Warwick Research Software Engineering

Intermediate MPI

CS Brady and H Ratcliffe
Senior Research Software Engineers

December 4, 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 March 2019 at the University of Warwick.

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

0.2 Disclaimer

0.3 Example Programs

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

Chapter 1

Recap of MPI Basics

A more in-depth version of this section is available in our Introductory MPI course, `https://warwick.ac.uk/research/rtp/sc/rse/training/intrompi`.

1.1 Parallelism and MPI

Parallelism relies on the ability to break a problem up into elements that different processors can work on. Ultimately this depends on the layout of your data. If your problem wants multiple processors to work on a single chunk of memory the solution is something like OpenMP, OpenSHMEM or other shared memory options. If you can distribute the data over processors so they each have some part of the problem, the solution is MPI. This is useful for problems which "aren't too coupled", so that mostly the processors work independently and only sometimes communicate.

MPI has plenty of advantages. It will work on any CPU based system for which an MPI aware compiler and MPI library can be built. It will scale well to the very largest machines, with large MPI codes working well on millions of cores. Overheads are quite low, with very little setup and teardown of messages and no costly locking of memory since each processor has its own memory. MPI also forces the developer to consider the locality of the data, which can lead to improved performance.

Forcing the programmer to work out the data locality is also a key disadvantage of MPI, since if this is not possible the system becomes very hard to use. For programs running on only a single node MPI is unnecessary as a shared-memory model will work well. Finally, anybody using the program needs an MPI-aware compiler or you must write explicit serial code as well as the parallel code.

1.1.1 MPI in Fortran

In C, all MPI objects are typed, such as `MPI_Datatype`, `MPI_Comm` etc. In Fortran these are usually plain integers. Fortran MPI functions also all take an extra INTEGER parameter which is the same as the C return code. `MPI_Wtime` and `MPI_Wtick` are exceptions. In these notes we mostly use the C versions because those need some parameters to be pointer type (as C is pass-by-value). Some examples are given in both C and Fortran, as is all example code.

NOTE: MPI is adding a Fortran 2008 interface which is much closer to the C version and uses Fortran 2008 constructs. This is not yet widely enough supported for general use, but is worth watching.

1.2 A Basic MPI Program

The simplest useful MPI program is shown schematically in Fig 1.1. The program calls `MPI_Init`, and then `MPI_Comm_rank` to get the unique rank for each processor. It then does a mixture of computation and MPI Sends and Receives. Finally it calls `MPI_Finalize` to clean up before exiting.

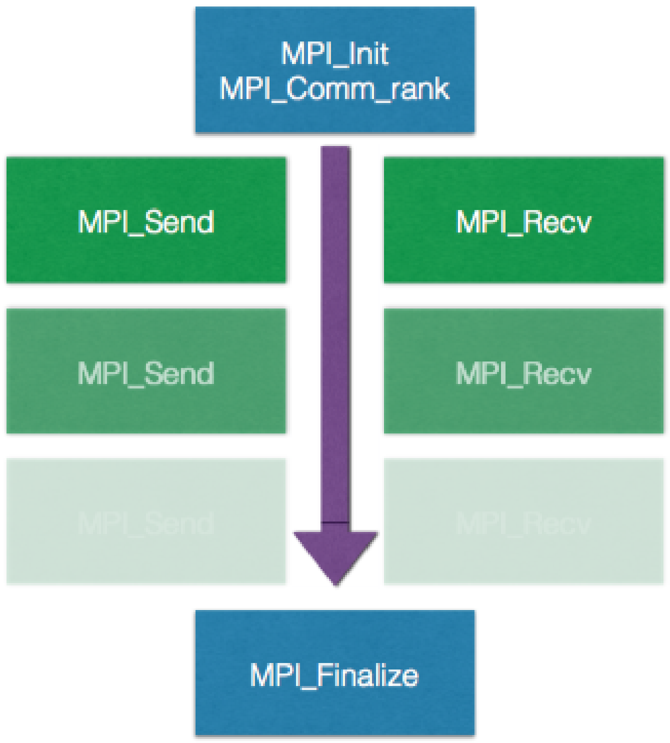

Figure 1.1: A basic MPI program in schematic form

1.2.1 MPI Send

The `MPI_Send` command looks like

```
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` number uniquely identifying this message (must be unique between any messages currently in flight)
- `comm` an MPI communicator (i.e. a list of processors) to send to

1.2.2 MPI Builtin Types

MPI has various built in types representing data, as well as the ability to create custom types representing chunks of data, discussed in Chapter 3. The built-in types strictly have different names in C and Fortran. Sometimes the "wrong" versions will work, but this is not strictly necessary so it is a good idea to use the right variant.

- `MPI_INT` Integer in C
- `MPI_INTEGER` Integer in Fortran
- `MPI_FLOAT` Single precision floating point in C
- `MPI_REAL` Single precision floating point in Fortran
- `MPI_DOUBLE` Double precision floating point in C
- `MPI_DOUBLE_PRECISION` Double precision floating point in Fortran
- `MPI_BYTE` A single byte (mostly used to build other types, for IO etc)
- Many others

1.2.3 MPI Receive

The `MPI_Recv` command looks like

```
int MPI_Recv(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
- `source` which rank data comes from

- `tag` number uniquely identifying this message (must be unique between any messages currently in flight)
- `comm` an MPI communicator (i.e. a list of processors) to receive from
- `status` an MPI status object containing information about the message

Note you might not care about either or both source and tag, i.e. you just want to receive the next message from the queue, whatever it is. This can be done by using the special constants `MPI_ANY_SOURCE` and `MPI_ANY_TAG`.

In Fortran the MPI_Status is an array of integers, `INTEGER, DIMENSION(MPI_STATUS_SIZE)`

1.2.4 Send Variants

There are actually 4 variants of send, all of which are blocking operations. Blocking here means roughly that they do not return until it is safe to reuse the sending buffer.

- `MPI_Send` Returns as soon as it is safe to reuse the send buffer. This says nothing about whether the message was actually received
- `MPI_Ssend` Does not return until the message has been received
- `MPI_Bsend` Copy data to be sent into another buffer and return immediately
- `MPI_Rsend` “Ready mode” send, only valid if a matching receive has already been posted. This has quite limited use.

1.3 The Ring Pass

The simplest possible MPI program is one that simply passes a value to its “neighbour”, here defined as the processor with rank one higher than the given processor. The highest ranked processor wraps back around and sends to processor rank 0 (usually called the root). Figure 1.2 shows this schematically.

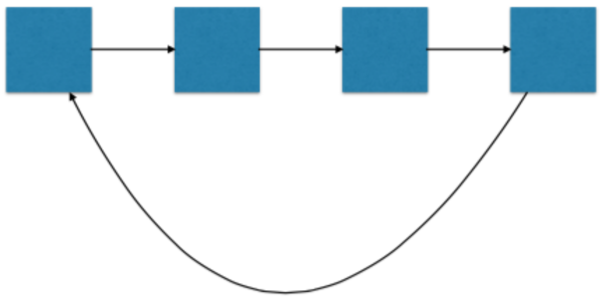

Figure 1.2: The basic MPI ring pass

A basic Fortran program to do this is:

```
PROGRAM wave
  USE mpi
  IMPLICIT NONE
  INTEGER, PARAMETER :: tag = 100
  INTEGER :: rank, recv_rank
  INTEGER :: nproc
  INTEGER :: left, right
  INTEGER :: ierr


  CALL MPI_Init(ierr)
  CALL MPI_Comm_size(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

  !Set up periodic domain
  left = rank - 1
  IF (left < 0) left = nproc - 1
  right = rank + 1
  IF (right > nproc - 1) right = 0

  IF (rank == 0) THEN
    CALL MPI_Ssend(rank, 1, MPI_INTEGER, right, tag, MPI_COMM_WORLD, ierr
        )
    CALL MPI_Recv(recv_rank, 1, MPI_INTEGER, left, tag, MPI_COMM_WORLD, &
        MPI_STATUS_IGNORE, ierr)
ELSE
    CALL MPI_Recv(recv_rank, 1, MPI_INTEGER, left, tag, MPI_COMM_WORLD, &
        MPI_STATUS_IGNORE, ierr)
    CALL MPI_Ssend(rank, 1, MPI_INTEGER, right, tag, MPI_COMM_WORLD, ierr
        )
  END IF

  PRINT *,"Rank ", rank, " got message from rank ", left, " of ",
      recv_rank

  CALL MPI_Finalize(ierr)

END PROGRAM wave
```

and in C

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

#define TAG 100

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

  int rank, recv_rank, nproc, left, right;

  MPI_Init(&argc, &argv);
12   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
13   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14
15   //Set up periodic domain
16   left = rank - 1;
17   if (left < 0) left = nproc - 1;
18   right = rank + 1;
19   if (right > nproc - 1) right = 0;
20
21   if (rank == 0) {
22     MPI_Ssend(&rank, 1, MPI_INT, right, TAG, MPI_COMM_WORLD);
23     MPI_Recv(&recv_rank, 1, MPI_INT, left, TAG, MPI_COMM_WORLD,
24         MPI_STATUS_IGNORE);
25   } else {
26     MPI_Recv(&recv_rank, 1, MPI_INT, left, TAG, MPI_COMM_WORLD,
27         MPI_STATUS_IGNORE);
28     MPI_Ssend(&rank, 1, MPI_INT, right, TAG, MPI_COMM_WORLD);
29   }
30
31   printf("Rank %3d got message from rank %3d of %3d\n", rank, left,
       recv_rank);
32
33   MPI_Finalize();
34 }
```

Lines 11-13 initialize MPI, setup a communicator containing all the processors, and get the unique rank of each processor. Next, lines 15-19 setup a periodic domain. Each processor has a left-hand adjacent processor, either its rank minus one, or on the left-hand-edge, the furthest right processor. Each also has a right-hand neighbour, which is rank plus one, except at the right-hand-edge, where its zero.

Every processor *except* rank 0 waits for data from their left, and then proceed to send their own (lines 26-28). Rank 0 starts by sending its data to rank one, and then waits to receive from the right-hand-processor. Finally the code calls `MPI_Finalize` before stopping.

The result of this code is

