
WARWICK RESEARCH SOFTWARE ENGINEERING

Parallelism Primer Technologies

A Brief Introduction to Threads, OpenMP, and MPI

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



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

March 26, 2025

Contents

Preface	i
0.1 About these Notes	i
0.2 Example Programs	i
1 Pthreads	1
1.1 Other low-level threading libraries	2
1.2 Python and Multithreading	3
2 OpenMP - Open MultiProcessing	4
3 MPI - Message Passing Interface	11
3.1 What to do Now	17
3.2 Other Resources	17
4 Glossary of Terms	18
Glossary	18

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 2020 at the University of Warwick.

This work, except where otherwise noted, is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.



The notes may be redistributed freely with attribution, but may not be used for commercial purposes nor altered or modified. The Angry Penguin and other reproduced material, is clearly marked in the text and is not included in this declaration.

The notes were typeset in L^AT_EX by H Ratcliffe.

Errors can be reported to rse@warwick.ac.uk

0.2 Example Programs

Several sections of these notes benefit from a hands-on look at the concepts and tools involved. Test code is available on Github at <https://github.com/WarwickRSE/ParallelismPrimer>.

Chapter 1

Pthreads

Pthreads stands for *POSIX Threads*, where [POSIX](https://en.cppreference.com/w/cpp/thread/posix) is a standard for a kind of operating system to give things some common abilities and interfaces. See e.g <https://stackoverflow.com/questions/1780599/what-is-the-meaning-of-posix> for details.

For our purposes, we just need to know that all *nix operating systems support Pthreads, so that is Linux, MacOS, AIX and other commercial Unix systems. Pthreads is not hard to use in theory, but it's a very low level library, so it is slow to program and not very common in academic programming. However, the model is very useful to know about, because threads underpin so much of parallel programming.

We are going to use C for our examples of Pthreads, because it is much nicer to do than in Fortran. If C is unfamiliar to you, just ignore all of the weird symbols and focus on the things with “pthread” as part of their name. A simple program is:

