Warwick Research Software Engineering

Introduction to MPI

C.S. Brady and H. Ratcliffe
Senior Research Software Engineers

“The Angry Penguin”, used under creative commons licence
from Swantje Hess and Jannis Pohlmann.

March 27, 2020

Contents

Preface

0.1 About these Notes

These notes were written by H Ratcliffe and C S Brady, both Senior Research Software Engineers in the Scientific Computing Research Technology Platform at the University of Warwick for a Workshop first run in December 2018 at the University of Warwick.

The notes were typeset in LaTeXby H Ratcliffe.
Errors can be reported to rse@warwick.ac.uk

0.2 Example Programs

Several sections of these notes benefit from hands-on practise with the concepts and tools involved. Test code is available on Github at `https://github.com/WarwickRSE/IntroMPI`

Chapter 1

Principles of Parallel Programming

1.1 History

Parallel computing precedes electronic computers. Early 20th century calculations done by hand by "computers"[1] could still be broken into pieces and worked on simultaneously. The following image shows workers at GCHQ doing this. The recent book (and film) Hidden Figures http://www.hiddenfigures.com/ is about the human computers involved in the space race. Human computers could even match the speed of early computers (https://en.wikipedia.org/wiki/Harwell_computer) although not their durations.

Figure 1.1: Women working as "computers" at Bletchley park

[1]Humans, using pencil and paper, simple adding machines, books of tables etc

1.2 Kinds of Parallelism

There are many levels at which computers can work on things in parallel. We're only going to discuss the MPI standards here, and not really any of the following:

- Bit level parallelism

  Processors work on chunks of data at once rather than bit by bit

- Instruction level parallelism

  Processors can operate on more than one variable at a time

  NOT multicore

  A large chunk of optimisation of code is trying to improve instruction level parallelism

- Task level parallelism

  Split up a "task" into separate bits that computer can work on separately

- Task Farming

  Tasks that are unrelated to each other can easily just be handed off to different processors

  Exploring parameters

1.2.1 Embarrassing Parallelism

Embarrassing parallelism is a fairly special case that comes up a lot in practice. Here, you have a lot of unconnected, un-sequenced tasks. Since there's no connections between tasks, you can simply run multiple copies of your program. It is important not to run more copies than you have physical processors. Modern computers use something called "Simultaneous Multithreading" or "Hyperthreading" which runs several things at once, swapping between them. This is great for normal computing loads, since generally a lot of time is spent waiting for user inputs, or for things to happen. For research computing loads, it doesn't generally work well, with the threads contending for resources and taking longer overall than running one then the other. Large or commercial machines use a scheduling system to run jobs only when there is a free processor.

1.2.2 Tightly coupled parallelism

The opposite of embarrassing parallelism is tightly coupled parallelism, where a problem splits into chunks, but each chunk needs information from other chunks, whether this is only some other chunks or all other chunks. Somehow the data has to be made available to all chunks which need it.

This leads to two main concerns. To allow more than one processor to work on a chunk of data you have to:

1. Make sure that the data is somewhere so that it can be used by a processor (*communication*)

2. Make sure that data transfer is *synchronised* so that you know you can use it when you need to

These two problems can be addressed in several ways, and different models of parallelism use different approaches.

1.2.3 Shared Memory Paralellism

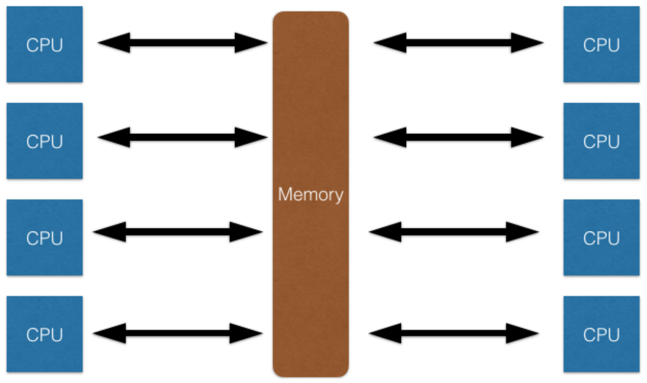

Figure 1.2: Schematic diagram of shared memory parallelism

Shared memory parallelism lets multiple tasks access the same data, i.e. they share their data memory. Figure 1.2 shows this. Each processor has some designated work chunk, but the memory is shared. *Communication* is automatic, but *synchronisation* is a significant problem.

The following figure shows how a single processor does an operation such as $i = i+1$. First, it reads the current value, here $i = 0$, from memory, into its own internal memory[2]. Then it increments this value. Finally it copies the new value back where it came from.

[2]I.e. a register

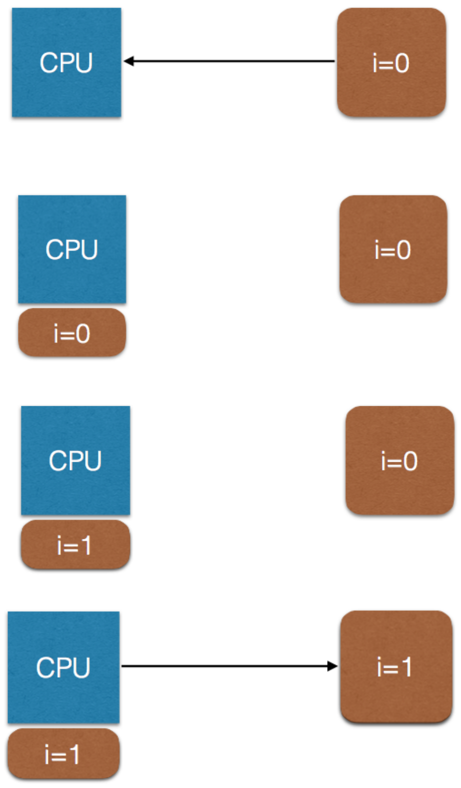

Figure 1.3: Sequence of events for a single processor doing a simple $i = i + 1$ operation. Left-pointing arrows are memory read operations, right pointing arrows are writes.

Now imagine this process for two processors simultaneously. Each reads the value, $i = 0$, each increments it, and each writes it back. With no synchronisation occurring, the end result is $i = 1$ rather than the hoped for $i = 2$ result. The full sequence ends up looking like this:

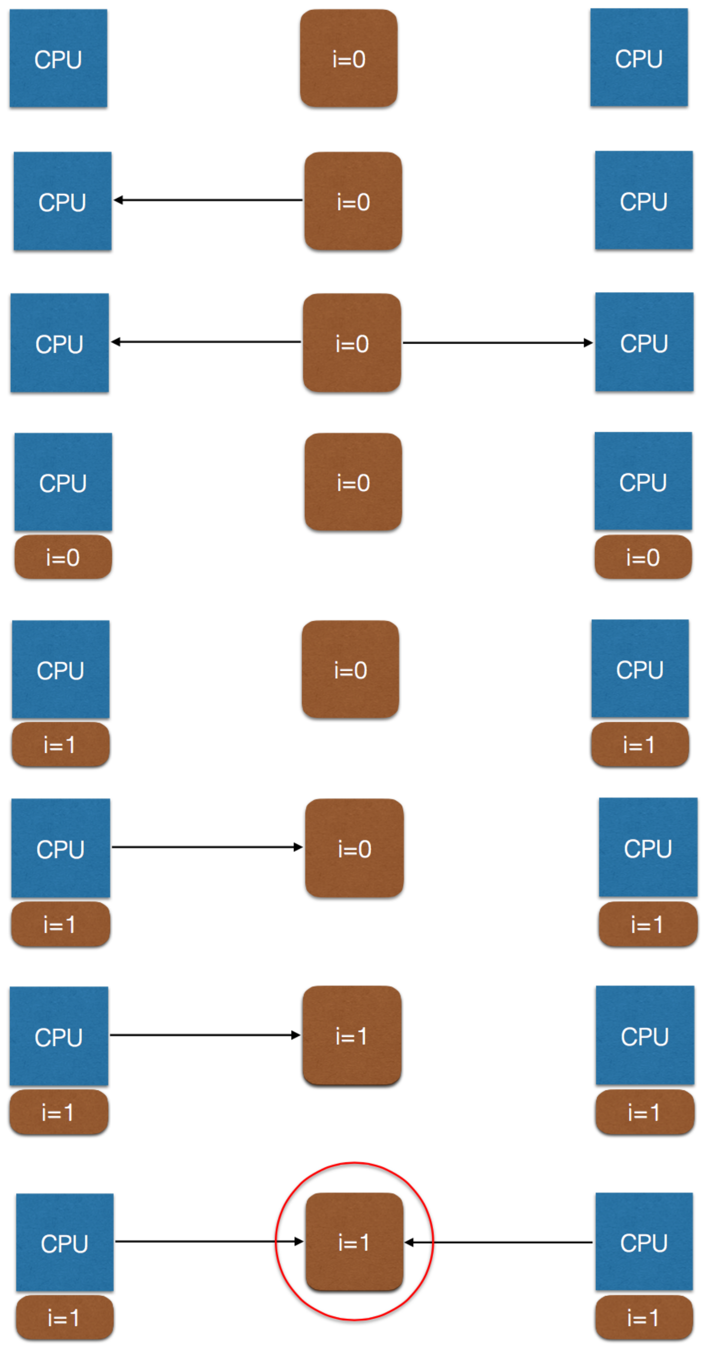

Figure 1.4: Two processors simultaneously doing an $i = i + 1$ operation. Left-pointing arrows are memory read operations, right pointing arrows are writes.

Solving this is a surprisingly tricky problem. We have to ensure that each processor only works on things which are safe to do independently, without any synchronisation, such as accessing read-only memory, or processing its own data chunk. Global variables can be asking for trouble in shared memory programs - how do you make absolutely certain they're never modified by two processes at once?

The Read-Modify-Write sequence and Atomicity

The 2-processor sequence above is related to a very common problem called a data race. If instead of incrementing i, each processor was to set the value to its own rank, the end result would depend critically on the exact order of each processor's write operation, so the two are in a race to do their work. The problem comes because we are treating read, modify and write as separate operations. What we wanted was for the complete sequence to be considered one operation.

This is very like a much deeper problem in multi-threaded computing, where several operations might affect the same piece of data. Imagine if rather than a simple integer, we were writing a string, one letter at a time. Processor 1 is trying to write the string "One", while two is writing "Two". Depending on how the operations order, we might end up with nonsense like "Owe" or "Tno".

This can also happen to some variable types which are wider than the processors "load-store" width. This means the processor has to do more than one write operation to write all the bytes of the variable, and you can, in theory, get two variables mashed together. Most systems allow you to use special operations which are guaranteed to happen all-at-once from the perspective of other parts of the system. This is often called "atomicity" because the operation cannot be divided into parts. The term might be encountered everywhere from databases (you want a record to be modified completely or not-at-all) at the largest scale, to interrupt programming (dealing with signals from the operating system or devices) at the smallest scale.

There is a fairly obvious answer to this problem, which is to use some sort of flag to "lock" a variable until you are done with it. But first you have to test whether it is already locked: and imagine what happens if between your test and your set operation, another processor has already set the lock... Systems implement atomic versions of "test-and-set" and "read-modify-write" to get around this. These can be used to implement locks, or "mutexes" (mutual exclusions).

In shared memory programming, *critical sections* are used to insist that operations must happen one after the other, and anything where this is important must be labelled correctly. In more complex cases it can be very tricky, but in many problems there are only a few critical sections.

OpenMP

OpenMP is a system commonly used in research codes providing directives to tell the compiler how to parallelise loops in code. This automates some of the synchronisation, although you still have to label critical sections, and it can't do everything. It also requires a compatible compiler, although these are very common.

Threads

Threading libraries can be used to explicitly run parts of code in other threads. The operating system will try and balance the load across processors. This technique is very common in non-academic code, especially things with a user-interface, but less common in academia. Mutual Exclusions (see aside box above) are used to force synchronisation and ensure only one processor works on data at once. A thread has to first get the mutex, do its work, and then release it. Only one thread can hold it at a time, and all others must wait. Obviously some care must be taken to avoid deadlocks, where the threads are all depending on each other in a way that can't be worked out.

1.3 Distributed Memory and MPI

In distributed memory systems all processors have their own memory, and data can only flow between them via a *fabric*. Typically programs explicitly send and receive data, i.e communication is done manually. Synchronisation is directly tied to this, and is usually guaranteed once a receive operation completes.

As a programmer, you have to manually work out the transfers of data and send explicit *messages* between processors which contain the data. There are a number of common strategies to help with this, but in general it can be tricky, and in general the comms fabric is quite slow compared with direct memory access, so we really need to minimise the amounts and frequency of data transfers.

