Introduction to Parallelism

nag®
In a serial code, statements in a program are accessed sequentially by a single processing element which has access to some memory.

In a parallel code, a number of separate processing elements are working together on the same problem.

In this module we will give an overview of the concepts of parallel computing in a fairly language agnostic manner

- Fortran or C or whatever doesn’t matter here

However we will give an intro to the concepts behind the main parallel programming paradigms in scientific computing

- Shared memory programming (for which we’ll use OpenMP)
- Message passing (via MPI)
  - Note “concepts” – the details will come in latter modules
A Quick Overview of What We Are Trying To Achieve
A Quick Introduction

- So we want to either adapt a program to run on a parallel computer or write one from scratch
- To see in very, very general terms let’s first revise what is actually in a parallel computer
- And then think of the ways we might map our program in a very general way onto that computer
What Is In A Parallel Computer?

- The building block for modern parallel computers is the Multicore Processor
  - A number of cores all share the same memory
  - Typically 2-16 cores
  - Sometimes called Symmetric MultiProcessors (SMP) or a “shared memory” computer
- An example is almost certainly your laptop or desktop
What is in a parallel Computer

- Larger parallel computers are clusters of “nodes” connected by an interconnect
  - Each node is a 1 (or more) multicore processors
  - E.g. ethernet, myrinet, infiniband …
    - A good interconnect can be expensive…
Each Core is totally independent from all the others
- What Core 0 is doing can be completely different from what Core 1 is doing
- What Core 0 is doing has no affect on what Core 1 is doing
- The internet is an example
Another Important Point

- Each node’s memory is totally independent
  - Values stored by Node 0 can be totally different from those held by Node1
  - Totally independent computers connected by some kind of wires
  - Again c.f. the internet

```
X = 7
```

```
X = -2342
```

Core 0

Core 1

Core 2

Core 3
The actual physical configuration is a bit complicated. In practice the programmer usually takes a simplified ("Logical") view of the machine to program it. The three most common in scientific computing are

- Shared Memory
  - This leads to OpenMP Programming
- Distributed Memory
  - This lead to message passing via MPI
- Mixed (or Hybrid) Mode
  - This uses a combination of the two
It is assumed each core can see *all* the memory of the computer. Leads to the shared memory paradigm of parallel computing

- Usually OpenMP but also others, e.g. Pthreads, PGAS languages, which we don’t cover here

**Usually limited to just using the cores on one node**

- Due to the underlying hardware, i.e. we don’t use the interconnect
It is assumed each core can has its own memory, and it, and only it, can see that memory directly
- i.e. A cluster of “normal” serial computers
- All communication between cores is through a “interconnect.” This leads to the message passing paradigm of programming
  - MPI
- Can use all the machine
Mixed (or Hybrid) Mode

- Mixed sees the computer as it really is
  - Use shared memory within the nodes
  - Use message passing between the nodes
- Looks the best but can be difficult to program efficiently in practice
So What Are We Trying To Achieve?

- Consider a simple vector add:

```c
void vector_add( int n, float *a, float *b, float *c ) {
    int i;
    for( i = 0; i < n; i++ )
        c[ i ] = a[ i ] + b[ i ];
}
```

- What we shall find out in these modules
  - How to parallelise this
  - Why, as hard as we try, this can only ever get a few percent of the peak performance of our computer
void vector_add( int n, float *a, float *b, float *c ) {
    int i;
    for( i = 0; i < n; i++ )
        c[ i ] = a[ i ] + b[ i ];
}

- Each iteration of the loops is independent
- So each iteration could be done by a different core!
- So what we want is independent operations
- More subtly this also means independent data
  - What if updating c[ i ] changed a[ i + 1 ]?
Now consider a dot product

```c
void dot_product( int n, float *a, float *b, float c ){
    int i;
    c = 0.0;
    for( i = 0; i < n; i++ )
        c = c + a[ i ] * b[ i ];
}
```

- Each core can perform a subset of the iterations to obtain a partial sum
- But then the cores have to work together to add up the partial sums to get the final result
  - A “reduction” operation
To perform the reduction operation the cores have to work together

So they are no longer independent

Thus cores

- May have to wait for other cores
- Will have to somehow send data to other cores
- And other more subtle effects

Much of the rest of this module is discussing the affect this could have on the performance of our parallel program
Shared Memory And Distributed Memory Programming
Before we go into the “theory” in more detail let’s just have a quick look at the two main methods of parallel programming

Not looking at “how” you program it yet, just the concepts behind them and what conclusions we can draw

A reminder:

Shared memory
- Every core executing the program can see ALL the memory and hence all the data associated with the program

Distributed memory
- Each core can only see some of the memory addressed by the program. If it requires data that it can not directly access it typically has to “ask” the core that is associated with that data to send it
We have a logical picture of the computer something like this.
• The standard method for programming in a shared memory style is **OpenMP**

• Uses threads
  • Typically each thread runs on a different core
  • The execution of each thread is completely independent from any other
    • Unless you say so in your program
  • Also note threads not unique to OpenMP
    • e.g. pthreads
    • In fact standard (modern) Unix feature
OpenMP uses the “fork-join” model:
- start as a single (master) thread
- continue as single thread until the program meets a “parallel construct”
- At that point create the required number of threads
- program statements within this parallel region are executed in parallel by each thread in this team of threads (includes master thread and slaves)
- All threads will act totally independently unless your program says otherwise
- at end of the parallel construct, threads synchronize and only master thread continues execution
- Parallelism carried out in distinct parallel regions
  - And you can have as many of these regions as you want
OpenMP – The Good

- As each thread can see all the data OpenMP tends to be somewhat easier to program than distributed memory approaches
- The fork-join model can mean it is easy to parallelise your program incrementally
  - Initially just parallelise the most “important” part, the rest of the program will just run in serial as before
  - Then look at the second most important part etc.
- Usually the resulting parallel code only differs slightly from the original serial code
  - Mainly just a fairly small set of compiler directives/”#pragmas”
- Usually limited to the number of cores in the node
  - Because this is the number of cores that actually share memory
  - i.e. can not use the interconnect
- Sharing resources, especially memory, can lead to new kinds of bugs which can be hard to find and are NOT repeatable
  - E.g. *race conditions*, see next slide
- It can be difficult to get good parallel performance
  - Forking threads is expensive
  - Cache coherency issues which can be difficult to diagnose
  - Often need lots of *synchronisations* between all the threads
    - These are required to make sure each threads view of the memory is the same as all of the others
Race conditions

- Consider the following on 2 threads
  \[ i = 0 \]
  \[ i = i + 1. \]
- What answer do you get?
- Impossible to say! Remember threads are independent (unless you say otherwise)
- In the above thread 0 might zero i and then increment it, and then thread 1 zero i and increment it, giving 1
- Or thread 0 zeros, thread 1 zeros and increments, and then thread zero increments giving 2
- Or any other ordering!
- This is an example of a race condition
- It occurs when you have *any shared facility and the order in which it is being accessed is not well defined*
- Remember the threads are totally independent (unless we say so) and so we can say *nothing* about the relative temporal ordering of their execution
- “Unless we say so” … This is where OpenMP allows us to synchronise the threads in some way to obtain a well ordered access pattern and so avoid the race condition
- To further help OpenMP allows threads to have *private variables* which can not be “seen” by other threads
  - As they are not shared can not be involved in race conditions
OpenMP is for shared memory programming
It uses a fork-join model
Threads perform the parallel execution
The shared memory view of the computer usually limits using OpenMP to one node
Shared memory makes it relatively easy to program
But also shared memory introduces new classes of bugs
And to avoid such conditions it means the threads may need to synchronize relatively often which can make obtaining good performance difficult
Distributed Memory

