Warwick Research Software Engineering

Parallelism Primer

A Gentle Overview of Methods and Strategies for Parallel Computing

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, 2020

Contents

Preface **i**
0.1 About these Notes i
0.2 Example Programs i

1 Introduction By Analogies **1**
1.1 Why Analogies 1
1.2 Restaurant Analogies 1
1.3 Computer Terms 7

2 Concepts in Parallel Programming **9**
2.1 Introduction 9
2.2 Task Based Parallelism 9
2.3 Map Reduce 12
2.4 Connected Task Parallelism 14
2.5 Tightly Coupled Parallelism 17
2.6 Wrapup 23

3 Technologies for Parallel Programming **24**
3.1 Pthreads 24
3.2 Other low-level threading libraries 25
3.3 Python and Multithreading 26
3.4 OpenMP - Open MultiProcessing 26
3.5 MPI - Message Passing Interface 32

4 Summary **39**
4.1 Wrap Up 39
4.2 What to do Now 40
4.3 Other Resources 40

5 Glossary of Terms **41**
Glossary 41

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 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 LaTeXby 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

Introduction By Analogies

1.1 Why Analogies

Parallel computing can feel quite daunting and seem to be very different to anything you encounter in daily life. Actually though, a lot of the general principles are things you encounter and deal with every day. Most of the problems and solutions to running things on “multiprocessor” computers are the same as trying to do several tasks at once in real life. So, if you’ve ever prepared a complicated meal, packed for a holiday, or arranged for home repairs or deliveries, you’ve probably got the basic skills down.

So to get started gently, we’re going to use some analogies that can be very helpful for relating the things you’ll see here to real life. A well chosen analogy shares key features with the real problem it’s being used to illustrate. But you have to be careful - analogies get leaky and breakdown and a feature you identify in the analogy might not be present or relevant to the real problem.

To try and make these as safe as possible, we’re going to be quite brief in our analogies, and if you find them helpful we recommend that you try and extend them yourself, although you’ll want to check that the extensions apply to the real problem too. And remember, the analogy might be useful, but often the creating of it is more of a benefit than making the final product neat and shiny.

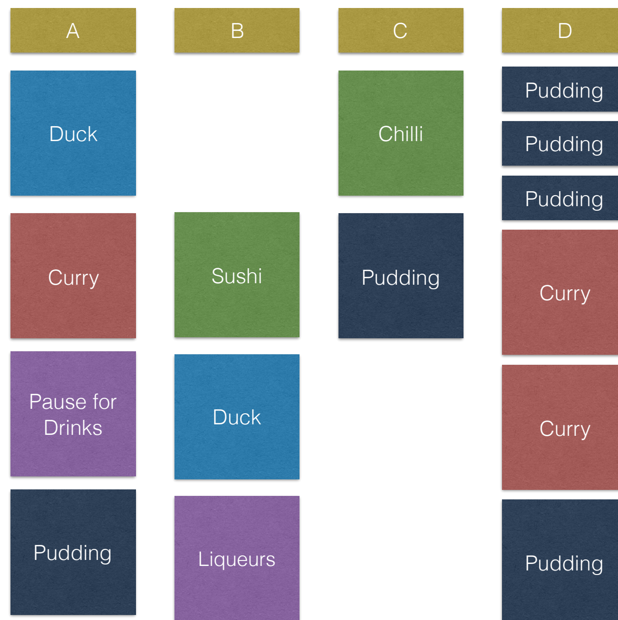
1.2 Restaurant Analogies

Several of the sorts of layouts and strategies in parallel programming can be compared to eating at restaurants.

1.2.1 The Buffet

Eating at a buffet is quite like the simplest model of doing multiple things at once. Everybody does what they want completely independently, and only interfere with each other if they try and use the same resource at the same time. One can eat dishes in any order, sample as many as one wishes, and eat as much as wanted. For example, 4

people (borrowing the classic Computer Science names), Alice, Bob, Charlie and Dave might eat like the following picture:



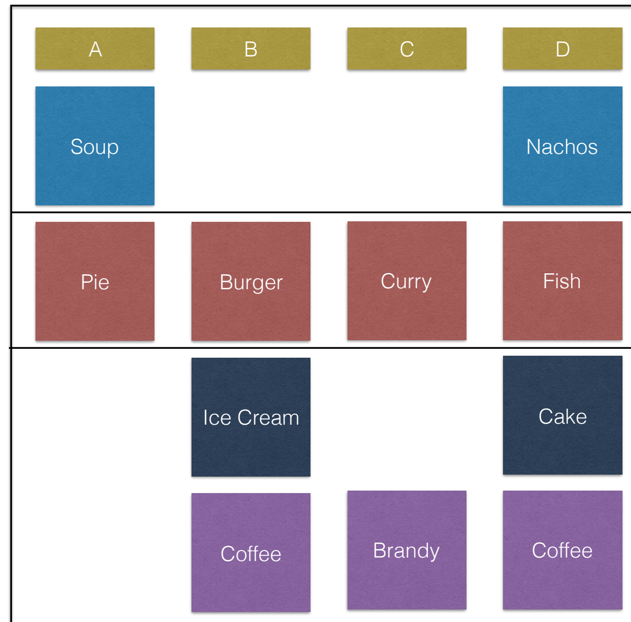
We can compare this to a computer in normal operations. Programs (tasks, dishes) are split onto different processors so that they are all used (everybody eats). So for example running two programs at once usually puts them on two different processors (note this is a gap in the analogy we will not try and patch over).

This is sometimes called "Embarrassing Parallelism" (because it is embarrassingly easy to program) where we simple run many programs, or copies of a program, all at once. They run independently, and while a processor can run one at a time (eat one dish at once) they can swap about and sequence them any way they wish.

If we briefly switch analogies and compare dishes to resources (computer processing power, memory, disk etc) then we see that we don't want too many programs running at once (people at the buffet) otherwise they will collide over dishes and have to wait their turns.

1.2.2 A La Carte Menus

The "A La Carte" Menu is rather more regimented than a buffet, but still leaves lots of choice. Not everybody has all parts of the meal (starters, puddings, drinks), and not everybody has the same things for any course, but there is general *synchronisation* - mains are served only after starters are done, pudding comes last etc. This might look something like the following picture:



In computing terms, this is a "weakly coupled" parallel program - a program where things mostly do their own tasks without reference to what other things are doing, but sometimes there is a need to synchronise so that somethings happen at the same time, or happen in some needed sequence.