```
Rank   1 got message from rank   0 of   0
Rank   2 got message from rank   1 of   1
Rank   3 got message from rank   2 of   2
...
Rank  15 got message from rank  14 of  14
Rank   0 got message from rank  15 of  15
```

There is something in the code you may not have seen before, namely the `MPI_STATUS_IGNORE` which is a constant you can use where otherwise there would be an `MPI_Status` object. Note that this can't be used in place of an *array* of status objects, instead there is an `MPI_STATUSES_IGNORE`

Repeating this loop very many times, in this case 100,000 (100,000 messages per processor) using a shared-memory, 16 core machine, we find a total time of 1.3 s. The total time needed just for the communications is about 1 μ s, so most of this time is due to the latency in the process, not the communications bandwidth. In theory, we'd like a latency as close as possible to the 100 ns required to fetch the value from our memory.

So why does it take so much longer? Because most of the processors are sitting idle for most of the time! Processor 1 is waiting for processor 0. Processor 2 is waiting for processor 1, which is waiting for processor 0. Processor 3 is waiting... and so on. We can't just have all the processors send at once, or they would deadlock, waiting forever for a message that will never arise, because the sender is also locked waiting in the same way.[1]

1.3.1 Red Black Ordering

Since the crucial thing here is having paired senders and receivers both ready at once, there is a fairly obvious better solution, often known as red-black ordering. We colour each even numbered processor red and each odd one black. Then rather than passing sequentially 0 to 1 then 1 to 2, we have all the red processors send, while all the black ones receive, and when that is finished, we swap, and have the black ones send and the red ones receive. The last processor again wraps around back to the 0th. This is shown schematically in Figure 1.3.

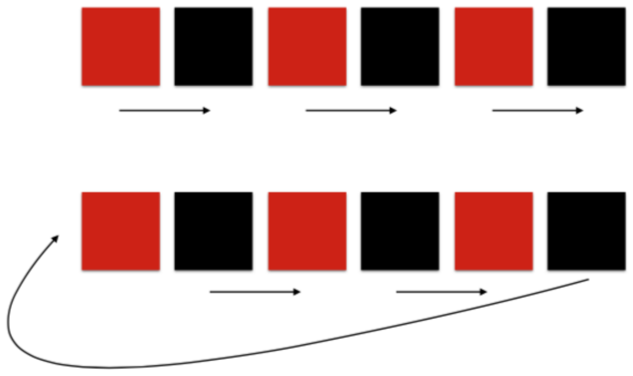

Figure 1.3: A basic red-black ordering

The code is barely changed, only the `if` statement on line 21, instead of

```
if (rank == 0)
```

becomes

```
if (rank%2 == 0)
```

or

[1]We'll see in the next section how MPI lets us in fact do this, and sorts out the deadlock for us. But for now, suppose we don't know about that

```
IF (MOD(rank, 2) == 0) THEN
```

Now we try this on our same 16 core machine, and find it to be nearly five times faster, requiring only 0.22 s. The latency is now reduced to about 138 ns, much closer to our 100 ns target.

1.3.2 Combined SendRecv

We mentioned in a footnote that MPI lets us just send everything at once and expect it to sort everything out behind the scenes. In particular, there is the combined send-receive command, `MPI_Sendrecv`. You may or may not have encountered this before. As long as both the send part and the receive part can complete, we can call this on all processors at once and not risk a deadlock. The command is long because it must contain all the information for the send, and for the receive, giving

```
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)
```

and the code on lines 21 to 29 becomes the single line

```
MPI_Sendrecv(&rank, 1, MPI_INT, right, TAG, &recv_rank, 1, MPI_INT, left,
      TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
```

and

```
CALL MPI_Sendrecv(rank, 1, MPI_INTEGER, right, TAG, recv_rank, 1,
    MPI_INTEGER, left, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
```

Is this any better? On our test machine, this takes only 0.09 s, and the latency has dropped to the target 100 ns. This is basically as fast as we can expect this to work, which is great. We're now sharing data between adjacent processors as fast as we really can.

Parallel Decomposition

The core idea of parallel programming is working out a way to use multiple processors at once to get work done. So called *embarassingly parallel*, or *trivially parallel*, workloads are ones where there are completely independent sets of work and so you can simply start up several separate processes and let them run independently. This is the optimal solution, although few people would consider this to be really parallel programming. Instead, our target is to have each processor working *mostly* with data it owns, and only *rarely* having to communicate with its neighbours.
There are several common ways of achieving this goal, mainly based around splitting into some sort of orthogonal set. If this can be done, and interactions are in some sense *local* in these directions, then MPI can work well and scale to the largest supercomputers. More on this below.

Domain Decomposition
Domain decomposition is perhaps the simplest non-trivial decomposition. We have a grid of cells in say 2-D, with n_x cells in the x-direction, and n_y in the y-direction. We split this grid into rectangles and assign one to each processor. The first Case Study in the next Chapter will show an example of this. Now assuming that particles interact only with those *nearby* in space, for example, via simple collisions, each processor only needs the chunk of space it owns, and some small amount of its neighbours, usually only 1 or 2 grid cells worth at the edges. This is all we need to communicate, using send and receive or sendrecv.
Sadly this is not always possible because not all processes are local in space...
Arithmetic Decomposition
If spatial decomposition isn't possible or useful, sometimes there are other "directions" we can use. Most commonly, we have some sort of orthogonal basis functions for our solution set, and we can decompose and give each processor a subset of these basis functions. Because they are orthogonal, so linearly independent, the processors can solve independently for their components at any given time.
In general, sadly, there will still be coupling of basis functions somewhere in your equations, requiring communication. Hopefully, this is only between functions that are in some sense "nearby" in which case we have limited comms and MPI can still work well, even if it is rather complicated to understand what is going on intuitively.
Arbitrary Decomposition
In some cases, the coupling between things really is almost arbitrary or fundamentally long-range, in which case the choice of decomposition barely matters. Only answer is to split things up somehow and try and deal with the comms later. In this case MPI is probably not the best solution, and you might be better investigating PGAS systems and be warned that performance will probably not be good.

1.4 Reduce Operations

Reduce operations are those which collect data from all processes, operate on it, and then put it somewhere else. In general, they either place the result onto a root processor (`MPI_Reduce`), or back onto all processors (`MPI_Allreduce`).

If these functions are given an *array* of values, they will operate on it elementwise, to give an array of the same size as final result. They will not reduce over the elements.

The commands are

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

and

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

and a simple example program looks like

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

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

  int rank, nproc, recv;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_Allreduce(&rank, &recv, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);

  printf("On rank %3d MPI_Allreduce gives maximum rank as %3d\n", rank,
      recv);

  MPI_Finalize();

}
```

The built-in operations you can use in reduce include max/min, summation and logical ops. The full list is

```
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

MPI_MAXLOC : Value and rank of maximum element
MPI_MINLOC : Value and rank of minimum element
```

Note that MAXLOC and MINLOC have some caveats about how they indicate which processor the result was found on, so read up carefully before using them. Basically, they return both the max/min value, and the rank on which it occurred (for each element if an array is passed). You give the operation a packed pair of values, one the value in question the other the rank of the processor. In C these are in a struct, in Fortran they should be an array.

1.5 Non-blocking Send and Recv

Both send and receive have so-called non-blocking variants, `MPI_Isend` and `MPI_Irecv` which return immediately rather than waiting for the operation to in some sense "complete".

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

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

Note that the syntax is very similar to the normal Send and Recv commands, but includes an extra `MPI_Request` object. We will cover non-blocking commands in detail in Chapter 4, but they are put here for completeness.

1.5.1 MPI Wait

This command is used to ensure the non-blocking operations have completed. You give it the request object from the send/recv command and it will block until the request has completed. This lets you put sends and receives at the most suitable places within your calculation, and then put the wait at the end. This can improve efficiency in some cases.

1.5.2 Sendrecv

The `MPI_Sendrecv` command from 1.3.2 actually consists of a non-blocking send, a non-blocking recv, and a wait all wrapped up together. This is why it lets you do the simple thing of having every processor call sendrecv at once without causing a deadlock.

1.6 Other Operations

There are also some other very useful MPI operations you should know about.

1.6.1 Barrier

Barriers block until all processors in a communicator have reached the barrier, at which point all processors may continue.

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

1.6.2 Broadcast

Broadcasts send information from one rank (the root) to all other ranks.

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

1.6.3 Scatter

Scatters send different data from root to each other processor.

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

Here sendbuf is $sendcount \times nproc$ long, with the first sendcount elements going to the lowest ranked other processor, the next sendcount elements to the second in line, etc. Recvbuf should then be recvcount long, and unless you're using custom types to reshape your data, recvcount should equal sendcount.

1.6.4 Gather

Gather collects data from all ranks onto the root rank

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

Here recvbuf is $recvcount \times nproc$ long and receives sendcount elements from each other processors. sendbuf is sendcount long and again unless using custom types, sendcount equals recvcount

1.6.5 All Gather

All gather is similar to gather, but the data is given to all ranks.

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

The buffer sizes are all the same as for Gather.

1.6.6 All To All

All to all is a generalised scatter and gather combined, where every rank sends some data to every other rank, and receives some data from every other rank.

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

Sendbuf is $sendcount \times nproc$ long, recvbuf is $recvcount \times nproc$ long and as usual, unless using custom types, sendcount equals recvcount.

1.6.7 V variants

Finally, Scatter, Gather, Allgather and Alltoall all have "v" variants, which allow sending or receiving different amounts of data on different processors. You have to have some mechanism so that each knows the correct amount of memory to allocate for receiving.

In general, these indicate that your MPI might be becoming overly complex, although they are sometimes a necessary evil. Use them with caution, and make sure to program robustly and document carefully!

Chapter 2

Case Studies

2.1 Problem Specification - A physical problem

For a good example of MPI, we want some sort of problem that can be split up over processors, preferably using a spatial decomposition. This means we need something with spatial extents. We'd also prefer 2-D for simplicity of viewing. A nice example is a rectangular sheet of metal of fixed density and varying temperature. We hold the top and right edges hot, and the bottom and left edges cool, and want to solve for the temperature across the sheet. The following figure shows our arrangement:

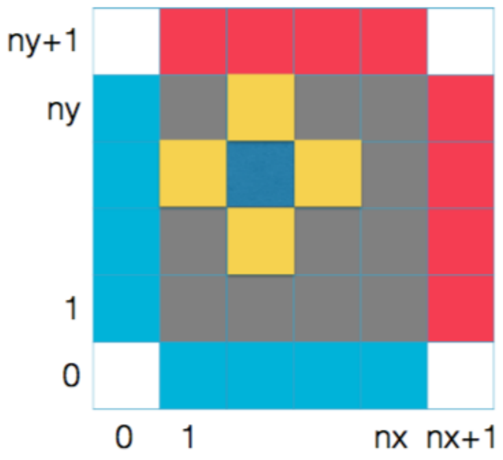

Figure 2.1: The layout for the case study problem. The red and blue edge cells are held fixed. The darker blue cell is the one being solved, the yellow is the cells which we average to update this cell.

Now we solve the heat equation (https://en.wikipedia.org/wiki/Heat_equation) for a steady state answer. This means we are iteratively solving for a single answer, not obtaining a solution over time. We start by splitting space into a set of discrete grid points, and then, in simple form, just average the values in the 4 cardinally adjacent cells (above, below, left and right). We do this for all cells, then start again with the resulting values. We're "done" once the solution no longer changes in time (or changes by no more than some threshold).

The "driven" edges, which we keep hot or cold, are implemented as a "halo" of *guard* or *ghost* cells, which are never changed.

2.1.1 The Code

The Fortran code for this uses strict F95, with each processor holding an allocatable, multi-dimensional array with explicitly set upper and lower bounds. Copy and assignments are done using array subsections.

The C code is C99 and uses a custom array similar to Fortran (and using Fortran ordering, first index varies fastest). The implementation is in `support/array.c`.