1.3.1 MPI

The Message Passing Interface library is the most popular distributed memory library. It is just a library and needs no special compiler. It includes routines for sending and receiving data, collective operations such as summing between processors, parallel file I/O and many others.

MPI on a single computer performs nearly as well as shared memory, but is harder to program than OpenMP in simple cases. Programming is about as hard as writing threaded code would be. Finally, while the library itself scales to the largest supercomputers in the world, your algorithm may well not.

On shared memory architectures, MPI works fine. We program it as though it is distributed, although there are some advanced features which are explicitly shared. If the algorithm is amenable to distributed memory we get comparable performance to OpenMP or threading. On the other hand, some algorithms just map better onto shared memory, so we can use a Hybrid approach combining MPI with OpenMP or pthreads (a common threading library) to get the best of all worlds.

1.4 Alternatives to all of this

There are some alternatives to avoid having to deal with any of this. These include OpenSHMEM, a standard for writing shared-memory libraries. There are some extensions to specific programming languages to tackle problems, such as Coarray Fortran, or Unified Parallel C. Some languages have built-in support for “concurrent” (simultaneous) operations, including Chapel (a language developed by Cray) or X10 (from IBM). Support for these can be poor or patchy, and none have really obvious advantages over MPI or OpenMP.

Chapter 2

Starting with MPI

2.1 What is MPI?

At its core, MPI is a set of standards, set out by the MPI forum (`http://mpi-forum.org`). There are different versions of the MPI standard:

- MPI1 was the original standard from 1989. It still works, but much has been deprecated and some bits don't work well
- MPI2 was an updated standard from 1997. This added a broad variety of features and modernised lots of bits. The latest MPI2.2 standard is from 2009. We will be teaching you this!
- MPI3 is a further update from 2012. Very useful and powerful additions, but not really relevant for a beginning MPI course
- MPI4 further updates MPI3 and has not yet been formally ratified

There are many different implementations of the MPI standard, including

- OpenMPI
- MPICH
- MVAPICH
- Intel MPI

Valid MPI code will work with any of them, unless you manage to invoke one of the many interesting and strange bugs in implementations.

The MPI Application Programming Interface (API) changes according to the MPI version you're using, and the language binding you're working with. For MPI there are 3 official bindings, C, Fortran and Fortran-2008. The latter isn't ready for general use yet, but is worth knowing about.

There used to be a C++ binding. *Do not use it.* It is formally deprecated, not up to date and will be removed in future. It also adds nothing over the C bindings for a C++ developer.

There are also unofficial bindings, including OpenMPI Java, which is an almost compliant MPI 3.1 implementation, but this is only for OpenMPI. There is also MPI4Py which adds a Python layer onto any of the above MPI installations. This has a different interface to MPI proper and lacks some features. These, and the various others, are either MPI vendor specific or not fully standards compliant[1]. This means you generally have to learn them separately, so we're not going to cover them here.

2.2 The Parts of an MPI program

In general, any MPI program has the following parts:

- Initialisation
- Main program:

  Compute

  Communication

  Output
- Finalisation

The main program part is often repeated in a loop.

Note that Compute and Communication are listed separately for a few reasons:

- Clarity - it's harder to understand, debug and maintain code which mixes MPI code in among general code
- Performance - you want to do as little comms as possible, using as few messages as possible

[1]The MPI standards are available online. They're a rather dry read but eventually you may need to be familiar with parts of them.

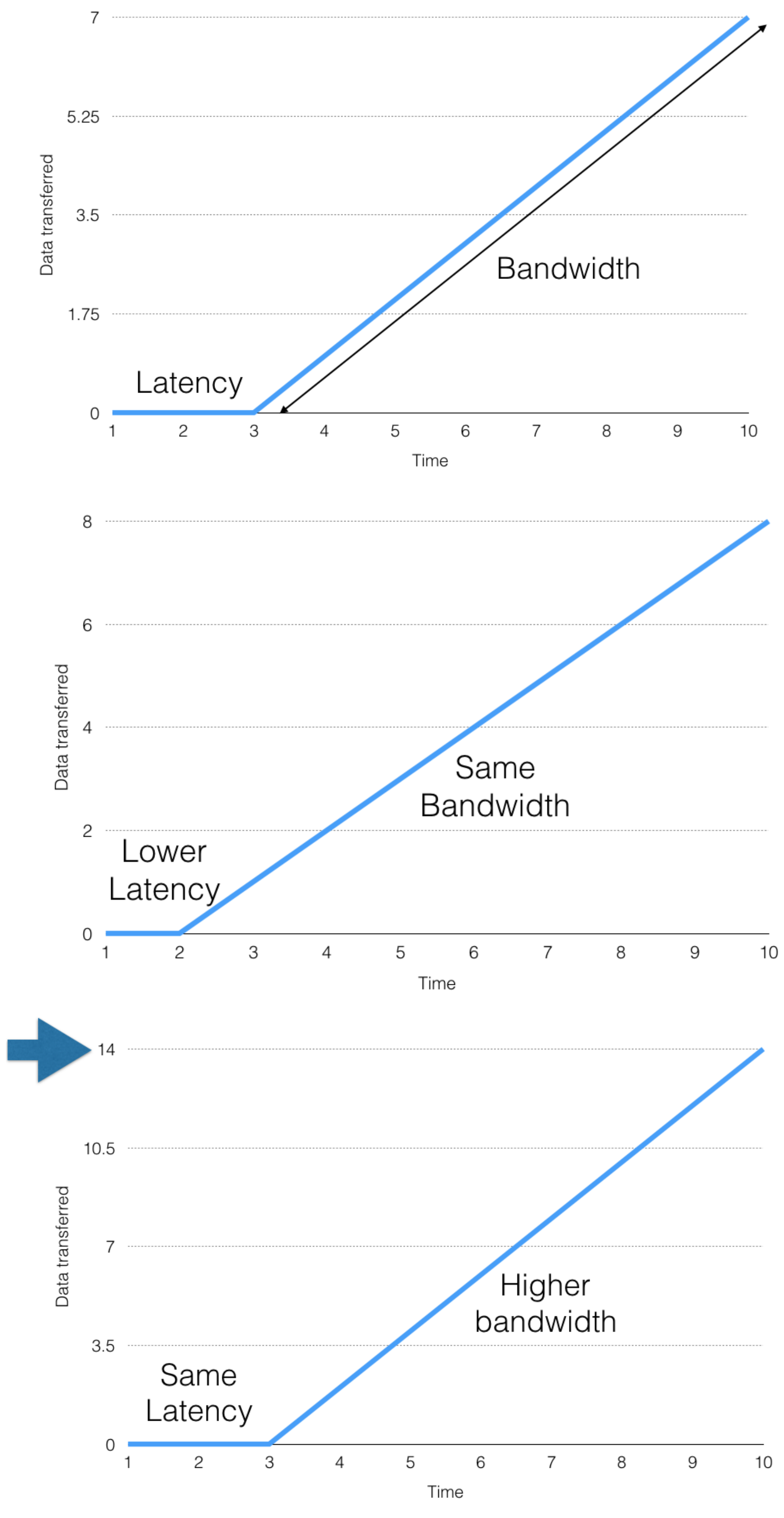

Figure 2.1: Data transferred over time for varying values of latency and bandwidth

Latency Bandwidth and Messaging

We just said "as little comms as possible, using as few messages as possible". Each MPI message takes time to assemble, prepare for sending, arrange to receive etc - this is latency. The time taken to move the data is limited by the bandwidth - how much data can be moved per second. The image above shows how the two values change the overall time to move data.
Imagine painting the centre line on a road. It takes you some time to get set up, open your paint can, and get going (latency). Once you're started, the speed at which you proceed depends (mainly) on how much paint your brush will hold (bandwidth). Each time you have to stop and open a new paint can you have a new chunk of latency.
You may have heard the old joke - a {target of humorous excursion} is tasked with painting the lines on a road. On the first day, they do brilliantly, painting 3 miles of new line. On the second day, they manage only 1.5 miles. By the third day they complete barely half a mile. "What is taking so long?" asks the supervisor. "Well," they reply, "it just takes so long to keep going back to the paint pot!". Tiny messages are almost as bad as going back to the paint pot if you have a lot of data to shift.
As an example, on real HPC kit we have a latency of 0.5μ s per message and a bandwidth of 37 GB/s. So to send 100 messages of 1MB each, we have 50 μs latency and 27 μ total sending time, a total of 77 μ s. Wheras to send a single 100MB message, the total time is just 27.5 μ s.

2.2.1 Initialisation

The initialisation step uses a single routine, `MPI_Init`, to initialise the MPI layer. This *must* be called before any other MPI routines. The signature is

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

- `argc` Pointer to `argc` in main. Usually just `&argc`
- `argv` Pointer to `argv` in main. Usually just `&argv`

This is one of the few functions that is a bit different in Fortran. There, we don't have the argc or argv arguments, but as usual we do have the final error parameter, so the full signature is

```
MPI_Init(ierr)
```

where `ierr` is an `INTEGER`.

2.2.2 Finalisation

The finalisation step also uses just a single routine, `MPI_Finalize`, to shut down the MPI layer. After calling this, you *cannot* call any other MPI routines, and you *must* call this on all ranks (processors) before the program ends. *Warning*: if you forget, your code will mostly work, but this is not guaranteed. The signature is

```
MPI_Finalize()
```

with no parameters. Remember: in Fortran every MPI function has the extra parameter `ierr`, including this one. So in Fortran this does have parameters

2.2.3 A very first MPI code

These two functions are enough to put together a very first MPI code:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  printf("Multiprocessor code\n");
  MPI_Finalize();
  return 0;
}
```

We now have to compile this using the MPI compiler, which supplies (links) all of the MPI functions. Usually the compiler will be `mpicc` (rather than `gcc`) or `mpif90` (rather than `gfortran`), but if you're using the Intel compiler suite or something like Cray, check the docs for the appropriate commands. To run the program, we have to get multiple copies going linked together by MPI - this uses the command `mpiexec -n <number of processors> <program name>`.

This program is really not much use. If you compile and run it on more than one processor you'll see several lines all saying "Multiprocessor code". Note that you don't see the letters jumbled together from different processors - while multiple print statements will get mixed together, each one usually[2] comes out in one piece. This is useful for showing user messages - just remember to assemble the whole message into a single print and keep it brief.

2.3 A basic example program

To demonstrate an actual use for MPI we need an example problem. We'll use one that may be familiar from GCSE maths - calculating π using geometry.

In the following picture

[2]As far as we know this is almost always the case for simple, single prints, but isn't guaranteed; sometimes you will get prints mixed into each other.

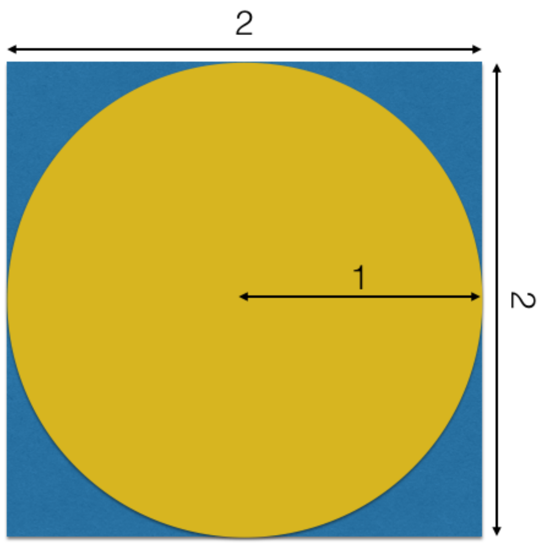

Figure 2.2: Geometry to calculate π

we know that the square has an area of $l^2 = 2^2 = 4$ while the circle has area $\pi r^2 = \pi$.

We now imagine an ideal darts player who randomly throws `ndart_total` darts at the square, never missing:

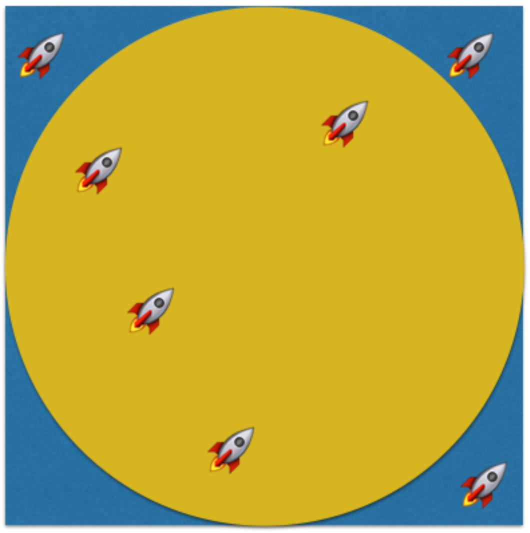

Figure 2.3: Throwing darts at the square

and we count how many of these land inside the circle `ndart_circle`. The probability of landing inside the circle is the fraction of the total area it covers, i.e. $\pi/4$, and ndart_circle / ndart_total is an estimate of this probability. As we throw more and more darts (increase `ndart_total`) we get a better and better estimate.

The code for this in serial (in C) is