1.2.3 The Prix Fixe

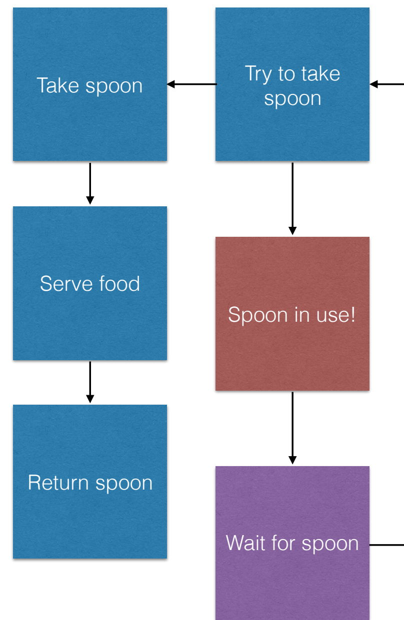
The expensive end of the restaurant analogy is the Prix Fixe or "set menu" where a restaurant offers a pre-selected set of dishes at a fixed price. Everybody gets their dishes at the same time (at least at a single table) and there may be small options (peas or salad, cheese course or sweet) but generally everything is very fixed. For instance:

In computing terms, this is "tightly coupled" parallel programming, with many sequencing points where nobody can advance until every task has reached the same point. Things may do slightly different tasks between these points, but you can't move on until everything is finished.

1.2.4 Another Angle - The Buffet Spoon

We mentioned along with the buffet that you can think of this where people are processors and dishes are programs, or you can usefully think of it where people are programs, and dishes are computer resources (see later for discussion of what sorts of things count). If we think of the second one, we have a very handy analogy in the buffet dish with a single serving spoon.

If you want some of the dish, you have to acquire the spoon, serve yourself some food, and then return the spoon. As long as you have the spoon, nobody else can access the dish, and they must wait for you to finish. The following picture shows this:



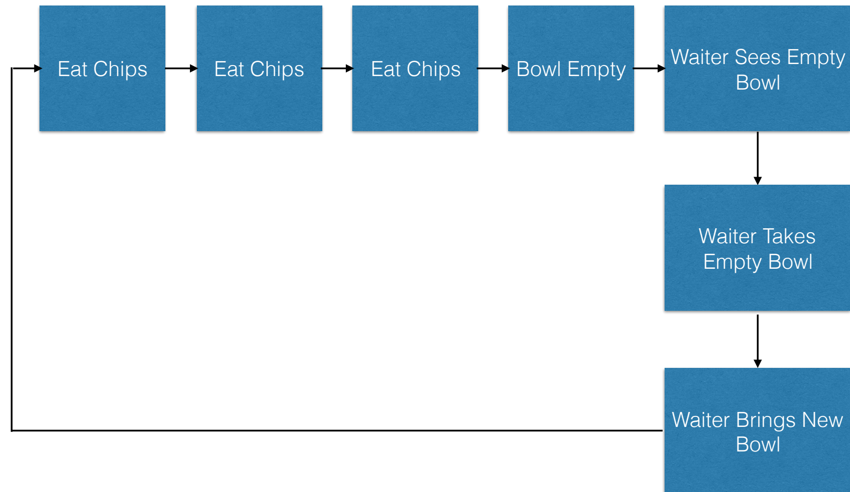
Note one very important thing which holds in the analogy and reality - if you forget to return the spoon, the dish is effectively "locked out" for anybody else, AND the wait for spoon loop need not ever end. If you are unlucky, you can wait a very long time while other people pass the spoon around and never get your turn.

In computing, this analogy applies to a whole lot of things, basically any resource which can only be used by one thing at a time, such as disk (you can only read from/write to a single location at one time), parts of the operating system, etc.

The buffet spoon is analogous to a Mutual Exclusion (Mutex) object (Mutex) - a thing that only one program/processor/task can hold at a given time. We'll talk about these more later. In real systems, a lot of work goes into making sure everybody gets their turn with the spoon, and this is rarely as simple as a "first come first served" queue.

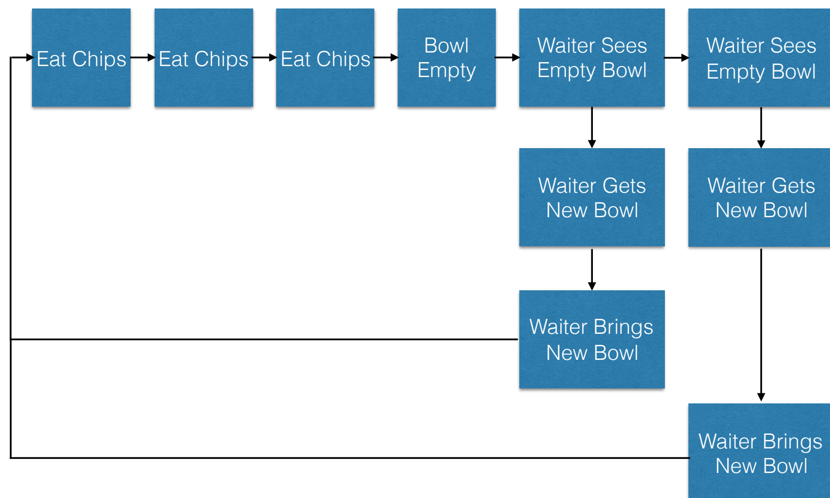
1.2.5 Another Angle - Bowls O' Chips

Another angle on the "getting the order right" question is the thing at some restaurants where you might get an endless bowl of table snacks to eat before the meal (bread rolls, chips (American parlance), poppadums etc). The *intended* scenario is like this:



You eat the snacks, and when it is empty, a waiter/server takes the empty bowl away and brings another.

Now suppose the waiter didn't take the empty bowl away, but simply brought a new bowl. The "signal" of the empty bowl stays, so another waiter/server might also see it and respond. Soon, you are piled high with chips (although the analogy leaks again - what's so bad about having too many chips?).



In computer terms, this best represents what we call a race condition. The empty

chip bowl represents a signal which the full bowl *or absent bowl* does not. If two processes (waiters) both catch this signal, they both act to remedy it, and the result is an excess of chips (in computers, more likely a crash, wrong result etc). The key point is that there is a delay between noticing the signal, and resetting the signal, during which the signal can be seen multiple times. When the waiter instead takes away the chip bowl, the signal is instantly reset, even before the action is complete.

You might notice that there are 3 “states” to this problem. There is the full bowl, the empty bowl, and the absent bowl. This is a feature of many real problems too - there is a difference between *I am handling this, but am not yet done* and *I have handled this and am complete*. This is an example of what is called a sentinel value - a value which represents some sort of signal. The chip bowl can contain any amount of chips, but an empty bowl, or absence of the bowl, means something special.

1.3 Computer Terms

A lot of this section might already be familiar, but it is important to understand some terms, and make sure we mean the same thing that you do when we use computing terms.

Firstly, the core - an individual bit of a computer chip, the smallest bit which can independently do general purpose computation (there are bits within the core which can do specific tasks).

The actual Central Processing Unit (CPU) or chip, the of a computer, the physical chip. Usually now contains multiple cores. Some people call the box part of a computer (the tower etc, where the On button is) the CPU and they’re pretty much wrong.