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 void* testthread(void* arg){
5     int i =*(int*)arg;
6     printf("Sleeping for %i\n", i);
7     sleep(i);
8     return NULL;
9 }
10 int main (int argc, char ** argv){
11     int i;
12     int ival[8];
13     pthread_t mythreads[8];
14     void* rvals[8];
15     for (i = 0;i<8;++i){
16         ival[i]=i;
17         pthread_create(&mythreads[i], NULL, testthread, &ival[i]);
18     }
19     for (i = 0;i<8;++i){
20         pthread_join(mythreads[i], rvals+i);
21     }
22 }
```

The important bits to note are:

- On line 2 we include the Pthreads library
- The block in lines 4 to 9 is defining a function which takes some argument, interprets it as a number, prints that it is sleeping, and sleeps for the number of seconds, before returning

Because this is C, and because of how pthreads works, we have to use void (typeless) references (C pointers) for the function parameters and return values

In Fortran, we can use Pthreads either via some intermediate library, OR by using a lot of C_pointer types and using the C library

- Lines 10 to 22 are the main program

We create 8 threads in a loop, and in each thread we call the function we just described. The function is passed the number of the thread (0 through 7) (lines 15 to 18)

Then we shut down the threads (rejoin them to the main program) (lines 19 to 21)

As you might infer from this, Pthreads is a very low level way of splitting work up over processors, where we have to create them, tear them down, and dictate what they should each do. This makes it very time consuming to program anything complex using Pthreads. In particular, you have to deal with distinguishing thread-local variables (each thread has its own copy) from shared variables (all threads use the same one), and thinking back to the buffet spoon analogy, you will then have to deal with the potential for multiple threads to touch the same variable etc.

On the other hand, if you just have a list of tasks and want them all to run separately (exploiting multiple cores in the process), it is a useful library, although it remains surprisingly annoying to work out how many cores you have available.

1.1 Other low-level threading libraries

A lot of more modern languages (more modern than the venerable old C) have their own threading models that lie on top of the system-level install of Pthreads. These all differ in the details, but follow the same pattern(s) as above:

- Create threads
- Recombine threads
- Test state of threads
- Use [Mutual Exclusions \(Mutexes\)](#) to control access to shared data

As in the previous section, these libraries are easier to use than Pthreads, but still mostly requires you to manually deal with everything, they just tend to give you more helpers functions to make it a bit easier. We recommend knowing a bit about these

options, but generally don't recommend using them in academic software without a very good reason. Remember, even if you can understand the model, anybody else using your code likely can't.

1.2 Python and Multithreading

Python is a popular language for a lot of reasons, but it is not well suited to parallel programming. There are several reasons for this, one being that it is generally a slow and inefficient language, so turning to parallelism to *speed it up* is a poor choice, as it uses more resources, and you would mostly be better off finding a more efficient solution. Note that this does not apply to libraries like Numpy, Scipy, Numba, Tensorflow, Blas etc, which often make use of multi-threading in their backing C, Fortran etc library. This is not what we're discussing here.

In particular, while Python's native parallelism model is a threading model related to Pthreads, the normal Python interpreter (CPython) can't actually run threads in parallel. PyPy and some other interpreters do not share this pathology, but CPython is far more common and unlikely to change this situation.

In CPython, threads will have to queue up one after the other to do any operations on any Python objects (i.e. any calculations you, or a library you use, write as Python code). Only operations in an external library are allowed to actually run simultaneously. This is called the Global Interpreter Lock. Do note that most of libraries like Numpy on the inside are not restricted like this, but the bits where they read or write python objects are. So if your code is mostly calling external libraries, or makes use of the Numba JIT compiler (in NoPython mode), then threading might help, but you want to write as little Python code as possible.

As a final note on Python and the technologies we are about to discuss, Python has no equivalent of the OpenMP library, but does have a version of MPI called MPI4Py, which is not fully standards conforming, but allows some MPI use in Python code.

For more about speeding up and parallelising Python code, see our course Accelerating Python <https://warwick.ac.uk/research/rtp/sc/rse/training/acceleratingpython>

Chapter 2

OpenMP - Open MultiProcessing

OpenMP is a library for shared memory programming (see the Intro materials) which consists of:

- A set of directives that tell the compiler how to parallelise bits of the code
 - Evidently this means the compiler must be OpenMP-aware to use them, but handily non-OpenMP-aware compilers simply ignore (most) directives, so you don't need two complete code versions
- A library that gives your code access at runtime to information about number of processors available, how to split work up etc

These parts won't compile with a non-OpenMP-aware compiler. You need to use tricks like conditional compilation to remove them for a serial version

OpenMP allows for almost completely general parallel programming on a single physical computer. However, by far the most common use is to split up loops so that different iterations are handled by different processors. This has an obvious limitation - only loops that have independent iterations can be parallelised this way. So, loops which advance a quantity in time, where iteration 2 depends on iteration 1, which depends on iteration 0, cannot be handled this way.

A simple OpenMP code to parallelise a loop is:

```
1 PROGRAM loop_decompose
2   USE omp_lib
3   IMPLICIT NONE
4   INTEGER, PARAMETER :: max_its = 10000
5   INTEGER :: nproc, i, thread_id
6   INTEGER, DIMENSION(:), ALLOCATABLE :: its_per_proc
7   nproc = omp_get_max_threads()
8   ALLOCATE(its_per_proc(nproc))
9   its_per_proc = 0
10  !$OMP PARALLEL DO
11  DO i = 1, max_its
12    thread_id = omp_get_thread_num() + 1
13    its_per_proc(thread_id) = its_per_proc(thread_id) + 1
```

```

14  END DO
15  !$OMP END PARALLEL DO
16  DO i = 1, nproc
17    PRINT '(A, I0, A, I0, A)', 'Processor ', i, ' performed ', &
18      its_per_proc(i), ' iterations'
19  END DO
20  PRINT '(A, I0)', 'Total work on all processors is ', SUM(its_per_proc)
21 END PROGRAM loop_decompose

```

Note on code listings - because of the formatting on these examples, we don't recommend trying to copy-paste this code. All code snippets are included in the Github repo for this course

This uses both facets of the OpenMP library we mentioned: some parts are comments and show how to divide the work, and some are functions provided by the library to see how many processors we have and such. Both of these are highlighted in orange in the above listing.

The directive `$OMP PARALLEL DO` starts what is termed a *parallel region*. Note that this starts with a `!`, which is a Fortran comment, so an OpenMP non-aware compiler will ignore this line completely. `$OMP END PARALLEL DO` ends the parallel region. Inside the region, multiple processors will do work, and it is important to program in a parallel-suitable way. Outside the parallel region, only one processor is working so everything is like normal programming.

OpenMP has plenty of annoying features, but the most irritating for a discussion like this is that the directives aren't quite the same in any of the supported languages. For example, the above example in Fortran uses the Fortran style *DO/END DO* for a loop, whereas in C/C++ it uses *FOR*. In Fortran you have the explicit *Start* and *END* markers, whereas in C you use curly braces (`{}`) like in C code. C++ mostly looks like the C.

In the above example, in the parallel region, even though all the processors are working, each only touches its own element of the array `its_per_processor`. So there's nothing more that needs doing for this code to be parallel-safe.

Note that we used the default number of threads as given by `omp_get_max_threads`. This is the number of *virtual* cores your computer has. If your CPU has [hyperfthreading](#) ability, this will be twice the number of actual cores, otherwise it will match the number of physical cores.

2.0.1 Private and Shared Variables

Note that the code above relies on some default behaviour of OpenMP to get the right answer - specifically how it handles the loop variable when we parallelise. By default, each thread gets its own *private* copy of that loop variable, while all the other variables remain shared between all processors. This is exactly what we wanted in that example, but that isn't always the case.

Re-writing the same program as above to manually handle this, we get:

```

1 PROGRAM loop_decompose
2   USE omp_lib

```

```

3  IMPLICIT NONE
4  INTEGER, PARAMETER :: max_its = 10000
5  INTEGER :: nproc, i, thread_id, its
6  !$OMP PARALLEL PRIVATE(its)
7      its = 0
8  !$OMP DO
9      DO i = 1, max_its
10         its = its + 1
11     END DO
12 !$OMP END DO
13 PRINT '(A, I0, A, I0, A)', 'Processor ', omp_get_thread_num(), '
14     performed ', &
15     its, ' iterations'
16 !$OMP END PARALLEL
17 END PROGRAM loop_decompose

```

In the above example, we separate the `PARALLEL DO` into its two components - first we do the splitting into threads, with the `PARALLEL` part on line 6, and then we decompose the loop part with the `DO` section on line 8. Instead of the array of `its` variables, we use a single private copy in each thread, that we explicitly set to private on line 6 with the `PRIVATE(its)` part.

2.0.2 OpenMP Variable Specifiers

There are four variable specifiers, like the `PRIVATE` we just saw. All of these can be applied to any OpenMP statement that starts a parallel region. These are:

- `PRIVATE`

Each thread has its own version of the variable. These *do not have any particular value, and DO NOT inherit any value the variable had before the threads started*

- `SHARED`

This variable is shared by all threads - changing its value in one thread will change it for all the other threads

- `FIRSTPRIVATE`

Like `PRIVATE` but any value the variable had before entering the parallel region is retained

- `LASTPRIVATE`

Like `PRIVATE` but the last value the variable was given in the thread is retained after the parallel region. In particular, in a loop it is the value from the last iteration, otherwise the last section

2.0.3 Do Not Do This Thing - Race Conditions

In the first example, we used an array so that each thread counted iterations in its own variable. In the second, we had a private variable, and printed it inside the parallel section. We didn't even try to count the total number across all threads. It might be tempting to write some code like this:

Horrible Broken Code