On one processor, the core code is, in Fortran,

```fortran
INTEGER, PARAMETER :: nx = 20, ny = 20
REAL, DIMENSION(0:nx+1, 0:ny+1) :: values, temp_values
INTEGER :: ix, iy, icycle
values = 5.5
values(0,:) = 1.0
values(nx+1,:) = 10.0
values(:,0) = 1.0
values(:, ny+1) = 10.0
CALL display_result(values)
PRINT *,'Please press a key to advance'
READ(*,*)
DO icycle = 1, 500
  DO iy = 1, ny
    DO ix = 1, nx
      temp_values(ix,iy) = 0.25 * (values(ix+1,iy) + &
            values(ix,iy+1) + values(ix-1,iy) + values(ix,iy-1))
    END DO
  END DO
  values(1:nx,1:ny) = temp_values(1:nx,1:ny)
  IF (MOD(icycle,50) == 0) THEN
    CALL display_result(values)
    PRINT *,'Please press a key to advance'
    READ (*,*)
  ENDIF
END DO
```

or in C,

```c
grid_type values, temp_values;
int ix, iy, icycle;
//Allocate a 2D array with indices that run 0->nx+1 and 0->ny+1
//This replicates Fortran's arrays with variable starts and ends
allocate_grid(&values, 0, nx+1, 0, ny+1);
allocate_grid(&temp_values, 0, nx+1, 0, ny+1);
//Assign the value 5.5 to the whole grid
assign_grid(&values, 0, nx+1, 0, ny+1, 5.5);
```

```
//Assign the boundary conditions. 1.0 along the left and bottom
//10.0 along the right and top
assign_grid(&values, 0, 0, 0, ny+1, 1.0);
assign_grid(&values, nx+1, nx+1, 0, ny+1, 10.0);
assign_grid(&values, 0, nx+1, 0, 0, 1.0);
assign_grid(&values, 0, nx+1, ny+1, ny+1, 10.0);
//To a C programmer, this looks backwards, but the array is using
//Fortran ordering deliberately
for (icycle=0;icycle<500;++icycle){
  for (iy=1;iy<=ny;++iy){
    for (ix=1;ix<=nx;++ix){
      *(access_grid(&temp_values, ix, iy)) = 0.25 * (
          *(access_grid(&values, ix+1, iy   )) +
          *(access_grid(&values, ix   , iy+1)) +
          *(access_grid(&values, ix-1, iy   )) +
          *(access_grid(&values, ix   , iy-1)));
    }
  }
  copy_grid(&values, &temp_values, 1, nx, 1, ny);
  if(icycle%50==0){
    display_result(&values);
    getchar();
  }
}
```

The result of this is shown in Figure 2.2. There is a smooth gradient across the box. The actual result doesn't matter here: all that matters is that the parallel version of this code gives the same result.

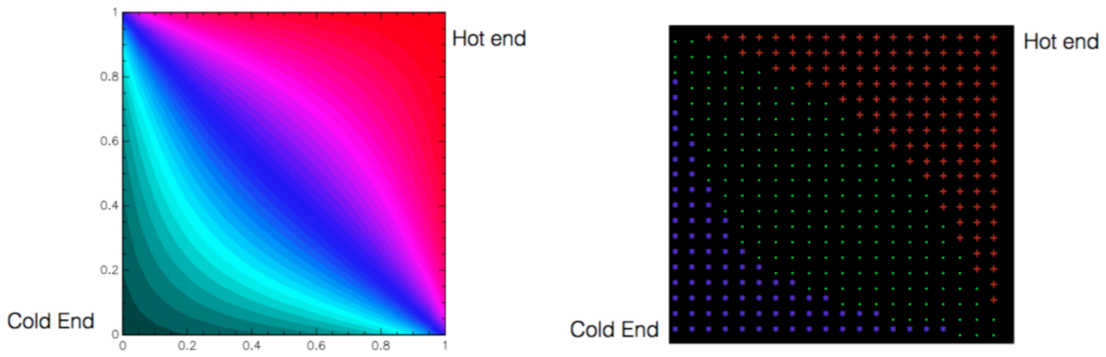

Figure 2.2: The temperature across the box after a stable state is reached. Left: actual smooth result. Right: an approximate visualisation produced by the example code. Points are coloured blue if they are in the bottom third of temperatures, are unchanged (here green) if in the middle third, and are red if in the hottest third.

2.1.2 Basic Parallelism - Domain Decomposition

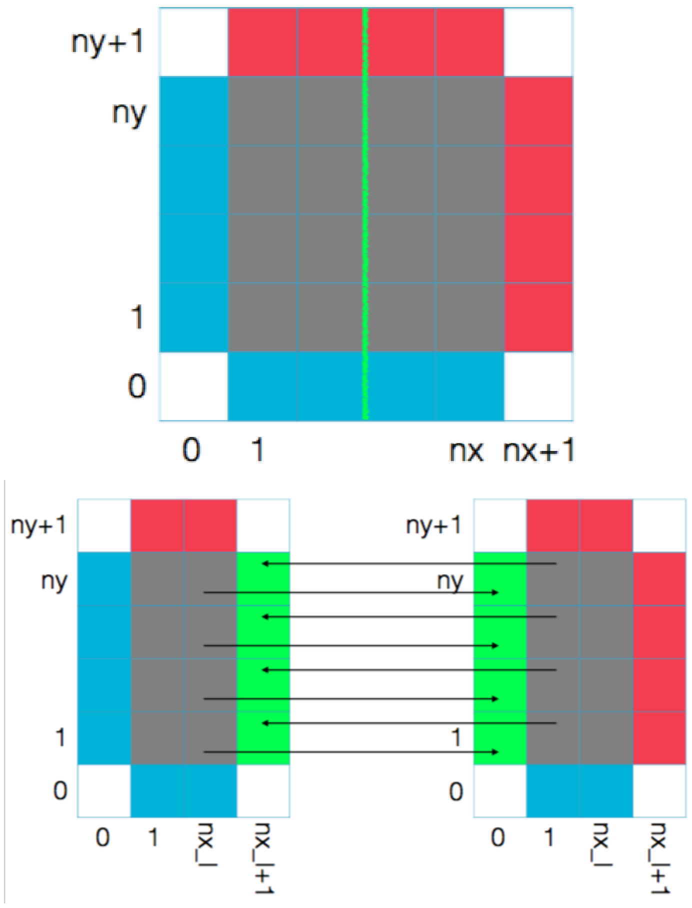

Figure 2.3: Domain decomposition on 2 processors. The whole domain is split on the green line (top) and new ghost cells are added which must be communicated between processors (bottom)

To parallelise this code, we simply break the domain into chunks, creating new, "virtual" boundaries between them. These have their own ghost or guard cells, coloured green in Fig 2.3, which overlap real cells in the adjacent domain (see arrows). Between steps of the solver, we communicate these ghost cells to the neighbour, and receive new data back, keeping the whole domain in sync everywhere.

In practise, it is a good idea to put the boundary conditions into their own routine, including both the virtual bounds and the real ones. In this example nothing needs to be done on the real boundaries, as they are simply fixed.

The global domain in this example runs over $(0 : \mathrm{nx}+1) \times (0 : \mathrm{ny}+1)$ with the part $(1 : \mathrm{nx}) \times (1 : \mathrm{ny})$ being the actual simulation domain, and the rest the boundary cells.

Each local domain runs $(0 : \mathrm{nx}_l + 1) \times (0 : \mathrm{ny}_l + 1)$ with, again, the edges being boundary cells and the region $(1 : \mathrm{nx}_l) \times (1 : \mathrm{ny}_1)$ the core simulation domain.

For ease of reading, in the code we use $\mathrm{nx_{local}}$ and $\mathrm{ny_{local}}$ rather than the shorter forms, as the "l" character can be ambigious.

Usually, the sizes nx and ny are defined by the problem to solve, but $\mathrm{nx_l}$ and $\mathrm{ny_l}$ can only be defined at run-time one you know how many processors are in use, and decide how to split them.[1]

In general, splitting the domain is a hard problem. On anything non-trivial you want to split not by number of cells but by some sort of overall load. For instance, suppose some cells took longer to calculate than others: ideally those should be evenly distributed, otherwise some processors will have more work to do, and others will sit idle.

In this problem, all cells have the same load, so all we really want is to minimise the perimeter to area ratio, since we need extra ghost cells around the perimeter.[2]

2.2 MPI Topologies

The topology of an MPI problem is how the subdomains are connected together. To show this to MPI, we create a new communicator, to replace `MPI_COMM_WORLD`. The MPI systems then tries to keep connected nodes close in the physical hardware, so that communications are less costly.

There are two MPI routines for creating topologies, `MPI_Graph_create` and `MPI_Cart_create`. Here we are only going to discuss the second. The resulting communicator replaces `MPI_COMM_WORLD`. A processor can be part of more than one communicator at the same time, and it is important to note that a given processor need not (and generally does not) have the same rank in different communicators.

First we are going to describe the functions used to work with a Cartesian topology, then we apply these to our case study. You may wish to re-read this section after looking over the code in 2.3

The `MPI_Cart_create` routine creates a Cartesian (rectangular) grid of processors, and is:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const
    int periods[], int reorder, MPI_Comm *comm_cart)
```

where

- `comm_old` Usually `MPI_COMM_WORLD`
- `ndims` Number of dimensions in new communicator. Usually the same as the number in the problem.
- `dims` Number of processors in each direction

[1]In general it is a bad idea to write code that needs a certain number of processors. Sometimes, e.g. a power of 2 can be very advantageous (e.g. Fourier transforms) but you should definitely warn and preferably have some strategy for dealing with an inappropriate number.

[2]If this is not obvious, compare the number of ghost cells for a 4x4 square domain and a 16x1 domain.

- `periods` Whether processor topology should wrap around at the edges (creating a periodic domain)
- `reorder` Whether processor ranks should be changed to better map to physical hardware
- `comm_cart` Resulting Cartesian communicator

The `dims` array needed by `MPI_Cart_create` can be created using the function

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

where

- `nnodes` Generally the number of processors
- `ndims` Number of dimensions wanted
- `dims` Output of number of processors in each direction

This function creates a "sensible" but not necessarily optimal breakdown.

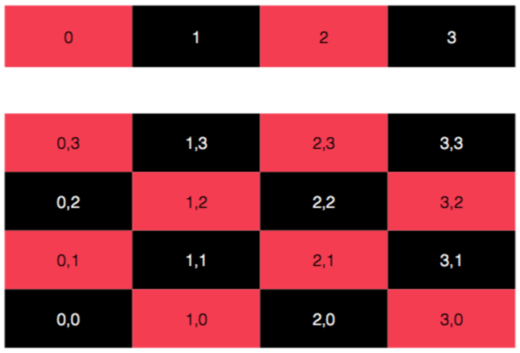

Figure 2.4: An example Cartesian topology

To convert between rank and location, and vice versa, there are the helper functions

```c
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

where

- `comm` A Cartesian communicator from `MPI_Cart_create`
- `rank` Rank (in comm not `MPI_COMM_WORLD`) of processor you want coordinates of
- `maxdims` effectively, the length of `coords` array
- `coords` output array containing coordinates of `rank`

and

```c
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
```

where

- `comm` A Cartesian communicator from `MPI_Cart_create`
- `coords` Coordinates of processor you want rank of
- `rank` output rank for specified `coords`