The socket - technically the thing the CPU plugs in to, usually used to refer to the number of physical CPUs in a computer, e.g. a two socket workstation. Each socket usually has its own memory

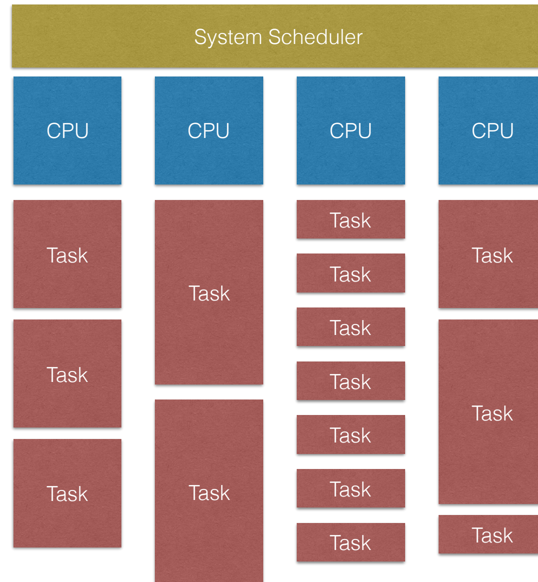
A node - a single computer, usually used in the context of a cluster (many nodes that can communicate)

The RAM (Random Access Memory) is the part that stores the data your program is generally working on. Bits of this might be moved into small caches which are part of the CPU temporarily. Be careful to distinguish this memory from the hard drive (disk) - the long term storage that keeps data even when the computer is powered down.

1.3.1 The CPU

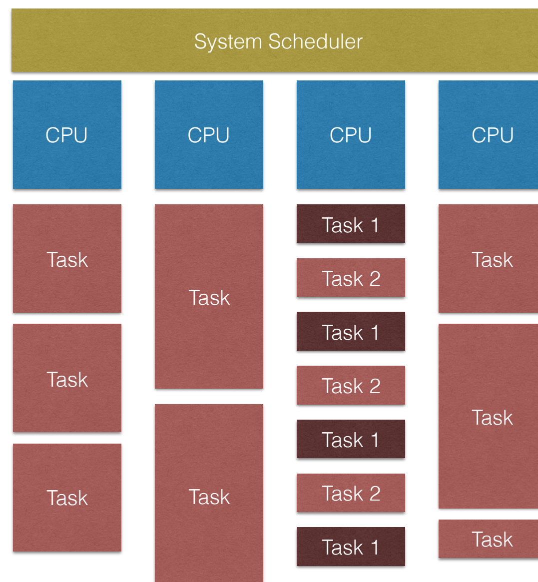
Most of this Workshop is going to focus on the CPU, because that is what actually does the parallel processing - we’re trying to move programs from using one single core of a CPU across to many, perhaps very many.

The normal way the CPU in your computer works is quite like the buffet analogy - lots of programs doing lots of tasks. If we draw the same sort of block diagram, we get something like (a very simplified schedule):



We mentioned in the buffet examples that people have to queue and wait if they want to get the same food, and can only sample one dish at a time - taken literally this is an analogy failure because computers don't work like this. If they did, you could only run one program per core you have, otherwise things would have to wait in line.

Real CPUs make a lot of use of time slicing, where one program is suspended to allow another to run. This is especially effective for interactive programs, where there might be a lot of time spent waiting for input that could usefully be spent by another program. This looks a bit like:



Chapter 2

Concepts in Parallel Programming

2.1 Introduction

Before discussing the dedicated libraries for parallel programming, we're going to talk over the general concepts, and also introduce one program, because it happens to fit here.

2.2 Task Based Parallelism

The first, and simplest, way to get more stuff run faster is to just run a bunch of things at the same time on a big computer. This is really easy if your problem is to explore some parameter space (run a bunch of copies on different input data), and/or run the same problem a bunch of times (for instance doing multiple Monte-Carlo realisations or changing a randomness generator to investigate uncertainties).

You can run these multiple copies on different processors (cores) of a single computer, or across multiple separate computers. You might need something simple to make sure each task selects a unique parameter set or random realisation, but this can be done "before the fact", often by picking the next thing from an input file, knowing one's order in a set (e.g. task number N takes parameter set N), or in the case of randomness, by properly generating a random number.

This sort of parallelism scales exceptionally well, usually limited only by the number of tasks you have to run and the number of processors you have to run on. If you have one processor per task, you can get them all done in the same time (on the clock, or the wall time) it would have taken to do one task.

The main wrinkle in this sort of parallelism is for very small, short tasks, where the time to start them can be a substantial amount of the total time, and you will want to "bundle" mutliple sets into a single invocation. A similar wrinkle occurs if you have a truly vast amount of jobs (e.g. 10s of thousands in a few days) where some systems might store so much information on what they have run that this overloads them. If in doubt, consult the person in charge of the computing resource.

2.2.1 Basic GNU Parallel Scripts

The GNU Parallel program is installed on all Warwick clusters and the COW (including dedicated nodes) and can be used by loading the "parallel" module. It's also available on most other *nix platforms, via apt, yum or brew. The official tutorial is at https://www.gnu.org/software/parallel/parallel_tutorial.html and is very good!

The idea of this is that you take a program which takes parameters from the command line (e.g. to run it you type ' ./program_name [parameters]') to tell it what to run, and then you tell *parallel* how to build the command line you need to run several copies for whichever parameters you want.

parallel can be given a number of job slots, which are used simultaneously, so usually this is the number of available processors (perhaps reserving some for other tasks), and will run the set of tasks it was given in sequence, distributing them across job slots until it runs out of tasks. A new task is started whenever a slot becomes free (by default).

You invoke *parallel* via a script (or for super simple examples, by typing at the command line) and the simplest script is

```
#!/bin/bash
parallel echo ::: A B C D E F
```

which produces

```
A
B
C
D
E
F
```

Taking the line 'parallel echo ::: A B C D E F' and breaking it down, bit by bit: firstly 'parallel' runs the GNU parallel program. 'echo' is a command line utility (program) which just prints whatever arguments you give it (followed by a new line). ':::' tells *parallel* where to separate your command from the parameters to parallel itself, and then the last bit is the parameters to *parallel*, separated by spaces, although this can be changed.

A slightly more complex script is:

```
#!/bin/bash
parallel echo ::: A B C ::: D E F ::: G H I
```

which produces

```
ADG
ADH
ADI
AEG
AEH
...
CFI
```

i.e. *parallel* combines each possible triplet of arguments from the 3 sets we gave it (the Cartesian product of the sets).