```c
#include <stdio.h>
/* Define how many iterations to try */
#define ITS 10000
int main(int argc, char **argv){
  int count = 0, index;
  double d1, d2;
  /* Setup a random number generator (provided in example code)*/
  seed_rng();
  for (index=0;index<ITS;++index){
    /* Get some random numbers between  1
    I.e. put the centre of circle at (0,0) with radius 1
    This function also provided in example code */
    d1=get_random_in_range();
    d2=get_random_in_range();
    if (d1*d1 + d2*d2 <= 1) count ++;
  }
  printf("%i,%i\n", count, ITS);
}
```

and will output something like `7833,10000` for the number inside the circle and the total number thrown, respectively. That corresponds to an estimate for π of 3.13320, which is wrong, but in the right ballpark. Increasing to 10 million iterations we get something like 3.14165 so we do indeed seem to be converging to the correct answer.

Now we add the basics of parallelism we learned above, namely the Init and Finalize calls, to get

```c
#include <stdio.h>

#include <mpi.h>

#define ITS 10000
int main(int argc, char **argv){
  int count = 0, index;
  double d1, d2;

  MPI_Init(&argc, &argv);

  seed_rng();
  for (index=0;index<ITS;++index){
    /* random numbers between  1 */
    d1=get_random_in_range();
    d2=get_random_in_range();
    if (d1*d1 + d2*d2 <= 1) count ++;
  }
  printf("%i,%i\n", count, ITS);

  MPI_Finalize();

}
```

This code runs, but gives us something a bit silly, namely

```
7805,10000
7805,10000
...
7805,10000
```

i.e. the exact same result on all processors.

The reason is hidden away inside the bit we skipped over, because it was "irrelevant". *Lots of things you don't think of are different in parallel. Be very cautious calling something irrelevant based on experience in serial code.* The problem is seeding (setting up) the random number generator. We used the usual standard approach, seeding with the system clock to get a different answer run to run, as

```c
#include <time.h>
void seed_rng(){
  time_t t;
  srand((unsigned) time(&t));
}
```

But all of our processors share this clock, so we'll generally get the same seed[3] and so the same random sequence.

We want to get something which is unique to each processor to ensure[4] different seeds. In MPI, each processor is assigned a rank, which is an integer value. The "first" processor is given a rank 0 and the rest are assigned in order up to the number of processors. The function `MPI_Comm_rank` returns this value.

We also need the idea of an MPI communicator at this point. This is essentially a list of processors. Here we will always be using the default value `MPI_COMM_WORLD` which means all the processors you're running on. To get the rank value, we use (remembering again that in Fortran we have an extra, final *ierr* parameter)

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

- `comm` - An MPI communicator. Generally just the constant `MPI_COMM_WORLD`
- `rank` - Pointer to integer to be filled with the unique rank value for this processor in communicator comm. If you have more than one communicator a processor can have a different rank in each one.

Remember that Fortran is pass-by-reference, so where C requires a pointer, Fortran can use a plain INTEGER So the Fortran signature is, instead

```fortran
MPI_Comm_rank(INTEGER comm, INTEGER rank)
```

[3]Another place to be extra careful - because there can be tiny differences in timings between processors, especially processors on several compute nodes, you might in fact NOT get the same seed in your test runs, and never spot this potential bug.

[4]If you can spot why in rare cases in the code below we might still get the same seed, well done! Now fix it!

- `comm` - An MPI communicator. Generally just the constant `MPI_COMM_WORLD`
- `rank` - Integer to be filled with the unique rank value for this processor in communicator comm. If you have more than one communicator a processor can have a different rank in each one.

Using the rank, we seed the random number generator as

```
#include <time.h>
#include <mpi.h>
void seed_rng(){
  time_t t;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  srand((unsigned) time(&t) + rank);
}
```

and compile and run this program. The results are something like

```
7860,10000
7814,10000
7856,10000
7850,10000
7814,10000
7889,10000
7839,10000
7934,10000
7803,10000
7855,10000
7869,10000
7820,10000
7771,10000
7829,10000
7869,10000
7812,10000
```

which is much better.

Now in serial we parse and add up those results to get an estimate of $\pi = 3.13710$ just as if we'd run 16000 iterations on one processor.

We know how to find the rank of our current processor, but how do we find out how many there are in total? This uses the command `MPI_Comm_size` which again works on a communicator and returns an integer.

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

- `comm` - An MPI communicator. Generally just the constant `MPI_COMM_WORLD`
- `size` - Pointer to integer to be filled with the number of processors in comm. Since ranks start from 0, this will be 1 greater than the maximum rank

Once again, in Fortran both `comm` and `size` are plain `INTEGER` and we have a final, `INTEGER` parameter for the return code.

2.3.1 The Story So Far

So far, we've see how to do an embarassingly parallel (explained in a moment) problem in MPI[5]. Every processor works independently on its own problem and then the results can be synthesised into a single answer. We haven't learned anything about communications yet at all. However this can already be useful.

This sort of divided work can be used to do "task-farming" - splitting work over many processors with one in charge of dispatching. We talk about a simple model for this in Chapter 6. Note that this is the main use of MPI in Python, via MPI4Py or a similar library. MPI with communication is generally a bad idea in Python code, but the dispatcher model is fairly useful. If you'd like some help with this sort of approach, let us know by email.

For general MPI programs though, we need to learn how to communicate, so read on!

[5]Well, mostly in parallel anyway

Chapter 3

Basics of Communication

3.1 Actually Communicating

MPI communications split into two general sets:

- Point to point communications

  This processor talks to *that* processor

- Collective communications

  Every processor communicates with *every other*[1] processor

Since these groups of functions, and the purposes they serve, are quite distinct, we treat them separately over the next two chapters. We think collectives are a tiny bit easier to start with, so we start with them.

3.2 Collective Communications

The core set of collective communications are:

- Reduce - Combine data from all processors in specified way onto a single root processor

  Allreduce - Combine data from all processors in specified way and give result to all processors

- Bcast - Broadcast same data from root processor to all others

- Scatter - Send different data from the root processor to all other processors

- Gather - Receive different data from each processor onto root processor

  AllGather - Receive different data from each processor and put the result on all processors

[1]Every other processor in the communicator you're using. But you can write many MPI programs and never use any communicator except `MPI_COMM_WORLD`, so don't worry about the distinction for now.

- AllToAll - Send different data between every pair of processors at the same time. *Don't use this unless absolutely necessary. This is pretty much the slowest MPI communication routine there is.*

At the low level, MPI communication is dumb: it just sends and receives bytes. To send useful messages, we need to be able to indicate that something is an integer, or a floating-point number etc.[2] MPI has built in types for the basic types, and it is possible to create custom ones representing e.g. a structure with multiple fields. For more on this, see our Intermediate MPI (coming soon).

The basic types include:

- `MPI_INT` - C Integer
- `MPI_INTEGER` - Fortran Integer
- `MPI_FLOAT` - C single precision
- `MPI_REAL` - Fortran single precision
- `MPI_DOUBLE` - C double precision
- `MPI_DOUBLE_PRECISION` - Fortran double precision
- `MPI_BYTE` - Single byte value
- Many, many others

Do note that there are slightly different names in C and Fortran. If you use the wrong one things will probably compile and work, but there's a risk of that changing over time or on different computers (the sizes aren't forced to be the same, for example, they just nearly always are).

The real power of MPI types is for reduction operations (taking several values and combining them into one), where MPI not only sends the data, but does something with it too. This means it needs to know details of the particular system's data layout etc. MPI can be used between computers that represent data differently, and MPI then has to take care of transforming the data types. Custom data types also allow fancy things like reshaping an array in flight (keeping the total number of elements the same). Once again, this is stuff you should know of, but shouldn't need for a fair time, if ever.

3.2.1 MPI Gather

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

[2]This isn't strictly essential for simple comms- your program could simply interpret the bytes as what is knows should be sent, but this loses all the power of type checking and disallows the things discussed in the next paragraph.

- `sendbuf` - Data to send to root
- `sendcount` - Number of elements in sendbuf
- `sendtype` - Type of data in sendbuf
- `recvbuf` - Array into which to receive data (should be recvcount * nproc elements long)
- `recvcount` - Number of elements to receive from a single processor (unless using custom types for sendtype and/or recvtype should equal sendcount)
- `recvtype` - Type of data in recvbuf (generally same as sendtype)
- `root` - Rank of processor to receive data
- `comm` - MPI communicator (usually `MPI_COMM_WORLD`)

This looks a bit complicated, but in theory its pretty simple. Each processor sends `sendcount` elements to the processor whose rank equals root. On root you need to allocate an array of length `sendcount * nproc` elements to hold the results. Despite this `recvcount` should be the number of elements received from each processor, not the total number. The results from rank 0 go into (0:sendcount-1). The results from rank 1 go into (sendcount : 2 * sendcount -1) etc. etc. We use `MPI_Comm_size` to get the number of processors, `nproc`, so we can get this array ready to receive into.

We can use Gather to make our darts program a bit more elegant. We change the code to

```c
int main(int argc, char **argv){
  int count = 0, index;
  double d1, d2;
  int rank, nproc;
  int *count_global;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  seed_rng(rank);
  for (index=0;index<ITS;++index){
    /* random numbers between  1 */
    d1=get_random_in_range();
    d2=get_random_in_range();
    if (d1*d1 + d2*d2 <= 1) count ++;
  }
  count_global = malloc(sizeof(int)*nproc);
  MPI_Gather(&count, 1, MPI_INT, count_global, 1, MPI_INT, 0,
      MPI_COMM_WORLD);
  if (rank == 0) {
    for (index = 0; index < nproc; ++index) {
      printf("%i,%i\n", count_global[index], ITS);
    }
  }
  MPI_Finalize();
}
```

We've made a few changes to the code. We now get the rank inside main and pass it to the seed function to avoid doing this twice. We get the number of processors using `MPI_Comm_size`. We only print our answer on rank 0, which prints the values from each processor, so we get the same output as before.

3.2.2 Reduction

We could add up the the numbers once we've collected them, and print one final result, but there's a better way - we can add them all up as part of the collective operation, using *reduction*. The MPI layer combines data using the operation you specify, before passing it on. Here we want to use `MPI_SUM` to sum values together.

As well as making the code a bit clearer (more self documenting), there's one big reason to let MPI handle this, which is that is can use a better algorithm. We gather all of the numbers onto rank 0, which means all data has to be sent to the same processor so the messages have to queue. We have one message per processor to receive. And finally, on rank 0 we have to sum the elements one by one. This is roughly the following picture.

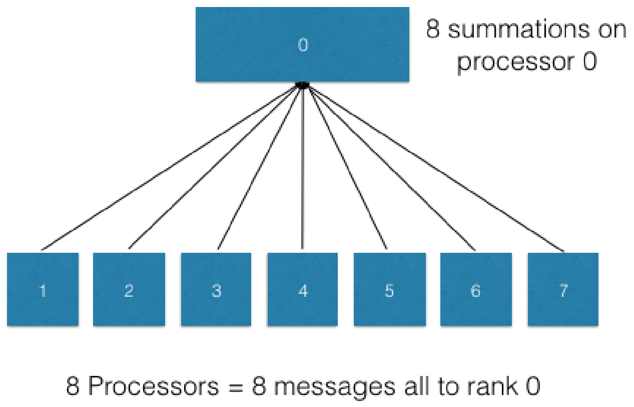
0
8 summations on processor 0
1
2
3
4
5
6
7
8 Processors = 8 messages all to rank 0

Figure 3.1: Summation done manually

MPI does not guarantee how reduce is implemented, only that it will give the expected result. It can[3] implement something more efficient than the gather and sum. In the case of a sum, it is transitive, i.e. $(a + b) + (b + c) = a + b + c + d$ regardless of the bracketing. One possibility is to build a tree of processors and do partial sums. This reduces the number of messages and distributes the effort of summation across processors. Also, no processor ever holds more than two data items, which can be useful if memory is short or items are large. The next picture shows how this works.

[3]Note, "can", there is no guarantee of anything

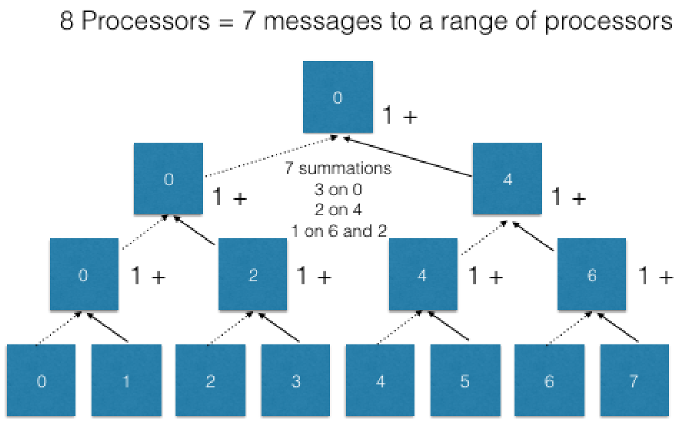

Figure 3.2: Summation done as a tree

This is still far from perfect - processor 0 has more work and some processors have none at all, but it is better. The reduction in message number gets more significant the more processors are involved.

Remember that this is a way reduction CAN be made more efficient than a gather and sum. MPI may do nothing of the sort. But in general, MPI implementations are written by good programmers, and can be made specific to a given machine, so generally you want to use the built-ins rather than rolling your own. In particular, the bigger a machine, the more likely they have some special implementation, and the more you lose by not using it.

The reduce function is

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

- `sendbuf` - Pointer to memory holding source data from the current processor
- `recvbuf` - Pointer to data to receive result into. Only populated on processor root
- `count` - Number of elements in sendbuf
- `datatype` - Type of data in sendbuf
- `op` - MPI_Op type describing reduction operation
- `root` - Rank of root process to receive data
- `comm` - Communicator describing processors over which to perform reduction

The common built-in operations are

- `MPI_MAX` : Maximum element

- MPI_MIN : Minimum element
- MPI_SUM : Sum all elements
- MPI_PROD : Multiply all elements
- MPI_LAND : Logical AND all elements
- MPI_LOR : Logical OR all elements
- MPI_BAND : Bitwise AND all elements
- MPI_BOR : Bitwise OR all elements

It is also possible to create custom ones as well but that is unusual and a bit tricky.

A closely related operation is the MPI_Allreduce which is a reduce putting the result onto all processors. This has a much higher comms requirement than plain reduce, so should be used only when needed.

3.2.3 Dart program with a reduce

Using reduce, the program gets rather simpler, since we have to do less manually:

```
int main(int argc, char **argv){
  int count = 0, index;
  double d1, d2;

  int rank, nproc, count_global;
  MPI_Init(&argc, &argv);}
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  seed_rng(rank);
  for (index=0;index<ITS;++index){
    /* random numbers between  1 */
    d1=get_random_in_range();
    d2=get_random_in_range();
    if (d1*d1 + d2*d2 <= 1) count ++;
  }
  MPI_Reduce(&count, &count_global, 1, MPI_INT, MPI_SUM, 0,
      MPI_COMM_WORLD);
  if (rank == 0) {
    printf("%i,%i\n", count_global, ITS * nproc);
  }
  MPI_Finalize();
}
```

3.3 MPI Barrier

The final really common collective operation seems to not really be communication at all - it is the MPI_Barrier. This blocks (here meaning forces to wait) all processors in a communicator until they have all entered the barrier, after which the call returns and the program can continue. This is useful to allow synchronisation of code, so that you know all the processors are in the same place. The signature is

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

- `comm` The communicator to block

where as usual in Fortran there is also the final error parameter.

Barriers are often used after sections where only one processor (usually root) is doing something special. That way, you can be sure the other processors aren't racing off and continuing their work, and you don't need to worry as much about what might go wrong. It's also really useful for debugging, where you don't want errors from later parts of the code to be appearing when the actual problem is with one stuck process earlier on.

Chapter 4

Point to Point Communication

4.1 Overview

The collectives we just discussed are very useful, but the most common form of MPI comms is point-to-point. Here one processor, the *source* sends a message to another processor, the *receiver*. As well as the send command, the receiver must make a matching receive command. *Matching* will be discussed in a bit.

The most common problem you'll likely encounter in MPI programming is *deadlocking*, where no commands can complete. This usually happens because a send occurs without a matching receive, or a receive happens for a message that hasn't been sent. This locks up because the default send and receive commands are blocking, so control doesn't return until the commands have "completed"[1] We'll show in a minute the usual problem, where a send can't complete until the receive line is reached, and the receive line can't be reached until the send completes, the programming analogue of opening a crate with the crowbar inside it.

The basic send command is

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

- `buf` a buffer containing the data
- `count` number of elements to send
- `datatype` type of the elements to send
- `dest` which rank to send to
- `tag` integer code that identifies this message. Has to match in receive
- `comm` an MPI communicator

[1]We'll come back to exactly what "completed" means, because it doesn't quite mean that the send-receive has finished

and the receive is

```c
int MPI_Receive(const void *buf, int count, MPI_Datatype datatype, int
    source, int tag, MPI_Comm comm, MPI_Status * status)
```

- `buf` a buffer to hold the received data
- `count` number of elements to receive
- `datatype` type of the elements to receive
- `dest` Rank of the source, i.e. which rank data comes from
- `tag` integer code that identifies this message. Has to match in send
- `comm` an MPI communicator
- `status` object containing information about the message.
  `INTEGER, DIMENSION(MPI_STATUS_SIZE)` in Fortran

4.1.1 Matching Sends and Receives

The rules for what matching means are

1. The `dest` parameter to `MPI_Send` is the rank of the receiver
2. The `source` parameter to `MPI_Recv` is the rank of the sender
3. The `tag` to both `MPI_Send` and `MPI_Recv` is the same

There are also special values you can use in place of `source` and `tag` in `MPI_Recv` commands. These are `MPI_ANY_SOURCE` which means a message from any source will be accepted, and `MPI_ANY_TAG` which means a message with any tag will be accepted. Usually you want to use only one of these at once, or it gets hard to tell messages apart.

4.2 A simple example - the ring pass

The simplest example of MPI point to point comms is called a ring-pass. Each processor sends a message to its neighbour on the right, and receives from its neighbour on the left. This doesn't have a lot in common with real MPI codes and has some rather (significant pause)... interesting pathologies. But its the standard first code, so lets have a look. The first attempt at this code is

```c
int main(int argc, char **argv){
  int rank, nproc, rank_right, rank_left, rank_recv;
  MPI_Status stat;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

```
 7
 8    rank_left = rank -1;
 9    /*Ranks run from 0 to nproc-1, so wrap the ends around to make a loop*/
10    if(rank_left == -1) rank_left = nproc-1;
11    rank_right = rank + 1;
12    if(rank_right == nproc) rank_right = 0;
13
14    MPI_Send(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
15    MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD, &stat)
        ;
16
17    printf("Rank %i has received value %i from rank %i\n", rank, rank_recv,
18        rank_left);
19
20    MPI_Finalize();
21  }
```

and when we compile and run, we get

```
Rank 5 has received value 4 from rank 4
Rank 1 has received value 0 from rank 0
Rank 2 has received value 1 from rank 1
Rank 3 has received value 2 from rank 2
Rank 4 has received value 3 from rank 3
Rank 14 has received value 13 from rank 13
Rank 0 has received value 15 from rank 15
Rank 6 has received value 5 from rank 5
Rank 7 has received value 6 from rank 6
Rank 8 has received value 7 from rank 7
Rank 9 has received value 8 from rank 8
Rank 12 has received value 11 from rank 11
Rank 13 has received value 12 from rank 12
Rank 15 has received value 14 from rank 14
Rank 10 has received value 9 from rank 9
Rank 11 has received value 10 from rank 10
```

It seems to work, and all the processors seem to print the right thing. *But there is a major problem, which might lead to some surprises*. On most computers it'll probably work, but on clusters it'll usually deadlock. Why is this?

The problem is lines 14 and 15. Remember that sending and receiving are blocking? All the processors hit the send command in line 14, and block there, so the matching receive is never reached. Deadlock is actually the *correct* outcome. So why does it work on a laptop/desktop?

The answer is that we we a bit shifty about what "blocking" means. `MPI_Send` isn't *required* to block until a message is received, only until you can safely reuse `buf` again. Often this is when the data has been copied into an MPI internal buffer. There's a good chance that this happens, and the program continues to the receive line. You can't count on this though, so you shouldn't rely on this in real codes.

There is a variant command, `MPI_Ssend` which is guaranteed to block until the message is received. Try this in the example - it should deadlock every time.

Similarly, `MPI_Recv` only has to block until `buf` contains the correct received value. This isn't quite the same as blocking until the receive has completed, which can happen only once the send is completed. You can be sure that the send is complete once the matching receive has *completed* (remember, this doesn't just mean the program has moved to the next line). Usually you don't actually care *when* a send completes, just that it has properly done so.

There's lots of ways we could fix the example code block. Here we're going to use the ring-pass idea. Rank 0 will do its send and then a receive, while all the other ranks receive and then send. The data will then travel in a "wave" through the processors, 0→1→2→... Note that this is a terribly inefficient way of doing real things, because processor n wont get its data until all (except 0) of the lower ranked ones have. I.e. the comms are back to being serial, one after another.

To make this change we swap lines 14 and 15 from

```
MPI_Send(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD, &stat)
    ;
```

to

```
if (rank == 0) {
  MPI_Ssend(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
  MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD, &
      stat);
} else {
  MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD, &
      stat);
  MPI_Ssend(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
}
```

Now the output is[2]

```
Rank 1 has received value 0 from rank 0
Rank 2 has received value 1 from rank 1
Rank 4 has received value 3 from rank 3
Rank 5 has received value 4 from rank 4
Rank 6 has received value 5 from rank 5
Rank 7 has received value 6 from rank 6
Rank 3 has received value 2 from rank 2
Rank 8 has received value 7 from rank 7
Rank 9 has received value 8 from rank 8
Rank 10 has received value 9 from rank 9
Rank 11 has received value 10 from rank 10
Rank 12 has received value 11 from rank 11
Rank 13 has received value 12 from rank 12
```

[2]If you're wondering why these lines aren't strictly in order, read on

```
Rank 14 has received value 13 from rank 13
Rank 0 has received value 15 from rank 15
Rank 15 has received value 14 from rank 14
```

This code is working as expected, and we've used `MPI_Ssend` so there's no trickery here, this will work on any machine.

Note that the print statements are nearly in rank order, much more so than for the "simple" code. This is because of the wave propagating through the ranks. The order isn't perfect because print isn't guaranteed to show in the order the processors try and print. As we said above, this ordering really impacts performance though!

4.3 Sendrecv

Note though that this code doesn't work on a single processor! Rank 0 has to send and receive at the same time. This kind of both at once problem is fairly common in MPI codes, and luckily there is a command to help!

`MPI_Sendrecv` is a send and a receive glued together, so its signature is pretty long, but it really is just the two commands glued together:

```
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype
    sendtype, int dest, int sendtag, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
    MPI_Status *status)
```

As long as both parts of the command can complete, the whole command will succeed and never deadlock.

Our example code gets much simpler using this, as we replace lines 14 and 15 with just

```
  MPI_Sendrecv(&rank,      1, MPI_INT, rank_right, 100, &rank_recv,1,
     MPI_INT, rank_left , 100, MPI_COMM_WORLD, &stat);
```

The output of this has all the ranks mixed up again, and we no longer have the propagating wave of data. Performance is much better, the code works seamlessly on any number of processors and there's no risk of deadlock. For all of these reasons, this command is the preferred solution to this kind of problem.

4.4 The Story So Far

To recap, we now know how to

1. Combine data from all processors
2. Send data from one processor to another
3. Deal with the simplest class of deadlock for "send right-receive left" problems

This is plenty to get started, so on to some actual parallel code!

Chapter 5

Domain Decomposition Models

5.1 Motivation

In a working MPI program you effectively want to make your program act as though there was one very powerful processor, and to ignore, as far as possible, that the work is being done by many processors. Remember "Embarrassing Parallelism"? That's where you simply don't care that there are many processors, because once you've assigned them their tasks, they work alone, never needing to communicate with each other at all.

As we said though, that's rarely the case in practice, and we usually need to find a way to split up (decompose) our problem to make the tasks as independent as possible. We want to minimise the amount of MPI code we need to write, and minimise the amount of communication we need to do, so that most of the time our processors are happily working away.