NB: `coords` must be in range or an error will occur; this function does not return anything like `MPI_PROC_NULL`

To get the neighbours of a given processor use the following function on the processor in question:

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *
    rank_source, int *rank_dest)
```

where

- `comm` A Cartesian communicator from `MPI_Cart_create`
- `direction` 0-based integer saying which direction to get neighbours in
- `disp` displacement of neighbour to get. 1 gets nearest neighbours, 2 the second nearest etc
- `rank_source` rank of processor `disp` lower in `direction`-th coordinate
- `rank_dest` rank of processor `disp` higher in `direction`-th coordinate

Unless periodic boundaries are used, then processors at the edge of the domain will have `MPI_PROC_NULL` for some neighbours. With periodic bounds, the neighbour is the processor on the opposite edge. Using the `periods` argument to `MPI_Cart_create` to wrap the neighbours around, periodic boundary conditions come for free, passing to and receiving from neighbours regardless of whether they are at the domain edges or not.
Note: neighbours are those processors in the same row or column. To get diagonal neighbours, one has to use `MPI_Cart_rank` and calculate the coordinates wanted

2.3 Cartesian Communicator for Case Study

Setting up an appropriate communicator for the Case Study problem goes as follows in Fortran

```
CALL MPI_Init(ierr)
CALL MPI_Dims_create(nproc, 2, nprocs, ierr)
CALL MPI_Cart_create(MPI_COMM_WORLD, 2, nprocs, periods, .TRUE., &
    cart_comm, ierr)
!Rank in new communicator
CALL MPI_Comm_rank(cart_comm, rank, ierr)
!Get the rank of the neighbouring processors in Cartesian communicator
```

```fortran
CALL MPI_Cart_shift(cart_comm, 0, 1, x_min_rank, x_max_rank, ierr)
CALL MPI_Cart_shift(cart_comm, 1, 1, y_min_rank, y_max_rank, ierr)
!Get my coordinates in Cartesian communicator
CALL MPI_Cart_coords(cart_comm, rank, 2, coordinates, ierr)
!Divide the global size (nx x ny) per processor
!Note that MPI_Dims_create works backwards
nx_local = nx / nprocs(1)
ny_local = ny / nprocs(2)
!Calculate what fraction of the global array this processor has
x_cell_min_local = nx_local * coordinates(1) + 1
x_cell_max_local = nx_local * (coordinates(1) + 1)
y_cell_min_local = ny_local * coordinates(2) + 1
y_cell_max_local = ny_local * (coordinates(2) + 1)
```

and in C

```c
MPI_Init(argc, argv);
MPI_Dims_create(nproc, 2, nprocs);
MPI_Cart_create(MPI_COMM_WORLD, 2, nprocs, periods, 1,
    &cart_comm);
//Rank in new communicator might be different
MPI_Comm_rank(cart_comm, &rank);
//Get the rank of the neighbouring processors in Cartesian communicator
MPI_Cart_shift(cart_comm, 0, 1, &x_min_rank, &x_max_rank);
MPI_Cart_shift(cart_comm, 1, 1, &y_min_rank, &y_max_rank);
//Get my coordinates in Cartesian communicator
MPI_Cart_coords(cart_comm, rank, 2, coordinates);
//Divide the global size (nx x ny) per processor
nx_local = nx / nprocs[0];
ny_local = ny / nprocs[1];
//Calculate what fraction of the global array this processor has
x_cell_min_local = nx_local * coordinates[0] + 1;
x_cell_max_local = nx_local * (coordinates[0] + 1);
y_cell_min_local = ny_local * coordinates[1] + 1;
y_cell_max_local = ny_local * (coordinates[1] + 1);
```

Now we have a suitable communicator, the core of the serial code just needs rewriting to use the local sizes, $\mathrm{nx}_{\mathrm{local}}$ and $\mathrm{ny}_{\mathrm{local}}$, and then add boundary conditions.

The boundary conditions have to map between ghost cells and domain cells as shown in Fig 2.3. Considering processors 1 and 2 in the Figure, we need to map $(\mathrm{nx}_\mathrm{l}, 1 : \mathrm{ny}_\mathrm{l})$ on rank 1 to $(0, 1 : \mathrm{ny}_\mathrm{l})$ on rank 2, and also $(1, 1 : \mathrm{ny}_\mathrm{l})$ on rank 2 to $(\mathrm{nx}_\mathrm{l} + 1, 1 : \mathrm{ny}_\mathrm{l})$ on rank 1.

More generally, we need to:
Send right: $(\mathrm{nx}_\mathrm{l}, 1 : \mathrm{ny}_\mathrm{l}) \to (0, 1 : \mathrm{ny}_\mathrm{l})$
Send left: $(1, 1 : \mathrm{ny}_\mathrm{l}) \to (\mathrm{nx}_\mathrm{l} + 1, 1 : \mathrm{ny}_\mathrm{l})$
Send up: $(1 : \mathrm{nx}_\mathrm{l}, \mathrm{ny}_\mathrm{l}) \to (1 : \mathrm{nx}_\mathrm{l}, 0)$
Send down: $(1 : \mathrm{nx}_\mathrm{l}, 1) \to (1 : \mathrm{nx}_\mathrm{l}, \mathrm{ny}_\mathrm{l} + 1)$

The code to do this in Fortran is:

```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)
```

where we use array subsections to show MPI what data to send and receive. Note that we are not using periodic boundaries, so at the edges of the domain the `sendrecv` commands are null operations.

In C the code for a single dimension exchange is:

```
//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));
//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];
}
```

where we have to copy data into temporary arrays because there are no array subsections. There is one code block like this for each of the 4 send/recvs we need to do (although if we're careful, we can reuse the src and dest memory).

In Fortran this works well, although for large problems the array subsections can lead to stack-space issues with some compilers. In C, the code is pretty long and ugly but works fine.

The final result is shown in Fig 2.5. On 16 processors the result is identical to the serial result, as we hoped.

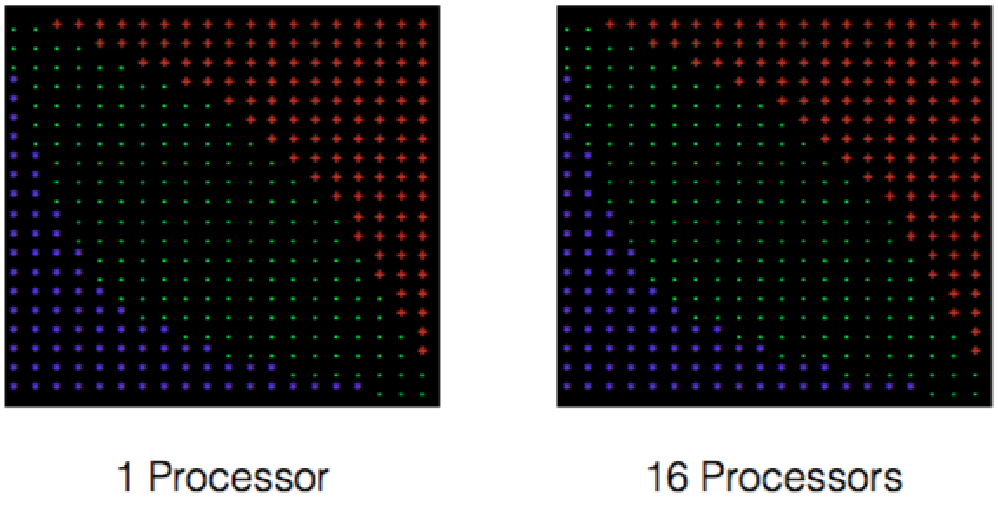

Figure 2.5: The result of the Case Study code

2.4 Problem Specification - A terrible prime finder

A lot of problems can be domain decomposed, although not always in space. Any basis set can be used, and ideally there are minimal connections between them. In space, we used Cartesian coordinates, but one could decompose the same way using Fourier decomposition (good for some wave problems, called spectral) etc.

Some problems though, just aren't amenable to this sort of treatment. A classic example is a large amount of semi-independent working, which must be generally driven by some co-ordination between processors. If the co-ordination is minimal, we can "domain decompose" across the work, but if not, this won't gain us as much as we'd hope. This is especially the case if we can't write down how to split up the work, but we can find splits as things run.

In this case, we might turn to a different model, the Worker-Controller (sometimes called Master-Slave). Here, one processor is special; it is the Controller, and it is in charge of splitting up and re-combining the work. The rest are Workers - they get given some work to do and they do it, returning some results. We usually call the work a "work package".

One immediate, major example of this work-package model is that if the packages take highly varying times to complete, there's a lot less time wasted[3] as any workers who become free can immediately take the next package.

Even better, in this model sometimes the controller can itself service work packages. This can be tricky to get right, as if the controller is busy working there will be delays in responding to requests, which can cost more than you gain. On the other hand, if there is some cheaper work the controller can do, and especially if this work can be

[3]Ideally, none, but that's rarely possible. If you get unlucky and one worker gets "too many" hard packages, there'll be some waste.

paused and resumed, and even more so if it can reduce the number or load of work packages somehow, there can be massive savings.

For example, with the case we describe in the next section, we can improve our basic algorithm by using the controller process to thin the pool of work packages we need to execute, using a different process to the full execution the workers complete. This work is continuous and can easily be interrupted and resumed, so using some non-blocking comms (see later) we can make this very elegant and efficient.

2.4.1 The Terrible Primer Finder Algorithm

Suppose we want to find all prime numbers in an interval $[a, b]$ (between a and b, inclusive of both ends). A prime number, by definition, is a number which is divisible by only itself and 1, i.e. it has no other factors. So, 5 is prime, but 6 is not - 6 is a product of 2 and 3, and is called a composite number.

The simplest way to identify primes is “trial division” - try and divide by every possible factor, and as soon as we find one that divides exactly, our number cannot be prime. Maths tells us two useful things - if a number, N, divides by some other number, m, then N must divide by everything which m divides by (say m = p*q, then N = p*q*r, where r = N/m). If this sounds trivial, it sort of is, but it does need proving in proper maths. This also means that there is exactly one way to write any number as a “product of primes”.

Taking the first few primes, 2, 3, 5, 7, 11, 13, 17 (1 is usually not counted as prime) - we see there is exactly one way to write 15 as a product: its 3×5. No other numbers in that list work. Similarly, $21 = 3 \times 7$ and can't be written any other way.

Those two things together mean that we only have to check whether a number divides by potential factors in the list of primes, and we can stop at the first one that works. Consider 12, for example. It is $2 \times 2 \times 3$. Once we've checked 2, we don't need to check 4, or 6, or any other even number as these absolutely cannot be factors, since they divide by 2. Also, since we already know there is a factor, 2, then 12 cannot be prime. On the other hand, for 13, we have to check 2, 3, 5, 7 and 11, and only when ALL of them fail can we say 13 is prime.

This is an example of an “early break” condition, and means that the work required to check a number varies greatly depending what the number is. For 12 we do 1 division, but for 13 we have to do 5. For 15, we have to do 2 (factors 2 and 3). Since the distribution of primes isn't simple (or even known exactly), we can't work out how to combine work packets to give each worker the same amount of work.

While the workers work on their trial divisions, the master process waits to dispatch the next candidates to check. We can see quite easily that the master could remove candidates that divide by 2 by skipping every 2nd number. It could also skip every 3rd, 5th etc. This is called a “sieve” (specifically `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`), and is work the master can do slowly while the workers work.

Important Note: this combined algorithm is not terribly sensible - we might be better just running a parallel sieve alone, without any trial division. But we want to illustrate the power of the Worker-Controller model and this is a good simple example. The real problem with it is that unless we use very large numbers, the communications overhead is very large and we'd be better using a simple split and accepting the inefficiency. But case studies are rarely perfect.

2.4.2 The Code

An example of this model is included in the code packet with this course, in C, Fortran and Python. You will probably notice the overheads if you try writing a simple sieve algorithm for comparison. If you want to explore this further, look into sending larger work packages - for instance having the worker check 100 numbers, not 1. This reduces the ratio of comms to work, and helps the efficiency.

Chapter 3

MPI Types

3.1 Basic Custom Types

MPI provides data types for things like integers and floats, that tell MPI how much data to send with a given element. We went over the built-in types briefly in 1.2.2 - recall that C and Fortran use slightly different namings.

The built-in types are essential for communications, but there is a far more powerful option - writing our own custom types. Just as we would use TYPES (in Fortran) or structs (in C), we can use a group of elements as though they were one - passing to a function for instance. In MPI, we can create custom types to streamline our comms operations.

For instance, an obvious need is for a way to describe not a single basic type element, but many of them, i.e. an array. We'd also like a type so we can send a structure smoothly, or communicate some sub-section of an array. While all of these can be done with the basic commands, we'd prefer something more elegant!

3.1.1 MPI Contiguous

Recall the ring-pass example, but suppose rather than a single element we want to send nitems at once. We can obviously do this using the `count` parameter, like

