
Hello from 2.
Hello from 7.
Hello from 0.
Hello from 6.
Hello from 4.
```

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

Keep This Picture

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

main(int argc, char** argv){
    int my_PE_num;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
    printf("Hello from %d.\n", my_PE_num);
    MPI_Finalize();
}
```

```
mpirun -n 8 a.out
```

A Few Interesting Details

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

main(int argc, char** argv){
   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

```
Hello from 5.
Hello from 3.
Hello from 1.
Hello from 2.
Hello from 7.
Hello from 0.
Hello from 6.
Hello from 4.
```

There are two behaviors here that may not have been expected. The most obvious is that the output might seems out of order. The response to that is "what order were you expecting?" Remember, the code was started on all nodes practically simultaneously. There was no reason to expect one node to finish before another. Indeed, if we rerun the code we will probably get a different order. Sometimes it may seem that there is a very repeatable order. But, one important rule of parallel computing is don't assume that there is any particular order to events unless there is something to guarantee it. Later on we will see how we could force a particular order on this output.

The second question you might ask is “how does the output know where to go?” A good question. In the case of a cluster, it isn’t at all clear that a bunch of separate Linix boxes printing to standard out will somehow combine them all on one terminal. Indeed, you should appreciate that a dedicated MPP environment will automatically do this for you – even so you should expect a lot of buffering (hint: use flush if you must). Of course most “serious” IO is file-based and will depend upon a distributed file system (you hope).

Do all nodes really run the *same* code?

Yes, they do run the same code independently. You might think this is a serious constraint on getting each PE to do unique work. Not at all. They can use their PE numbers to diverge in behavior as much as they like.

The extreme case of this is to have different PEs execute entirely different sections of code based upon their PE number.

```
if (my_PE_num = 0)
          Routine_SpaceInvaders
else if (my_PE_num = 1)
          Routine_CrackPasswords
else if (my_PE_num =2)
          Routine_WeatherForecast
                    .
                    .
                    .
```

So, we can see that even though we have a logical limitation of having each PE execute the same program, for all practical purposes we can really have each PE running an entirely unrelated program by bundling them all into one executable and then calling them as separate routines based upon PE number.

Manager and Worker PEs

The much more common case is to have a single PE that is used for some sort of coordination purpose, and the other PEs run code that is the same, although the data will be different. This is how one would implement a manager/worker paradigm.

```
if (my_PE_num = 0)
        ManagerCodeRoutine
else
        WorkerCodeRoutine
```

Of course, the above Hello World code is the trivial case of

```
EveryBodyRunThisRoutine
```

and consequently the only difference will be in the output, as it at least uses the PE number.

An Analogy To Remember

- Think of an MPI program as a book our group is reading. We might all start at the same time, but I wouldn't expect us to all be on the same line at any point.
- Furthermore, some of our books are "choose your own adventures". We might each be taking a different path.
- So, it doesn't make sense to ask what line an MPI program is on.

- I might also ask each of you to rate each chapter. These are analogous to variables; you each have your own copy.
- So it doesn't make sense to ask what *the* value of any variable is. There are multiple copies.

Communicators

Our last little detail in Hello World is the first parameter in

```
MPI_Comm_rank (MPI_COMM_WORLD, &my_PE_num)
```

This parameter is known as the "communicator" and can be found in many of the MPI routines. In general, it is used so that one can divide up the PEs into subsets for various algorithmic purposes. For example, if we had an array - distributed across the PEs – for which we wished to find the determinant, we could define some subset of the PEs that holds a certain column of the array so that we could address only that column conveniently. Or, we might wish to define a communicator for just the odd PEs. Or just the “ocean” PEs in a weather model. You get the idea.

However, this is a convenience that can be dispensed with in simpler algorithms. As such, one will often see the value MPI_COMM_WORLD when a communicator is required. This is simply the global set and states we don't really care to deal with any particular subset here. We will use it in all of our examples. We will delve into its considerable usefulness in the Advanced talk.

Compiling

Before we move on, let's see how we compile and run this thing - just so you don't think we are skipping any magic. We compile using a normal ANSI C or Fortran 90 compiler (many other languages are also available).

Most MPI configurations provide a convenient wrapper to spare us finding the required libraries. Usually called mpicc or mpif90*. Don't let that distract you from the fact that we are using absolutely standard serial compilers.

For our C codes:

```
mpicc hello.c
```

For our Fortran codes:

```
mpif90 hello.f
```

We now have an executable called `a.out` (the default).

Running

To run an MPI executable we must tell the machine how many copies we wish to run at runtime. On Bridges you can choose any number up to the size of your job request. We'll try 8. The command is `mpirun`:

```
mpirun –n 8 a.out
Hello from 5.
Hello from 3.
Hello from 1.
Hello from 2.
Hello from 7.
Hello from 0.
Hello from 6.
Hello from 4.
```

Which is (almost) what we desired when we started.

It really does just copy and run.

If you start to think there is more to mpirun than starting multiple copies of an executable, try this simple experiment.

Basic serial "hello world":

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("Hello\n");
}
```

Compile with cc:

```
cc hello.c
```

Run with mpirun:

```
mpirun -n 4 a.out
```

This is what you get:

```
Hello
Hello
Hello
Hello
```

No MPI anywhere in the code.

Fundamental Concept of MPI
Do you get it?

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

main(int argc, char** argv){
   int my_PE_num;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

```fortran
program hello
include 'mpif.h'

integer my_pe_num, errcode

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)
print *, 'Hello from ', my_pe_num,'.'
call MPI_FINALIZE(errcode)
end
```

```
Hello from 5.
Hello from 3.
Hello from 1.
Hello from 2.
Hello from 7.
Hello from 0.
Hello from 6.
Hello from 4.
```

Second Example: Sending and Receiving Messages

Hello World might be illustrative, but we haven't really done any message passing yet.

Let's write about the simplest possible message passing program:

It will run on 2 PEs and will send a simple message (the number 42) from PE 1 to PE 0. PE 0 will then print this out.

Sending a Message

Sending a message is a simple procedure. In our case the routine will look like this in C (the common pages are in C, so you should get used to seeing this format):