- In this view you have a number of *processes*
  - The distinction between processes and threads is important
    - Remember
      - Threads for shared memory
      - Processes for distributed memory
- Typically each process will be running on a different core
- Associated with each process is an area of memory
- And that memory can only be accessed by the process which owns it
- To get data from another process you have to use the interconnect
Distributed Memory

- Mem
  - Proc 0

- Mem
  - Proc 1

- Mem
  - Proc 2

- Mem
  - Proc 3

Interconnect
The standard method to use the interconnect is *Message Passing*

The standard way to do message passing is *MPI*

- *Message Passing Interface*

If process A needs data from process B then process B has to send a message containing the data to process A

- Note that *both* processes are involved
  - A *Two sided Protocol*
- Therefore they are synchronized to a degree
  - Remember, by default they run independently
- This is an example of *communication*
MPI is a library

- i.e. it works by the program calling functions or subroutines

Unlike OpenMP in typical scientific codes all of the processes are running all of the time

- You start them all at once typically with a command like mpirun

Each process has direct access only to its own memory

To access data associated with another process a call to the MPI library is required, and typically both processes involved

- Remember – two sided protocol

To make best use of this you should split up your large objects so that each MPI process only has a part of them

- For detailed reasons see later in this module!
- Can scale to very many processes and hence cores
  - Not limited to shared memory area as it uses the interconnect
- Lack of shared facilities makes race conditions much harder
- Lack of shared facilities means generally fewer synchronisations are required as don’t need to keep those facilities coherent, and so you can get more efficient code
  - Similarly cache issues are much less relevant for parallel performance
    - Still have serial cache performance issues
- You generally have to plan the WHOLE code from scratch for message passing, which again tends to lead to a more efficient solution overall
Because you don't have access to all the data associated with the problem it can be harder to program than shared memory
- Especially due to the two sided protocol
- You generally have to rewrite much of the code
  - Incremental parallelism is much harder
- The code is often significantly different from the serial implementation
First View of MPI in Summary

- It is a library
- Processes are used for the parallel execution
- All of the processes run all of the time
- Can use many cores via the method
- Can be hard to program
- But in general tends to lead to a better overall efficiency than a comparable OpenMP program
- To get the best of it split up your program’s large objects between the processes
Parallelism – some background “theory”
Previously we had a quick look at parallelism

- If we have independent operations and independent data we can split the work up between different cores

- And also the most common methods of programming a parallel computer
  - OpenMP to exploit a shared memory logical view of the computer
  - MPI for a distributed memory view

In this section we’ll introduce a little of the background theory before looking at how to get the best out of a parallel machine

- We’ll see how we might classify parallel computers
- We’ll introduce some measures of performance
  - Need to do that before we can address efficiency!
So What Is a Parallel Computer

- So first of all let’s try to characterise the various forms of parallel computing you might come across
So What Is a Parallel Computer

- What do we need to be able to compute in parallel?
  - Independent operations
  - Independent data that those operations can act upon

- So we might try to characterise computers by whether they can act in parallel either due to instruction parallelism
  - i.e. perform many different instructions at once
- Or data parallelism
  - That is act on many different pieces of data at once (possibly all with the same instruction)
- or both
This leads to *Flynn’s Taxonomy* which characterises computers according to the two criteria

- How many instructions are being executed at any one instant
- How many data items are being acted upon at any one instant
## Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
</tr>
</tbody>
</table>
**Single Instruction Single Data**: At each instant there is only one piece of data being acted upon by one instruction.
- True SISD is rare today, but you can think of a “traditional computer” being like this.
- In fact there is a fair likelihood you will never use a true SISD computer in your life.
- **Single Instruction Multiple Data**: One instruction is issued at a time, and each instruction acts on a number of different data items.
- GPUs work like this.
- **Multiple Instruction Single Data**: For each data item more than one instruction is executed at once. Rare. Redundancy is one reason – Space Shuttle.
Multiple Instruction Multiple Data: Each instruction acts on multiple data items, and many instructions are issued at once. This is what we are mostly interested in!
Why MIMD?

- **Multiple Instruction**: At any one time each core can issue (at least) one instruction, and we have multiple cores.

- **Multiple Data**: At any one time each core can access a different memory location from any other, and we have multiple cores.
- Actually due to the programming models commonly used it’s usually a subset of full MIMD functionality that we use

- Called **Single Program Multiple Data**

- In SPMD the *same program* is loaded onto all of the cores
MIMD v. SPMD v. SIMD

- MIMD: Each core is totally independent of any other and may be running totally different programs.
- SPMD: Each core is running the same program. However at any given time each core can be executing any part of that program, not necessarily the same part as any other core.
- SIMD: Each core is running the same program, and at any given time each core is running **exactly** the same part of the program (i.e. instruction) as any other core.

- Actually there’s a whole range of possible behaviours between the MIMD and SIMD extremes ...
So We’re Interested In SPMD

- So SPMD will be the general model of the computer that we are running our problems on.

- We’ll next have a little look at some general ideas that try explain how efficiently our parallel programs run

- So we better define efficiency!

- But first it’s useful to introduce the big O notation ...
The big O notation is used to indicate how something to do with a program/algorithm/virtually anything scales asymptotically with a given parameter

- i.e. at some limit of that parameter how a change in the parameter affects the observation of interest
- Generally interested in large values of the parameter

One of the most common and possibly simplest uses is to look at how the time taken for an algorithm changes with the size of the problem that is to be solved.
Consider $Ma = b$, where $a$ and $b$ are vectors of size $N$, and $M$ is a $N \times N$ matrix. How many operations does this take?

- To generate each element of $b$ takes $N$ multiplies and $N-1$ additions, and there are $N$ elements of $b$
- So it takes $N \times N + N \times (N-1) = 2N^2 - N$ operations
- For large $N$ only the squared term is important
- The time taken for matrix vector multiplication is said “to scale as $O(N^2)$”
- Double the size of the problem and the time taken increases by (approximately) a factor of 4
Other Examples

- Dot product: \(O(N)\)
- Matrix Matrix Product: \(O(N^3)\)
- Configuration Interaction: \(O(N!)\)
  - (A method in quantum chemistry)
- See if you can work out the above

- I tend to think about what happens to the time when you double the problem size
  - If the time goes as \(2^\beta\) then the algorithm is \(O(N^\beta)\)
You can use also O notation e.g. For memory requirements
- Matrix vector multiply is $O(N^2)$ in memory requirements
- As is matrix matrix multiply
  - Why?
- So we can describe the scaling of the time or the memory required by an algorithm by the big O notation

- What about the scaling with the number of cores?
  - i.e. how much faster (or slower!) does your program get if you use more cores
  - After all this is the fundamental question for measuring efficient parallel execution of a program, so we better have a measure

- For this the most common measure is *Speed Up*
- *Speed up* answers the question “How much faster does my program run if I use P processors”

- $S(P) = \frac{T_1}{T(P)}$
Absolute Speed Up

- What we should really measure is *Absolute Speed Up*

- \[ S(P) = \frac{T_s}{T(P)} \]

- Where \( T_s \) is the time to run the *best implementation of the serial program*, and \( T(P) \) is the time to run the parallel code, *which may be an entirely different program*, on \( P \) cores

- For instance you may use a different algorithm in the parallel code from the serial code
Relative Speed Up

- However what is almost always measured is *Relative Speed Up*

- \[ S(P) = \frac{T(1)}{T(P)} \]

- i.e. we compare the speed against the time taken *on 1 core by the parallel program*

- Saves writing both the serial and parallel code
A program is said to display *Linear speed up* if

- $S(P) = P$

So if you run on $P$ processors, it runs $P$ times quicker.