If you provide a single argument, like in the first script, you can specify where this gets inserted into your command line with {}:

```
#!/bin/bash
parallel echo HELLO {} ::: A B C
```

which produces

```
HELLO A
HELLO B
HELLO C
```

and if you specify more than one, then you pick them by number inside curly braces:

```
#!/bin/bash
parallel echo {1} Says hello to {2} ::: A B C ::: D E F
```

which produces

```
A Says hello to D
A Says hello to E
A Says hello to F
B Says hello to D
...
C Says hello to E
C Says hello to F
```

parallel gives each job a sequential number, and you can access this using {#}, such as:

```
#!/bin/bash
parallel echo {1} is job number {#} ::: A B C
```

which produces

```
A is job number 1
B is job number 2
C is job number 3
```

If you need each job to have a unique ID, it's a bit fiddly, because you need to invoke a program to generate it in the same job as running the actual program, and it can be tricky to get everything to run at the right point. Something like the following works:

```
#!/bin/bash
runfunc(){
  echo $1 has uuid `uuidgen`
}
export -f runfunc
parallel runfunc ::: A B C
```

which produces

```
A has uuid 536b2247-64fb-49cd-af9b-258ca52c6200
B has uuid fef2fce0-26fb-4553-9c5e-7a6c8b07cd58
C has uuid bc58cea0-8ffb-4733-9650-441f3587023d
```

(Note you will obviously get different actual IDs!)

What can GNU parallel run?

Parallel can run:

- Any shell command, such as ls, cat, echo
- Any other program that can run from the command line
- Bash functions, if they are exported using export -f as in the above example

GNU Parallel on SCRTP cluster machines

Details on running GNU Parallel on the cluster machines is available on the SCRTP wiki, at https://wiki.csc.warwick.ac.uk/twiki/bin/view/HPC/ClusterUserGuide#Serial_jobs (SCRTP login required).

Command Line Parameters and Beyond

You can make almost any modern language take command line parameters, although they vary a bit in how. If the parameters to pass get very complex, consider passing the name of an input file, and having one of these for each job, which provides the full setup.

2.3 Map Reduce

Map Reduce is a method for processing large amounts of data in parallel, consisting of two core operations (or more in real systems), one of which runs independently on each processor, and the other runs between processors.

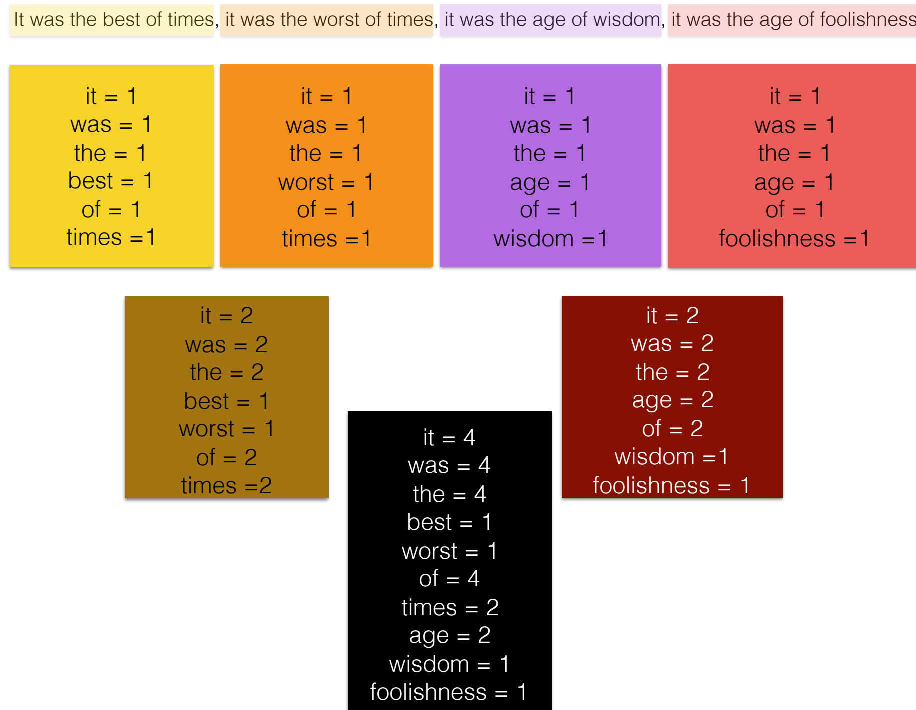
The *Map* operation happens independently on each processor, and converts some sort of raw data to the data to be processed further.

The *Reduce* operation then takes two of items out of the Map operation and combines them into a new data item somehow containing the data of both. Strictly this needs to be a linear, commutative and associative operation, so that operating on two items at a time lets you combine them all. Two of the items produced by the reduce operation can also be combined. This lets the operation be done like a tree - on each processor the items are reduced to one, and then this is combined with other processors. It's complex to optimise this on real hardware, but that's a job for the library writers.

If we take a simple example, of counting words in a text, for instance:

```
It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness
```

We can split this into the 4 clauses (comma separated bits, which I conveniently already did above so it fits neatly on the page), and count the words on each. This is the second row in the figure below. Then, we pair-wise combine the blocks in the second row to get the third row, and pair-wise combine these two to get the fourth row, which shows the final counts.



2.3.1 Restrictions in MapReduce

There are clearly substantial restrictions on what can be done with Map Reduce, for all it is very powerful. Most importantly, you can only do things where you can combine operations on already partially reduced data. A lot of the classic operations you might think of, adding, multiplying, making maximums and such are directly possible.

Averaging however, is not, at least not directly. If you have a set of numbers, you can't simply split them into two groups, average each, and average the result. For instance, averaging the numbers from 1 to 9 should give a result of 5. But if we average 1 to 4 and 5 to 9 separately, we get $(10/4 + 35/5) = 4.75$, which is a bit wrong, and the answer gets worse if we divide other ways (except the one way where each subgroup also has an average of 5). But all is NOT lost, and we can still use MapReduce to get averages, we just have to do two steps. We can total up all the numbers in one step,

and count the elements in another, and then at the end calculate the average. Other things like the Standard Deviation need more tricks (One Pass SD formula), but can often be done.

2.3.2 MapReduce packages

There's a lot of packages that implement the MapReduce model, including Apache Hadoop and the MongoDB database engine, and several front ends to these in R, Python etc (although note you may need to use other languages inside the engine, such as JavaScript in MongoDB).

2.4 Connected Task Parallelism

GNU parallel is great when you have set of independent tasks, but what if your tasks aren't completely independent and what you want to run next might depend on the results of an earlier run? For relatively loosely connected tasks you can use the Worker-Controller model. This used to be called Master-Slave most commonly, but some people now prefer to avoid that term, so others are used. We pick Worker-Controller.

2.4.1 Example - Linguistics

As an example of this sort of task, imagine an over-simplified problem in text analysis. Wikipedia lists almost 100 people who might supposedly have written the plays attributed to William Shakespeare. We make no comment on whether these are serious prospects, and most likely the plays were written by who everybody thinks.

But imagine you wanted to write a program to test if a given author might be Shakespeare. More seriously, suppose you are trying to work out if a piece of text is written by a give author. There might be some easy tests you can do first to rule things out - for instance was the text written while the candidate was actually alive? Could they have written in English? These aren't terribly selective tests, but they eliminate potentials at very low cost.

Doing the hard tests might require a lot of work - e.g. machine learning analysis of precise sentence structure might take many days to conclude. Between these two extremes are are range of difficulties of test, and it would be nice not to have to do the really hard ones if the easier ones can rule out a given candidate.