```
MPI_Send( &numbertosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD)
```

&numbertosend	a pointer to whatever we wish to send. In this case it is simply an integer. It could be anything from a character string to a column of an array or a structure. It is even possible to pack several different data types in one message.
1	the number of items we wish to send. If we were sending a vector of 10 int's, we would point to the first one in the above parameter and set this to the size of the array.
MPI_INT	the type of object we are sending. Possible values are: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LING, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED Most of these are obvious in use. MPI_BYTE will send raw bytes (on a heterogeneous workstation cluster this will suppress any data conversion). MPI_PACKED can be used to pack multiple data types in one message, and we can also define out own types. We will save that for the advanced talk.
0	Destination of the message. In this case PE 0.
10	Message tag. All messages have a tag attached to them that can be very useful for sorting messages. For example, one could give high priority control messages a different tag then data messages. When receiving, the program would check for messages that use the control tag first. We just picked 10 at random.
MPI_COMM_WORLD	We don't really care about any subsets of PEs here. So, we just chose this "default".

Receiving a Message

Receiving a message is equally simple and very symmetric with MPI_Send (hint: *cut and paste is your friend here*). In our case it will look like:

```
MPI_Recv( &numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,MPI_COMM_WORLD, &status)
```

&numbertoreceive	A pointer to the variable that will receive the item. In our case it is simply an integer that has has some undefined value until now.
1	Number of items to receive. Just 1 here.
MPI_INT	Datatype. Better be an int, since that's what we sent.
MPI_ANY_SOURCE	The node to receive from. We could use 1 here since the message is coming from there, but we'll illustrate the "wild card" method of receiving a message from anywhere.
MPI_ANY_TAG	We could use a value of 10 here to filter out any other messages (there aren't any) but, again, this was a convenient place to show how to receive any tag.
MPI_COMM_WORLD	Just using default set of all PEs.
&status	A structure that receives the status data which includes the source and tag of the message.

Send and Receive C Code

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv){

int my_PE_num, numbertoreceive, numbertosend=42;
MPI_Status status;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

if (my_PE_num==0){
    MPI_Recv( &numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    printf("Number received is: %d\n", numbertoreceive);
}
else MPI_Send( &numbertosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

MPI_Finalize(); }
```

Send and Receive Fortran Code

```
program sender
implicit none
include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend, status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 42

if (my_PE_num.EQ.0) then
     call MPI_Recv( numbertoreceive, 1, MPI_INTEGER,MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status, errcode)
     print *, 'Number received is:‘  ,numbertoreceive
endif


if (my_PE_num.EQ.1) then
     call MPI_Send( numbertosend, 1,MPI_INTEGER, 0, 10, MPI_COMM_WORLD,   errcode)
endif

call MPI_FINALIZE(errcode)

end
```

A Peek Behind The Curtain

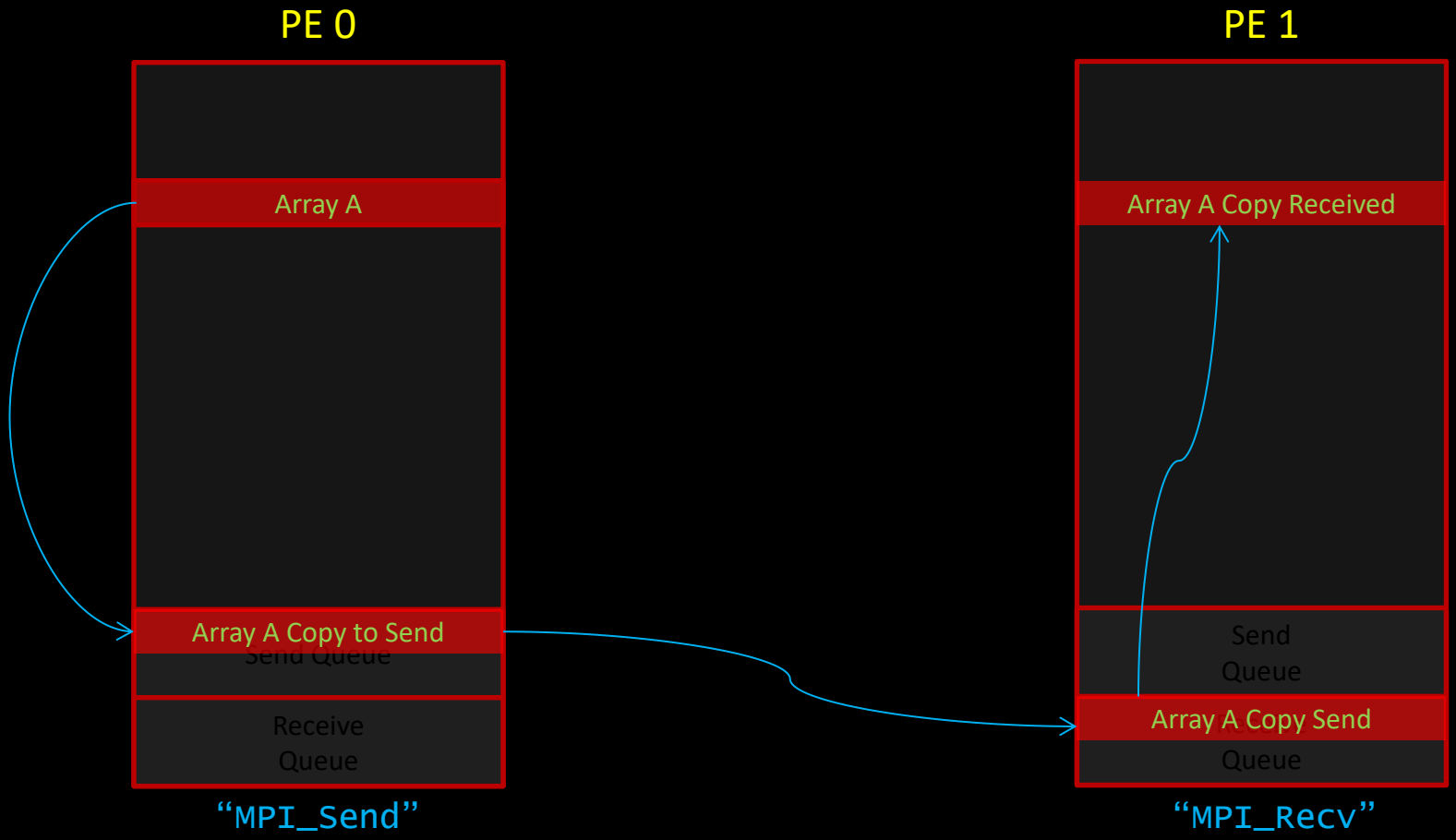
You can ignore the ne

However, many of you e is passed?", or "Why are there these

Since I always get thos gh these details.