This is the ideal situation:
- We’ll find out soon why you almost never see it except possibly at very low $P$

Also called *perfect scaling*
But What Does It look Like In Practice?

- For a Linear Equation Solve, $N=5000$
So Why Is It Not Perfect!?!?

- Many Possible Reasons! Much of the rest of this talk

- But before we move onto that there’s a caveat about speed up you should be aware of, and also a couple of other measures of parallel performance.
The Caveat

- Speed Up says nothing about time to solution!
- Let’s compare two diagonalization routines

![Graph showing Speed Up vs Number Of Cores]

- So Standard is best?
The Caveat

![Graph showing the comparison between Time To Solution/s and Number Of Cores for Standard and Divide And Conquer methods.](image)
The Caveat

- So remember that a good speed up does not necessarily mean it is the best method
  - It says nothing about time to solution
- And we’ll see soon that inefficient serial methods tend to display better parallel scaling
- For this reason I prefer to plot something related to $1/t$ against core counts
  - e.g. timesteps per second
- This captures both scalability and performance
  - Straight line still show good parallel scalability
  - But now can see if we are getting many or few timesteps per second
The Caveat

Introduction to Parallelism
Another measure of parallel performance is *Parallel Efficiency*

This is related to the speed up and defined by

\[ E(P) = \frac{T_1}{P \cdot T(P)} = \frac{S(P)}{P} \]
On some computers you will be CHARGED! for use

This is the usual situation on national resources

Typically you get a finite budget which gets deducted from every time you use the machine

Quite how the charging works will vary from machine to machine.

Typically you get charged a fixed rate per node that you use, and for every second you use

Cost=nodes*time

i.e. If you leave cores idle in a node you get charged for them
Cost on a 8 Core/Node Machine

![Graph showing relative cost vs. number of cores. The graph compares Standard and Divide And Conquer methods.](image)

- **Standard**
- **Divide And Conquer**
Now we know how to measure parallel performance we can start to ask why all codes don’t display perfect speed up.

For instance below is an example from DL POLY. Why do we not see perfect speed up, and why do different parts of the code show differing scalability?

```
Number Of Cores
```

```
Speed Up
```

```
- Total MD time
- van der Waal’s
- Short Range Ewald
- Long Range Ewald
```
So Why Is It Not Perfect?

- Many possible reasons
  - Residual serial code
  - Load Imbalance
  - Communications
  - Intra-node effects
  - I/O
  - Contention
  - ...

- Will focus on the first 3
Residual Serial Code

- It’s impossible to parallelise all the code ...
- There will always be some portion of it that is running in serial
  - Common example – reading the input

- In fact let’s go back to the OpenMP execution model
• Parallelism carried out in distinct parallel regions
- “Parallelism carried out in distinct parallel regions”
- Between which are serial regions!
- How much can this serial code affect things?
- And note this is for all parallelism, not just OpenMP
  - Let’s just be slightly more careful about what we are saying here …
Serial here can mean you have multiple cores executing a program but all the cores are doing exactly the same thing on exactly the same data (values)

- Thus this is possible in either OpenMP or MPI
- In OpenMP we are talking *either* about the parts of the code outside the parallel regions *or* inside a parallel region but all the threads are doing the same thing
- In MPI the issue here is situation when all the processes (remember they are all running all the time) are all doing the same thing on the same data values – e.g. reading an input file

- Let’s look into the effect of this
- It is the key idea behind *Amdahl’s Law*
Amdahl’s Law

- Let the program spend a fraction $\pi$ of its time doing parallel work and $\sigma$ doing work that is not parallel (serial).
- By definition the time take on 1 core is
  \[ t_1 = \pi t_1 + \sigma t_1 \]
- So the time taken on $P$ cores is
  \[ t_P = \frac{\pi t_1}{P} + \sigma t_1 \]
- So
  \[ S(P) = \frac{t_1}{t_P} = \frac{\pi + \sigma}{\pi/P + \sigma} \]
- And so because $\pi + \sigma = 1$

\[ S(P) = \frac{1}{(\pi/P + (1-\pi))} \]
Amdahl’s Law

- This law is frightening!

- First let us look at the very large number of cores limit, \( S_\infty \)

\[
S_\infty = \frac{1}{1-\pi}
\]

- So if \( \pi = 0.5 \), we can NEVER get a speed up of more than 2, however many cores we use!

- Further to use 100 cores we better at least have \( \pi = 0.99 \), as otherwise we have no hope of going 100 times faster
Actually this limit is obvious. Pretend we have a job that has two parts:

1. The first takes 1 day for one person to complete, ½ a day for two etc.
2. The second part also takes 1 day for one person. However it takes 1 day however many people do it.

The fastest this job can ever be done in is 1 day, a speed up of 2 – as predicted by Amdahl’s law as $\pi=0.5$

Now let’s quickly consider how the predicted speed-up varies as a function of the number of cores
Amdahl’s Law – Small P

![Graph showing speed up vs cores for different values of P.]

- Red: P=0.5
- Yellow: P=0.75
- Green: P=0.9
- Cyan: P=0.95
- Blue: P=0.99
- Navy: P=0.999
- Grey: P=0.9999
- Blue dashed: P=0.99999
- Black dashed: Perfect Scaling

Cores vs Speed Up
Amdahl’s Law – Large P

![Graph showing speed up versus cores for different parallelization levels](image-url)

- $\pi=0.99$
- $\pi=0.999$
- $\pi=0.9999$
- $\pi=0.99999$
- Perfect Scaling

**Axes:**
- Y-axis: Speed Up
- X-axis: Cores
So to exploit even a medium number of processors a very large part of the runtime our program must be in parallel.

And it is worse for large numbers.

\( \pi=0.99999 \) corresponds roughly to doing 1 second of serial work per DAY of parallel work!
- But it can’t be that bad!
- We have programs that can exploit 1000s of processors and even more ...

- The missing point is that $\pi$ and $\sigma$ may scale differently with the size of the physical system, for example
  - $\pi$ might be $O(N^3)$
  - $\sigma$ might be $O(N)$

- This leads to *Gustafson’s law*
Let’s consider a parallel dot product:
- $\pi$ scales with system size as $O(N)$
- $\sigma$ scales with system size as $O(1)$
  - I’m assuming it’s only the setup which is independent of system size
- For a given system size let $\pi=\sigma=0.5$
- Now consider system size $2N$. How do $\pi$ and $\sigma$ change?
  - $\pi$ is $O(N)$. So for system size $2N$ $\pi'=2\pi=1$
  - $\sigma$ is $O(1)$. So for system size $2N$ $\sigma'=\sigma=0.5$
- So the new parallel fraction is given by

$$\frac{\pi'}{(\pi'+\sigma')}=0.66666666666$$
Another example. Suppose for a given size $\pi=\sigma=0.5$

For system size scaling in this problem
- $\pi$ is $O(N^3)$
- $\sigma$ is $O(N)$

Let us now double the problem size. From 1 and 2

$\pi'=8\pi$ \quad $\sigma'=2\sigma$

So

$$\frac{\pi'}{(\pi'+\sigma')} = \frac{8\pi}{8\pi+2\sigma} = 0.8$$
Gustafson’s Law Speed Up on 128 Cores

- Using these two examples on 128 cores

![Graph showing speed up vs system size/N]
So Size Does Matter ...