2.4.2 Example - DNA

As another example, suppose you have a DNA sample which is somehow degraded, whether by UV exposure, chemical exposure or other wise. All you really have are incomplete chunks. You also have some candidate samples and you want to identify which of them might be a source of the chunks.

Reconstituting DNA sequences from chunks is (comparatively) expensive, so you would like to reject candidates bases on the smallest sequences which let you rule them out (see shotgun sequencing, https://en.wikipedia.org/wiki/Shotgun_sequencing.

Since the reconstitutions and the matches are not perfect, you have to keep going until you have a certain level of certainty of match and not match, and you also want to reject a sequence reconstruction if no candidate could give it (assuming your candidates are complete), or do this with some probability anyway.

So what you have is a series of tests (or simulations) of increasing difficulty, some of which need not be run at all if others have produced certain results.

2.4.3 Trees and Pruning

Schematically, what you want to program to solve problems like these examples is some sort of tree shaped set of simulations, like this picture:

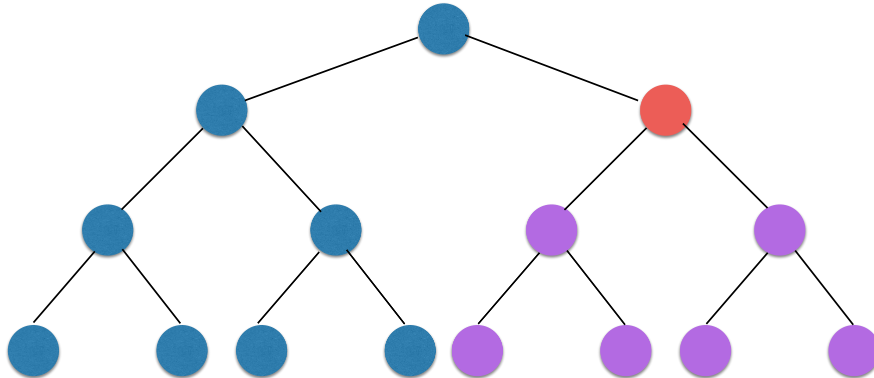


Figure 2.1: A tree of simulations. Each layer (depth) represent simulations that in some way depend on the layer above. Imagine the node highlighted in red tells us this is an unpromising avenue - then we can skip all 6 or the purple nodes.

Note that the highlighted node can answer the question you're working on in such a way that you know all of the subsequent simulations needn't be run. Usually, this answer turns out to be either "no, this branch cannot be the one sought" (Dan Brown is certainly not Shakespeare) or "based on this result, this is not an interesting avenue" (this low-resolution simulation shows that nothing much happens, so I needn't do the harder one). For the node we picked here, we don't have to do almost an entire half of our tree.

2.4.4 Worker Controller in Code

For models like this, working out which job to do next and getting it set up is quite a lot of work, and so we might dedicate an entire processor as the Controller, and this one tells all the others what to run next - it dispatches the work packages to the workers. Unlike with GNU parallel, the controller might be making complex decisions on what package to hand out next, and if a result indicates some packages aren't needed, it can choose not to hand those out at all.

Unfortunately, there isn't a standard package to help you set this up. However, all you need is some way for the Controller to communicate with the Workers. This can be via a network, using OS specific parallel processing features, or even just via files and the filesystem.

Files are actually an easy solution, but the performance is generally poor (and gets comparatively worse the quicker the workers tasks can run). The other two options are best used via some other library or framework, so it's not up to you to deal with details of complicated protocols, and not your problem to stay up to date with small changes in hardware etc.

The Worker-Controller paradigm is simple enough to be implemented in basically any parallel system, and we'll describe some of these options later, and give examples in our Example code. Some systems have built-in support of this kind of model, and some are intended specifically for it.

For the classic languages, C, C++ or Fortran, the options are:

- MPI
- OpenMP
- Threading
- Low level networking (very unusual in academia)
- CUDA

For Python, the options include

- Python Multiprocessing (a Package)
- SCOOP (rather old, last release circa 2015)
- Ray (https://ray.readthedocs.io/en/latest/)
- MPI4Py
- Many others (Faust, gevent, Thespian ...)

For other languages, e.g. Java, there are varying quality implementations of some of the C list, and some languages offer there own paralellism ideas.

2.4.5 Aside - Python programmers beware

One note - in Python one should be cautious using parallelism explicitly in your own code. Libraries like numpy, tensorflow, BLAS etc might invoke multithreaded library code, but Python has issues with parallelism (see also section 3.3) and often the speed increase you are looking for from paralellism is better achieved by outsourcing the hard bits, or the entire code, to C/Fortran etc. For more on this, see our Accelerating Python course `https://warwick.ac.uk/research/rtp/sc/rse/training/acceleratingpython`.

2.5 Tightly Coupled Parallelism

Unlike the previous examples, some problems are far more tightly coupled, and there is no way to split them up into separate tasks with only occasional communications, or manage jobs purely via a scheduling solution (like *parallel*). Usually, this is a single problem that is split up, but that distinction can get very fuzzy, so don't worry about it, just consider whether there is a lot of data to share, or a lot of times when things need to synchronise.

The classic example is when you have a physical grid (array) of data and you split it into chunks and solve one on each processor. You then have to glue the chunks back together at their edges, and this requires communication between processors, and it is this gluing, or coupling, that makes the problem different to what we discussed before.

An example of the weakly, or un-coupled parallelism would be trying to find all prime numbers between 1 and 1,000,000. This is a single problem, but is not (intrinsically) tightly coupled, in fact each number can be treated separately, so we can split the set of numbers (the domain) into chunks and give one to each processor. Do note though that plenty of algorithms to solve this are NOT weakly coupled - for instance the classic "Sieve of Eratosthenes" method requires every processor access the same data array.

So, **Take Care - tight and weak coupling of parallelism can be a feature of the problem itself, but it can also be a feature of the method chosen to solve it.**

2.5.1 Tight Coupling Example - Image Smoothing

A simple example of a tight-coupled problem is the iterative smoothing of a (2-D) image. A simple smoothing algorithm is to simply average each pixel with its neighbours, and repeat this until the image is as smooth as desired. The basic version is to use the 4 pixels above, below, left and right.

In code[1] one "pass" of this smoothing is

```
1 DO iy = 1, ny
2   DO ix = 1, nx
3     temp(ix,iy) = 0.25 * (im(ix-1, iy) + im(ix+1, iy) + im(ix,iy-1) + im(
        ix,iy+1))
```

[1] We use Fortran for most examples, since we find it is most understandable by general programmers

```
    END DO
END DO
! Comment: Once above is done, we can copy temp back into im
im = temp
```

Note that we have to use a temporary array, so that we always use the original value in the smoothing. We'll talk about the edges in a moment.

On a single processor, with its own chunk of our image, the figure below shows which pixels contribute to a pixel we're considering. It also shows the issue when we come to the edge - we are missing 1 or 2 of the pixels we need. We need more of the image to be accessible to the current processor, shown in purple round the edges.