```
MPI_Sendrecv(values, nitems, MPI_INT, right, TAG, values_recv, nitems,
    MPI_INTEGER, left, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
```

and

```
MPI_Sendrecv(values, nitems, MPI_INTEGER, right, TAG, values_recv, nitems
    , MPI_INTEGER, left, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
```

where both values and values_recv are now arrays so in C we no longer require the explicit address-of.

The alternative to this is using `MPI_Type_contiguous` to create a type consisting of `nitems` copies of the basic type. To create the type:

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *
    newtype)
```

- `count` number of elements of `oldtype` in `newtype`
- `oldtype` datatype to base this new type off. Usually, but not necessarily, a primitive MPI type
- `newtype` output containing the newly created type.

Note that the *contiguous* part means that the elements must be stored in order and without gaps such as in a C or Fortran array.

3.1.2 Committing Types

Now the type is created, can it be used immediately? Not yet! If you try it it may work, but it can't be relied on and can change, especially if you create any more types. Instead it must be *committed* first, so that it is registered with the MPI type system. The commit function is

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

where `datatype` is a type created with type_create.

Once committed, custom types can be used just as normal types[1]

The opposite of the commit is the free,

```
int MPI_Type_free(MPI_Datatype *datatype)
```

where `datatype` is a type created with a creation routine *that has been committed.* After the free operation, the type can no longer be used in MPI operations, just as if it had never been committed.

3.1.3 Using a Type

The ring-pass example using this is then, in Fortran,

```
CALL MPI_Type_contiguous(nitems, MPI_INTEGER, contig_type, ierr)
CALL MPI_Type_commit(contig_type, ierr)
CALL MPI_Sendrecv(values, 1, contig_type, right, tag, values_recv, &
1, contig_type, left, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
```

and in C

```
MPI_Type_contiguous(NITEMS, MPI_INT, &contig_type);
MPI_Type_commit(&contig_type);
MPI_Sendrecv(values, 1, contig_type, right, TAG, values_recv, 1,
    contig_type, left, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

[1] There is a caveat about reduction operations, such as Reduce and Allreduce - custom types need custom operators to know how to do the reduction - but you can avoid needing to do this by falling back to the less elegant long-form option

We create and commit the type, and then send as before, only now the `count` parameters are back to 1 because we are sending only one block of our custom type.

There aren't many actual reasons for doing this with a contiguous type. In theory it might be faster, although we have never observed it to be. It can be used to send more than 4GB[2] of data: note that the counts are integers so this is their maximum size. Type lengths are also integers, so we could use them to send up to 19 EB (Exabytes) of data. However, the contiguous type has some uses and is the simplest type to start with.

3.1.4 Indexed or block-based type

Imagine trying to send the red cells in Figure 3.1. For this we can use the special type `MPI_Type_indexed` which encodes a series of blocks of data and their relative displacements.

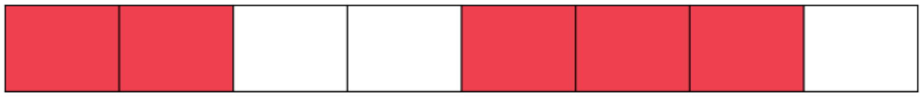

Figure 3.1: A Subset of Cellular Data

```
int MPI_Type_indexed(int count, const int *array_of_blocklengths, const
    int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *
    newtype)
```

- `count` number of blocks
- `array_of_blocklengths` array of length of blocks in multiples of `oldtype`
- `array_of_displacement` array of starts of blocks in multiples of `oldtype`
- `oldtype` Type to base custom type on. Usually (not necessarily) a primitive type
- `newtype` value to return newly created indexed type

For the data in Figure 3.1 we then have

```
  lengths[0] = 2; lengths[1] = 3;
  displacements[0] = 0; displacements[1] = 4;
  MPI_Type_indexed(2, lengths, displacements, MPI_INT, &index_type);
  MPI_Type_commit(&index_type);
```

[2] 2GB using some implementations

or

```
 lengths = (/2, 3/)
 displacements = (/0, 4/)
CALL MPI_Type_indexed(2, lengths, displacements, MPI_INTEGER,
    index_type, &
     ierr)
CALL MPI_Type_commit(index_type, ierr)
```

General indexed type

There also exists a more general version of the indexed type which allows you to specify the offsets in bytes rather than in units of `oldtype`, allowing you to deal better with mixed data. This is called `MPI_Type_create_hindexed`. Do not confuse this with the function `MPI_Type_hindexed` which is *deprecated* and must not be used in new code.

3.1.5 Mixing Types

Mixing MPI types in send and receive operations provides a lot of power, allowing the remapping of data in the process. For instance we can map from one set of cells in the array above, to another set, as in Fig 3.2. Initially the source array is set to values 1-8, while the destination is all 0. After the comms step, the destination is as shown.

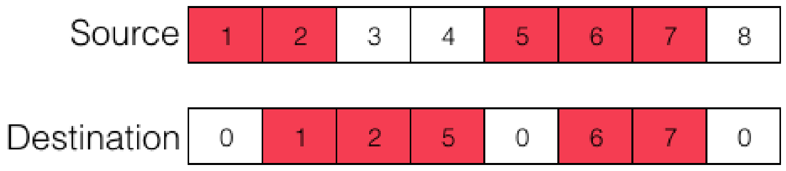

Figure 3.2: Mixed Send-Recv types to remap data

The code to do this is, in Fortran,

```
 lengths = (/2, 3/)
 displacements = (/0, 4/)
CALL MPI_Type_indexed(2, lengths, displacements, MPI_INTEGER,
    index_type, &
     ierr)
CALL MPI_Type_commit(index_type, ierr)

 lengths = (/3, 2/)
 displacements = (/1, 5/)
CALL MPI_Type_indexed(2, lengths, displacements, MPI_INTEGER, &
     index_type_recv, ierr)
CALL MPI_Type_commit(index_type_recv, ierr)
```

and in C

```c
lengths[0] = 2; lengths[1] = 3;
displacements[0] = 0; displacements[1] = 4;
MPI_Type_indexed(2, lengths, displacements, MPI_INT, &index_type);
MPI_Type_commit(&index_type);

lengths[0] = 3; lengths[1] = 2;
displacements[0] = 1; displacements[1] = 5;
MPI_Type_indexed(2, lengths, displacements, MPI_INT, &index_type_recv);
MPI_Type_commit(&index_type_recv);
```

3.2 Types Describing Structures

Types are also a handy way to simplify or streamline the sending of structures (and some Fortran TYPEs). Lets call each field of our structure a block (for clarity between languages). Then to describe a structure in memory we need to specify:

- The offset of each block of the struct from the start of the struct as a number of *bytes*. Do note that this isn't something you can write down - compilers might pad (see padding) data types - so locations might not be what you thought.
- The datatype of each block, which tells us its length in memory
- The number of sub-elements in a block, for instance rather than a simple int it might be an array of 3.

When we're trying to build an MPI type, we want to specify the type as an MPI datatype, and the number of sub elements relative to this.

The function we use is

```c
int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
    const MPI_Aint array_of_displacements[], MPI_Datatype array_of_types,
    MPI_Datatype *newtype)
```

- `count` Number of blocks
- `array_of_blocklengths` array of lengths of blocks. The lengths are in multiples of the datatype given in `array_of_types`
- `array_of_displacements` array of offsets to the start of each block, in bytes
- `array_of_types` Types of each block
- `newtype` Value to return newly created type

In C, the signature introduces a new variable type, `MPI_Aint` which is an integer guaranteed to be long enough to hold a memory address (pointer) on this system. In practice, this is almost always the same size as a `size_t`.

In Fortran, this is instead an INTEGER KIND, `MPI_ADDRESS_KIND`, which we use like `INTEGER(MPI_ADDRESS_KIND) :: address`

3.2.1 Describing a simple structure

As an example, consider the structure in figure 3.3 below. This is a single integer, a pair of integers, and a single float.

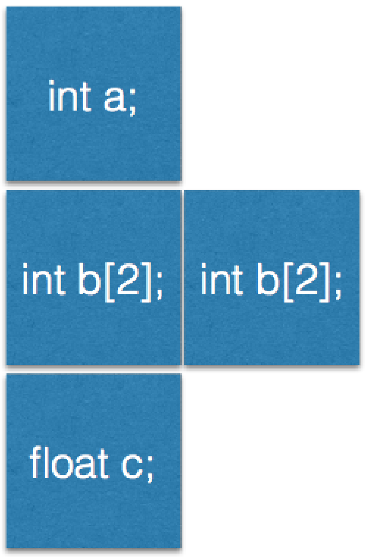

Figure 3.3: Illustration of array ordering in MPI Code

The types for this are obvious and easy - we have an `MPI_INT`, a pair of type `MPI_INT` and an `MPI_FLOAT`. The blocklengths are also easy - they are 1, 2, 1.

The offsets aren't something we can see by looking at what we've defined, as we noted above. Instead we use a function to find out for us, like this:

```
typedef struct {
  int a;
  int b[2];
  float c;} mystruct;

displacement[0] = offsetof(mystruct, a);
displacement[1] = offsetof(mystruct, b);
displacement[2] = offsetof(mystruct, c);
```

The complete code to create this type in C then becomes