- So Gustafson’s Law tells us 2 things
  - Parallelise the parts of the code that scale worst with system size
    - As these are usually the most expensive parts this is usually expressed as “Parallelise the most expensive parts of the code”
    - Should be fairly obvious
  - Parallel computers are best for solving large problems
    - Might not be so obvious

- Debatably this last point is the most important point. In fact I think it is worth a slide on its own:
PARALLEL COMPUTING IS FOR SOLVING BIG PROBLEMS
Strong and Weak Scaling

- Strictly the measure we have considered so far is called **Strong Scaling**
  - For a fixed problem size we consider how the time to solution varies as a function of the number of cores
- Beloved of computer manufacturers is **Weak Scaling**
- Here you scale the problem size with the number of processors
  - So if you double the number of processors you also double the problem size
  - Whether this is always appropriate is debatable …
    - The number of atoms in a protein does not depend on the number of cores used to simulate it
Weak Scaling In Practice

- By Gustafson’s law doubling the system size will help the parallel speed up
  - Good weak scaling is somewhat easier to achieve than good strong scaling
  - This why computer vendors love it!
- However the science of a problem doesn’t always allow us just to double the problem size
  - The chemistry/physics/engineering/whatever constrains us
- So strong scaling is generally more useful
- And check what number people are quoting!
You can do an analogous analysis on memory usage, between

- **Distributed Data**
  - This corresponds to $\pi$ above
  - These are objects for which there is only 1 copy in the *whole* parallel computer
  - Thus as you increase the number of cores the amount of memory required *per core* decreases

- **Replicated Data**
  - This is $\sigma$ above
  - Here each core has its own copy of the whole object
  - Thus the memory required *per core* remains constant as you increase the number of cores
Amdahl’s and Gustafson’s Law For Memory

- So:
  - Distributed data
    - In OpenMP these are objects shared by all the cores
    - In MPI these objects split up across all the cores, so the memory associated with each process only contains part of the whole object
  - Replicated Data
    - In OpenMP these are private variable where each thread has its own version
    - In MPI these are object where the process holds all of the memory associated with an object
Amdahl’s and Gustafson’s Law For Memory

- Important conclusions:
  - 100 times the memory does not necessarily mean 100 times bigger calculation due to the replicated objects
  - (OpenMP) Make the most memory expensive objects such as large arrays shared between the threads
  - (MPI) Distribute the most memory expensive objects – i.e. split them up so that each process only holds part
Some Background Reading

Related in spirit is Brook’s Law:

- “Adding manpower to a late software project makes it later”
  - Corollary: “Nine women can't make a baby in one month”

What Else Can Go Wrong

- So far we have considered the effect of all the code not running in parallel
  - Note: no communications between cores yet! All so far has them running totally independently
- The second common effect to consider is load imbalance
Say we have 100 jobs and 100 people. Each of the jobs is totally independent from all the others.

- No residual serial effort, so no Amdahl’s law effects.
- So 100 people means that the everything is done 100 times faster – 1 job, 1 person?

But what if 99 of the jobs take 1 hour, and the 100th 1 month?

The total time taken is dictated by the worker that takes the longest time to finish the job.
Load Imbalance

- Load imbalance occurs when the work each core needs to do differs from the others.

- It is often observed in “task farms”.
A task farm is when \( T \) independent tasks must be performed by \( W \) workers. Each of the workers performs a subset of the tasks. Unless all the tasks take exactly the same time, load imbalance is all but unavoidable.

Note that a worker is NOT necessarily a core in a parallel computer. In modern computer programs it can often be a number of cores working together in parallel on the problem.

- You’ll cover mechanisms in MPI to facilitate this
- Or in some cases (debatably) a fraction of a core!
- But this is more common in finance and search engines, rare in science
There are two main possibilities …
Each worker performs one task, when it is finished it asks for the next task that needs to be done

- Workers performing quick tasks are not held back by those performing slow ones
Static Load Balancing

- At the start each of the workers is given a list of jobs to do
  - Or at least can work out which of the jobs it needs to do (e.g. Every $P^{th}$ job if there are $P$ cores)

- It then goes and does those jobs, and no others.
  - However one worker may get all “quick” jobs and another all “slow” jobs
  - So workers may end up waiting on others at the end
Changing the Load Balancing algorithm in OpenMP is easy
- When you learn about schedule clauses you’ll see why

So why should we ever use static load balancing?

Less overhead than dynamic
- Just “asking” what is the next job can be quite expensive

Therefore for simple cases static *might* be quicker
- Depends on the implementation, the problem, the phase of the moon, the relative humidity ...
Dynamic v. Static Load Balancing: MPI

- Dynamic load balancing is **much** harder in MPI
  - Somehow have to know global state of system at anyone time
  - Easy in OpenMP – shared memory
- Difficult in MPI – everything “private”
- So MPI programs often use Static Load Balancing
- That said an observation:
  - If you have lots of tasks compared to the number of cores you are using statistically it’s less likely that any one core will get all the expensive ones, so ...
PARALLEL COMPUTING IS FOR SOLVING BIG PROBLEMS
What else?

- So far we have assumed the cores are all operating totally independently

- Amdahl’s law effects will always occur
  - Load balance effects will always occur

- But what if they are not totally independent?
The Cores Are Not Independent

- This is *communication*
- We’ve seen examples of it already
  - In OpenMP we’ve talked about the need for synchronization, either at the end of a parallel region, or to ensure the accessing shared data is done in an organised manner to avoid race conditions
  - In MPI the two sided protocol means data has to be sent via whatever is the interconnect if one processor needs data from another process
- Let’s consider a simple example that illustrates the need for communication: 1 dimensional quadrature by the trapezium rule, i.e. numerical integration
Integration: The Trapezium Rule

- Consider the definite integral
  \[ I = \int_{a}^{b} f(x) \, dx \approx \sum n \frac{h(f(a+nh)+f(a+(n+1)h))}{2} \]

- We are approximating the integrals by a sum over areas – each area is independent so we can give a number of them to each core, e.g. for 3 cores
Integration: The Trapezium Rule

- To get the final answer we add up the partial results from all the cores:
  \[ I = \sum_{p} I_p \]

- This is the “new” part

- This is an important class of operation called a reduction operation
  - Individual values on many cores are “reduced” to a single value by application of an operator (summation in this case)
  - Done by a reduction clause in OpenMP
  - And by a call to MPI_Reduce or MPI_Allreduce in MPI
  - And of course MPI will have to use the interconnect
  - So let’s get in early the vital message about interconnects
The Interconnect

INTERCONNECTS ARE SLOW COMPARED TO THE CORES
The Speed of Typical Interconnects

![Graph showing the speed of Ethernet and Myrinet interconnects.

- **Y-axis** (Time/s): 0.000 to 0.006
- **X-axis** (Length/Bytes): 0 to 8000000

- **Ethernet**
- **Myrinet**

Introduction to Parallelism
The Interconnect

- Note the roughly straight line
- So the time to transfer N bytes of data over an interconnect roughly obeys: \( t = \alpha + \beta N \)
- \( \alpha \) is the latency
  - Typically \(~1\text{-}100\mu\text{s}\) for modern networks
  - It is the time to transfer zero bytes of data
  - Dominates the time for short messages
- \( 1/\beta \) is the bandwidth
  - Typically \(~100\text{MByte/s}\text{-}2\text{GByte/s}\) for modern networks
  - Dominates the time for long messages
- Different networks have different bandwidths and different latencies
α is the latency

Typically ~1-100µs for modern networks

So the SHORTEST POSSIBLE time taken using the interconnect is around 1µs

In that time a modern core can do many thousand floating point operations!

