Software Design and Development
Languages and Parallelism

N.M. Maclaren
nmm1@cam.ac.uk

October 2019

4.1 Introduction

5.1.1 Summary

Many issues are specific to particular programming languages (or classes of language) or to models of parallelism, and this lecture includes a rough overview of the main ones. Some of these are not taught in this MPhil, and included here for background, because you may need to use them in your later career. There is one **critical** rule to follow when selecting a language and parallelism model to use:

- Agree your choices together with your supervisor or Director of Studies, or both.

Your chosen project may have constraints, such as needing to start from a particular program, and there may also be restrictions imposed by the examiners.

5.1.2 Other Courses

There are a lot of good courses and other references on the Web, but even more bad ones. Some of the University Information Service's courses may be useful to you, too.

By far the most common failing of most courses, Web references and even books is to teach enough to get the reader into trouble but not enough to know how to avoid it or get out. You may find some of my old courses useful, because they concentrate on the latter aspects, and because some cover topics where there are few or no other courses on the Web. They are all in:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/`

The ones you are most relevant to you are `Fortran`, `C++`, `MPI`, `OpenMP`, but several others may be useful.

4.2 Relevant Languages

The two main relevant languages are C++ and Fortran, with the former being the one you will probably use for the MPhil. In both cases, you are **strongly** advised to use a recent, but not too recent, version of the international (ISO) standard language, such as C++03 or C++11 or Fortran 2003 or 2008. That will get you all the functionality you need and minimise the problems you have with compilers.

Both the C++ and Fortran standards have introduced their own parallelism, and most compilers now support them, though you sometimes need a specific version. C++ has included a very low-level form of shared-memory threading, and Fortran has included

coarrays – a PGAS (Partitioned Global Array Storage) model – Intel already supports it, in its cluster toolkit, and a version of gfortran does, too. You may well hear about UPC (Unified Parallel C), but are recommended **not** to use it, for complicated reasons.

5.2.1 C++

• C++ is very flexible, but has very poor checking; errors are easy to make and extremely hard to locate. Also compilers cannot optimise it very much, so you may have to do some of that yourself.

However, it is dominant in many areas, and looks as if it will remain so for a while. Unfortunately, I do not know of a good, introductory C++ course, other than the one given in this MPhil, mainly because of the following problems:

• C++ is a huge and complicated language, with no simple, generally useful, subset.

• Books etc. rarely cover scientific computing requirements, and some things (like the handling of multi-dimensional arrays) are very tricky.

• C (and hence C++) has lots of subtle '*gotchas*' (i.e. complications that look as if they do one thing but actually do another, sometimes only occasionally), which are usually glossed over, but often cause trouble in real programs. The signed/unsigned minefield mentioned earlier is just one example of many.

I used c. 100 programming languages before C++, and I was astounded at its complications and *gotchas*, but they do not make it correspondingly powerful, despite widespread claims that they do. To become a competent programmer takes about 5 times as much effort as doing so for Fortran.

You are advised to learn more advanced C++ from a book, and the following look fairly good:

Stroustrup, B. (2008). Programming: principles and practice using C++.

This is 1100 pages, and is considerably more relevant and rather more thorough, but is definitely not easy going. It should turn you into a competent but not expert C++ programmer, and estimates that using it to learn C++ from scratch will take about 14 weeks at 15 hours per week! I taught a course using it as a basis:

Programming in Modern C++
`https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++/`

Eckel, Bruce (2000, 2003). Thinking in C++, 2nd ed.
`http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`

There are two volumes: *Introduction to Standard C++* (800 pages) and *Practical Programming* (500 pages). Note that it still has the restrictions mentioned above, though not as badly as most books on C++.

5.2.2 C++ and Parallelism

Above all don't try to be clever, to avoid being bitten by the complications mentioned above – KISS is the critical rule. The main other problem is the compiler generating

implicit calls to copy constructors and assignment. Do not underestimate how complicated and full of *gotchas* this aspect of C++ is.