How we can, or should, do this depends very much on the problem at hand. In this chapter, we're mainly going to focus on the most common situation, where we can split up the problem so that each processor does the same tasks on different data. In the next chapter, we're going to talk about the other option, where one processor is special.

5.2 What is Domain Decomposition

Domain decomposition means splitting up the domain ("space") of your problem into chunks, and working on each chunk separately (usually one per processor). Here, "space" often means physical space, dividing into regions, but it can also mean momentum, energy or anything where you can work on different values independently.

When decomposing in space, you usually have a large array covering the entire domain, and divide it into pieces, giving one to each processor, and use MPI messages to send data that a processor needs from others around. In particular, you're usually doing something to keep processors "in sync", because ordinarily each must proceed one step (timestep, iteration etc) and can only go to the next step after some communications.

To work well, domain decomposition requires that your algorithm be *local.* This means the updates to one element depend only on those nearby.

5.2.1 Algorithms which Work Well

There are, happily, lots of algorithms like this. For instance, numerical derivatives require anything from one neighbouring cell (giving Euler's method, https://en.wikipedia.org/wiki/Euler_method) up to an arbitrary but fixed number (the entire family of Runge-Kutta methods https://en.wikipedia.org/wiki/Runge-Kutta_methods). All of these are examples of finite difference schemes, where a finite number of grid cells give enough information to approximate a local derivative.

Finite-volume (roughly flux conservation) and finite-element methods are also amenable, as are many iterative matrix inversion schemes for certain inputs[1], (Jacobi https://en.wikipedia.org/wiki/Jacobi_method , Gauss-Seidel https://en.wikipedia.org/wiki/Gauss-Seidel_method (with extra care)).

5.2.2 Algorithms that Don't Work Well

Any algorithm without locality wont work well. This doesn't mean they wont work at all, but they can be very difficult to make work well. For instance, direct matrix inversion (e.g. Gaussian Elimination) or inversion of general matrices rely on all other elements of the array. Fourier transforms also cannot be done piecewise, which means spectral methods are mostly unpromising.

5.2.3 Nice Systems Without a Grid

Note that you don't need to have a grid to be able to domain decompose. If you're calculating pairwise interactions between particles you can consider only some number of nearest neighbours, which is again a form of locality. Tree based schemes like Barnes-Hutt find ways to extend this by using less data from further away[2]. Some mesh free schemes can work too (Mesh Free Galerkin etc.)

However all of these are generally messier and not as illuminating, so here we're focusing on a simple grid based scheme.

Memory and Arrays

Behind the scenes, computer memory is one dimensional. Memory locations have an address, which is a single number.[a] If your program uses a 2-D or more array, this has to be somehow mapped into 1-D. C programmers may already be familiar with this, as really flexible multi-dimensional arrays are usually created from a 1-D array and a function to work out the proper location. For our purposes, what matters is that two elements that are beside

[1]Strictly, Jacobi iteration depends on every other element in the array, but for e.g. a tri-diagonal matrix only local elements are non-zero

[2]Roughly Barnes-Hutt averages far away particles together - more coarsely the further away you are

each other in the real array may not be beside each other (contiguous) in memory.
Programming languages differ by whether they keep the rows, or the columns contiguous. In fact, C and Fortran take the opposite conventions on this! See https://en.wikipedia.org/wiki/Row-_and_column-major_order for some details. For now, just bear in mind that taking a section of an array will not, in general, give you a single block of memory, and that this can matter when you're splitting up, or patching together, arrays.

[a]This is greatly simplified, but if you know enough to know it is wrong, you should already understand memory layouts.

5.3 Basic Principle

In practice domain decomposition for grid based schemes is rather simple. Rather than having one big array, dimensions (`nx`, `ny`) you have `N` arrays on `N` processors each (`nx_local`, `ny_local`). Remember that the big array doesn't actually exist anywhere. Also, remember that the local sizes don't have to be the same in general, but things are much easier if your sub-domains meet corner to corner (in 2-D think of dividing with a single line running all the way from one side of your domain to the other - you can vary the distance between the lines, but they always run straight all the way across).

Note that there are parallel schemes called Partitioned Global Address Space schemes that do have the whole big array existing. These are less common than MPI and its often harder to write performant code that way.

5.4 A "Simple" Example

Imagine updating each element of your array using

```
temp[i][j] = (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1])/4.0;
```

where once you've filled every element of `temp` you copy it back into `u`. This scheme simply smooths `u` in space over the 4 points immediately adjacent (notice `u[i][j]` doesn't appear). This uses only the current point and the points immediately adjacent, so it's definitely local.

Actually, the scheme above can also be used to iteratively solve the heat equation using the Jacobi method. This is a static problem (not changing in time) where we're iterating towards the correct solution, not moving on in time. We're going to use this as an example. The following figure (5.1) shows the heat equation set up so that we have a square of metal, with the bottom and left edges being held at a fixed, cold, temperature, and the top and right edges held at a fixed, higher temperature. The solution is a heat distribution as in the bottom panel of the figure, with smooth curves showing the lines of equal temperature, with pale blue coldest and red hottest.

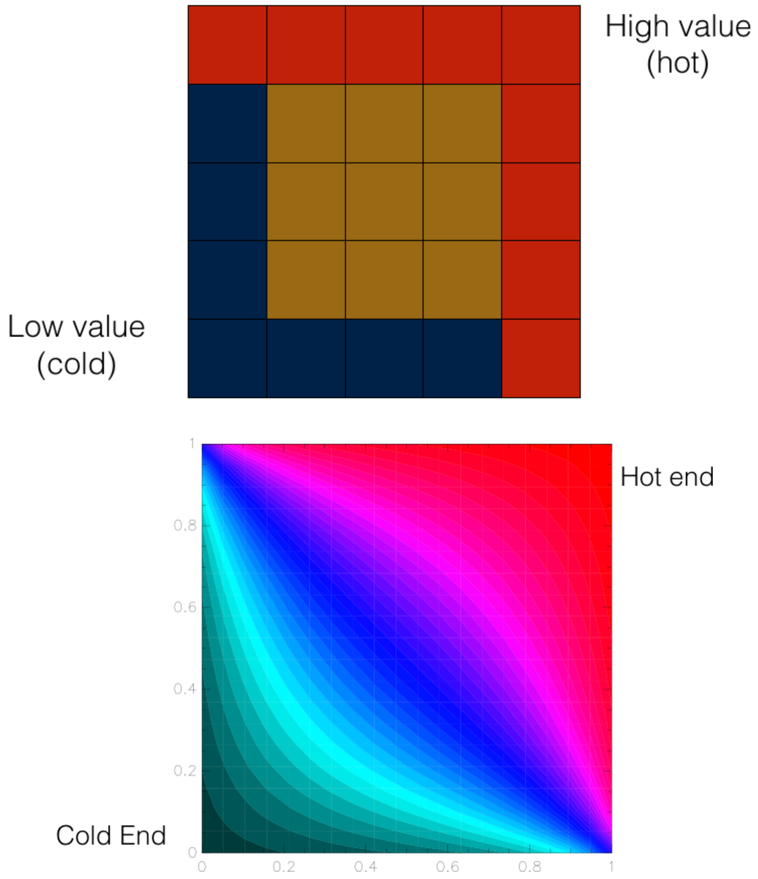

Figure 5.1: The Heat equation - on a grid and the expected result

5.5 Splitting the Domain

5.5.1 Ghost or Guard Cells

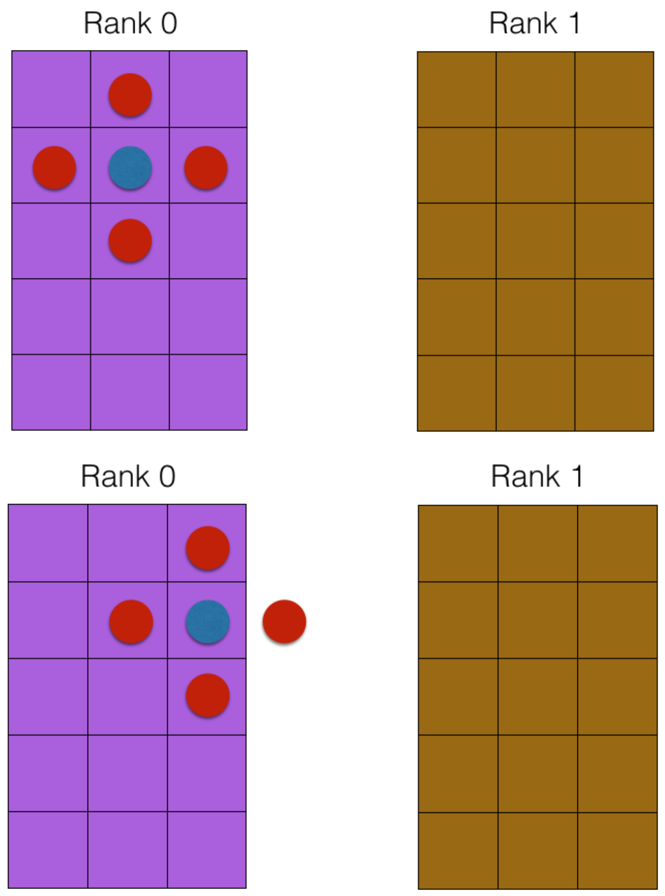

Figure 5.2: Solving the Heat Equation - the blue point is being solved. This needs the red points. In the top case these are all there in the array. In the bottom case, we need data from another processor.

Since the updates to each element require all of the neighbours, as shown in the figure (5.2) above, cells at the edges of our chunks or sub-domains need some extra information - they need the values in cells on other processors. Since we don't want to have MPI messages all over our code, we need to have this information on our local processor, so we have special cells around the edges, called *ghost* or *guard* cells.

On the true edges of our domain, these cells contain the real boundary values - on every other edge they contain a copy of the data in that real cell. In both cases, we never update these cells locally using our algorithm - we either change the external boundaries due to some changing boundary condition, or we fill the cells with the values from another processor. Usually, after each iteration is done, we send some messages to update the ghost cell values. Otherwise, we only ever read from these cells: i.e. as far as the local processor is concerned, these cells are always just boundary cells.

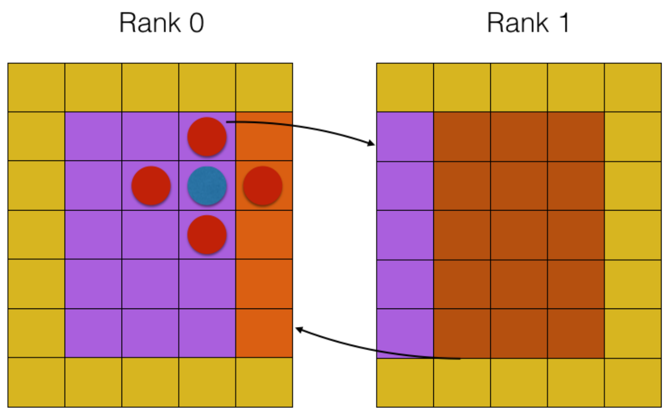

Figure 5.3: Adding Ghost cells. The yellow cells are ghost cells, filled from the neighbouring processor. The orange cells on Rank 0 are copied from Rank 1, the purple on Rank 1 are similarly copied from Rank 0

Updating the ghost cells is a perfect use of `MPI_Sendrecv`. Remember - we always want to minimise the number of MPI messages (because there's overheads in sending them), so we want to do one Sendrecv on each edge, not one per cell!. In 1-D we have 2 edges to deal with; in 2-D we have 4 and so on.

5.5.2 Working out How to Divide the Domain

So, we know how we want the domain to be divided, but before you can actually write a code you have to work out how to implement this.

Load Balancing

We mentioned above that the sub-domains don't have to all be the same size. For a lot of problems, every cell of the domain causes exactly as much work as every other, so for maximum efficiency we want to divide as evenly as possible. If this isn't true, then you have the problem of load-balancing: finding a way to divide the domain so that each sub-domain has the same amount of work. This is not usually a lot of fun. You have to find a way to quantify the work to be done, and then find a way to divide it. In 1-D the latter isn't too tricky, but for a 2-D or more domain it gets very tricky and actually optimising the problem leads to domains that are not only the same size, but aren't even rectangular!

In 1-D, splitting the domain is easy. You divide the number of cells by the number of processors, and use that. If the result isn't an integer you can refuse to run the code, insisting the user restart with a better number, although that is quickly limiting and pretty annoying. A better option is to allow the size of domains to vary a bit so it fits (allow one more or less cell on some blocks). This can affect performance, but as long as the domains aren't very small it usually doesn't matter much. If it really does matter, your only real option is to resize the domain to whatever you think is the most suitable size (largest divisible domain smaller than that asked for, or smallest possible larger than the one asked for).