To put it another way, you can do thousands of operations that are “doing science” in the time it takes to do 1 that does no science at all
Use of the interconnect, is slow and is not what we want to do
- Communications add nothing to the Science, they are only a means to solve the problem required on parallel machines
- So to get good use out of parallel computers you want
  \[ T_{\text{compute}} \gg T_{\text{communicate}} \]
- or
  \[ T_{\text{compute}} / T_{\text{communicate}} \gg 1 \]
- You want to spend your time computing, not communicating!
- Exactly the same argument applies to syncs in OpenMP
  - As these are just a form of communication
We want the compute time to be much longer than the comms time. Detail on that soon, but what are the basic ideas?

1. You want the compute to scale more rapidly with system size than the communication, as then as N gets sufficiently big the communication time will disappear.

Parallel computing is for solving big problems!

2. As you increase the number of cores ideally you also want communication time to decrease as otherwise it at sufficiently large core count it will dominate the decreasing (due to parallelism) compute time.
   - What!? More cores, less communication time?
   - That’s where you have to get clever, see later!
What Else Can Go Wrong?

- We have met 3 reasons why a parallel program does not run 100 times faster on 100 cores
  - Amdahl’s law effects i.e. Residual serial code
  - Load imbalance
  - Communications
- There are many others
  - OS Jitter
  - false sharing
  - Contention
  - I/O
  - The computer hates you
But Just Occasionally

- Just occasionally you can get *Super-Linear Speed Up*
- So on P cores it runs more than P times quicker!
- Why might that be?
- More cores doesn’t just mean more cores!
  - It means more cache
  - It means more available memory as you have more nodes
  - It may mean more disc (not on HPCWales)
- If Amdahl’s law, load balance, comms are not important and your program clever you might be able to take advantage!
  - For instance store temporary results in the extra memory rather than recalculate them
What else Can Go Wrong

- Finally parallelism introduces new classes of bugs
  - Race conditions which we have seen
    - Common in OpenMP, rare in MPI
  - Deadlocks
    - Rare in OpenMP, common in MPI
Race Conditions

- We’ve met this already
- It occurs when multiple threads/processes depend on some shared object
- Much more common in OpenMP than MPI
  - All variables “private” in MPI
- However can happen in MPI programs
  - E.g. Wildcarding on receives – see later!
Race conditions are an example of *Non-Deterministic Behaviour*

In serial for a given input a program (should) always produce the same output – Deterministic Behaviour

No longer true for parallel programs

- Depends on exact relative timing of the computational thread/processes

Might be trivial – e.g. Ordering of output

Might be difficult

- Irreproducible results
- Occasional “glitches”
Deadlock

- Deadlock is when two or more competing actions are waiting for the other to complete, and so none do:

- “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

- Law passed by the Kansas Legislature
Deadlock

- Nett result: The program “just hangs” and appears to do nothing

- Most common in MPI often due to the 2 sided protocol
  - Core A is waiting for core B to be ready to receive data, but ...
  - Core B is waiting for core A to be ready to receive data
  - Another example is not all processes call a collective
  - You will do this!

- But certainly not unknown in OpenMP
  - E.g. When dealing with locks
Deadlock can be very difficult to debug
  
  - Very often non-deterministic
  
  - Can be difficult to find out quite which bit of the program is “hanging”
  
  - An Interactive debugger (e.g. DDT on HPCWales) *might* help
Summary

- Covered a lot of background material here
- Basic classification of parallel computers
- Measures of parallel efficiency, especially speed up
- Some ideas as to why running on P processors does not mean P times quicker, especially
  - Residual serial code
  - Load Imbalance
  - Communications
- A brief look at the world of parallel bugs
- In the next section we will try to apply these ideas in order to work out how we might parallelise solving problems, and try to get an indication as to how well we expect our solution to scale
Parallelism – Applying The Theory nag®
Applying The Theory

- Last section we covered some ideas behind
- Here we will apply those ideas to look at how to parallelise a number of problems, and try to do it efficiently
- The essence of the idea is to make simple mathematical models of how you expect the time to solution of the problem to vary with
  - Size of the problem
  - Number of cores you are using
- From that you can quickly see if your method has a chance of scaling to many cores
- And it can help you chose when there is more than 1 way of splitting up the problem
To start with let’s go back to the one-D quadrature problem
Remember

1. We split up the area to be integrated into trapezia
2. We give a subset of the trapezia to each core
3. Each core calculate its contribution to the area
4. We then add all the contributions together (a reduction operation) to find the total area

How does what we learn previously apply to this?
There is essentially no residual serial code in this portion of the solving the problem as each core will work on a different subset of the trapezia.
Load Imbalance in the Integration Problem

- Provided there are many more trapezia than cores, there will be little load imbalance as each core can be assigned more or less the same number of trapezia to work on.

- However, if the number of cores is comparable to the number of trapezia, load imbalance could become a problem.
  - For instance, if we have 9 trapezia and 6 cores, 3 of the cores will have 2 trapezia to work on, while the other 3 will only have one, and the fractional difference is now large.

- Parallel computing is good for large problems!
- There is an obvious communication stage in the parallel algorithm
  - The Reduction
- In general for reductions if we have P cores involved and N data items to reduce the time is proportional to $N \cdot \log_2(P)$
- Here we have one data time
So we can now write down simple expressions for our compute and communicate times

- As there are no significant Amdahl’s law or load imbalance

\[ T_{\text{compute}} \propto \frac{N}{P} \]

- as we have \( N \) trapezia to work with and each core is doing \( 1/P \) of them

- The communications are simply a reduction of 1 number across the cores, so we can write the communication time as

\[ T_{\text{communicate}} \propto \log(P) \]

- And so we can write our ratio as

\[ \frac{T_{\text{compute}}}{T_{\text{communicate}}} \propto \frac{N}{P\log(P)} \]
Integration: The Trapezium Rule

- So for the integration
  \[ \frac{T_{\text{compute}}}{T_{\text{communicate}}} \propto \frac{N}{(P \log(P))} \]
- Remember we want this ratio to stay large as that will mean the communication overhead stay low
- So we can make two conclusions
  - The more processors we use, the lower the ratio
    - More cores, more communication
    - So 100 cores does NOT mean 100 times quicker
  - The more data points, the bigger the ratio
    - This means the bigger the problem the more efficiently we perform the integration on the parallel computer, so ...
PARALLEL COMPUTING IS FOR SOLVING BIG PROBLEMS
• Back to our example

\[ T_{\text{compute}} \propto \frac{N}{P} \]
\[ T_{\text{communicate}} \propto P \log(P) \]

• So

\[ T_{\text{tot}} = aN/P + bP \log(P) \]

• Where a and b are constants of proportionality
• At large P this *increasing*
• So the more processors the longer it takes!
• Amdahl’s law and load imbalance can’t do this
• If you use too many cores your program may run slower!
We can avoid Amdahl’s law effects and load imbalance fairly easily due to the regularity of the problem.

At the end of the problem we require a reduction operation.

This requires communications.

We can model the compute time and communication time.

This shows that as the number of cores is increased the communication time becomes significantly more important relative to the compute time.

Thus this problem will NOT scale to many cores.

And this is ultimately because the communication time is increasing with the number of cores used.
To have any hope of a program that scales to very many cores the communication time MUST be decreasing with number of cores as otherwise it will totally dominate the compute time.

Seems counter-intuitive – more cores, less communication, how???