Most simple uses of MPI are no problem, but see the lectures *More on Point-to-Point*, *Miscellaneous Guidelines* and (if used) *One-sided Communication* in:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/`

The ones you are most relevant to you are `Fortran`, `C++`, `MPI`, `OpenMP`, but several others may be useful.

4.3 Relevant Languages

The two main relevant languages are C++ and Fortran, with the former being the one you will probably use for the MPhil. In both cases, you are **strongly** advised to use a recent, but not too recent, version of the international (ISO) standard language, such as C++03 or C++11 or Fortran 2003 or 2008. That will get you all the functionality you need and minimise the problems you have with compilers.

Both the C++ and Fortran standards have introduced their own parallelism, and most compilers now support them, though you sometimes need a specific version. C++ has included a very low-level form of shared-memory threading, and Fortran has included coarrays – a PGAS (Partitioned Global Array Storage) model – Intel already supports it, in its cluster toolkit, and a version of gfortran does, too. You may well hear about UPC (Unified Parallel C), but are recommended **not** to use it, for complicated reasons.

5.3.1 C++

• C++ is very flexible, but has very poor checking; errors are easy to make and extremely hard to locate. Also compilers cannot optimise it very much, so you may have to do some of that yourself.

However, it is dominant in many areas, and looks as if it will remain so for a while. Unfortunately, I do not know of a good, introductory C++ course, other than the one given in this MPhil, mainly because of the following problems:

• C++ is a huge and complicated language, with no simple, generally useful, subset.

• Books etc. rarely cover scientific computing requirements, and some things (like the handling of multi-dimensional arrays) are very tricky.

• C (and hence C++) has lots of subtle '*gotchas*' (i.e. complications that look as if they do one thing but actually do another, sometimes only occasionally), which are usually glossed over, but often cause trouble in real programs. The signed/unsigned minefield mentioned earlier is just one example of many.

I used c. 100 programming languages before C++, and I was astounded at its complications and *gotchas*, but they do not make it correspondingly powerful, despite widespread claims that they do. To become a competent programmer takes about 5 times as much effort as doing so for Fortran.

You are advised to learn more advanced C++ from a book, and the following look fairly good:

Stroustrup, B. (2008). Programming: principles and practice using C++.

This is 1100 pages, and is considerably more relevant and rather more thorough, but is definitely not easy going. It should turn you into a competent but not expert C++ programmer, and estimates that using it to learn C++ from scratch will take about 14 weeks at 15 hours per week! I taught a course using it as a basis:

Programming in Modern C++

Many people use threading in C++, but it is very hard indeed to use correctly, and I do not generally advise it. The worst problem is that the container library is not well-defined for such use; this problem appliess to OpenMP, all forms of threading and all forms of asynchronism, including those 'defined' in the ISO standards. There are some safe but restrictive *empirical* rules that will work under most circumstances. For some guidelines, see the lecture *Critical Guidelines* in:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/OpenMP/paper_7.pdf`

the same rules apply to all forms of threading.

Beyond that, it is actually **safer** to write your own classes, unless you can find a suitable class library that is guaranteed to be thread safe by someone that knows the C++ standard and parallelism very well indeed.

5.3.2 Fortran

• With Fortran, you are advised to use the **modern** language; Fortran 95 is much more powerful than Fortran 77, and has much better checking and optimisability.

All valid Fortran 77 is also valid Fortran 95 code, so using existing programs and libraries is not a problem. The following course may be relevant, and the first lecture gives several recommended books:

Introduction to Modern Fortran

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/Fortran/`

Either following that or using one of the references should turn you into a competent but not expert modern Fortran programmer for about 40 hours' work (20% of that for C++!) You should allow another 5–10 hours if you need to be able to work with older code. There are a great many books on Fortran 77, but you are not advised to use that, both because the modern language is better and because many of the Fortran 77 books are very bad indeed!

Using parallelism and asynchronism in Fortran is generally not a problem, except for one aspect (where it is still easier than C++). Fortran also optimises more easily (and often more effectively) for OpenMP, SSE, VMX, Altivec etc.; it has been the language of choice for SIMD parallelism for over 30 years!