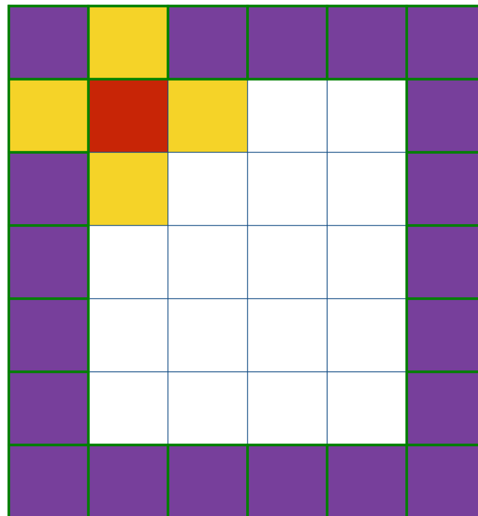
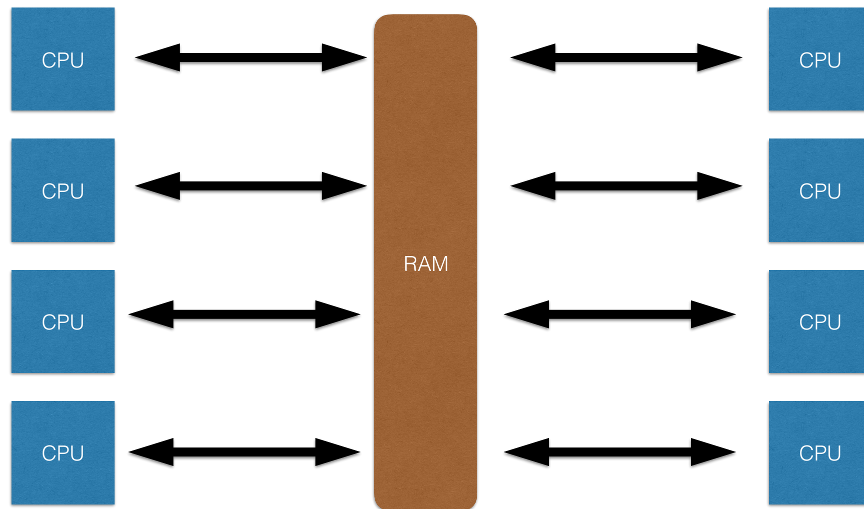


Figure 2.2: An image, with some "fake" cells in purple round the edges. To compute the smoothed value for the pixel shown in red, we use the pixels shown in yellow. At the edges, we need the "fake" data to make our smoothing work smoothly.

How we handle this purple boundary depends on which parallelism model we're using. If we're in a "shared memory" system, a processor can simply reach out and take the data it needs, when it needs it. In "distributed memory" models, the processor that has the data needs to make it available to you; the simplest way it can do this is to hand it over.



This figure shows the layout of a Shared Memory system, where all cores, 8 in this case, access the same pool of memory. This is like the setup on a single, normal, multi-core computer, like your laptop or desktop. You don't have to do anything special to make sure everything can access the memory, but you do need to take some care to make sure processors don't modify memory others are relying on, or rather, make sure they only modify when it is suitable.

In the image smoothing case, you have to make sure that the copy of "temp" into "im" doesn't happen until all processors are done using the values from "im", i.e. we have to synchronise where the comment line above is.

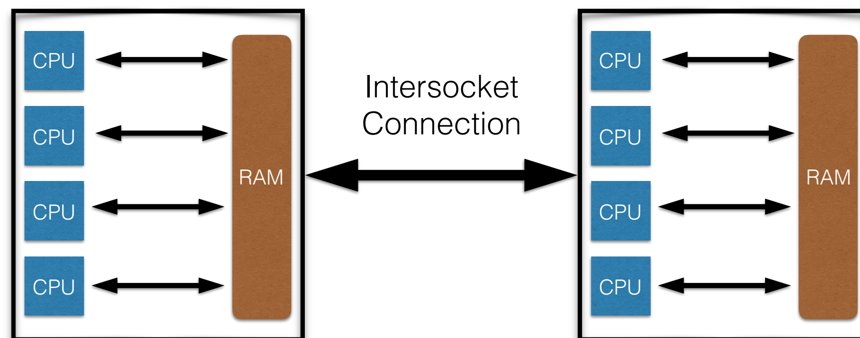
2.5.2 Shared Memory Technologies

In academic code, there are two common approaches for Shared Memory programming - Threading and OpenMP. Threading, essentially, means running some function from a program on a separate processor (strictly in its own thread, which can then be run on a separate processor if available). There are several ways to write this, but on *nix systems the core library is called *pthreads*. OpenMP is a library available for C/C++/Fortran programs which is very versatile, and is most easily and commonly used to split up loops to run on separate processors.

There are plenty of other libraries that can also be used, including Intel Threaded Building Blocks (in C++), and OpenAcc which works on CPU and GPU (graphics) cards. There are also some languages either aimed at, or including features for, parallel programming. Beware with these though, since there are many discontinued attempts and failures.

2.5.3 Aside - NUMA

For Workstations (big desktops) and cluster machines, there is one additional consideration, if they are multiple socket machines. The figure below shows schematically how the processors and memory are laid out - note that each socket has its own memory bank, and access goes via this intersocket connection in the middle. Since the electrical signals can travel at most at light-speed, and typically around 50% of this, we can see that for a 4GHz processor, a single clock cycle lasts about 2.5×10^{-10} s and so a signal can travel at most $2.5 \times 10^{-10} * c = 7.5$ cm and probably more like half this. So, given the physical size of a motherboard, **data on another physical processor MUST be slower to access**.



Luckily, you as a programmer don't have to worry about which socket your program, or some data, are running on, since everything is sorted out for you behind the scenes. This setup, and the library which lets the operating system (OS) deal with it is called NUMA - Non Uniform Memory Access. Basically, this is the idea of having memory available which has different access speeds, in this case due to physical location.

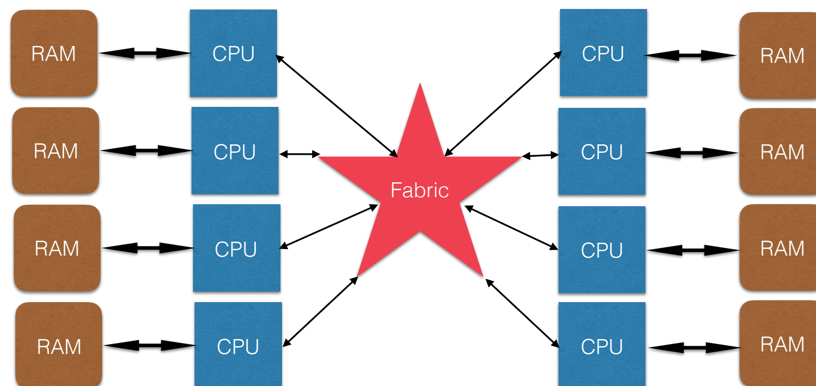
For instance, when a variable is first set, the memory holding it will be put in the directly accessible memory of the processor that set it, to try and make its access as fast as possible. NUMA is pretty much just a "last 5%" performance improvement, so is worth knowing about, but not worrying about when programming.