In 2-D you have a trickier problem, since you have to decide how to arrange the processors into a grid. You can split in only one direction, but this often doesn't perform well, and limits the maximum number of processors you can use (1 per cell in the split dimension). We discuss why it doesn't perform too well in the next section.

So, first we have to work out how to arrange N processors into a grid of `(L , M)`, so that $L * M = N$ and L and M are somehow optimal. In 3-D we need one more split, so a grid of `(K, L , M)` with the same restrictions. You can calculate this yourself, but it's easier not to worry about for now, unless your domain is oddly shaped.

5.5.3 MPI_Dims_create

MPI provides a function to do the splitting for the general case, where your whole problem is "pretty much square", namely `MPI_Dims_create`. This has the signature

```
MPI_Dims_create(int nnodes, int ndims, int dims[])
```

- `nnodes` - Number of processors (from `MPI_Comm_size`)
- `ndims` - Number of dimensions that you want to decompose in (2D, 3D etc.)
- `dims` - Returned array containing numbers of processors in each dimension

Called with `ndims` = 2, `MPI_Dims_create` will give you an (L, M) decomposition of your N processors which is as close to square as possible i.e. both L and M are as close as possible to SqrtN. This isn't guaranteed to be optimal, but its one less thing to worry about. If your domain is really long and thin, this definitely isn't optimal, but it will always work.

5.5.4 Domain Shapes

In any distributed system you want to spend as much time computing as possible and as little time communicating. Since you compute over the volume of a domain, and communicate along the edges, this means you want a lot of cells inside the domain, and as few as possible around the edges. Formally, you want to minimise the surface-area to volume ratio of your domains. This means keeping the subdomains as square as possible.

Surface Area to Volume Ratio

In 2-D, you might recall that a circle is the absolute best surface area to volume ratio. Since we're talking about rectangular grids, we're restricted to only horizontal and vertical lines (i.e. some sort of rectangle). Non rectangular grids do exist, but frankly they're a nightmare.
The surface area of a rectangle of size a by b is its circumference, namely $C = 2(a + b)$. Its volume is $V = a * b$. If we start with $a = b$ and fix C it's not hard to show that the volume can only decrease, so this definitely maximises the ratio (let $a \rightarrow a + n$ then $V \rightarrow V - n^2$). So the best shape is as square as possible.

With that in mind, you can work out the optimal `(L, M)` split for your actual domain, of size `(x, y)`. You want the ratio of number of processors to number of grid cells to be as equal as possible in the two cases. Generally, this means minimising $x/L - y/M$ subject to $L * M = N$ AND x/L and y/M are both integers. That is... not so easy. In the general case, any solution is impossible (imagine x and y both prime), and the only possibility is to let the domains be different sizes. Even when exact division is possible, minimising may require trying every combination of L and M. Once you allow differently sized domains,the problem changes to one of load-balancing and you must quantify the load in a grid-cell and find some way of making a 1-D load curve out of your 1, 2 or 3 dimensional domain.

5.6 Arranging the Processors

So, one way or another, we now have a division of processors into an L x M grid. Now we need to divide our array(s) in a way consistent with that.

5.6.1 Matching Edges

At simplest level, what you're worried about here is correctly matching the edges. Fig .. shows the entire array, and two ways we could distribute it across 4 processors. Processor 1 communicates its left-hand edge to the processor logically on its left, and its bottom edge to the processor logically below it. In the first case, this works perfectly. In the second case, processor 1 actually has the bottom left corner, so shouldn't send the bottom or left edges *anywhere*, so we end up with nonsense.

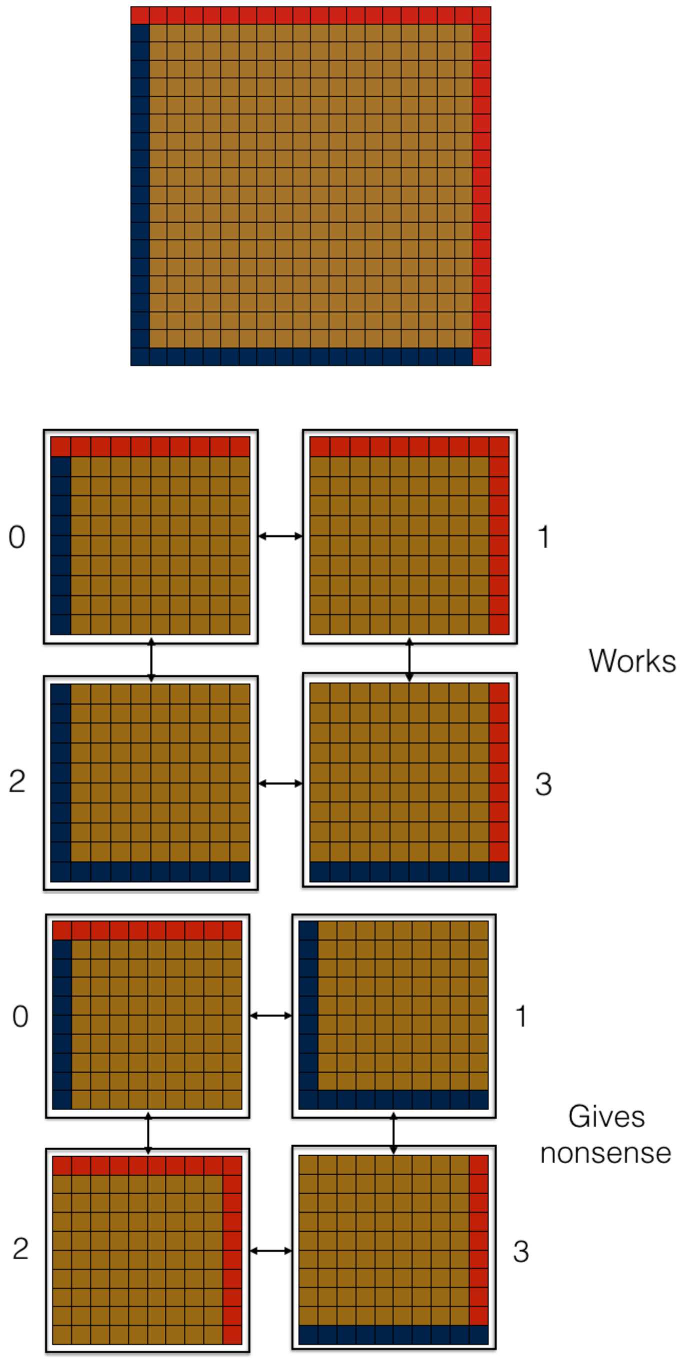

Figure 5.4: Dividing a domain across processors. The middle panel shows how to give each processor a section so that things work. The bottom panel shows how to make nonsense.

5.6.2 Real World Problems - Fabric Topology

In real MPI codes, we also have to think about how the actual computer works. Sometimes, some processors are "nearer" (in some way) to your current processor than other processors in the communication network.

Incidentally, being physically nearer is part of this - imagine a 3GHz processor. Each instruction takes 1 ns. This is the time taken for light in a vacuum to travel about 33 cm. Electrical signal in cables, or light in optical cables is usually travelling at 0.3 to 0.8 light speed.[3] So physical processors 1 m apart need between 4 and 10 ns just for the light-travel time.

Since we're only communicating with neighbouring processors, it makes sense to arrange processors so that communication is with processors nearby. MPI provides some functions to help with this, which *may* take account of all sorts of properties of the underlying computer fabric, such as which cores may be on the same node and which nodes are physically close. Note that this is all possible but not guaranteed. MPI implementations can do all sorts of optimisations, or they may not.

The helper functions fall into two families:

- `MPI_Cart_*` - Several routines for dealing with Cartesian topology (what I just described)
- `MPI_Graph_*` - Several routines for dealing with arbitrary topologies described as graphs

These are very useful, but add 3 or 4 new commands and quite a bit of complexity so we're going to ignore them here, except to tell you that they exist. They will be covered in our (future) intermediate MPI workshops.

5.6.3 Manual Calculation

In these notes we're just going to arrange our processors into a simple grid, and number them from bottom left to top right. We're going to go through the first index we got from `MPI_Dims_create` first, and call that x. Once we run out of processors in this direction, we'll increment the second dimension, y.

Modulo Arithmetic

Doing the decomposition calculations yourself isn't hard, it just requires a bit of modulo arithmetic. If this is unfamiliar, think of converting from Minutes into Hours and Minutes. Every block of 60 minutes is an hour, so we need to know how many whole hours fit, and how many minutes are left over. The equations below do precisely this, but for rows and columns. Our y co-ordinate is how many whole rows fit into our rank. The x co-ordinate is how many cells are left over. So we calculate the whole rows (hours) first.

[3] In fibre optics this is a combination of refractive index and longer path length, due to propagation in a wave guide.

Then we take this away, and see how many cells into that row (minutes) are left.
Say we're looking at 164 minutes (processor with rank 164). We first work out $FLOOR(164/60) = FLOOR(2.733...)$ I.e. the largest integer less than 2.733..., which is 2. So this processor has y-coordinate of 2. Then we have $164 - 60 * 2 = 164 - 120 = 44$ minutes left, i.e. x-coordinate of 44.

In full, the equations in 2-D are[4]

```
coords_y = FLOOR(rank / nprocx)
coords_x = rank - coords_y * nprocx
```

Now you also need to find the ranks of your direct neighbours, since you'll need them to communicate. This is actually a bit easier. From the following image, we can tell that

- Processor to left has rank `1` lower
- Processor to right has rank `1` higher
- Processor upwards has rank `nproc_x` higher
- Processor downwards has rank `nproc_x` lower

[4]Floor means take the largest integer (whole number) which is less than the argument you give it. I.e. FLOOR(100/60) = FLOOR(1.66) = 1. For positive numbers, this means just dropping the decimal part (truncating). For negative numbers, you still want the largest Integer less than the argument. I.e. -2 ¡ -1.66 ¡ -1, so FLOOR(-1.66) = 2. The opposite of FLOOR is CEIL or CEILING, the smallest Integer greater than the argument.

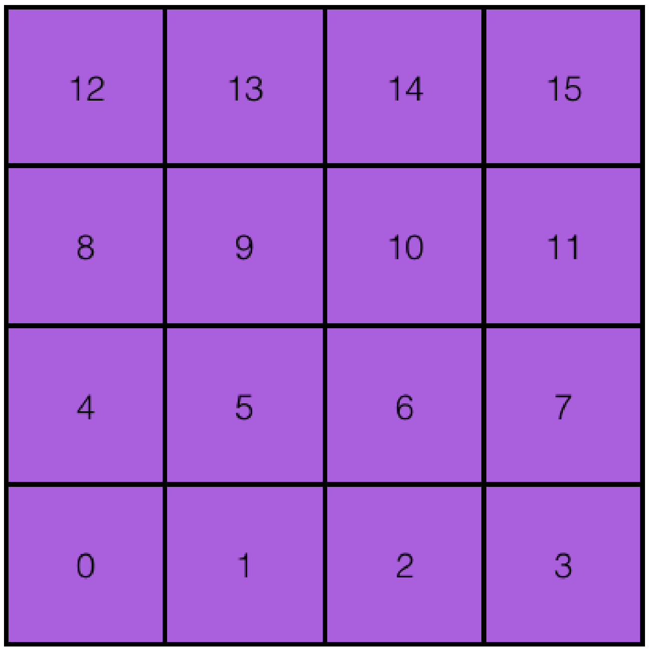

Figure 5.5: Arranging the ranks

5.6.4 Handling the Real Edges

Now we have to deal with the edges of the real domain, i.e. the processors along the outside edges. There's roughly two possibilities here - either we have a real problem with some boundary conditions, and so the values along these edges are fixed or externally specified, or we have periodic boundaries.

Periodic boundaries are one of the nicest boundary conditions. In this case, we just force the values on the right hand edge to wrap around to match those on the left, and the same top to bottom. This can be done exactly, can be physically meaningful, and comes almost for free with MPI code. Rather than make the outside edges special, we just set their neighbours and do nothing else. Processor 3 in the above figure then has a right-hand neighbour of processor 0, and bottom neighbour of processor 15. Then we just run all the communications using the neighbours without worrying about the physical location of any cells.

A quick note - periodic boundaries can occur in real physics (imagine currents in a loop of wire, or waves on a string loop), and can often approximate reality even if not. If you can say "whatever is outside this domain is the same as what's inside" then it's not a bad approximation to suppose anything leaving the bottom would similarly leave the bottom of the not-simulated part of reality above the chunk you're considering, and similarly on all the other edges. This can be a great way of getting decent results from a small piece of what you'd like to simulate, but has to be handled with care. For instance, a wave which fits exactly onto our string loop might appear to work, but one which is half-a-wavelength off will interfere destructively, and cancel itself out.