Well that's where you have to be smart!
Let’s look at a few more examples
First let’s look at a very important case, solving Laplace’s equation
Solving Laplace's equation via Jacobi iteration

- Laplace's equation: $\nabla^2 \phi = 0$
- This can be solved by “Jacobi Iteration”
  - Described in the next slide
- While this is far from the quickest method it is the basis behind many more important methods
  - E.g. multigrid
  - So what we learn here can be applied there
- And these methods can be applied to many differential equations important in science
  - Used in CFD, Computational Chemistry, …
- The methods in this section are very important!
Solving Laplace's equation via Jacobi iteration

- Laplace's equation: $\nabla^2 \varphi = 0$
- For simplicity let's assume 2D
- So *given some boundary conditions*
- Evaluate an initial guess at the function on a regular grid of points
  - Usually just all values zero except at the boundaries
- Iterate
  - $\Phi_{ij}^{n+1} = \frac{1}{4} (\Phi_{i+1,j}^n + \Phi_{i-1,j}^n + \Phi_{i,j+1}^n + \Phi_{i,j-1}^n)$
  - i.e. replace the mid point by the average of its nearest neighbours
  - And (eventually) you'll converge to a solution
$\Phi_{ij}^{n+1} = \frac{1}{4} [\Phi_{i+1,j}^n + \Phi_{i-1,j}^n + \Phi_{i,j+1}^n + \Phi_{i,j-1}^n]$
Parallel Methods of Solution

- To solve this in parallel using MPI we have to decide how to divide up the work *and the data* between the cores.

- Two possible ways of doing this are:
  - Each core holds all the grid and works on a subset of the rows (or columns) of the data. Once we have updated the grid on a core we have to merge together all the updates.
    - As each core holds a copy of all the data this is termed a replicated data solution.
    - Note the maximum size of problem is limited by the local memory.
  - Each core only holds a subset of the grid.
    - This leads to *Domain Decomposition* which we will discuss later.
Let’s what we looked at earlier for a replicated grid

- There are no significant serial parts of the code left, so Amdahl’s law effects are unimportant
- The regularity of the grid makes load balancing easy (each core gets 1/P of the rows or columns) provided the grid size is big enough compared to the number of cores
- Communications … At the end of each update of the grid the new contributions from each of the different cores need to be combined – effectively we are again reducing P results down to 1 merged result, so we can use a reduction operation. So for a NxN grid

\[ T_{\text{communication}} \propto N^2 \log(P) \]

- We can actually do slightly better than this, but it doesn’t change the argument
Replicated Grid – Can it scale

- Thus for a NxN replicated grid solution
  \[ T_{\text{compute}} \propto \frac{N^2}{P} \]
  \[ T_{\text{communicate}} \propto N^2 \log(P) \]

- Note the communication time is again increasing with P so we can’t be too hopeful about this working well …

- Thus our ratio is
  \[ \frac{T_{\text{compute}}}{T_{\text{communicate}}} \propto \frac{1}{P \log(P)} \]

- So again this decreases rapidly because the communication in the merging stage scales badly with increasing cores.

- In general replicated data approaches will never scale well because the merging costs increase with core count.
So replicated data approaches will never scale to large core counts

Thus the converse is for an MPI solution you must split up your large objects between the cores if you want an efficient solution

So let’s look at how you might think about this with domain decomposition
  - It might not work well but we haven’t got an alternative!
Domain Decomposition

- Each core is responsible for a part of the space onto which the problem maps
  - Its *Domain*
- And to complete its computation each core needs to exchange points at the edge of its domain with the neighbouring cores
  - *Halo Exchange*
- So first exchange, then ...
  - Do computation
  - And repeat until converged
- General method where the interactions are
  - Local, i.e. only depend on the separation of the things of interest
  - Ideally short ranged
- How well do we expect this to work?
One Possible Decomposition on 4 Cores

\[
\phi_{ij}^{n+1} = \frac{1}{4} [\phi_{i+1,j}^n + \phi_{i-1,j}^n + \phi_{i,j+1}^n + \phi_{i,j-1}^n]
\]
Halo Exchange

\[ \phi_{ij}^{n+1} = \frac{1}{4} \left[ \phi_{i+1,j}^{n} + \phi_{i-1,j}^{n} + \phi_{i,j+1}^{n} + \phi_{i,j-1}^{n} \right] \]
- So as shown in the diagram each core holds the data associated with its domain plus a thin “halo”
- This halo contains the points from the neighbouring domains which are owned by other cores
- This allows it to be able to update the points in its domain
  - Note it *only* updated points in its own domain, not the halo
- After which the halos will need updating by communicating the edge data from the cores which hold the neighbouring domain
- And similarly the core in question will need to send its own edge points to neighbouring cores so they can update their own domains
- Again Amdahl’s law and load imbalance should be unimportant provided the grid is large compared to the number of cores in use
- So we now want our compute to communicate ratio
- Compute we can work out as follows:
  - The amount of work to be done is proportional to the number of points a core owns
  - For load balancing want to give the same number of points to each core
  - Hence on P cores and a square of side N the compute is proportional to $N^2 / P = A/P$, where $A$ is the area of the grid
  - And in 3 dimensions for a cubic grid we get $V/P$
We can choose our domain any way we like as long as their volume is \( \frac{V}{P} \)
- Same volume implies same number of points so good load balancing
- We want choose it so that the comms time is minimised.
- The message passing time is proportional to the surface area of the domain
- So we want to minimise the surface area of the domain subject to the constraint that the volume is \( \frac{V}{P} \).
- There are also packing constraints
- Use of Lagrange multipliers shows that for a square box the shape of domain that minimises the message passing is …
A Square!
- Huzzah, this makes life easy!

Similarly in 3D it is a cube, and as 3D is more important (but harder to draw) let’s quote results from that now on

Hence the surface area of a domain, and hence our measure of the comms cost, is \(6N^2/P^{2/3}\)

Note this is *decreasing* with number of cores so we have a chance of this scaling well!
- To an extent we’ve “parallelised the message passing”
- As each core sends less data as P increases, so the comms costs decreases
Therefore our compute to communicate ratio is 
\( \frac{N^3/P}{(6N^2/P^{2/3})} = \frac{N}{6P^{1/3}} \)

- Big N good – Parallel computing is good for …
- Also note because we have split up the large object (the grid) we are not limited to problem sizes that fit in the local memory, but to problem sizes that fit in the memory on the whole machine

- **Make sure you understand this technique – It is very important in parallel computing**
- Note how the scalability improves as the resolution (i.e. the number of grid points) increases
What About OpenMP?