Do not get intimidate use MPI. Just follow along and we w

Default MPI Messaging

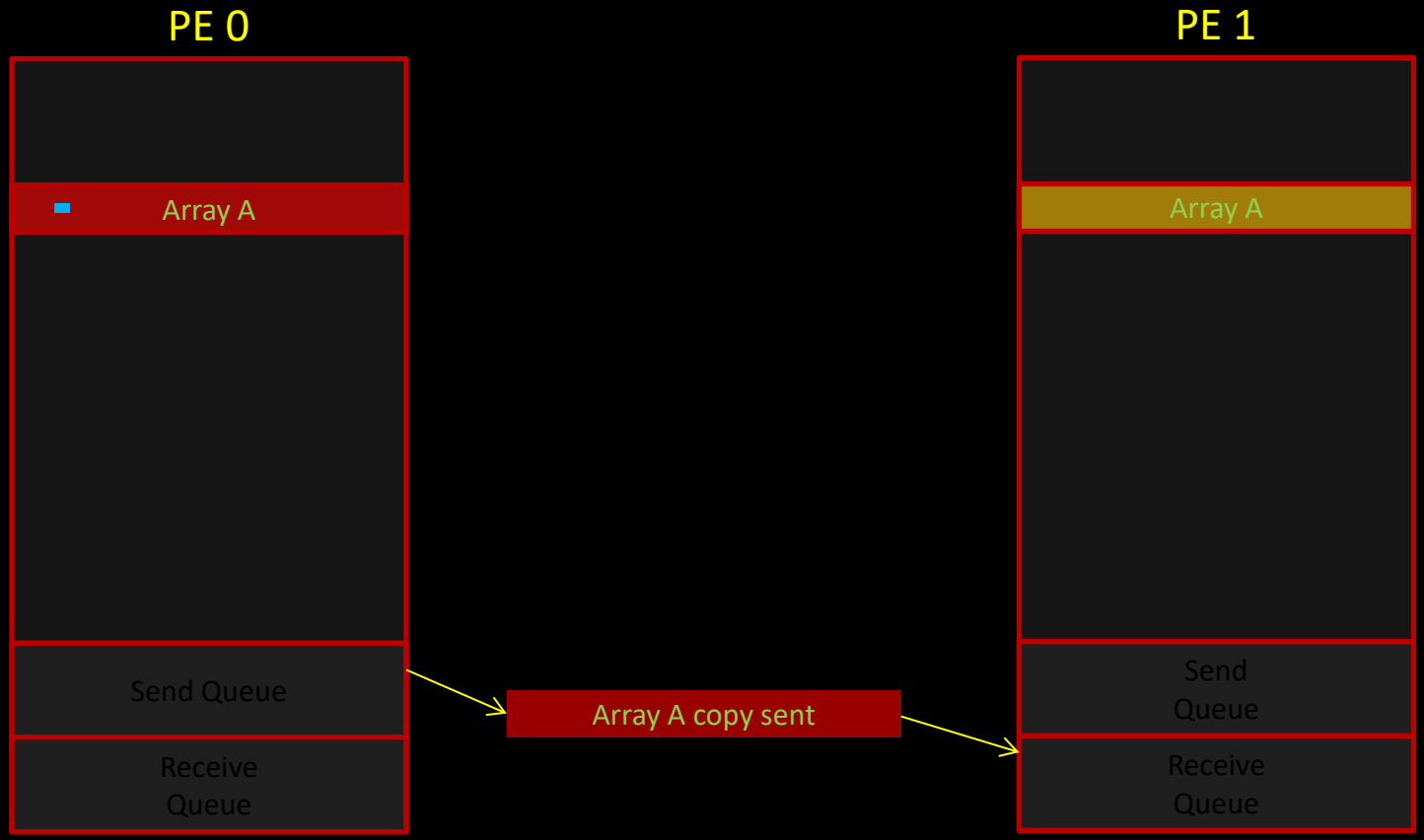


Send makes a copy

"MPI_Send(A, PE 1)"

Why make a copy?

Message in flight



Sitting in receive queue

Received

PE 0
Array A
Send Queue
Receive Queue
"goto some_where"
PE 1
Array A
Send Queue
Receive Queue
"MPI_Recv(A, PE 0)"

Receive Queue

The message queue contains all of the message that have already arrived. You can just grab the next one in the queue with MPI_ANY_SOURCE and MPI_ANY_TAG, or you can be selective.

Source	Tag	Results in Data
1	2	4444
3	2	1008*
2	1	43*
2	2	56
ANY_SOURCE	2	1008 or 4444 or 56*
1	ANY_TAG	4444*

* MPI's only ordering guarantee is that messages from the same source will stay in order. So, you will receive and retrieve the first sent message from any source first.

Note that communicators can also function as filters, but data types and data counts **do not**. Mismatched data types are an error, and may not even be detected at runtime!

Message Queue on PE 0

Data	Source	Tag
43	2	1
1008	3	2
4444	1	2
80999	2	1
5345	3	1
9044	1	1
5666	3	1
339	2	1
346	3	2
789879	2	1
78942	2	1
56	2	2
44509	1	1