```
typedef struct {
  int a;
  int b[2];
  float c;} mystruct;
```

```
MPI_Datatype types[3];
MPI_Aint displacement[3];
int lengths[3];
displacement[0] = offsetof(mystruct, a);
displacement[1] = offsetof(mystruct, b);
displacement[2] = offsetof(mystruct, c);
types[0] = MPI_INT;
types[1] = MPI_INT;
types[2] = MPI_FLOAT;
lengths[0] = 1;
lengths[1] = 2;
lengths[2] = 1;

MPI_Type_create_struct(3, lengths, displacements, types, &struct_type);
```

Unusually, the Fortran is more tricky than the C, since we're not as exposed to pointers and the underlying memory. There are horrible ways to find the offsets etc using C/Fortran interoperability, but this is not necessary or recommended.

For Fortran, the fields in a type are free to be re-ordered behind the scenes. To get their order guaranteed, mark your structure either as `BIND(C)` (meaning it can inter-operate with C code), or as `SEQUENCE`.

Now the function to get the locations is

```
int MPI_Get_address(const void *location, MPI_Aint *address)
```

- `location` The item you want the address of
- `address` The returned address as distance from the bottom of the MPI memory space

The address is returned as a slightly mysterious distance from the "bottom of MPI memory space" - i.e. it is a location relative to all of the memory in the program. We want only the offset from the start of our type - so we want to subtract this offset. But we can't do this using just an arithmetic operation, because `MPI_Aint` is special. Instead we use the following to add

```
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
```

and to difference

```
MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)
```

Specifically, the latter returns addr1-addr2.

Note that even in the Fortran there is no final error parameter.

The full Fortran code to describe our type is then

```
TYPE mytype
    SEQUENCE
    INTEGER :: a
    INTEGER, DIMENSION(2) :: b
    REAL(KIND(1.D0)) :: c
```

```
END TYPE mytype
INTEGER(MPI_ADDRESS_KIND) :: base
INTEGER(MPI_ADDRESS_KIND) :: &
DIMENSION(3) :: displacements
TYPE(mytype) :: data

CALL MPI_Get_address(data, base, ierr)
CALL MPI_Get_address(data%a, &
    displacements(1), ierr)
CALL MPI_Get_address(data%b, &
    displacements(2), ierr)
CALL MPI_Get_address(data%c, &
    displacements(3), ierr)

displacements(1) = MPI_Aint_diff( &
    displacements(1), base)
displacements(2) = MPI_Aint_diff( &
    displacements(2), base)
displacements(3) = MPI_Aint_diff( &
      displacements(3), base)
```

3.2.2 Using Structure Types

While creating them is a bit fiddly due to finding all the lengths and offsets, using structure types is just the same as any other. For instance we can send arrays of structure types etc.

3.2.3 Non-local Structure Data

Do remember that structures may "contain" data that is not actually local to them - for instance a structure containing a pointer to some data elsewhere. In Fortran, this may be the case for structures containing ALLOCATABLEs as well as POINTERs. In these cases you need to be careful to describe and send the actual data too.

3.3 Back to the Case Study

3.3.1 Array Subsections as a Type

The most generally useful `MPI_Type` routine is one that allows creation of a type representing a subsection of an array. This can replace the clumsy array temporaries we used in the domain decomposition case study before, making the C code much more readable and getting rid of some copying operations in the Fortran. The function is

```
int MPI_Type_create_subarray(int ndims, const int array_of_sizes[], const
    int array_of_subsizes[], const int array_of_starts[], int order,
   MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- `ndims` Dimensions of array
- `array_of_sizes` Sizes of outer array
- `array_of_subsizes` Size of subarray of outer array
- `array_of_starts` Offset in each dimension of subarray
- `order` Whether the array is C or Fortran ordered
- `oldtype` Type of the underlying data in the array
- `newtype` Value to return newly created type

3.3.2 Array Ordering

By default, arrays in C are in Column Major order so the last index varies the fastest. In Fortran arrays are Row Major so the first index varies fastest. Both languages can mimic the ordering of the other. Fig 3.4 shows what this means for our MPI calls.

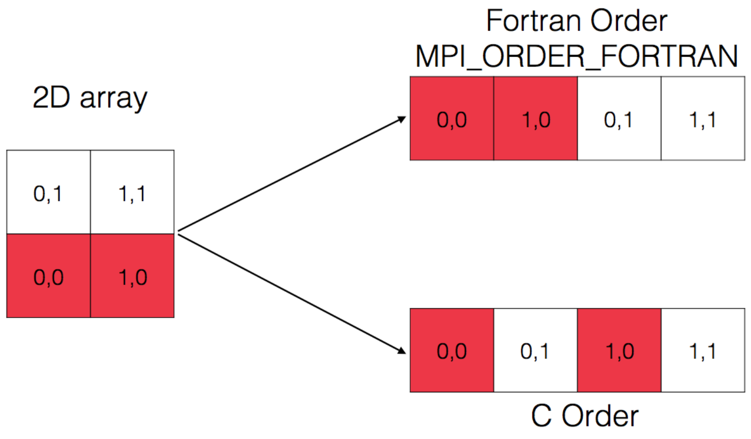

Figure 3.4: Illustration of array ordering in MPI Code

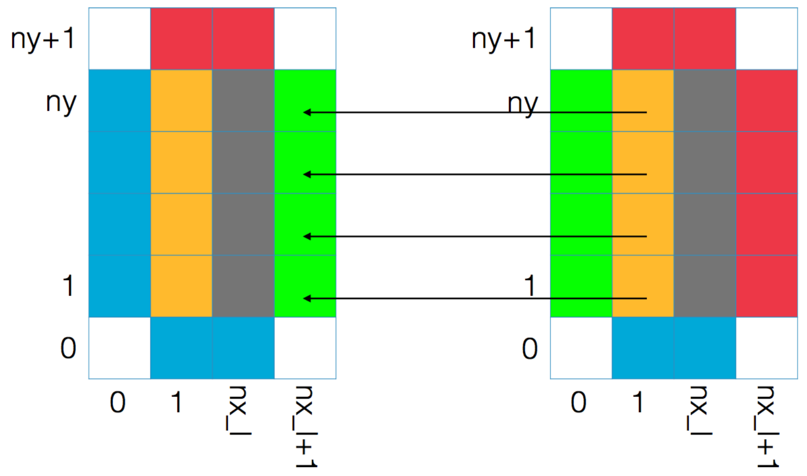

Figure 3.5: Paralellising the case study needs us to copy data from the Yellow cells to the Green cells.

Fig 3.5 shows what we need to do for the Case Study code. One paired send-receive operation needs to send the yellow cells to the processor on the left and to receive the green cells from the processor on the right. We create a type describing the green cells as (using C, since this benefits the most from the types)

```
int sizes[2], subsizes[2], starts[2];

//Whole array is nx_local x ny_local
sizes[0] = nx_local + 2;
sizes[1] = ny_local + 2;
//Want a strip 1 x ny_local
subsizes[0] = 1;
subsizes[1] = ny_local;

//Start 1 cell up, nx_local + 1 in
starts[0] = nx_local + 1;
starts[1] = 1;

MPI_Type_create_subarray(2, sizes, subsizes,
starts, MPI_ORDER_FORTRAN, MPI_FLOAT,
&newtype);

MPI_Type_commit(&newtype);
```

and for the yellow cells

```
int sizes[2], subsizes[2], starts[2];

//Whole array is nx_local x ny_local
sizes[0] = nx_local + 2;
sizes[1] = ny_local + 2;
//Want a strip 1 x ny_local
subsizes[0] = 1;
```

```
  subsizes[1] = ny_local;

  //Start 1 cell up, 1 cell in
  starts[0] = 1;
  starts[1] = 1;

  MPI_Type_create_subarray(2, sizes, subsizes,
  starts, MPI_ORDER_FORTRAN, MPI_FLOAT,
 &newtype);

  MPI_Type_commit(&newtype);
```

In practice we'd usually write a single routine to create and commit the type, like `create_single_type(sizes, subsizes, starts, &newtype);`

We can now use these types instead of the temporary array sections in the case-study code. In Fortran this doesn't make much difference, but in C the code gets much simpler.

The C code for the send-receives is now just

```
  MPI_Sendrecv(data->data, 1, type_l_s, x_min_rank,
      TAG, data->data, 1, type_r_r, x_max_rank,
      TAG, cart_comm, MPI_STATUS_IGNORE);

  MPI_Sendrecv(data->data, 1, type_r_s,
      x_max_rank, TAG, data->data, 1, type_l_r, x_min_rank,
      TAG, cart_comm, MPI_STATUS_IGNORE);

  MPI_Sendrecv(data->data, 1, type_d_s, y_min_rank,
      TAG, data->data, 1, type_u_r, y_max_rank,
      TAG, cart_comm, MPI_STATUS_IGNORE);

  MPI_Sendrecv(data->data, 1,type_u_s, y_max_rank,
      TAG, data->data, 1, type_d_r, y_min_rank,
      TAG, cart_comm, MPI_STATUS_IGNORE);
```

The Fortran doesn't look much simpler, but we no longer need to slice the arrays, so we're a bit less likely to make a silly typing error that could cause bugs:

```
    !Send left most strip of cells left and receive into right guard
        cells
    CALL MPI_Sendrecv(array, 1, type_l_s, x_min_rank, &
        tag, array, 1, type_r_r, 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, 1, type_r_s, x_max_rank, &
        tag, array, 1, type_l_r, x_min_rank, &
        tag, cart_comm, MPI_STATUS_IGNORE, ierr)

    !Now equivalently in y
    CALL MPI_Sendrecv(array, 1, type_d_s, y_min_rank, &
        tag, array, 1, type_u_r, y_max_rank, &
```

```
         tag, cart_comm, MPI_STATUS_IGNORE, ierr)

    CALL MPI_Sendrecv(array, 1, type_u_s, y_max_rank, &
         tag, array, 1, type_d_r, y_min_rank, &
         tag, cart_comm, MPI_STATUS_IGNORE, ierr)
```

And that's it - that's all the basics of Types and how to use them!

Chapter 4

Non-blocking MPI Communications

4.1 Concepts of Non Blocking Communications

Way back in Chapter 1 we saw that we could replace a Send and Recv call with a single Sendrecv, so that we didn't suffer from deadlocks in a single set of sends and receives. But what if we want to do many send-recv processes at once? For instance, instead of the simple ring pass, suppose we wanted to pass something to the processor one over, something else to the one two over, etc, and we want them all to go at once.

The solution to this is non-blocking communications - comms where we can send or receive data and then immediately go on to other things while the communication occurs in the background. Once the communication is all done, we can then do whatever we needed to with that data.

4.1.1 Why do we use this?

Why might we want to do this? Most of the time we can re-arrange our communications so that the usual blocking operations will work, but sometimes this can be difficult or inefficient. For instance, if we don't need synchronisation, but we need to send data to another processor to be used at some future time (the recipient will wait until it receives the data to act on it), we might find it easier and safer from deadlocks or livelocks to use non-blocking receives. Suppose we are running distributed calculations and want to return all our answers to the master process - the master can't know when a message will come in, so it is difficult to ensure it is waiting when another process is trying to send (sometimes giving a livelock, or if you're unlucky can turn into a deadlock too) or it has to check for messages periodically, and be allowed to carry on if one is not received.