```

1 PROGRAM loop_decompose
2   USE omp_lib
3   IMPLICIT NONE
4   INTEGER, PARAMETER :: max_its = 10000
5   INTEGER :: i, its_global
6   its_global = 0
7   !$OMP PARALLEL DO
8     DO i = 1, max_its
9       its_global = its_global + 1
10  END DO
11 !$OMP END PARALLEL DO
12 PRINT '(A,I0,A)', 'Counter records ', its_global, ' iterations'
13 END PROGRAM loop_decompose

```

But this *DOES NOT WORK*. Here all the threads try and increment a simple variable and this will give *THE WRONG ANSWER*. The problem is that incrementing a counter is not an instantaneous single step, and we get a race condition like the Bowl Of Chips analogy.

Imagine we have two threads each trying to increment a single variable, which starts with a value of 0. Something like this can happen:

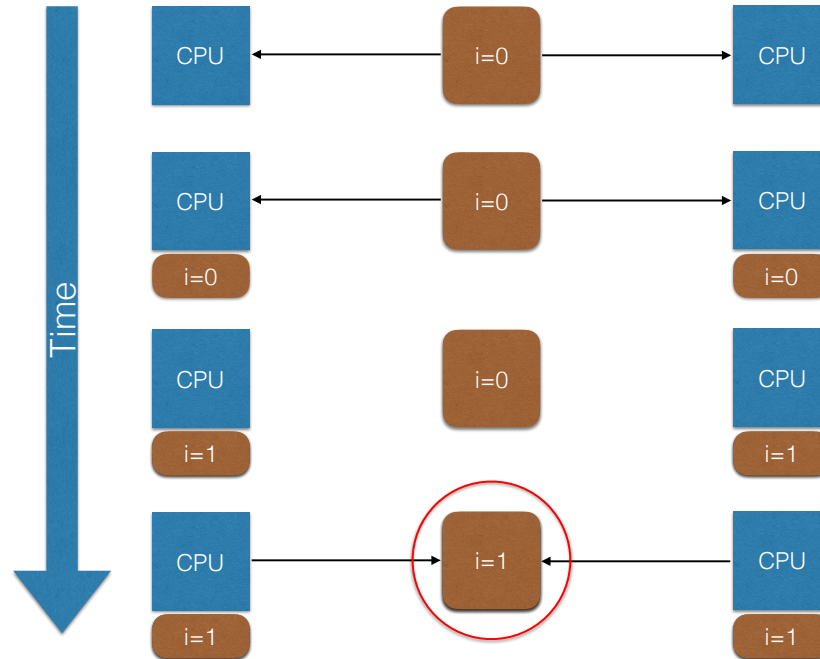


Figure 2.1: A race condition. On left and right are two CPUs, in the middle is the variable we're trying to increment. Each processor must read, increment, and write back, and instead of two incrementings, we only get one! Effectively, the processors are in a race with each other to finish their task before another tries to start.

2.0.4 Solutions to Race Problems

There are 3 solutions to this race problem. In order of increasing generality but decreasing efficiency they are:

1. Atomic Addition - Make the increment or other simple operation a single step that cannot be interrupted
2. Reduction - use OpenMP's ability to do a reduce step (like we discussed earlier), use a variable in each thread and tell OpenMP to sum them all up at the end
3. Critical Sections - create a region of the code that only one thread can be in at a time, and put the increment inside that. C.f. the [Mutex](#).

The code for each of these follows.

Atomic Addition

```

1 PROGRAM loop_decompose
2   USE omp_lib
3   IMPLICIT NONE
4   INTEGER, PARAMETER :: max_its = 10000

```

```

5  INTEGER :: i, its_global
6  its_global = 0
7  !$OMP PARALLEL DO
8  DO i = 1, max_its
9  !$OMP ATOMIC
10     its_global = its_global + 1
11 !$OMP END ATOMIC
12 END DO
13 !$OMP END PARALLEL DO
14 PRINT '(A,I0,A)', 'Counter records ', its_global, ' iterations'
15 END PROGRAM loop_decompose

```

Here we use OMP's directive to force atomic (unsplittable). Inside this block we can have a *single operation* and this will be forced to become a single instruction at the processor level, so cannot be interrupted by some other thread. However, there are limits on what can be done this way - basically you can do only simple assignments ($x=10$) or increment (+1) or decrement (-1) operations.

Reduction

```

1 PROGRAM loop_decompose
2   USE omp_lib
3   IMPLICIT NONE
4   INTEGER, PARAMETER :: max_its = 10000
5   INTEGER :: i, its_global
6   its_global = 0
7   !$OMP PARALLEL DO REDUCTION(+:its_global)
8   DO i = 1, max_its
9     its_global = its_global + 1
10  END DO
11 !$OMP END PARALLEL DO
12 PRINT '(A,I0,A)', 'Counter records ', its_global, ' iterations'
13 END PROGRAM loop_decompose

```

Here we use OpenMP's reduction ability - we specify a variable to be reduced and the operation to be used in it. The variable becomes effectively private, but at the very end the reduction operation is applied across threads.

Critical Section

```

1 PROGRAM loop_decompose
2   USE omp_lib
3   IMPLICIT NONE
4   INTEGER, PARAMETER :: max_its = 10000
5   INTEGER :: i, its_global
6   its_global = 0
7   !$OMP PARALLEL DO
8   DO i = 1, max_its
9   !$OMP CRITICAL
10     its_global = its_global + 1
11 !$OMP END CRITICAL
12   END DO
13 !$OMP END PARALLEL DO

```