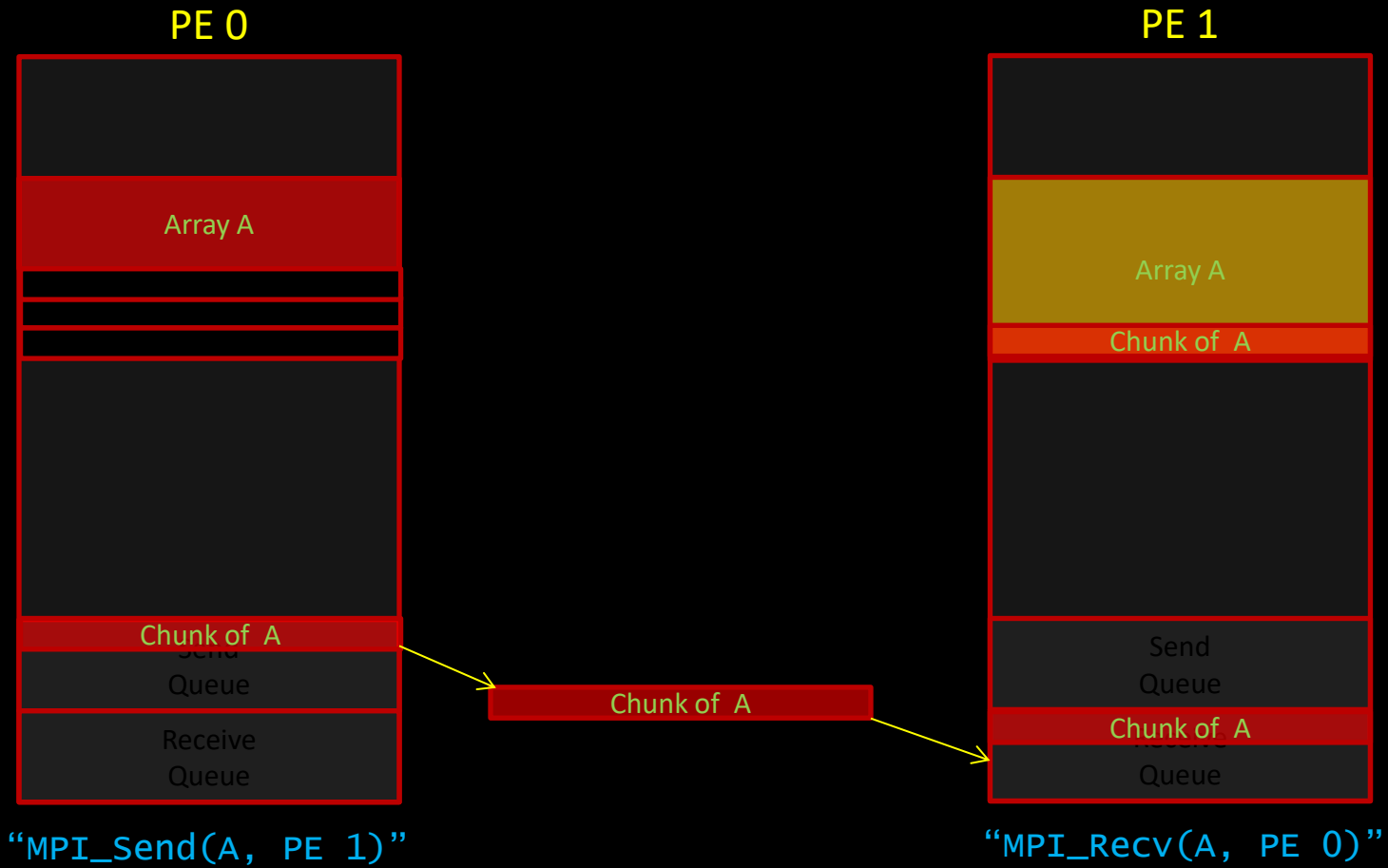
Non-Blocking Sends and Receives

All of the receives that we are using here are blocking. This means that they will wait until a message matching their requirements for source and tag has been received into the queue.

The default Sends try not to block, but don't guarantee it. As your data structures grow large, this blocking behavior may well emerge.

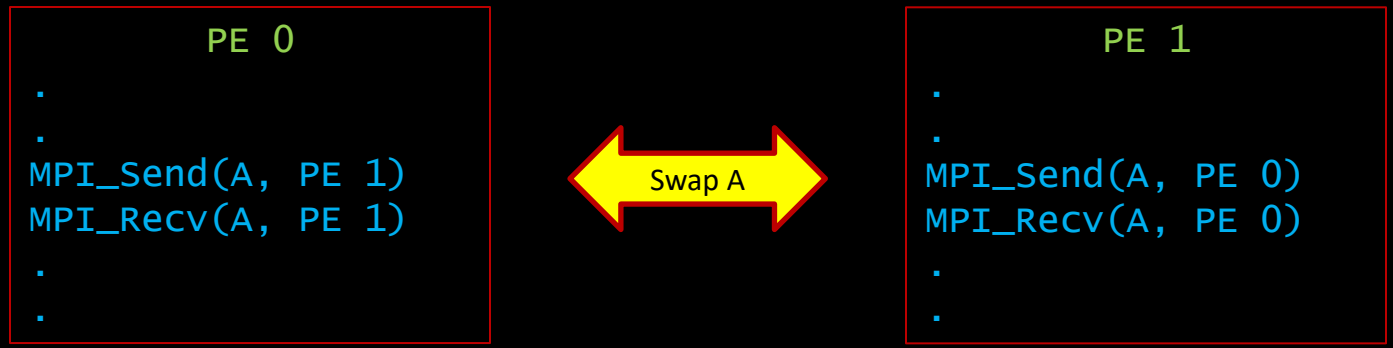
Blocking MPI_Send

Receive Issued



Non-Blocking Sends and Receives

This may seem like a strictly performance issue, but it is not. What about the very common case of:



If both PEs block on Send, we have a deadlock. The code will hang forever.

Non-Blocking Sends and Receives

This is a very useful subject, but we can comfortably defer it until the Advanced MPI talk because:

- It straightforward to convert blocking sends and receives to non-blocking when needed.
- It is often (but not always) easier to implement and debug a code with blocking and then optimize to non-blocking.
- It would clutter up our examples with additional complexity and would not gain us anything here.

Communication Modes

There are also variations on the normal blocking send and receive that alter the standard buffering behavior and may buy some optimization with little effort. If your algorithm is set up correctly, it may be just a matter of changing one letter in the routine and you have a speedier code. There are four possible modes (with differently named MPI_XSEND routines) for buffering and sending messages in MPI. We use the standard mode for all of our work.