4.1.2 Latency and Latency Hiding

More commonly, you would have the master sat waiting for the child processes to send data, and while that works and is safe, it wastes a lot of the master's time. We saw earlier that with blocking commns there can be substantial delays where one processor has to wait until another is ready to receive, and we'd prefer that to be done "in the background". This is called latency hiding.

Recall that latency is roughly the time taken for comms to begin, whereas bandwidth is the rate of data transfer once it's going. The figure below (4.1) illustrates this graphically.

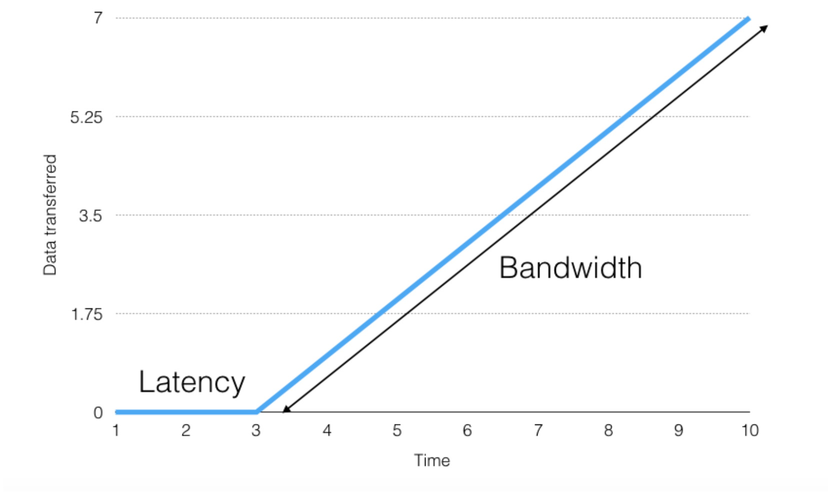

Figure 4.1: Latency versus Bandwidth

Non-blocking comms lets us "hide" the latency, because the sending processor isn't waiting for the recipient to be ready - it has passed off its data and is off working again. The recipient reaches a point where it is ready to receive, and the data is already prepared for sending.

For a simple analogy, think of ordering food from a takeaway. You could walk into restaurant and place an order, and then wait until it is ready. This is like "normal" communications - you are "blocked" and can't do anything else until the food is ready and you can collect it and leave. Alternately, you could telephone-in the order before hand, and go pick it up when it is ready. This is latency-hiding - it takes just as long to cook your food, but you don't have to sit waiting for that period - you can be doing other things. Note that if you are early, or your order unexpectedly takes longer to prepare there will be some remaining latency period - hiding cannot always remove it all.

We'll come back to latency hiding at the end of the chapter

4.1.3 Why don't we use non-blocking everywhere?

So why don't we use non-blocking comms everywhere and not think about it? Well, there's a few reasons. Firstly, it is usually harder to write - you have to make sure the data is ready and valid when you need it. There is also a bit more code you have to write. Next, there are some more restrictions on how you can use data - for instance you can't change data which is in the process of being sent. Finally, there are all sorts of new and exciting bugs you can produce.

However, if non-blocking communication is a natural way to describe your communication patterns, they can greatly simplify and speed up your code. *This is a common theme in programming - using a powerful tool where warranted simplifies - using it where not warranted complicates.*

4.2 Non-Blocking Commands

The basic send and receive commands are very similar to the blocking variants we have already seen. They are:

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

- `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` number uniquely identifying this message (must be unique between any messages currently in flight)
- `comm` an MPI communicator to send to
- `request` a request object that is used to uniquely identify this send operation later

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

- `buf` a buffer to hold the received data
- `count` number of elements to receive
- `datatype` type of the elements to receive
- `dest` which rank data comes from
- `tag` number uniquely identifying this message (must be unique between any messages currently in flight)
- `comm` an MPI communicator to receive from

- `request` a request object that is used to uniquely identify this receive operation later

We see that the send operation has gained a parameter, request, and receive has gained this and also lost the status object. Requests are a sort of handle that identifies the operation - so we can have multiple sends in operation and tell them apart. You need to keep hold of the request objects!

4.2.1 Inter-operation of Send and Receives

All of the MPI send and receive operations can inter-operate. That is, we can use different variants on the send and receive end. For instance, we could do a non-blocking send, but a blocking receive.

This is actually quite a common use-case - we want the sender to go on with its work, but on the receiver we need the data receive to be complete before we can go on.

4.2.2 Requests

The request handle if used to identify the message later if you want to check whether it has completed. The request is a unique handle for very send and receive call. As usual, there is a special parameter, `MPI_REQUEST_NULL` which causes all functions that operate on requests to do nothing and return immediately. This might sound silly but it saves branches - all processors can run the same commands, but only those with valid requests will actually do anything.

Generally, we will use the request handle to find out whether a send or receive has completed - and we really must keep hold of it until it has and we've dealt with that. There are two basic classes of functions to do this: Wait and Test. The Wait variants will wait (block, pause etc) until a request or set of requests has completed[1]. The Test variants will test the state and return immediately whether completed or not. The specific functions are:

- `MPI_Wait` - Wait until a given request has finished and return information about its status

- `MPI_Waitall` - Wait until all requests in a list have finished and return information about all of them

- `MPI_Waitany` - Wait until one request in a list has finished and return information about the one that finished

- `MPI_Waitsome` - Wait until at least one request in a list has finished and return information about all requests that are marked as finished

[1]See below for what "completed" actually means

What is a Completed Operation?

This is very important. Completion has different meanings for sends and receives!
For a receive operation, completion means that the data has arrived on the receiving rank and is ready for use. For a send operation, completion means that the data has left the sending rank (so the buffer may be reused) NOT that it has been received.
As the sender, you have to send an explicit message to verify if a recipient has received its data.

The `MPI_Waitany` and `MPI_Waitsome` might seem quite similar and a bit redundant, but they are very useful. Recall that we are usually doing some computation while non-blocking communication is happening, because this is how we get the latency hiding effect. This means that more than one message may have finished while we were doing that. Waitany will pull off the first request in the queue which is completed, and leaves the others, whereas Waitsome will pull off ALL of the completed ones. One or other of these might be much more elegant in our wider code.

The signatures of the waits are:

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

- `request` the request object that we're waiting for, from an `MPI_Isend` or `MPI_Irecv` call
- `status` a status object (same as we use in `MPI_Recv`) to be filled with information about the request

Note that `MPI_Wait` will block until the given request has completed.

```
int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status
    array_of_statuses[])
```

- `count` the number of requests we're waiting for
- `array_of_requests` an array of request objects to wait for
- `array_of_statuses` an array of status objects to be filled

The array lengths should be `count`. Note that `MPI_Waitall` will block until all of the given requests have completed.

```
int MPI_Waitany(int count, MPI_Request array_of_requests[], int * index,
    MPI_Status * status)
```

- `count` the number of requests we're waiting for

- `array_of_requests` an array of request objects to wait for
- `index` an integer that will be filled with the number of the request that completed
- `status` a status object which will be filled with the information about the request that completed

Note that `MPI_Waitany` will block until one of the the given requests completes and returns information about this one.

```
int MPI_Waitsome(int count, MPI_Request array_of_requests[], int *
    outcount, int array_of_indices[], MPI_Status array_of_statuses[])
```

- `count` the number of requests we're waiting for
- `array_of_requests` an array of request objects to wait for
- `outcount` the number of requests (in the given list) which have completed
- `array_of_indices` an array of length outcount containing the indices of the completed requests
- `array_of_statuses` an array of status objects to be filled (also of length outcount)

Note that `MPI_Waitsome` will block until at least one request has completed, and return all of them. This means it will return multiple requests if and only if multiple have completed before it is called.

The testing variants are very similar, but in this case they don't block, they simply return immediately with information on completion. For all but `MPI_Testsome`, there is an additional integer (LOGICAL in Fortran) parameter `flag` that returns the truth of the test. For `MPI_Test` and `MPI_Testall` this is true if all of the given requests are complete. For `MPI_Testany`, it is true if *any* of the requests have completed. For `MPI_Testsome`, instead of a flag, there is an array of indices returned for the completed requests, like `MPI_Waitsome`.

4.2.3 IMPORTANT: Request Expiry

Once you have waited or successfully tested for completion of a request, it becomes invalid, and is replaced by `MPI_REQUEST_NULL`. You can no longer wait or test for this handle, even if you tried to sneakily keep a copy of it - that will lead to errors or bugs. So, as soon as you know a request is complete, you have to service it - you can't use the request handle to identify it any more.

Note there is one exception to this - persistent handles - that we will come to later.

4.3 Caveats of Non-Blocking Comms

4.3.1 Changing Data in Send Buffers

For non-blocking sends, one does have to be careful. While `MPI_Isend` returns immediately, *it does not take a copy of the data you gave it* (and often you wouldn't want it to as that could be a lot of memory use). If you change that data before the send actually occurs or completes, the changed data will be sent. If you happen to make changes *during* the sending, you might get a mix of old and new data.

So remember, once you have called `MPI_Isend` on some data, you mustn't touch that data until you know the sending is complete.

4.3.2 Memory Allocation of Buffers

Similarly, you must ensure that send and recv buffers remain allocated and valid until the corresponding send or recv completes, or MPI will access invalid memory, leading to crashes, bugs, and horror.

This is particularly easy to accidentally do by using a non-blocking send on a function-local variable. On exit of the function, this variable leaves scope and the read or write will no longer be a valid memory usage. So you either have to wait within the function for the completion, or you can use global variables.

So remember, be very careful about ensuring your send or recv buffers are in scope when the actual send or recv happens.

4.3.3 Fortran Array Temporaries

For Fortran, there is another common way of mis-scoping a variable - array subsections or slices *may* create temporary variables. These scoped locally - if you use one in a function call it is scoped only to the function call itself. But the temporary creation does not have to happen, and in fact often doesn't if the slice is contiguous in the underlying memory. So you may have it work, or appear to work, in some circumstances and on some compilers.

DO NOT rely on this working - it is risky and can lead to horrible bugs.

4.3.4 MPI Types for Non Blocking

Often the reason for using an array subsection or a temporary variable is that you need to send part of a persistent array in which the data is changing as the code runs. You can do this more simply by using a Type to describe the part you wish to send, and passing the whole array for this type to act on. MPI will sort out moving just the parts dictated by the type.

Doing this is a lot simpler, but you must still remember not to change the data until the sends are complete!

4.3.5 Data Races

If you have multiple non-blocking receives that all write to the same part of memory, then you cannot know beforehand the order in which they will complete, so (assuming they are trying to write different values) the final value of the contentious element is unpredictable. For blocking receives, you know it is always the last one that will have set the state. For non-blocking, it will be the last one, and you might be able to find out (with the `Test`) functions which this was, but overlapping like this is almost never a sensible thing to do.

Either all of the receives are actually setting the value to be the same, in which case you are sending unnecessary data and can probably reduce your comms load, or they are setting different values, and you have a needless reproducibility problem. This isn't actually invalid, but it's unexpected and rarely a good idea.

4.4 Persistent Communication

As we noted in the previous section, with non-blocking communications there are a lot more restrictions on when data you give to MPI has to be available. There is another variant which adds even more restrictions, but has some advantages in some circumstances.