- So far we’ve concentrated on MPI, what about OpenMP?
- **Synchronisation** is the most common cause for poor scaling for shared memory codes
  - Remember syncs at end of loops, sections, etc.
  - Remember it might be an issue with the algorithm itself
- Synchronisation is essentially a “global” communication like a reduction
- So same ideas apply to OpenMP as MPI
- Use an algorithm that requires the fewest synchronisationa
  - Avoid Synchronisation wherever possible
- BUT remember how easy race conditions are – make sure it works!
So let’s look at an OpenMP example

This also shows that picking the right algorithm is crucial to get good performance

And that the best performing algorithm in parallel occasionally is NOT the same as the best in serial

The example is orthogonormalisation
  - Given a number of vectors $q_i$ transforming them so $q_i.q_j = \delta_{ij}$

First we will look at a standard serial method, Modified Gram-Schmidt (MGS), and then the Iterated Classical Gram-Schmidt method (ICGS)
Modified Gram Schmidt

1. for (k=1,2,3,... , m)do
2. for (i=1,2,3,... , k)do
3. \[ s = \mathbf{q}_i^T \mathbf{q}_k \]
4. \[ \mathbf{q}_k = \mathbf{q}_k - s\mathbf{q}_i \]
5. end do
6. \[ s = \mathbf{q}_k^T \mathbf{q}_k \]
7. \[ \mathbf{q}_k = \mathbf{q}_k / \sqrt{s} \]
8. end do

- The MGS algorithm for a set of m vectors is shown to the left
- Each iteration of both the k and i loops depend on the previous iterations
  - So we can’t split those up amongst the threads as they are not independent
- But the dot product at step 3 and the vector add at step 4 can be split up (with a reduction for the dot product)
1. for (k=1,2,3,... , m)do
   2. for (i=1,2,3,... , k)do
      3. \[ s = q_i^T q_k \]
      4. \[ q_k = q_k - sq_i \]
   5. end do
   6. \[ s = q_k^T q_k \]
   7. \[ q_k = q_k / \sqrt{s} \]
   8. end do

- So we must complete step 3 before we do step 4, as the value of s is needed
- And we must complete updating \( q_k \) before the end of the i loop as it will be required by the next iteration of the i loop
- Thus in parallel we must synchronize at the point indicated by the red lines due to steps 3 and 4 having mutual data dependencies
1. for \((k=1,2,3,\ldots, m)\) do
2. for \((i=1,2,3,\ldots, k)\) do
3. \(s = q_i^T q_k\)
4. \(q_k = q_k - sq_i\)
5. end do
6. \(s = q_k^T q_k\)
7. \(q_k = q_k/\sqrt{s}\)
8. end do

- So for a set of \(m\) vectors each of length \(m\)
  - Again Amdahl’s law and load imbalance is not important for a big enough problem
- The compute time on \(P\) threads is \(O(M^3/P)\)
- The communicate time is proportional to the number of synchronisations which is \(O(M^2)\)
  - Which while not decreasing is at least constant!
So for MGS

\[ T_{\text{compute}} \propto \frac{M^3}{P} \]
\[ T_{\text{communicate}} \propto M^2 \]

So our ratio is \( \frac{M}{P} \)

Which decreases fairly rapidly with increasing \( P \) so we don’t expect it to scale that well

Further we have assumed the synchronisation cost is the same whatever the number of threads. Almost certainly this will not be true, more threads will take longer, so the ratio decreases even more quickly
Iterated Classical Gram Schmidt

1. for \((k=1,2,3,\ldots , m)\) do 
2. \(s^0 = q_k^T q_k\) 
3. \(q_0^k = q_k\) 
4. for \((r =1,2,3\ldots )\) do 
5. \(p_r = Q_k^T q_{kr}^{r-1}\) 
6. \(q_k^r = q_k^r - Q_k p_r\) 
7. \(s_r = q_k^r T q_k\) 
8. if \((s_r > s_{r-1}/4)\) then 
9. break 
10. end if 
11. end do 
12. \(q_k = q_k^r / \sqrt{s_r}\) 
13. end do 

- ICGS is a bit more complicated 
- However the crucial point is the \(r\) loop is only done until a tolerance is reached, it is NOT over all vectors 
  - Typically only 2 iterations are required 
- The main points where synchronisation is required are shown 
- So typically the algorithm requires \(M*2*3=6M\) syncs
Iterated Classical Gram Schmidt

1. for (k=1,2,3,... , m) do
2. \( s^0 = q_k^T q_k \)
3. \( q^0_k = q_k \)
4. for (r =1,2,3...) do
5. \( p_r = Q_k^r q_k^{r-1} \)
6. \( q_k^r = q_k^r - Q_k p_r \)
7. \( s^r = q_k^r^T q_k \)
8. if \( (s_r > s_{r-1}/4) \) then
9. break
10. end if
11. end do
12. \( q_k = q_k^r / \sqrt{s^r} \)
13. end do

- So ICGS only requires O(M) syncs compared to O(M^2) for MGS!
- However it is more complicated and this means the serial (i.e. non-parallel run time is longer)
  - Typically it takes 2-3 times longer
- So we expect ICGS to scale better, but have a worse 1 core “starting point”
- Can ICGS beat MGS?
Comparison of the Two Methods

- Run on a teaching cluster at Southampton, M=1024
  - Remember the details will vary from machine to machine!

![Graphs showing comparison of two methods]
So on the Southampton machine MGS is faster when running on 1 thread, but ICGS is better on any more of that due to the better scaling

This better scaling is because it requires MANY less synchronisations

We need synchronisation points here due to one step of the algorithm requiring data prom a previous step to be completely updated

A data dependency

Keeping such syncs to a minimum is the key for good OpenMP performance

MPI shows very similar behaviour – try the analysis yourself!
Spectral methods using Fast Fourier Transforms (FFTs) are very important in a number of areas of scientific computing

- Condensed matter, CFD, Environmental modelling …

Let’s have a look at how a 2D FFT is usually parallelised with MPI

- Very similar to the 3D case, and it is much easier to draw
- Can't use halo exchange as the interactions aren't local
  - Need all of the values to perform a 1D FFT

- Again we have a grid of points, keep it square for simplicity
- Assign the first \( N/P \) columns to core 0, the next \( N/P \) to core 1 etc
- Use a library FFT routine to perform \( N/P \) FFTs for each column that this core owns
- Rotate the square and rearrange the data amongst the cores so that core zero now owns the first \( N/P \) columns (i.e. What were the first \( N/P \) rows) and so on
- Again use a library FFT to perform the \( N/P \) FFTs for each column that this core currently owns
- Best shown by pictures!
Again we can eliminate serial code and load balance this as long as the size of the grid is big enough.

So the computation cost is given by:
- Each core has $N/P$ FFTs to perform, each of which costs $N \times \log(N)$.
- And there are two lots of them.
- So the cost is $2N^2 \times \log(N)/P$ (assuming a square grid).
The communication costs are a bit harder:
- Each core has to send data to all the other core
- For simplicity let’s assume the each core sends the same amount of data to each of the other cores
- Each core starts with $N^2/P$ data, and so from the above we send $P$ messages each of length $N^2/P^2$
- So the comms cost is $N^2/P$
- Great, it’s decreasing with $P$!
The ratio is \((2N^2 \times \log(N)/P)/(N^2/P) = 2 \times \log(N)\)
- EXCELLENT
- Big N, better ratio
- Independent of P
- Expect this to scale really well
- Perfectly parallelized the message passing
  - Doubling P halves the message passing time
3D FFT 128x128x128

![Graph showing speedup vs. number of processors for different FFT algorithms. Ideal Speedup, PESSL Forward, PESSL Backward, FFTW Forward (Estimated Plan), FFTW Backward (Estimated Plan), FFTW Forward (Measured Plan), FFTW Backward (Measured Plan).]
What have we Forgotten

- Oh Dear! What’s gone wrong
- We’ve forgotten the latency!
  - We’re assuming the time to send N bytes is simply proportional to the message length which for short messages is not a good approximation
  - In other words we’ve only considered the bandwidth term, if the messages get very short we need also to include the latency.
- Here the messages are of length $N^2/P^2$ and so they get very short, very quickly as P increases
- So let’s include the latency term in our model
So The Problem With The FFT is

- Time for comms in the FFT: $P(\alpha+\beta N^2/P^2)$
- So messages rapidly become very short as $P$ increases
  - And lots of them!
- So our assumption about the times for the comms being bandwidth dominated is just wrong!
- Better time for comms at large core count: $P$
- As only the latency term is important
- Ratio: $(2N^2 \cdot \log(N)/P)/(P)=2*N^2 \cdot \log(N)/P^2$
- WHOOPS!
  - Anti-parallelized the communications!!
  - Doubling $P$ doubles the comms time!
So What Can We Learn From This?