Standard mode	Send will usually not block even if a receive for that message has not occurred. Exception is if there are resource limitations (buffer space).
Buffered Mode MPI_Bsend	Similar to above, but will never block (just return error).
Synchronous Mode MPI_Ssend	Will only return when matching receive has started. No extra buffer copy needed, but also can't do any additional computation. *Good for checking code safety.*
Ready Mode MPI_Rsend	Will only work if matching receive is already waiting. Must be well synchronized or behavior is undefined.

```
#include <sto
#include "mpi
main(int argc

int my_PE_num
MPI_Status st

MPI_Init(&arg
MPI_Comm_rank

if (my_PE_num
    MPI_Recv(                                                    &status);
    printf("M
}
else MPI_Send

MPI_Finalize(
```

But it really is quite simple

```
program sender
implicit none
include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend, status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 42

if (my_PE_num.EQ.0) then
     call MPI_Recv( numbertoreceive, 1, MPI_INTEGER,MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status, errcode)
     print *, 'Number received is:‘  ,numbertoreceive
endif


if (my_PE_num.EQ.1) then
     call MPI_Send( numbertosend, 1,MPI_INTEGER, 0, 10, MPI_COMM_WORLD,   errcode)
endif

call MPI_FINALIZE(errcode)

end
```

Third Example: Synchronization

We are going to write another code which will employ the remaining tool that we need for general parallel programming: synchronization. Many algorithms require that you be able to get all of the nodes into some controlled state before proceeding to the next stage. This is usually done with a synchronization point that requires all of the nodes (or some specified subset at the least) to reach a certain point before proceeding. Sometimes the manner in which messages block will achieve this same result implicitly, but it is often necessary to explicitly do this and debugging is often greatly aided by the insertion of synchronization points which are later removed for the sake of efficiency.

Third Example: Synchronization

Our code will perform the rather pointless operations of:

1) Have PE 0 send a number to the other 3 PEs

Foreshadowing:
Broadcast
MPI_Bcast()

2) have them multiply that number by their own PE number

3) they will then print the results out, in order (remember the hello world program?)

Synchronize
MPI_Barrier()

4) and send them back to PE 0

Reduction
MPI_Reduce()

5) which will print out the sum.

Synchronization: C Code

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

main(int argc, char** argv){

        int my_PE_num, numbertoreceive, numbertosend=4,index, result=0;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

        if (my_PE_num==0)
            for (index=1; index<4; index++)
               MPI_Send( &numbertosend, 1,MPI_INT, index, 10,MPI_COMM_WORLD);
        else{
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
            result = numbertoreceive * my_PE_num;
        }

        for (index=1; index<4; index++){
            MPI_Barrier(MPI_COMM_WORLD);
            if (index==my_PE_num) printf("PE %d's result is %d.\n", my_PE_num, result);
        }

        if (my_PE_num==0){
            for (index=1; index<4; index++){
                MPI_Recv( &numbertoreceive, 1,MPI_INT,index,10, MPI_COMM_WORLD, &status);
                result += numbertoreceive;
            }
            printf("Total is %d.\n", result);
        }
        else
            MPI_Send( &result, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

Synchronization: Fortran Code

```
program synch
implicit none

include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend
integer index, result
integer status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 4
result = 0

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Send( numbertosend, 1, MPI_INTEGER, index, 10, MPI_COMM_WORLD, errcode)
        enddo
else
        call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, status, errcode)
        result = numbertoreceive * my_PE_num
endif

do index=1,3
        call MPI_Barrier(MPI_COMM_WORLD, errcode)
        if (my_PE_num.EQ.index) then
             print *, 'PE ',my_PE_num,'s result is ',result,'.'
        endif
enddo

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, index,10, MPI_COMM_WORLD, status, errcode)
            result = result + numbertoreceive
        enddo
        print *,'Total is ',result,'.'
else
        call MPI_Send( result, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, errcode)
endif

call MPI_FINALIZE(errcode)
end
```

Step 1 – Manager, Worker

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

main(int argc, char** argv){

        int my_PE_num, numbertoreceive, numbertosend=4,index, result=0;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

        if (my_PE_num==0)
            for (index=1; index<4; index++)
                MPI_Send( &numbertosend, 1,MPI_INT, index, 10,MPI_COMM_WORLD);
        else{
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
            result = numbertoreceive * my_PE_num;
        }

        for (index=1; index<4; index++){
            MPI_Barrier(MPI_COMM_WORLD);
            if (index==my_PE_num) printf("PE %d's result is %d.\n", my_PE_num, result);
        }

        if (my_PE_num==0){
            for (index=1; index<4; index++){
                MPI_Recv( &numbertoreceive, 1,MPI_INT,index,10, MPI_COMM_WORLD, &status);
                result += numbertoreceive;
            }
            printf("Total is %d.\n", result);
        }
        else
            MPI_Send( &result, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

Step 2 – Manager, Worker

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

main(int argc, char** argv){

        int my_PE_num, numbertoreceive, numbertosend=4,index, result=0;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

        if (my_PE_num==0)
            for (index=1; index<4; index++)
                MPI_Send( &numbertosend, 1,MPI_INT, index, 10,MPI_COMM_WORLD);
        else{
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
            result = numbertoreceive * my_PE_num;
        }

        for (index=1; index<4; index++){
            MPI_Barrier(MPI_COMM_WORLD);
            if (index==my_PE_num) printf("PE %d's result is %d.\n", my_PE_num, result);
        }

        if (my_PE_num==0){
            for (index=1; index<4; index++){
                MPI_Recv( &numbertoreceive, 1,MPI_INT,index,10, MPI_COMM_WORLD, &status);
                result += numbertoreceive;
            }
            printf("Total is %d.\n", result);
        }
        else
            MPI_Send( &result, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

Step 3: Print in order…

Remember Hello World's random order? What if we did:

```
IF myPE=0 PRINT "Hello from 0."
IF myPE=1 PRINT "Hello from 1."
IF myPE=2 PRINT "Hello from 2."
IF myPE=3 PRINT "Hello from 3."
IF myPE=4 PRINT "Hello from 4."
IF myPE=5 PRINT "Hello from 5."
IF myPE=6 PRINT "Hello from 6."
IF myPE=7 PRINT "Hello from 7."
```

Would this print in order?

What if one PE was 1,000,000 times faster than another?

Step 3: Print in order…

No. How about?

```
IF myPE=0 PRINT “Hello from 0.”
BARRIER
IF myPE=1 PRINT “Hello from 1.”
BARRIER
IF myPE=2 PRINT “Hello from 2.”
BARRIER
IF myPE=3 PRINT “Hello from 3.”
BARRIER
IF myPE=4 PRINT “Hello from 4.”
BARRIER
IF myPE=5 PRINT “Hello from 5.”
BARRIER
IF myPE=6 PRINT “Hello from 6.”
BARRIER
IF myPE=7 PRINT “Hello from 7.”
```

If you liked our Book Club analogy, this is like saying “everyone stop when you get to Chapter 2, then we can continue.”