MPI_PROC_NULL

For real (non-periodic) boundaries, we want to make sure the values they have for their neighbours are special. MPI provides a special constant, a null (not simply the number zero) rank. This is the value `MPI_PROC_NULL`. This lets us tell that those processors have no neighbours, but actually is much more helpful. Passed to any MPI send or receive commands, this value turns the operation into a null operation (no-op), so we don't even need to test for it in these cases. The command is perfectly valid, but doesn't send or receive anything.

For the special Sendrecv command we can use this for either (or both) ranks (either `rank_recv`, and/or `rank_send`) and the relevant half of the command becomes a no-op. If given to only one half, then the other half works as normal.

Note that `MPI_PROC_NULL` is *not* valid as the rank of the root process in a gather, reduce etc commands and will cause *runtime* failure.

5.6.5 Calculating the Real Coordinates

The last step we need is to calculate which part of the "virtual" (not existing itself anywhere in memory) global array a given processor has. That is, we need to divide up our co-ordinates. This is pretty easy since we've done most of the hard bits now.

We're looking for the number of cells in each local bit of the array `nx_local`, as well as the local start and end part on each processor.

```
nx_local = nx / nproc_x
x_cell_min_local = nx_local * coords_x
x_cell_max_local = nx_local * (coords_x + 1) - 1
```

Here `coords_x` is the position we found earlier (Section 5.6.3). Note that these values are the *offsets from the start of the array*, so in Fortran where arrays start at 1 we have to add 1 to both values to get the array index.

The calculation in y is identical, swapping all the `x` for `y`.

Do note that you only need to know about this local position when you're setting up the initial values (initial conditions) or bringing your sections back together for output. Otherwise, all of the computation should work with local arrays (and any axes or physical coordinates required). Remember, (Section 5.1) as far as possible the processors shouldn't know they're part of a larger problem. They need to know who their neighbours are for the comms, but for the core calculation, they shouldn't really care.

5.7 Quick Recap

To recap what we've worked out so far:

- The domain is decomposed onto the required number of processors
- We know how the "virtual global" array maps onto our original array

- We know which processors are neighbours to our current processor
- We've set up some ghost or guard cells so each processor has all of the information it needs

Now we need to actually do the sends and receives. The idea of this is pretty obvious. We've set up the ghost cells, so all we have to do is make sure they contain the correct data, and then our calculations can proceed as usual.

5.8 Handling Boundary Exchanges

Let's assume that our local array runs (`0:nx_local+1, 0:ny_local+1`). The `0` and `nx_local+1` strips are ghost cells, either filled from actual boundary conditions or from MPI calls. We've stored the ranks of neighbouring processors in variables called `x_min_rank`, `x_max_rank`, `y_min_rank`, `y_max_rank`.

Then, in Fortran,

```fortran
!Send left most strip of cells left and receive into right guard cells
CALL MPI_Sendrecv(array(1,1:ny_local), ny_local, MPI_REAL, x_min_rank, &
    tag, array(nx_local+1,1:ny_local), ny_local, MPI_REAL, x_max_rank, &
    tag, cart_comm, MPI_STATUS_IGNORE, ierr)

!Send right most strip of cells right and receive into left guard cells
CALL MPI_Sendrecv(array(nx_local, 1:ny_local), ny_local, MPI_REAL, &
    x_max_rank, tag, array(0,1:ny_local), ny_local, MPI_REAL, x_min_rank,
        &
    tag, cart_comm, MPI_STATUS_IGNORE, ierr)

!Now equivalently in y
CALL MPI_Sendrecv(array(1:nx_local,1), nx_local, MPI_REAL, y_min_rank, &
   tag, array(1:nx_local,ny_local+1), nx_local, MPI_REAL, y_max_rank, &
    tag, cart_comm, MPI_STATUS_IGNORE, ierr)

CALL MPI_Sendrecv(array(1:nx_local,ny_local), nx_local, MPI_REAL, &
    y_max_rank, tag, array(1:nx_local,0), nx_local, MPI_REAL, y_min_rank,
        &
    tag, cart_comm, MPI_STATUS_IGNORE, ierr)
```

Here we use array subsections to tell MPI what data we want to send and receive. Since this isn't using a periodic domain, at the domain edges the Sendrecvs will will no-ops and so do nothing.

In C things are a little more complicated because we don't have the handy array subsections so we have to make some copies. A single exchange (one of the four from the Fortran) becomes

```c
//Unlike in Fortran, can't use array subsections. Have to copy to
    temporaries
src = (float*) malloc(sizeof(float)*(ny_local));
dest = (float*) malloc(sizeof(float)*(ny_local));

```

```c
//Send left most strip of cells left and receive into right guard cells
for (index = 1; index<=ny_local; ++index){
  src[index-1] = *(access_grid(data, 1, index ));
  //Copy existing numbers into dest because MPI_Sendrecv is a no-op if
  //one of the other ranks is MPI_PROC_NULL
  dest[index-1] = *(access_grid(data, nx_local + 1, index ));
}

MPI_Sendrecv(src, ny_local, MPI_FLOAT, x_min_rank,
    TAG, dest, ny_local, MPI_FLOAT, x_max_rank,
    TAG, cart_comm, MPI_STATUS_IGNORE);

for (index = 1; index<=ny_local; ++index){
  *(access_grid(data, nx_local + 1, index )) = dest[index-1];
}
```

Note that the send-receives on a single processor are sending on one edge and receiving on the other!

5.9 Putting it all Together

We now have all the parts of a full domain-decomposed MPI code. We've put a fully working copy together solving the heat equation (temperature profile of a metal plate where the edges are held at specific temperatures) that can be downloaded either wherever you got these notes or at `https://github.com/WarwickRSE/IntroMPI`.

Have a look at this code. Identify all the parts we've discussed. Try making a few changes. For example, if you change the processor decomposition, can you observe a slowdown? Try the `time` utility, or look up the function `MPI_Wtime`. What about changing to split only in X, or only in Y?

5.9.1 Example Code in C

The C code includes a non standard array library that we wrote to make it easier to deal with array sections. This makes the code a lot less messy since there's no longer loops all over the place. This is BSD licensed, so you can use it if you wish, but its only minimally tested. In Fortran, you get all of this for free. The core functions are:

- `allocate_grid(ptr_to_grid_type, x_min_index, x_max_index, y_min_index, y_max_index)` - Allocate a 2D grid with the specified index ranges (in this case 0:nx+1, 0:ny+1)
- `assign_grid(ptr_to_grid_type, x_min_index, x_max_index, y_min_index, y_max_index, value)` - Assign value to all cells within the specified index ranges
- `access_grid(grid_type, ix, iy)` - Returns a pointer to element [ix][iy]
- `copy_grid(ptr_to_dest, ptr_to_src, x_min_index, x_max_index, y_min_index, y_max_index)` - Copy values in specified range from src to dest

Chapter 6

Worker Controller Models

6.1 Motivation

After Domain Decomposition, the next most common approach for MPI programs is the worker-controller approach, also called master-slave. Here one processor is in charge of handing out work packages to all of the others, which do their work and then either ask for more, or finish.

It is possible to have the controller processor also be a worker, but this makes the MPI more complex. In particular, you have to make sure the controller is doling out the work packages fast enough that the workers aren't sitting idle, or you'll lose more from this than you gain from the work done on the controller process. We're not going to consider this for the main part of this chapter, but we give a brief discussion of the functions you'd need in 6.5.

6.2 Single Program Multiple Data (SPMD)

A core question in parallelising workloads is whether a program has multiple entities all following the same steps on different data, or whether they can be doing completely different tasks, either on the same data, or different sets. See `https://en.wikipedia.org/wiki/Flynn%27s_taxonomy` for details.

Single-Program Multiple-Data is pretty much the model of MPI. You write one program that runs the same steps on different sets of data. Obviously, in practise you often have e.g. rank-0 doing something a little bit "special" but this is from branches within the code. As a counter, Multiple-Program Multiple-Data would be something like a data-processing pipeline, where a different program carries out each step, transforming the data to be fed to the next step. This can be done in parallel (although with a little bit of startup latency). Alternately, a typical server-client model is solidly MPMD.

Do note that Single-Program is a lot more general than Single-Instruction (if you have ever heard of SIMD, or looked at the link above). Despite running the same code, different processes can be (and are) at different steps.

Here, we're just concerned with the fact that we have to write one program that is both the controller and the worker. This is actually really easy - we just pick some rank to be the controller, and make all the others workers. It's not unusual to have a special rank in an MPI code anyway, since despite what we said in the previous Chapter real codes are seldom ideal.