- Design your algorithm to keep the compute time high relative to the communicate time as the number of processors increases.

- To do this the bandwidth term in your comms time must decrease with increasing P by sending less data at higher values of P
  - As otherwise the decreasing compute time will eventually become comparable to it
  - And you can’t decrease the time due to the latency as that is an unbreakable, fundamental lower time limit for sending a message
Try To Keep Your Messages Long

- Try to keep your messages as long as possible
  - Avoid getting into the latency dominated regime, if you are your scaling will often be very poor and there is little that can be done about it
    - Can “parallelise” bandwidth limited, can’t for latency dominated

- If you have multiple messages to send at one time consider packing them all into one message to try to avoid getting caught by latency
  - But will probably need extra memory copies
  - Try to arrange your data to avoid this!
- Related to the above is if one core has to talk to all the others
  - E.g. A reduction or a synchronisation
- In this case your communication time will increase with core count due to using these “collectives” – the Log(P) term
- But when we study OpenMP and MPI we will find they both provide special facilities for collective communications
  - USE THESE if you need to perform a collective communication
    - Very often highly optimised for the architecture
    - The make the best of a bad job
- Avoid superfluous synchronisations aka barriers!!
But keep a sense of perspective!
The very occasional collective won’t be a problem
E.g. An iterative algorithm has to check that all processors think the algorithm has converged
- Needs a global communication
However the cost of each iteration is much, much, much more expensive than the convergence check, so can forget the latter
And the scaling of each iteration will probably be such that any degradation due to the collective just won’t be noticeable
The story so far can be viewed in a more general way

This is that things tend to parallelise well if the “interaction” is “short ranged”

This first well into domain decomposition

And if a domain only interacts with neighbouring domains, as in the Poisson equation case, the number of cores which another core needs to communicate with is fixed, however many cores in the job

- 2 in 1D, 8 in 2D, 26 in 3D

This naturally leads to comms times **not** increasing with P

- And usually decreasing

So THINK if you can map your problem onto such a solution
Another common trick is *hierarchical parallelism*. This is to exploit more than one level of parallelism. For instance, part of CRYSTAL (an *ab initio* electronic structure code) schematically is

```
Do spin = 1, 2
   Do k = 1, nk
      \[ H_{k\sigma} = \mathcal{Q}_{k\sigma}^T H_{r\sigma} \mathcal{Q}_{k\sigma} \]
      Diagonalise \( H_{k\sigma} \)
   End Do
End Do
```

Other codes like VASP and CASTEP have similar loops with FFTs inside.
Do spin = 1, 2
  Do k = 1, nk
    \( H_{k\sigma} = Q_{k\sigma}^T H_{r\sigma} Q_{k\sigma} \)
    Diagonalise \( H_{k\sigma} \)
  End Do
End Do

- The obvious to parallelise this is to act on each matrix in turn, i.e. \( H_{11} \) then \( H_{21} \), the \( H_{31} \) etc., and parallelise the operations required to perform the matrix multiplies and diagonalisations
- Problem is matrix diagonalisation scales poorly with core count
  - We saw this earlier – the standard and divide and conquer graps are diagonalisation routines
Hierarchical Way

\[
\begin{align*}
\text{Do } & \text{ spin } = 1, 2 \\
& \text{Do } k = 1, nk \\
& \quad H_{k\sigma} = Q_{k\sigma}^T H_{r\sigma} Q_{k\sigma} \\
& \quad \text{Diagonalise } H_{k\sigma} \\
& \text{End Do} \\
& \text{End Do}
\end{align*}
\]

- However each spin and \( k \) are independent, and so therefore each of the diagonalisations
- We could do all the diags at once in parallel
  - With each diag itself in parallel on a subset of the cores
- Which is much better is diags only scale to a few cores
- So it is much better to do 10 diags each on 10 cores at the same time rather than 10 diags on 100 cores in turn
Hierarchical Parallelism

- So the parallelism forms a hierarchy
  - Parallelise over spin
    - Within each spin parallelise over k
      - Within each spin and k perform a parallel diag
- So my diagonaliser only needs to scale well to \( \frac{P}{2n_k} \) processors, not all the way to \( P \)!
  - Good! Diags tend to scale poorly
  - Exploit the perfect parallelism over spin and k
- MPI communicators are great for this – see later
- Be aware of load balancing though - what if while independent not all the diags take the same time?
Hierarchical Parallelism In Practice

The graph illustrates the comparison between Hierarchical Parallelism and Simple Parallelism in terms of Diagonalizations/minute against the Number of Cores.

- **Hierarchical Parallelism** shows a steep increase with the number of cores, indicating improved performance as more cores are used.
- **Simple Parallelism** remains relatively flat, suggesting a limited increase in performance with additional cores.

This data highlights the benefits of using a hierarchical approach to parallelism, especially in high-performance computing scenarios.
I’ve given a chemistry example but this is a very common techniques and an important part of your weaponry

Other examples include

- When you have many geometries: e.g. in Monte Carlo methods
- When you have many different, loosely coupled initial conditions: e.g. ensemble Weather forecasting
- When you want to apply the same operations simultaneously across different parts of the same object: Many linear algebra operations if correctly formulated can take advantage of this

- Make sure you exploit all the parallelism at all levels that your problem allows!
- A real program will contain many stages
- All the important stages must be parallelised otherwise Amdahl’s law will kill the performance
- OpenMP is not too bad
  - Incremental parallelism is easy
- MPI is harder
  - To get the best you must plan the *whole code* from the beginning and distribute objects appropriately
  - You want to avoid big data redistributions between stages as this can kill the performance
  - Thus you may have to comprise certain stages to get the best overall performance
CASTEP is an electronic structure code

DL_POLY_4 is a classical molecular dynamics code

They both use 3D FFTs, but parallelise them in totally different ways – why?

Because the important part of the code is different in each case

CASTEP is absolutely dominated by FFTs

- So the distribution chosen should focus on how to get the best out of the FFT

In DL_POLY_4 the FFT is much less important

- Other concerns drive how the data should be distributed
We have said absolutely nothing about exploiting the multi-core based architectures that we use

- i.e. Nothing about *process placement*
- e.g. In CRYSTAL we might want all the cores doing each diagonalisation to be on the same node
  - Can’t do this portably in MPI (yet! – but facilities in new standard)

However you might consider using *mixed-mode* programming

- i.e. OpenMP within the nodes, MPI between
- Number of potential benefits, but also pitfalls

We’ll look at this in a later module
An effective parallelisation strategy REQUIRES your communication time to at least stay essentially constant and at best decrease with the number of cores you use.

- The key in MPI to doing this is keeping your messages long so you are in the bandwidth regime.
- In OpenMP it is synchronisations which are your enemy.
- In MPI to get the best you must distribute your large objects.
  - And the way you choose to distribute your objects is the key to getting.
- And do this in such a way as to avoid large serial sections and to avoid load imbalance.
- Easy! But it can be done...
Any Questions

- Support
  - support@hpcwales.co.uk

- Web Site
  - www.hpcwales.co.uk

- Customer enquiries
  - info@hpcwales.co.uk
Reuse of this Material

- Materials developed by NAG Ltd
  - Some material © The Numerical Algorithms Group Ltd 2012-2015

- Licenced to HPC Wales, and its partner Universities (Aberystwyth, Bangor, Cardiff, Swansea, South Wales and Wales) for non-commercial purposes.