Would this print in order?

What if one PE was 1,000,000 times faster than another?

Step 3: Print in order…

Now let’s be lazy:

```
FOR X = 0 to 7
    IF MyPE = X
          PRINT “Hello from MyPE.”
    BARRIER
```

This is a common idiom.

Step 3 – Manager, Worker

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

main(int argc, char** argv){

        int my_PE_num, numbertoreceive, numbertosend=4,index, result=0;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

        if (my_PE_num==0)
            for (index=1; index<4; index++)
                MPI_Send( &numbertosend, 1,MPI_INT, index, 10,MPI_COMM_WORLD);
        else{
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
            result = numbertoreceive * my_PE_num;
        }

        for (index=1; index<4; index++){
            MPI_Barrier(MPI_COMM_WORLD);
            if (index==my_PE_num) printf("PE %d's result is %d.\n", my_PE_num, result);
        }

        if (my_PE_num==0){
            for (index=1; index<4; index++){
                MPI_Recv( &numbertoreceive, 1,MPI_INT,index,10, MPI_COMM_WORLD, &status);
                result += numbertoreceive;
            }
            printf("Total is %d.\n", result);
        }
        else
            MPI_Send( &result, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

Step 4 – Manager, Worker

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

main(int argc, char** argv){

        int my_PE_num, numbertoreceive, numbertosend=4,index, result=0;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

        if (my_PE_num==0)
            for (index=1; index<4; index++)
                MPI_Send( &numbertosend, 1,MPI_INT, index, 10,MPI_COMM_WORLD);
        else{
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
            result = numbertoreceive * my_PE_num;
        }

        for (index=1; index<4; index++){
            MPI_Barrier(MPI_COMM_WORLD);
            if (index==my_PE_num) printf("PE %d's result is %d.\n", my_PE_num, result);
        }

        if (my_PE_num==0){
            for (index=1; index<4; index++){
                MPI_Recv( &numbertoreceive, 1,MPI_INT,index,10, MPI_COMM_WORLD, &status);
                result += numbertoreceive;
            }
            printf("Total is %d.\n", result);
        }
        else
            MPI_Send( &result, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

Step 5 – Manager, Worker

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

main(int argc, char** argv){

        int my_PE_num, numbertoreceive, numbertosend=4,index, result=0;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

        if (my_PE_num==0)
            for (index=1; index<4; index++)
                MPI_Send( &numbertosend, 1,MPI_INT, index, 10,MPI_COMM_WORLD);
        else{
            MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
            result = numbertoreceive * my_PE_num;
        }

        for (index=1; index<4; index++){
            MPI_Barrier(MPI_COMM_WORLD);
            if (index==my_PE_num) printf("PE %d's result is %d.\n", my_PE_num, result);
        }

        if (my_PE_num==0){
            for (index=1; index<4; index++){
                MPI_Recv( &numbertoreceive, 1,MPI_INT,index,10, MPI_COMM_WORLD, &status);
                result += numbertoreceive;
            }
            printf("Total is %d.\n", result);
        }
        else
            MPI_Send( &result, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

Step 1 – Manager, Worker

```
program synch
implicit none

include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend
integer index, result
integer status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 4
result = 0

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Send( numbertosend, 1, MPI_INTEGER, index, 10, MPI_COMM_WORLD, errcode)
        enddo
else
        call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, status, errcode)
        result = numbertoreceive * my_PE_num
endif

do index=1,3
        call MPI_Barrier(MPI_COMM_WORLD, errcode)
        if (my_PE_num.EQ.index) then
             print *, 'PE ',my_PE_num,'s result is ',result,'.'
        endif
enddo

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, index,10, MPI_COMM_WORLD, status, errcode)
            result = result + numbertoreceive
        enddo
        print *,'Total is ',result,'.'
else
        call MPI_Send( result, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, errcode)
endif

call MPI_FINALIZE(errcode)
end
```

Step 2 – Manager, Worker

```
program synch
implicit none

include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend
integer index, result
integer status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 4
result = 0

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Send( numbertosend, 1, MPI_INTEGER, index, 10, MPI_COMM_WORLD, errcode)
        enddo
else
        call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, status, errcode)
        result = numbertoreceive * my_PE_num
endif

do index=1,3
        call MPI_Barrier(MPI_COMM_WORLD, errcode)
        if (my_PE_num.EQ.index) then
             print *, 'PE ',my_PE_num,'s result is ',result,'.'
        endif
enddo

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, index,10, MPI_COMM_WORLD, status, errcode)
            result = result + numbertoreceive
        enddo
        print *,'Total is ',result,'.'
else
        call MPI_Send( result, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, errcode)
endif

call MPI_FINALIZE(errcode)
end
```

Step 3 – Manager, Worker

```
program synch
implicit none

include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend
integer index, result
integer status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 4
result = 0

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Send( numbertosend, 1, MPI_INTEGER, index, 10, MPI_COMM_WORLD, errcode)
        enddo
else
        call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, status, errcode)
        result = numbertoreceive * my_PE_num
endif

do index=1,3
        call MPI_Barrier(MPI_COMM_WORLD, errcode)
        if (my_PE_num.EQ.index) then
             print *, 'PE ',my_PE_num,'s result is ',result,'.'
        endif
enddo

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, index,10, MPI_COMM_WORLD, status, errcode)
            result = result + numbertoreceive
        enddo
        print *,'Total is ',result,'.'
else
        call MPI_Send( result, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, errcode)
endif

call MPI_FINALIZE(errcode)
end
```

Step 4 – Manager, Worker

```
program synch
implicit none

include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend
integer index, result
integer status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 4
result = 0

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Send( numbertosend, 1, MPI_INTEGER, index, 10, MPI_COMM_WORLD, errcode)
        enddo
else
        call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, status, errcode)
        result = numbertoreceive * my_PE_num
endif

do index=1,3
        call MPI_Barrier(MPI_COMM_WORLD, errcode)
        if (my_PE_num.EQ.index) then
             print *, 'PE ',my_PE_num,'s result is ',result,'.'
        endif
enddo

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, index,10, MPI_COMM_WORLD, status, errcode)
            result = result + numbertoreceive
        enddo
        print *,'Total is ',result,'.'
else
        call MPI_Send( result, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, errcode)
endif

call MPI_FINALIZE(errcode)
end
```