In code terms, all we have to do is make the main routine look something like

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  controller_fn();
} else {
  worker_fn();
}
```

6.3 A Basic Arrangement

6.3.1 Schematic

The basic process of a master-controller program is as follows. We pick Rank 0 to be the master, and have 3 workers, ranks 1-3. The following figures show the starting up, doing the main work, and finishing up of the program.

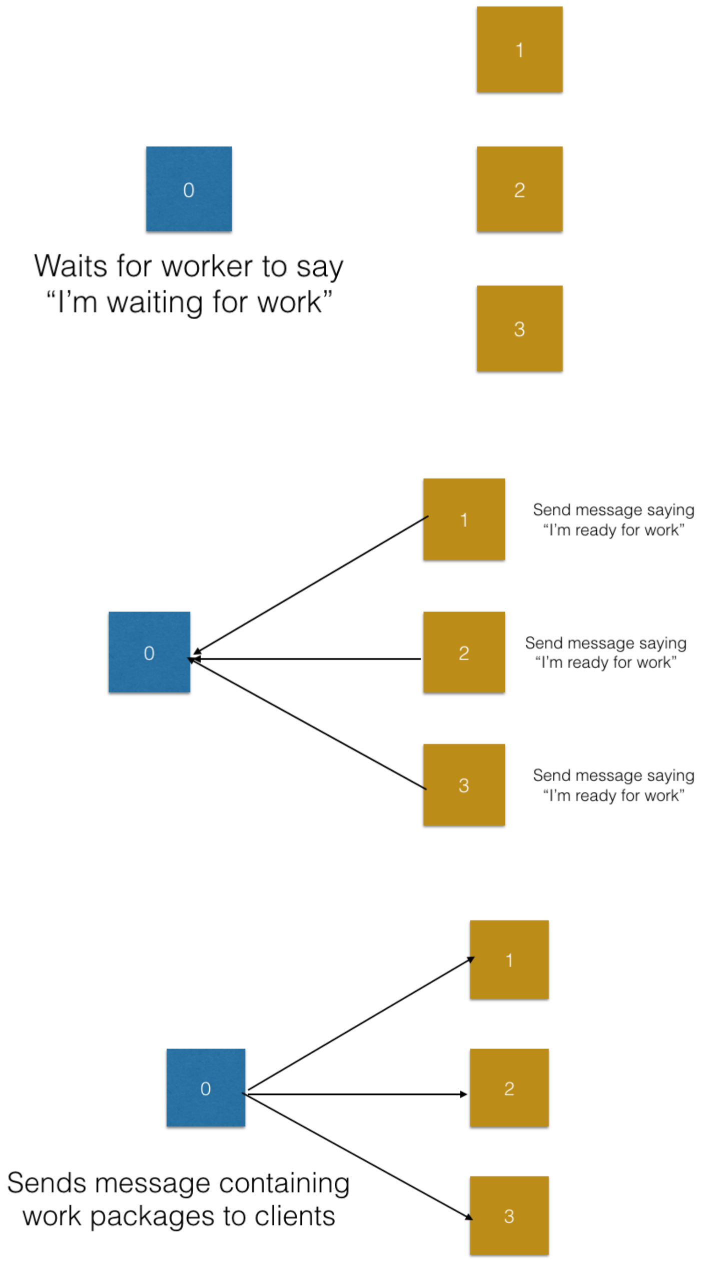

Figure 6.1: Master Controller Program - Getting Started

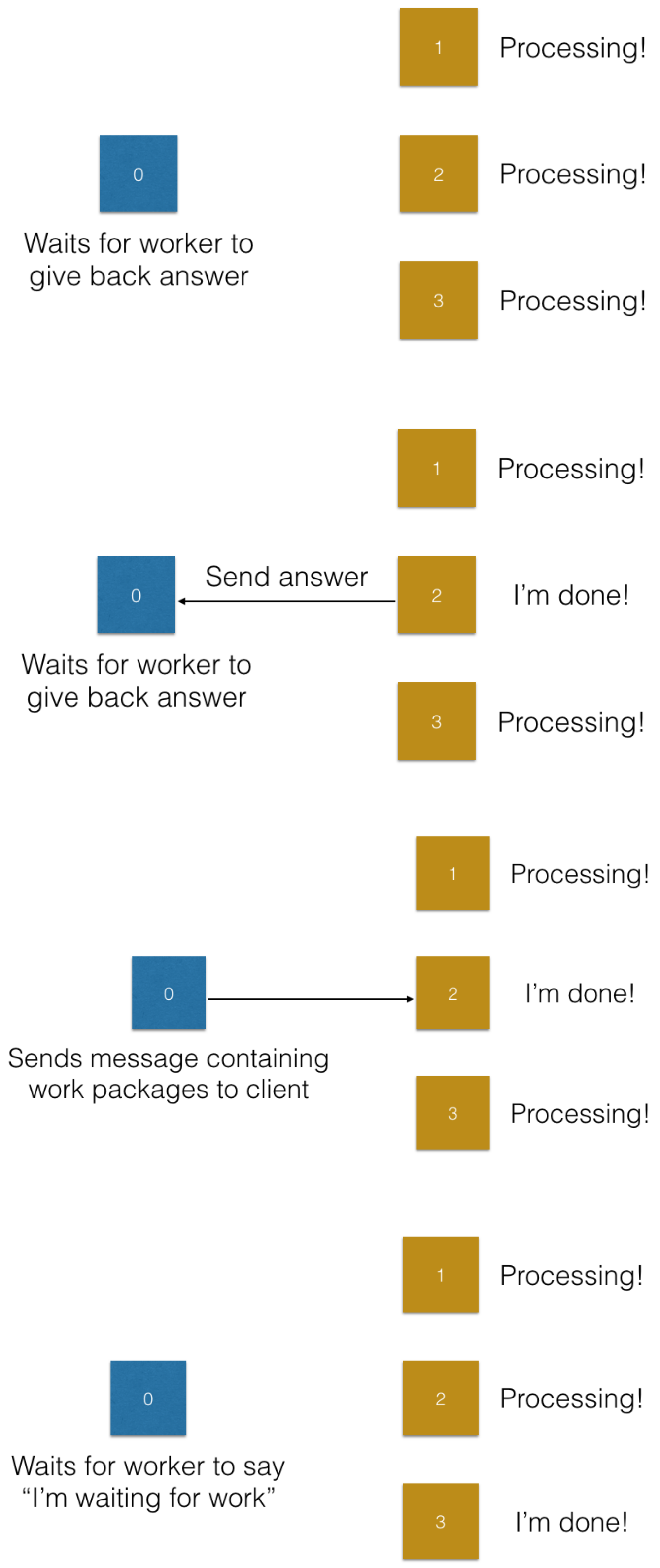
1
Processing!
0
2
Processing!
Waits for worker to give back answer
3
Processing!
1
Processing!
0
Send answer
2
I'm done!
Waits for worker to give back answer
3
Processing!
1
Processing!
0
2
I'm done!
Sends message containing work packages to client
3
Processing!
1
Processing!
0
2
Processing!
Waits for worker to say "I'm waiting for work"
3
I'm done!

Figure 6.2: Master Controller Program - Doing the Work

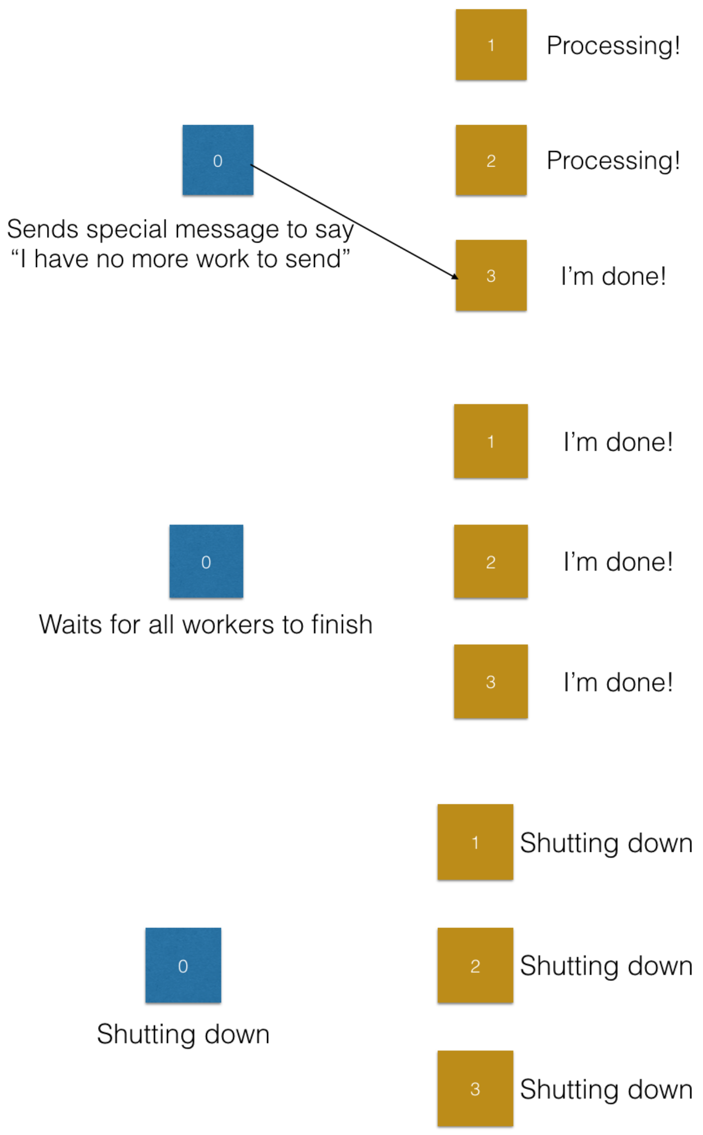

Figure 6.3: Master Controller Program - Finishing Up

6.3.2 The Controller

The controller basically just sits waiting to be asked for work, and then sends it out. This is a normal `MPI_Recv` command, except that we don't know which processor will communicate with us first. We use `MPI_ANY_SOURCE` so that the controller to wait for a message from any worker.

In the example code we use here, we use the message tag to contain the work package ID, so that we can send nice simple message bodies containing only the input data or the answer. This is rather too simple for any real-world task, but makes the example code nice and easy. In general you would want to send a message containing some sort of data structure giving input parameters and any other information needed, and similarly send back a structure. However, dealing with sending these is a bit tricky - we intend to cover this in our Intermediate (or possibly Advanced) MPI course.

Since the Tag contains the ID of a work package, the controller wants to wait for a message with any tag value, so we use the special `MPI_ANY_TAG` value in the tag field. The Receive call then looks like:

```
int data;
MPI_Recv(data, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
    stat);
```

6.3.3 Parsing MPI Status

In the previous chapter on domain decomposition we didn't need to use the MPI status objects, since we already knew which receive came from which neighbour, and weren't using the tags to transmit any information. Here, we need to capture and interrogate this to find out which processor sent the message and what the tag was. The code to do this is, in C and Fortran respectively:

```
MPI_Status s;
printf("source %i\n", s.MPI_SOURCE);
printf("tag %i\n", s.MPI_TAG);
```

```
INTEGER :: s(MPI_STATUS_SIZE)
PRINT *, "Source" , s(MPI_SOURCE)
PRINT *, "Tag" , s(MPI_TAG)
```

Note that in the C code we are accessing a structure by named values, whereas in the Fortran the status is simply an array of integers, with some named constants provided telling us which field contains which value.[1]

The code in the workers is rather simpler. They simply do an `MPI_Send` to rank 0 to send results back to the controller, and an `MPI_Recv` from rank 0 to get the next work package, or the message telling them there is no more data to send (via a sentinel value for the tag), and to shutdown.

6.4 Pitfalls to Avoid

The main pitfalls of this sort of Worker Controller code arise because it relies on all processes getting the messages they expect, when they expect them. This is particularly true of the workers, which must be told when to terminate. In particular, be careful to make sure that your code will terminate correctly under all conditions.

The controller must send out proper messages if there aren't any valid work packages at all. It must keep listening even after the last work package is sent out, and only stop

[1]Newer MPI Fortran Bindings (2008 onwards) are extending the types to look more like the C. This is *definitely* outwith the scope of this Intro.

when the last worker *finishes*. It must make sure to shut down all the workers before shutting down itself.

If you accidentally stop receiving on the controller before the last worker has sent back its result then your code won't terminate properly. Usually, your code will then be killed by the wall-time limit on a cluster, but this can be up to 48 or 72 hours. You can easily waste a lot of core hours this way (and that can be expensive).

6.5 Non Blocking MPI

The worker-controller strategy here is a very simple one that you probably wont see in the "real world". Instead, you will probably see the MPI commands `MPI_Isend` and `MPI_Irecv` (those are capital-i's if the font is unclear). These are special non-blocking MPI send and receive commands. They return you a `MPI_Request` object which you can interrogate to check on the status of the send or receive operation.

It is then up to you to make sure things have correctly happened. To do this, you need the following commands:

- `MPI_Wait` - Wait until a given `MPI_Request` completes
- `MPI_Waitall` - Wait until all of the `MPI_Request`s in an array have completed
- `MPI_Waitany` - Wait until any `MPI_Request` in an array of `MPI_Request`s has completed

These versions halt your code until the request is finished. As well as these, there are the Test versions,(`MPI_Test`, `MPI_Testall`, `MPI_Testany`) that return a logical flag about the current state of an `MPI_Request` (completed or not) rather than waiting for completion. These functions also all return `MPI_Status` objects (just like `MPI_Recv`) which allow you to check the tag, source etc of the message once it has been received.

The non-blocking commands give you a (relatively) easy way of letting rank-0 act as a worker as well as a controller. This process posts a non-blocking receive (using `MPI_Irecv`), does its work, and when convenient uses `MPI_Testany` to check if any of the workers have completed.

Non-blocking MPI is quite a bit harder to use in general, so we aren't going into details here. In particular, you have to be very careful using any temporary arrays, as they mustn't be changed or deallocated until the receive or send has completed. Fortran array sections, which are effectively temporary arrays, are right out. Usually you want to use special custom MPI types to make things more robust too.

Chapter 7

Glossary of Terms

Glossary

API Application Programming Interface; the set of routines (functions) which you call to interact with a library, service or similar. 9

bandwidth The rate at which data is transferred, usually in mega or giga bits per second. Your internet contract "speed" is actually a bandwidth. *See also* latency, 11

binding aka Language Binding. Libraries usually have their core written in a single language, but may provide interfaces in multiple languages. Your code talks to the interface, which takes care of translating between the core and the language you're using. Good bindings are usually quite thin, giving you access to everything the library can offer, adding nothing and not costing much either . 9

blocking For MPI commands, blocking commands are those where control doesn't return to the program until you can safely re-use the buffers you passed. For a send, this DOES NOT mean the send is done, only that the data has been acquired by the comms layer. For a receive, it DOES mean you now safely have the data. *See also* non-blocking, 26

communicator A grouping of processors. Programs can have multiple communicators, but simple programs use only `MPI_COMM_WORLD` which contains all processors in the job. Communicators can be split and combined. 16, 19, 26, 27, 54

interface Basically the set of function calls a library provides. You use these in your programs to interface (interact) with the library. Often interface (what) is distinguished from implementation (how things are done behind the scenes). *See also* API, , 53

latency The fixed time cost of sending or receiving data, i.e. the delay before things start moving. In video-gaming circles and internet speed tests, tends to be represented by the "ping" time. *See also* bandwidth, 11

non-blocking For MPI commands, non-blocking commands hand back control immediately, but let you check later whether data has been safely off-loaded or received. *See also* blocking, 52

rank A number, unique to each processor in a set (a communicator) that can be used to identify it. Usually processors are numbered sequentially, and the 0th is called the root and used for anything that only one processor needs to do. 16

root One of the processors in a communicator which does any unique work. 20, 54

scope Scope of a variable is the region of the program in which it exists and can be used. Most languages have "function scope" so variables you create inside a function can't be used outside it. C-like languages add "block scope" so a variable defined, for example, inside an if-block is lost when the block ends.

sentinel A special value, outside the range a parameter can ordinarily take, which signals that something special has or should occur. For instance, -1 can never be a valid count of items, so can be used as a signal. Similarly, `MAX_INT` or NAN can be used (with care) to signify missing data in a data set. Sentinels should be used with a little care, since they can cause chaos if other code or a user doesn't recognise them. 51