```
14 PRINT '(A,I0,A)', 'Counter records ', its_global, ' iterations'  
15 END PROGRAM loop_decompose
```

The part wrapped in the **CRITICAL** section can only have one thread in it at a time and the others must wait their turn. Almost any operation can go within that section, and as many lines of code as you like. However, to allow this generality, there is a lot more efficiency cost than *just* serializing the threads. So, use this with caution and only where the other options don't apply.

2.0.5 Other OpenMP sections

Finally, a few other important OpenMP sections. All of these have to be inside a **PARALLEL** section to work.

- **SINGLE**
One and only one thread will go through this section (at all, not at-a-time)
- **MASTER**
Thread-0, and only thread-0, will go through this section
- **WORKSHARE**
Used to split up non-loop operations. Used mostly to do Fortran array operations

Chapter 3

MPI - Message Passing Interface

MPI is a library used for Distributed Memory parallelism (recall ??). It is entirely up to you to write code to use it, and you also need an installation of the MPI library to compile code using it AND to run code compiled with it. Moreover, you need the same version of both. It is possible to write code to compile with and without MPI support, but you have to use something like conditional compilation to remove them for the serial version. MPI is the most common way of programming for distributed cluster systems.

Strictly, MPI is “only” a set of standards, set out by the MPI Forum (<http://mpi-forum.org>). There are several implementations of these standards, but as long as you stay within the standard, your code will work using any of them. Well, mostly - there are many strange and fascinating bugs that have cropped up in these libraries.

3.0.1 Compiler Wrappers

Generally, when compiling an MPI program you use the compiler wrapper generated by the MPI library when it is installed. From your perspective this works exactly like the normal compiler, but extra things happen behind the scenes.

Normally, this compiler is called mpicc (for C), mpic++ (for C++) or mpif90 (for Fortran), although this is by no means always the case. In particular, the Intel suite uses ifort for it’s Fortran compiler, hence mpiifort here. But mostly, you pre-pend mpi to the compiler name.

3.0.2 Minimal MPI Program

The smallest possible MPI program contains two steps. First, you *Initialize* MPI, before you can use it, or call any other MPI functions. And then, you *Finalize*, after which you must not call any MPI functions.

If you forget to Initialize, then any other MPI commands will fail, and you’ll probably notice right away. If you forget to Finalize, you might not notice, and bad things can happen. Mostly these things are unlikely to cause trouble, but for instance you can get odd, and inconsistent failures where data hasn’t been properly sent etc. Don’t worry though - nothing bad can happen to your computer, just your MPI program.

So, the very simplest MPI code is:

```

1 PROGRAM main
2   USE mpi
3   IMPLICIT NONE
4   INTEGER :: errcode
5   CALL MPI_Init(errcode)
6   PRINT *, "Multiprocessor code"
7   CALL MPI_Finalize(errcode)
8 END PROGRAM main

```

MPI uniquely identifies processors using a number called the **rank**, which runs from 0 to the number of processors minus 1. For more complicated codes, one might want to only use a subset of processors, and this uses an object called a **communicator**. Note that the rank is specific to the communicator (a given processor might have a different rank on different communicators). By default MPI creates the default `MPI_COMM_WORLD` that includes all processors, and that is what we will use here. Do remember that you can create others, and if you're working with somebody else's MPI code they may have done this.

The simplest marginally useful MPI code is:

```

1 PROGRAM main
2   USE MPI
3   IMPLICIT NONE
4   INTEGER :: errcode
5   INTEGER :: rank, nproc
6   CALL MPI_Init(errcode)
7   CALL MPI_Comm_size(MPI_COMM_WORLD, nproc, errcode)
8   CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, errcode)
9   PRINT '(A, I3, A, I3)', 'I am processor ', &
10      rank + 1, ' of ', nproc
11   CALL MPI_Finalize(errcode)
12 END PROGRAM main

```

where we initialize, get the communicator size (total number of processors we're running on), the rank of the current processor, we print these, and then we exit.

I've used quite a lot of colours in that listing to draw attention to the various bits. In particular, the orange and purple (e.g. `MPI_Comm_size` and `MPI_COMM_WORLD`) are MPI keywords - I have used orange for functions and purple for the constants. I have also coloured in light blue some parts which are NOT MPI provided, but are just our choices for good names for things like the rank.

So how does this actually work? Well, if we just run the code like normal (e.g. `./a.out`) then we only get one processor. All of the MPI code is still there, and you won't be able to run this way if your MPI library has gone missing between the compile and the run step, but nothing special happens (and in plenty of cases, things won't work - see a bit later where we discuss Deadlocks).

To run in parallel, we have to invoke the code on multiple processors, so we use another part of MPI called either `mpiexec` or `mpirun`. On a personal machine, both should work, on cluster machines consult the documentation to know what to do. Often

you will want to use a command provided by the cluster scheduling system, such as `srun`. You tell `mpirun` (or equivalent) how many processors to run on, and what command to run. So to run `a.out` on 4 cores we would do something like:

```
mpirun -n 4 ./a.out
mpiexec -n 4 ./a.out "inputfilename"
```

Notice that we do need the `./` still - we're not giving just the name of a program, we're giving the full command used to invoke it. So we can follow the command with any arguments we want to give to the program etc.

Now, what actually happens? n copies of the program are started, and each is placed on its own processor.¹ When `MPI_Init` is called, the MPI library sets up communications between these processes, including the default communicator, and then MPI functions can be called which use this infrastructure.

3.0.3 Classes of MPI Communication

MPI comms breaks roughly into 3 sets:

1. Collective communication - the processors all communicate in some way (e.g. to sum a quantity over all processors or the like)
2. Point to Point communication - THIS processor talks to THAT processor
3. Weird stuff - Everything else. Very valuable, but quite specific stuff

Most MPI codes spend most of their time in point to point communication. Luckily the rules for this aren't very complex - a Sender processor can send a message to some specific Recipient rank - a Recipient processor waits to receive a message either from a specific rank, or from any rank. Messages are all served in the order that they arrive.

The normal sending and receiving routines are *blocking* in that they don't return control to your program until the send or receive is complete. Note that the send being complete *does not mean* the receive has happened, just that the MPI communication layer has got the data to be sent, so your program can continue.

3.0.4 A Simple MPI Program - Ring Pass

The simplest actually useful MPI program is one where each processor involved sends a single data item to one other processor - and the easiest to see is when we place the processors in order of rank, and each sends to the next rank along. At the highest rank, we wrap back around, so this sends to processor 0.

A working code for the simple ring pass is:

¹Technically they need not be placed on their own processor, but this can cause issues - look into "oversubscription" for more. If you're interested in HOW they are forced onto separate processors, look up "process affinity"

```

1 PROGRAM main
2   USE MPI
3   IMPLICIT NONE
4   INTEGER :: rank, nproc, rank_right, rank_left, rank_recv, errcode
5   INTEGER, DIMENSION(MPI_STATUS_SIZE) :: stat
6   CALL MPIINIT(errcode)
7   CALL MPICOMMWRANK(MPICOMM_WORLD, rank, errcode)
8   CALL MPICOMMSIZE(MPICOMM_WORLD, nproc, errcode)
9   rank_left = rank - 1
10  !Ranks run from 0 to nproc-1, so wrap the ends around to make a loop
11  IF(rank_left == -1) rank_left = nproc-1
12  rank_right = rank + 1
13  IF(rank_right == nproc) rank_right = 0
14  IF (rank == 0) THEN
15    CALL MPISEND(rank, 1, MPI_INTEGER, rank_right, 100, MPICOMM_WORLD,
16      &
17      errcode)
18    CALL MPIRECV(rank_recv, 1, MPI_INTEGER, rank_left, 100,
19      MPICOMM_WORLD, &
20      stat, errcode)
21  ELSE
22    CALL MPIRECV(rank_recv, 1, MPI_INTEGER, rank_left, 100,
23      MPICOMM_WORLD, &
24      stat, errcode)
25    CALL MPISEND(rank, 1, MPI_INTEGER, rank_right, 100, MPICOMM_WORLD,
26      &
27      errcode)
28  END IF
29  PRINT ('("Rank ", I3, " has received value ", I3, " from rank ", I3)'), &
30    rank, rank_recv, rank_left
31  CALL MPIFINALIZE(errcode)
32 END PROGRAM main

```

There's a lot going on in this code, but in particular note how the Send and Recv bits work. In particular, note what you give to these functions:

- What variables to send and/or receive into
- How many elements (because it could be an array)
- The type of the variable (here `MPI_INTEGER`)
- The rank to send to/recv from
- An integer tag, which has to match in send and recv commands
- The communicator we are sending/recv-ing on

In the recv call, we have one more parameter, the *status*. This contains information about the message received (e.g. who sent it). Usually you don't need to know this, so you can use the special value `MPI_STATUS_IGNORE` so you don't need to capture it.

Also note that in Fortran all MPI functions take a final error code parameter, whereas the C ones return their error code.

Here we're only giving enough details to show the sort of things MPI can do. For more details, such as what types are available and what they are called, we suggest our Introductory MPI workshop (materials at <https://warwick.ac.uk/research/rtp/sc/rse/training/intrompi>) or one of the tutorials widely available online, or a good book.

3.0.5 Deadlocks

The code we just showed looks a bit over complicated - it's tempting to skip that whole if/else bit on lines 14 to 24 and just have every processor recv a message, and send it on. In general *this will not work*. All of the processors will enter the recv step, and wait there. Since no processor sent a message, no processor's recv can complete and the situation is a *deadlock*. More complex situations can occur, but basically whenever a processor is waiting for a message that can never come until said processor does something else, is a deadlock.

The deadlock is probably the error you will encounter most often in MPI code. There exist also livelocks, where processors are doing work but can never complete it (think trying to step aside to let somebody pass, when they also step aside and you are stuck moving left and right endlessly).

Unfortunately, common ways to get deadlocks can result in bugs that only occur in some circumstances. For instance, we mentioned that send is blocking, but that this only means it waits until MPI has control of the message. This means for a small message, which fits entirely into MPI's internal buffer, the send might return immediately. But if the message gets bigger, this can no longer occur.

Sometimes, different systems treat the sends differently - for instance your machine might be pro-active and complete the sends aggressively, but another system might not. This is NOT an error on the second system - it is a bug in your code. You cannot rely on this behaviour of sends.

So, how do you avoid a deadlock? There are many ways. For a start, MPI provides Send and Recv functions which aren't blocking, but these are tricky to use, so we will not discuss them here. There are many other ways, but for something like the ring-pass we favour using `MPI_SendRecv`, which combines the Send and Recv into a single command, where the MPI layer takes care of which order these happen in. The entire command blocks, but once it returns you know the Recv part is complete.

The code above using `SendRecv` becomes rather simpler:

```

1 PROGRAM main
2   USE MPI
3   IMPLICIT NONE
4   INTEGER :: rank, nproc, rank_right, rank_left, rank_recv, errcode
5   INTEGER, DIMENSION(MPI_STATUS_SIZE) :: stat
6   CALL MPLINIT(errcode)
7   CALL MPLCOMM_RANK(MPI_COMM_WORLD, rank, errcode)
8   CALL MPLCOMM_SIZE(MPI_COMM_WORLD, nproc, errcode)

```

```

9  rank_left = rank - 1
10 !Ranks run from 0 to nproc-1, so wrap the ends around to make a loop
11 IF(rank_left == -1) rank_left = nproc-1
12 rank_right = rank + 1
13 IF(rank_right == nproc) rank_right = 0
14 CALL MPI_Sendrecv(rank, 1, MPLINTEGER, rank_right, 100, &
15                  rank_recv, 1, MPLINTEGER, rank_left, 100, &
16                  MPLCOMM_WORLD, stat, errcode)
17 PRINT ('("Rank ", I3, " has received value ", I3, " from rank ", I3)'), &
18        rank, rank_recv, rank_left
19 CALL MPI_FINALIZE(errcode)
20 END PROGRAM main

```

We have split the SendRecv over several lines - note it basically has the send part of the command, followed by the recv.

This SendRecv example also works quite well for the smoothing example we discussed earlier. You send to your left, and you recv from your right. There is another handy MPI constant - `MPI_PROC_NULL` which can be used in a send or recv command to mean “there is no processor, do nothing”. This is very handy at the edges of your real domain, simply using this value in place of `rank_left` or `rank_right`.

3.0.6 MPI Collectives

Collective communications are not as much used in most MPI codes as point-to-point, but they are still very important. There are collectives to:

- Combine data from processors (`MPI_Reduce` and `MPI_Allreduce`)
- Send data from this processor to all other processors (`MPI_Bcast`, `MPI_Scatter`)
- Get data from all other processors to this processor (`MPI_Gather`)
- Send data from every processor to every other processor (`MPI_Alltoall`)
- Synchronise all of the processors without sending data (`MPI_Barrier`)

A simple example of collective comms using a Reduce operation is:

```

1 PROGRAM reduce
2   USE MPI
3   IMPLICIT NONE
4   INTEGER :: nproc, rank, rank_red, errcode
5   CALL MPI_Init(errcode)
6   !Get the total number of processors
7   CALL MPI_Comm_size(MPLCOMM_WORLD, nproc, errcode)
8   !Get the rank of your current processor
9   CALL MPI_Comm_rank(MPLCOMM_WORLD, rank, errcode)
10  !MPI.Reduce combines values from all processors. Here it finds the
    maximum
11  !value (MPLMAX) over all processors. It gets that value on one
    processor

```

```

12  !called the "root" processor, here rank 0. The related MPI_Allreduce
    gives the
13  !reduced value to all processors
14  CALL MPI_Reduce(rank, rank_red, 1, MPLINTEGER, MPLMAX, 0,
    MPICOMM_WORLD, &
15  errcode)
16  IF (rank == 0) PRINT *, 'Largest rank is ', rank_red
17  !MPI_Allreduce combines values from all processors. Here it finds the
    sum of
18  !the value (MPLSUM) over all processors. It gets that value on all
    processors
19  CALL MPI_Allreduce(rank, rank_red, 1, MPLINTEGER, MPLSUM,
    MPICOMM_WORLD, &
20  errcode)
21  IF (rank == nproc-1) PRINT *, 'Sum of ranks is ', rank_red
22  CALL MPI_Finalize(errcode)
23  END PROGRAM reduce

```

The code comments describe what is happening here.

All of the collectives differ, but have some commonalities. In particular, since they involve a lot of processors communicating they involve more comms and hence have more overhead than a single point-to-point call. They are also blocking - for those where non-blocking variants exist these are used less commonly.

3.1 What to do Now

If you're following for general interest, or future potential work, we would suggest examining the example codes (<https://github.com/WarwickRSE/ParallelismPrimer>) tweaking them, and making sure you see what they are doing. Then, try solving some problem of your own using parallelism. You don't need to worry about efficiency, just get things working. Bonus points if you break things and work out why, and how to fix them.

3.2 Other Resources

If you want or need to use parallelism in your work, we have some other courses that might be of interest. In particular, we cover all stages of MPI programming, from Introductory to Advanced, with slides and example code available in [Introduction to MPI](#), [Intermediate MPI](#) and [Advanced Topics in MPI](#). For OpenMP we have [Introduction to OpenMP](#). We also discuss a bit about parallelism in Python code in [Accelerating Python](#)

If you're looking to use cluster machines, we have some material that may be useful in [HPC at Warwick and Beyond](#).

Chapter 4

Glossary of Terms

Glossary

bandwidth The rate at which data is transferred, usually in mega or giga bits per second. Your internet contract “speed” is actually a bandwidth. *See also* [latency](#),

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. [12](#), [19](#)

core A single processing unit, which independently executes instructions (actions) but has access to other things like memory on the same physical processor. This is the smallest unit capable of independent, general purpose computation. , [18](#)

CPU Central Processing Unit; the heart of a computer, the physical chip containing the ability to do processing. Usually now contains multiple [cores](#).

CPU time The total amount of time, summed over all processors involved. This is what funding bodies and things usually want to know. Note that this can vary depending on how many processors you run on, especially for hard-to-parallelise problems. *See also* [wall time](#),

hyperthreading The ability of certain processors to separately use parts of their hardware, so that more than one task can run simultaneously. [5](#)

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

Mutex Mutual Exclusion Object; a thing which only one task/program/processor can hold at once, ensuring that a resource is only accessed by one task/program/processor at once. [2](#), [8](#)

POSIX A set of standards for low level operating system features etc, supported by a lot of platforms, e.g. Linux, OSX and similar. Note Windows is NOT one, except via the new WSL system in Windows 10. [1](#)

race condition A problem where two (or more) things can both attempt an action at once, and their actions will not “add together” to the correct result. For instance, if two tasks both increment a shared variable, you would hope to get an end result of +2, but race conditions can mean that both tasks read the initial value, and write back this plus 1, giving the wrong answer.

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. [12](#)

root One of the processors in a [communicator](#) which does any unique work. , [19](#)

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

socket The physical slot a CPU is plugged into, used as a term for the number of physical processors (each of which is probably multi-core) a machine has. A normal laptop or desktop is usually single-socket, a Workstation or cluster machine might be 2 or even 4.

wall time Time in the real world (i.e. as measured by a clock on the wall), as compared to the total amount of cpu time (summed over all cpus) and/or active time (for processes which wait for input etc). *See also* [CPU time](#),