Step 5 – Manager, Worker

```
program synch
implicit none

include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend
integer index, result
integer status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 4
result = 0

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Send( numbertosend, 1, MPI_INTEGER, index, 10, MPI_COMM_WORLD, errcode)
        enddo
else
        call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, status, errcode)
        result = numbertoreceive * my_PE_num
endif

do index=1,3
        call MPI_Barrier(MPI_COMM_WORLD, errcode)
        if (my_PE_num.EQ.index) then
             print *, 'PE ',my_PE_num,'s result is ',result,'.'
        endif
enddo

if (my_PE_num.EQ.0) then
        do index=1,3
            call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, index,10, MPI_COMM_WORLD, status, errcode)
            result = result + numbertoreceive
        enddo
        print *,'Total is ',result,'.'
else
        call MPI_Send( result, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, errcode)
endif

call MPI_FINALIZE(errcode)
end
```

Results of “Synchronization”

The output you get when running this codes with 4 PEs is:

```
PE 1’s result is 4.
PE 2’s result is 8.
PE 3’s result is 12.
Total is 24
```

What would happen if you ran with more than 4 PEs? Less?

A Few Words About *MPI_Barrier*

Barriers are often necessary.

Barriers are often convenient. They can help with debugging, and are sometimes put in just to avoid thinking about possible race conditions. These unnecessary barriers can cause wasteful idle time and should be eventually eliminated.

Think of them like a race car driver thinks about brakes. You don't want to use them more than required, but you aren't going to get anywhere without them.

Analysis of “Synchronization”

The best way to make sure that you understand what is happening in the code above is to look at things from the perspective of each PE in turn. THIS IS THE WAY TO DEBUG ANY MESSAGE-PASSING CODE.

Follow from the top to the bottom of the code as PE 0, and do likewise for PE 1. See exactly where one PE is dependent on another to proceed. Look at each PEs progress as though it is 100 times faster or slower than the other nodes. Would this affect the final program flow? It shouldn't unless you made assumptions that are not always valid.

Speaking of Debugging...

The good old-fashioned print statement is a fine way to get started debugging. Maybe with little elaboration like

```
printf("Got to point B on PE %d \n", my_PE_num);
```

You will find these scattered around many codes in development. Especially as the most common symptom of an MPI bug is that PEs just "hang" because of blocking messages, and you need to figure out where.

However, this is often many coders introduction to the phenomenon of IO buffering. It can make your prints come out in funny order, or seem to disappear. You can investigate how to get your system to properly *flush* output, but this can be frustrating as many flush commands are simply ignored or overridden by the system.

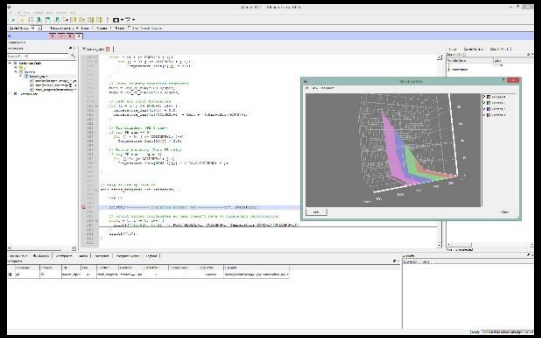
One technique I have found very useful is to take advantage of the fact that stderr is usually unbuffered:

Fortran:

```
write(0,*) my_pe_num
```

C:

```
fprintf(stderr,"PE %d\n", my_PE_num);
```



Final Example: Beyond the Basics

You now have the 4 primitives that you need to write any algorithm. However, there are much more efficient ways to accomplish certain tasks, both in terms of typing and computing.

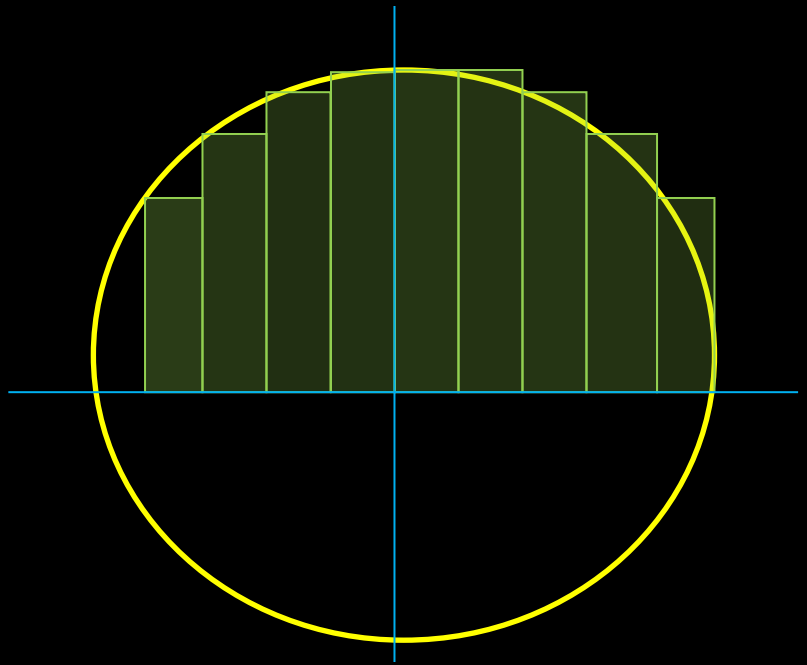
We will look at a few very useful and common routines (reduction, broadcasts, and Comm_Size) as we do a final example.

You will then be a full-fledged (or maybe fledgling) MPI programmer.