Suppose you have some persistent data arrays that are around for the entire lifetime of your code (or at least a significant part of it), and want to send some parts of them a lot. Then you can given the information about what to do in a send/recv early on, and re-use that setup multiple times.

This is even more restrictive than the normal non-blocking comms, because these arrays/variables have to be available for specification when you do the setup and any time you use it. On the other hand, doing this step only once does save some overheads. Also, if you design things carefully you can have less repeated code - rather than multiple Send/Recv sections, you have one setup part and some calls to a function `MPI_Start` at the places the actual comms happens. This can make maintenance easier, as you have less to change if, for example, the sources or the sizes of data needs to change.

4.4.1 Commands for Persistent comms

The commands you'll use are:

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

- `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` number uniquely identifying this message (must be unique between any messages currently in flight)
- `comm` an MPI communicator to send to
- `request` a request object that is used to uniquely identify this send operation later

```
int MPI_Recv_init(const void *buf, int count, MPI_Datatype datatype, int
    source, int tag, MPI_Comm comm, MPI_Request *request)
```

- `buf` a buffer to hold the received data
- `count` number of elements to receive
- `datatype` type of the elements to receive
- `dest` which rank data comes from
- `tag` number uniquely identifying this message (must be unique between any messages currently in flight)
- `comm` an MPI communicator to receive from
- `request` a request object that is used to uniquely identify this receive operation later

and you might notice that, apart from the names, these are identical to the non-blocking `Isend` and `Irecv` from earlier. This is because they are essentially the same, but these don't do the communication when they're called, but just set up the infrastructure. Communication is started using some extra functions, `MPI_Start` and `MPI_Startall`, which look like:

```
int MPI_Start(MPI_Request *request)
```

- `request` a request object from a `MPI_Send_init` or `MPI_Recv_init` call

```
int MPI_Startall(int count, MPI_Request array_of_requests[])
```

- `count` the number of requests to start
- `array_of_requests` an array of length `count` of request objects from a `MPI_Send_init` or `MPI_Recv_init` call

The Start functions initiate the actual communication, after which you can test the state using the Wait and Test functions. Importantly, unlike with non-blocking comms, the request handles for persistent comms *do not become invalidated* when tested, but they do become, in some sense, “completed”.

4.4.2 Sequencing

As we just said, the persistent communications handles can be re-used. However, it is not valid to overlap comms - calling `MPI_Start` on a request that is already active is not allowed. You must wait until the request completes and has been serviced (i.e. you have done what you needed to with the received data). You are also not allowed to change the data between a Start, and a Wait or successful Test, or rather, you may get odd results if you do. Only once you have a completed Wait or a successful Test, may you alter the data or call `MPI_Start` again.

In other words,

1. before Start is called, a Request is a description of comms that will be done
2. after Start but before completion it is a "in-flight" request and the states of the data are unknown
3. after completion, which you identify with Wait or Test, the comms is completed, and the request is back to being valid but not active
4. you may now alter the data
5. when you are again ready to communicate you return to step 1

4.5 Latency Hiding Again

We mentioned latency hiding earlier, and in this section we show an example of actually doing it. You may also find this described as "overlapped comms and compute" or "hidden communication".

The practical implementation is always roughly the same - you start the comms as non-blocking as early as possible, then on the receiving end you do all of the computation you can while waiting for the data. Once you receive the data, you finish the computation you need it for. Note that this may still leave a waiting period, but can reduce it if you're lucky.

4.5.1 Latency Hiding in the Case Study

Recall the stencil we need for the domain decomposed case study, shown below. Notice that we only use cells immediately next to the cell being solved, so we can calculate all of the interior cells before we have data available in the ghost cells. In the figure, we can do the 4 very-central cells - in a real problem the domain would be much larger, and we would be able to do "most" of our comms before the data is ready.

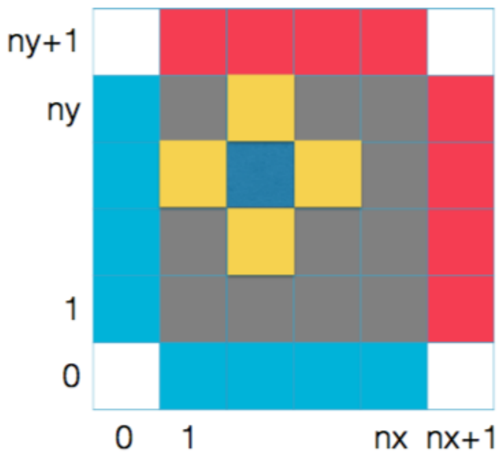

Figure 4.2: The layout for the case study problem. The red and blue edge cells are held fixed. The darker blue cell is the one being solved, the yellow is the cells which we average to update this cell.

This is a good example of how latency hiding works - we can do plenty of compute before the comms are done and then finish off.

We have examples in the code pack of the case study using the non blocking comms, the persistent comms, and also the persistent comms doing latency hiding.

4.6 Final Notes on Non Blocking Comms

4.6.1 Collectives

We've mostly focused on the Send and Recv commands in this chapter, but there exist analogues of the collectives too -

- `MPI_Ireduce`
- `MPI_Igather`
- `MPI_Iscatter`
- `MPI_Ialltoall`
- `MPI_Ibarrier` ??

If you think Barrier is an odd choice, it is, but it's there for completeness and consistency - so that you can always use a Wait or Test regardless of which collective you're working with.[2]

[2]Because Barrier doesn't do anything, we could always swap an Ibarrier plus Wait or Test for a regular Barrier without loss of generality, but it can be neater not to - often we'd prefer all processors do the same thing and this can aid that

The collectives are used the same broad way as the Send and Recvs - you make the call to the function to get a request object, let comms happen in the background, and then test for completion with Wait or Test.

4.6.2 Buffers

There are also buffered versions of the MPI non-blocking send routines. You create a memory buffer large enough to hold all the messages you will ever have "in-flight" at one time (there can be more than one). You use the `MPI_Buffer_attach` function to make it available, and then you send data with `MPI_Bsend`. The data is copied into the buffer before this routine returns, and the send is then non-blocking. You are able to immediately re-use the original location of the sent data.

However, if you underestimate the size of buffer you need, e.g. by accidentally sending more messages than you meant to, your code will generally crash. These functions are handy when required, but are generally considered to be tricksy and hard to use.

Chapter 5

Summary

The vast majority of MPI programs "in the wild" never use anything beyond the simple commands covered in any beginner course, such as ours https://warwick.ac.uk/research/rtp/sc/rse/training/intrompi.

However, that doesn't mean the commands here are useless - far from it! Where you need them, the commands we cover here can vastly simplify your codes as well as improve performance. Combined Sendrecv is incredibly useful, and Types, for example, while a bit tricky at first, will soon become so useful you'll wonder how you ever managed without them.

On the other hand, non blocking comms deserves a brief note of caution. While these commands are sometimes essential, they are tricksy. Like any other sort of asynchronous programming programming, you can no longer rely on things being done when you think they will be - instead you have to check if they are done when you need them. This way of thinking can be hard to adapt to, and can make code much more complicated.

In general, our recommendation would be to be aware of non-blocking comms and use them in any cases where they simplify your communications, but to be wary of any use for optimisation. As with all optimisation questions, you should wait until you have a performance problem - if it aint broke, don't fix it.

5.1 Where to Go Now?

The content in this course takes you a substantial way through all of MPI, but it doesn't cover everything! There are two major topics we haven't even touched on, as well as other bits "round the edges". We cover some of this in a further course, https://warwick.ac.uk/research/rtp/sc/rse/training/advancedmpi.

The first of these, MPI-IO, enables you to write your own output files in parallel. Without this, you either have to use a parallel-IO library (such as PnetCDF or HDF5), or perform some sort of serialisation operator. MPI-IO is not hard once you are comfortable with types, but, file formats *are* hard, and good file formats are very hard! Writing your own is just not recommended without very good reasons, which is why we didn't cover this.

The second big part is called One-Sided communications. Everything we have seen so far is collaborative in some sense - there is a sender and a receiver working together to finish the process. But sometimes we want instead to either "push" data off to another processor, or "pull" data in from another processor, without the other end having to take active part. Clearly this can't be entirely one-sided - for instance the other end has to at least accommodate us doing it - and it is a very different way of working. Occasionally this can improve performance, and it might be worth knowing about, but it is very low level and rarely something you'll actually need.

As usual, the best thing we can recommend is to write programs! The more you write, the more natural MPI thinking will become. Good luck!

Chapter 6

Glossary

Glossary

asynchronous programming Programming where the order of operations is no longer strictly defined. For example, if we send ten emails and deal with the responses as and when they come, whenever that may be and in whatever order, we are working asynchronously. If we sent one at a time, sending the next only after the previous response has been received and dealt with, we are working sequentially. *See also* blocking operations, 50, 52

bandwidth The rate at which data can be transferred in a communication process. *See also* latency, 7

blocking operations Blocking operations are ones that "hold" until something is true or completed, in contrast to "non-blocking" which usually describes an operation which has to be checked for truth or completion. The latter allows for other things to intervene, which is a sort of asynchronous programming process. Note that blocking operations do not always block until an operation is fully completed - just until some guarantee can be met. 4

buffer A fancy name for "memory used to store something". Often buffers are a temporary location used to store something transient. 4

Column Major A 2-D array is usually described as row x column. Column major ordering is when the elements are laid out in memory so that the elements of a column are next to each other, so the first index varies fastest. *See also* Row Major, 34

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. 3, 4, 40, 46, 53

deadlock A state in which nobody can move, because there are dependency loops - for instance A is waiting for B to act, and B is waiting for A, so neither can move. If you've ever see 2 people each trying to wave the other on at a junction, that's a sort of deadlock - they're both stuck waiting for the other to go first. *Contrast* livelock, 38

latency The time required for processes to begin. Usually this is a fixed delay. For instance, in a phone call there is a time required to connect a call before you can start talking. *See also* bandwidth, 7

latency hiding a way of reducing the apparent time taken for communication by allowing it to overlap with other workloads, usually computation. 39

livelock A livelock looks a bit like a deadlock, but importantly both parties can move, they just cannot resolve their ordering. The usual example is when two people meet in a corridor, and both try to step aside - they are moving continually, but never reach a state where either can move ahead. *Contrast* deadlock, 38

non-blocking . *See* blocking operations, 11

padding Computer memory is divided into words, or chunks, and many things work better and faster in chunks that match this. Since these are often longer than a basic numeric type can be, compilers might insert dead space between data items in memory to keep the alignment correct. This is called padding.. 30

pass(ed) by value Different languages pass parameters into functions differently. When passed by value, the current value (at call time) of the variable is copied to a dummy variable inside the function. For example a function

```
FUNCTION   inc(x)
x = x+1
END FUNCTION
y=1
inc(y)
PRINT y
```

would give 1 as y is not changed by the call to inc. *See also* ,

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. 2, 4

root One of the processors in a communicator which does any unique work. 4, 53

Row Major A 2-D array is usually described as row x column. Row major ordering is when the elements are laid out in memory so that the elements of a row are next to each other, so the second (last) index varies fastest. *See also* Column Major, 34

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.