Some other systems try to make memory access smooth at greater distances and in these cases, the performance hit can be considerable and does require that you think about it when programming. Co-Array Fortran is one example of this - where arrays have an extra dimension which corresponds to the processor they are on. This makes it easy to access memory on any processor, BUT unless you are careful, can produce

unduly slow code. These packages can patch over the syntax to make it easier, but ultimately they can't break the laws of physics!

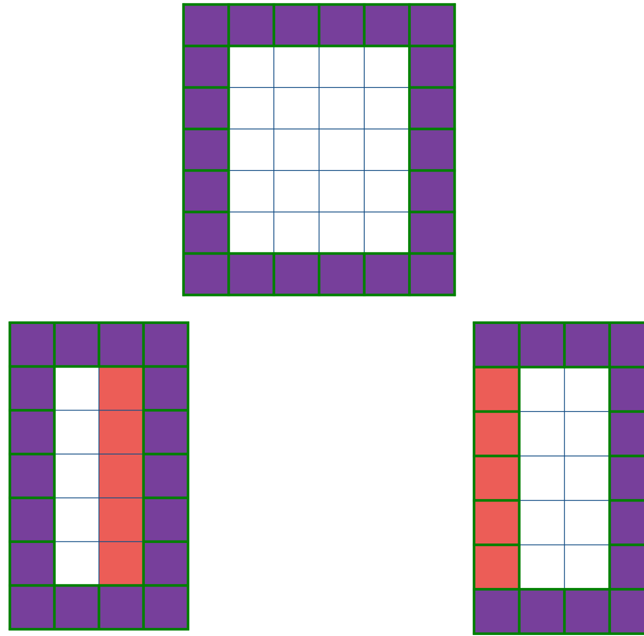
2.5.4 Distributed Memory

Distributed memory systems, such as clusters of computers, are a bit different to shared memory systems. A schematic view is:



You have a set of processors, each with their own memory, and the connection between them is via some "fabric" (mostly a fancy name for a network). It is now up to you (the programmer) to control when and what data is sent between processors. While this is an extra thing to think about, it mostly replaces concerns of synchronisation, because you know when a processor is in the right position to send data, and when to receive it.

Thinking about our image smoothing example again, we split the image up into pieces. The following figure shows how we might split things on 2 processors:



We have to add extra purple cells round the edges, so our smoothing algorithm has access to all the data it needs, and then we have to deal with keeping the data in these cells correctly up to date. This means the left most "purple" (actually highlighted in the red) cells of the second chunk are a copy of the right-most image cells of the first chunk (also in red). Similarly, if we split further, or split horizontally, other purple cells would be copies.

Then, after running the smoothing procedure, we share the purple cell temp data, so:

```
DO iy = 1, ny
  DO ix = 1, nx
    temp(ix,iy) = 0.25 * (im(ix-1, iy) + im(ix+1, iy) &
        & + im(ix,iy-1) + im(ix,iy+1))
  END DO
END DO
!Send temp(1,1:ny) to left processor
!Send temp(nx,1:ny) to right processor
!Send temp(1:nx,1) to bottom processor
!Send temp(1:nx,ny) to top processor
!Receive data from left processor into temp(0,1:ny)
!Receive data from right processor into temp(nx+1,1:ny)
!Receive data from bottom processor into temp(1:nx,0)
!Receive data from top processor into temp(1:nx,ny+1)
im = temp
```

How exactly we do those sending and receiving operations depends what technology we're using, but the general principle is the same. Do note that we're running the array from 0 to n+1, so that the image is in 1 to n and the purple cells are strips 0 and n+1.

2.6 Wrapup

This chapter has introduced the three basic models of parallelism:

- Embarrassing parallelism, and "task farming"

  GNU Parallel is a quick and easy solution to programming this

- Worker-controller type parallelism

  This overlaps with task-farming where tasks are contingent on each other

- Domain decomposition

  Splitting the problem into chunks, whether physical or otherwise. We can split based on any axis of our data, but ideally one where things only depend on "local" data, to minimise communication

The next chapter will talk more about technologies to use for the latter two models.

Chapter 3

Technologies for Parallel Programming

3.1 Pthreads

Pthreads stands for *POSIX Threads*, where 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 operatings 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 is 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.

3.2 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 Mutexs 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.

3.3 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`

3.4 OpenMP - Open MultiProcessing

OpenMP is a library for shared memory programming (see Sec 2.5.2) 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 the gives your code access at runtime to information about number of processors available, how to split work up etc

  These parts wont compile with an 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 paralellise a loop is:

```
PROGRAM loop_decompose
  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: nproc, i, thread_id
  INTEGER, DIMENSION(:), ALLOCATABLE :: its_per_proc
  nproc = omp_get_max_threads()
  ALLOCATE(its_per_proc(nproc))
  its_per_proc = 0
!$OMP PARALLEL DO
  DO i = 1, max_its
    thread_id = omp_get_thread_num() + 1
    its_per_proc(thread_id) = its_per_proc(thread_id) + 1
  END DO
!$OMP END PARALLEL DO
  DO i = 1, nproc
    PRINT '(A, I0, A, I0, A)', 'Processor ', i, ' performed ', &
        its_per_proc(i), ' iterations'
  END DO
  PRINT '(A, I0)','Total work on all processors is ', SUM(its_per_proc)
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, th above example in Fortran uses the Fortran style *DO/END DO* for a loop, wheras 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 hyperthreading ability, this will be twice the number of actual cores, otherwise it will match the number of physical cores.

3.4.1 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 !$OMP END DO
13  PRINT '(A, I0, A, I0, A)', 'Processor ', omp_get_thread_num(), '
       performed ', &
14          its, ' iterations'
15 !$OMP END PARALLEL
16 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.

3.4.2 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

3.4.3 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

```
PROGRAM loop_decompose
  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: i, its_global
its_global = 0
!$OMP PARALLEL DO
  DO i = 1, max_its
     its_global = its_global + 1
END DO
!$OMP END PARALLEL DO
PRINT '(A,I0,A)','Counter records ', its_global, ' iterations'
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:

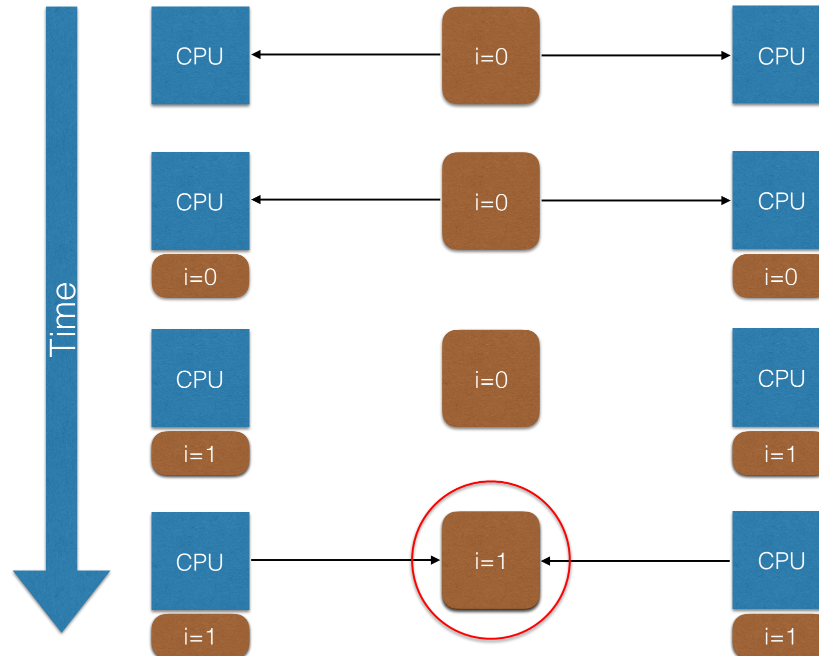


Figure 3.1: A race condition. On left and right are two CPUs, in the middle is the variable we're trying to increment. Each processor must read, increment, and write back, and instead of two incrementings, we only get one! Effectively, the processors are in a race with each other to finish their task before another tries to start.

3.4.4 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 OpenMPs 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

```
PROGRAM loop_decompose
  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: i, its_global
   its_global = 0
!$OMP PARALLEL DO
  DO i = 1, max_its
!$OMP ATOMIC
     its_global = its_global + 1
!$OMP END ATOMIC
END DO
!$OMP END PARALLEL DO
PRINT '(A,I0,A)','Counter records ', its_global, ' iterations'
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

```
PROGRAM loop_decompose
  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: i, its_global
 its_global = 0
!$OMP PARALLEL DO REDUCTION(+:its_global)
  DO i = 1, max_its
     its_global = its_global + 1
END DO
!$OMP END PARALLEL DO
PRINT '(A,I0,A)','Counter records ', its_global, ' iterations'
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

```
PROGRAM loop_decompose
  USE omp_lib
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_its = 10000
  INTEGER :: i, its_global
   its_global = 0
!$OMP PARALLEL DO
    DO i = 1, max_its
```

```
!$OMP CRITICAL
      its_global = its_global + 1
!$OMP END CRITICAL
    END DO
!$OMP END PARALLEL DO
  PRINT '(A,I0,A)','Counter records ', its_global, ' iterations'
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.

3.4.5 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

3.5 MPI - Message Passing Interface

MPI is a library used for Distributed Memory parallelism (recall 2.5.4). 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.

3.5.1 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.

3.5.2 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 wont 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 wont 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.

3.5.3 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

[1] Technically 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"

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.

3.5.4 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:

```
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 MPI_INIT(errcode)
7    CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, errcode)
8    CALL MPI_COMM_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   IF (rank == 0) THEN
15     CALL MPI_Send(rank, 1, MPI_INTEGER, rank_right, 100, MPI_COMM_WORLD,
          &
16        errcode)
17     CALL MPI_Recv(rank_recv, 1, MPI_INTEGER, rank_left, 100,
          MPI_COMM_WORLD, &
18        stat, errcode)
19 ELSE
20     CALL MPI_Recv(rank_recv, 1, MPI_INTEGER, rank_left, 100,
          MPI_COMM_WORLD, &
21        stat, errcode)
22     CALL MPI_Send(rank, 1, MPI_INTEGER, rank_right, 100, MPI_COMM_WORLD,
          &
23 errcode)
24   END IF
25   PRINT ('("Rank ",I3," has received value ", I3, " from rank ", I3)'), &
26       rank, rank_recv, rank_left
27   CALL MPI_FINALIZE(errcode)
28 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.

3.5.5 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 may 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 MPI_INIT(errcode)
7    CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, errcode)
8    CALL MPI_COMM_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, MPI_INTEGER, rank_right, 100, &
15                     rank_recv,1, MPI_INTEGER, rank_left , 100, &
16                    MPI_COMM_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`.

3.5.6 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:

```
PROGRAM reduce
  USE MPI
  IMPLICIT NONE
  INTEGER :: nproc, rank, rank_red, errcode
  CALL MPI_Init(errcode)
  !Get the total number of processors
  CALL MPI_Comm_size(MPI_COMM_WORLD, nproc, errcode)
  !Get the rank of your current processor
  CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, errcode)
  !MPI_Reduce combines values from all processors. Here it finds the
      maximum
  !value (MPI_MAX) over all processors. It gets that value on one
      processor
  !called the "root" processor, here rank 0. The related MPI_Allreduce
      gives the
  !reduced value to all processors
  CALL MPI_Reduce(rank, rank_red, 1, MPI_INTEGER, MPI_MAX, 0,
      MPI_COMM_WORLD, &
       errcode)
   IF (rank == 0) PRINT *,'Largest rank is ', rank_red
  !MPI_Allreduce combines values from all processors. Here it finds the
      sum of
  !the value (MPI_SUM) over all processors. It gets that value on all
      processors
  CALL MPI_Allreduce(rank, rank_red, 1, MPI_INTEGER, MPI_SUM,
      MPI_COMM_WORLD, &
       errcode)
   IF (rank == nproc-1) PRINT *, 'Sum of ranks is ', rank_red
  CALL MPI_Finalize(errcode)
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.

Chapter 4

Summary

4.1 Wrap Up

This workshop has aimed to be a very brief overview of the kinds and strategies for parallelism. It has deliberately left out a lot of details, but as far as possible we have tried to be absolutely accurate. We have only covered enough to get you started, but hopefully you can see how you might go about parallelising different tasks, and you have the analogies to help spot more examples in future.

A super brief, non-exhaustive summary:

- Multiple unconnected tasks

  Spin up a lot of tasks and let them run

  Use Gnu Parallel for slightly complicated schedules

  Use whatever a cluster owner recommends for task farming

- Multiple, weakly connected tasks

  Gnu Parallel with external file to manage contingencies or similar

  Worker-Controller model

  Threads (with care)

- A single task containing lots of loops or sections that can be done simultaneously

  Threads (with care)

  OpenMP

  MPI (sometimes)

- A single task that can be split into chunks (domain decomposition)

  MPI

  OpenMP (sometimes)

Note that the choice between OpenMP and MPI is usually dictated by whether you need to run on shared or distributed memory architecture. But generally, if you see lots of loops, OpenMPI can work well, whereas if you see lots of sequence points and need for synchronising data, MPI might be the key.

4.2 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.

4.3 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. We hope to produce something similar for OpenMP when time permits. 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 5

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. 33, 42

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. 7, 41

CPU Central Processing Unit; the heart of a computer, the physical chip containing the ability to do processing. Usually now contains multiple cores. 7

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. 28

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. 5, 26, 30

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. 24

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. 6

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. 33

root One of the processors in a communicator which does any unique work. , 42

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. 7

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. 20

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, 9