Final Example: Finding Pi

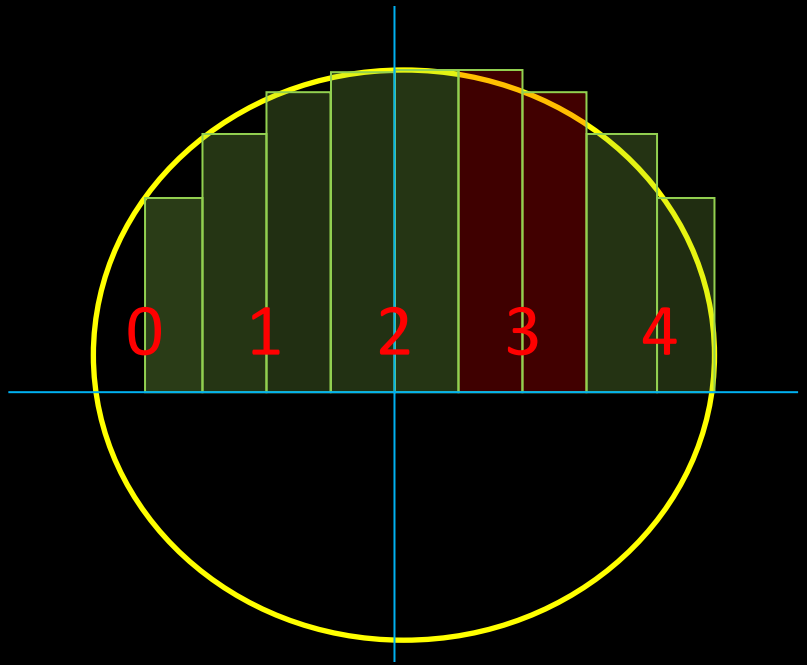
Our last example will find the value of pi by integrating over the area of a circle of radius 1. We just use the fact that the Area = π R^2 and that the equation for such a circle is $x^2 + y^2 = 1$.

We will use the standard (and classic serial) method of finding the area under a curve by adding up a bunch of rectangular slices:



Final Example: Finding Pi

We can parallelize this very effectively by just having a number of PE's work on subsections and add up their final results. With 5 PE's, we should expect it to take roughly 1/5 as long to reach an answer.



Final Example: Finding Pi

We could easily do this with the commands we have in hand. However, this is a good time to introduce a few of the more common and powerful additional MPI commands: broadcast and reduce.

MPI_Bcast() is very useful when we want to communicate some common data to every PE. We know we can do this with a loop, as in the previous example, but with Bcast we not only save some typing and make the code more readable, but we enable the hardware to use a much more efficient broadcast mode. Picture the difference between me giving you each this lecture point-to-point, or "broadcasting" to you all as I am.

We use MPI_Bcast() here to let all of the nodes know how many intervals the user would like to employ for the approximation. You will find yourself using it frequently.

Final Example: Finding Pi

We also need to add together each PE's partial sum. We just did something like this in the previous example, but once again we find that MPI provides us with a command that saves typing and is makes more efficient use of the hardware.

MPI_Reduce() allows us to do this loop with one command. However, it can be used for more than just adding numbers. Other common operations are mins and maxes, but there are plenty more.

Note that both Bcast and Reduce require all of the PE's to participate, ar
particular PE as the sender or collector. This will often be 0 if it is the ma

Using any number of available PEs

And we are going to add one last capability that any good MPI application should have: the ability to run on various numbers of PEs. We have hard-coded the number of PEs into our previous examples, but think about how much better it would be if they could run on varying PE counts. After all, scalability is probably the main reason we are writing MPI codes to begin with.

MPI_Comm_size() gives the basic information we need for the code to adjust itself.

We use it here so that each PE can calculate the number of slices it is responsible for. If the user want 1,000,000 slices for good precision, and then runs the code with mpirun -n 10 we should have each PE work on 100,000 slices.

Finding Pi

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

int main( int argc, char **argv ){

  int    n, my_pe_num, numprocs, index;
  float  mypi, pi, h, x, start, end;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs );
  MPI_Comm_rank(MPI_COMM_WORLD, &my_pe_num );

  if( my_pe_num == 0 ){
    printf("How many intervals? ");
    scanf("%d", &n);
  }

  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

  mypi = 0;
  h= (float) 2/n;                        /*Size of each slice*/
  start = (my_pe_num*2/numprocs)-1;      /*Slices for this PE*/
  end   = ((my_pe_num+1)*2/numprocs)-1;

  for (x = start; x < end; x = x+h)
    mypi = mypi + h * 2* sqrt(1-x*x);

  MPI_Reduce(&mypi, &pi, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

  if( my_pe_num == 0 ){
    printf("Pi is approximately %f\n", pi);
    printf("Error is %f\n", pi-3.14159265358979323846);
  }

  MPI_Finalize();

}
```

References

There are a wide variety of materials available on the Web, some of which are intended to be used as hardcopy manuals and tutorials. A good starting point for documents is the MPI Forum Docs page:

http://www.mpi-forum.org/docs/

A good index of MPI man pages can be found at:

http://www.mpich.org/static/docs/latest/

Two very authoritative and comprehensive books:

- *Using MPI: portable parallel programming with the message-passing interface.* William Gropp, Ewing Lusk, Anthony Skjellum. MIT Press
- *Using Advanced MPI*. Gropp, Hoefler, Thakur and Lusk. MIT Press

Exercises

Exercise 1: Write a code that runs on 8 PEs and does a “circular shift.” This means that every PE sends some data to its nearest neighbor either “up” (one PE higher) or “down.” To make it circular, PE 7 and PE 0 are treated as neighbors. Make sure that whatever data you send is received.

Exercise 2: Write, using only the routines that we have covered in the first three examples, (MPI_Init, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Barrier, MPI_Finalize) a program that determines how many PEs it is running on. It should perform as the following:

```
mpirun –n 4 exercise
I am running on 4 PEs.

mpirun –n 16 exercise
I am running on 16 PEs.
```

You would normally obtain this information with the simple *MPI_Comm_size()* routine. The solution may not be as simple as it first seems. Remember, make no assumptions about when any given message may be received.

Exercises Summary

A concise index of all MPI calls, with each parameter described is:

http://www.mpich.org/static/docs/latest/

Exercise 1: Write a code that runs on 8 PEs and does a “circular shift.”.

Exercise 2: Write (using only MPI_Init, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Barrier, and MPI_Finalize) a program that determines how many PEs it is running on.

Example session for Exercise 2 (after you’ve edited *your_program.c* into existence):

```
% interact –n 8                                    (if you aren’t already on a compute node)
% mpicc your_program.c or mpif90 your_program.f90
% mpirun -n 4 a.out
I am running on 4 PEs.
%
```