Fortran allows compilers to copy data and sometimes requires it, even when the user does not specify it, essentially like C++ copy constructors etc., but more cleanly. Fortran 2003 allows you to stop this by putting the ASYNCHRONOUS attribute on the relevant arguments; MPI 3 uses that correctly, but unfortunately is not yet generally available. There are several ways round the problem, but the simplest is to use the ASYNCHRONOUS attribute when you need to.

5.3.3 Parallel and Auxiliary Languages

There are several parallel languages that extend existing serial languages, and are important in scientific computing. The main GPU interfaces are both like that: CUDA, OpenCL abd OpenAcc. There are also shared-memory languages, of which the leader is OpenMP, which has C, C++ and Fortran forms. There are also dozens of specialist parallel languages around, of which few have had any impact outside computer science (and often not much in that).

C is a high-level assembler, and you should treat it as such – it is not recommended for writing applications in, but you will probably need to use it for interfaces, including system calls. It can be called from both C++ and Fortran. You are **very** strongly advised to use either the 1990 or 1999 version of the ISO standard language (C90 or C99), because the IT industry has essentially rejected the later ones.

Matlab (and its GNU equivalent, octave) and Python/numpy are very useful for quick test codes, and you can also use them to write prototype programs. You should use Mathematica or Perl only if you know them well, because they are harder to use equivalents of Matlab and Python. And hundreds of other languages exist! For a comparison of some of the ones mentioned, see:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/WhyFortran/`

Python is also a very useful scripting language, including for converting data from one form to another. You are strongly advised to learn it unless you already know Perl well.

5.3.4 Mixed-Language Applications

Applications that use more than one language can be written, but there are only two relatively easy cases.

- Calling C and simple C libraries, especially from C++ or Fortran.

Almost every language can do that, in some way, and it is the way that most languages access the system facilities and implement libraries like CUDA and MPI.

- Calling Fortran from C or C++, using the Fortran 77 subset for at least the interface.

You can still use a Fortran 95 compiler for this, and (with some provisos) the code can be in Fortran 95, but the interface cannot use any of the new facilities. An example of this is LAPACK.

Using both C++ and Fortran 95 in one program can be tricky, because their memory management and a few other facilities can interfere with one another. However, if you avoid using Fortran's memory management in advanced ways (which is usually possible), it is usually feasible; C++'s memory managment is generally not avoidable, and rather more delicate. There is some very dated information in:

Mixed Language Linking
`https://www-internal.lsc.phy.cam.ac.uk/nmm1/MixedLang/`

Most of that is still true, but quite a lot has been superseded, because there are now better ways to do the same task. If you do need to mix C++ and Fortran, a better approach is to use separate processes; you can still build them into a single application,

but the process interface is much cleaner in most modern systems. This is far beyond the scope of this course, so this information is for background only; there is some more information in:

Building Applications out of Several Programs
`https://www-internal.lsc.phy.cam.ac.uk/nmm1/MultiApplics/`

Processes can still share memory on SMP systems, by using POSIX `mmap` or some form of `shmem`, but remember that explicit synchronisation is needed.

5.3.5 Relevant Libraries

MPI is a library-based interface; OpenMPI and MPICH are the two open-source versions, and Intel and most HPC vendors have their own.

NAG is the best general, portable numerical library, and is the most reliable source of numerical algorithms. LAPACK is open source linear algebra code of very high quality. FFTW is open source fast Fourier transforms. MKL and ACML are Intel's and AMD's mathematical libraries, and will deliver the best performance. And there are a great many more, both proprietary and open source.

- Do **NOT** trust *Numerical Recipes* or the Web.

The former is like the Curate's Egg†; an expert can tell which parts are good and which are bad, but an expert will not use it, anyway. The Web is not called the *Web of a Million Lies* for nothing, and that is an underestimate; there is some extremely good information on it – and considerably more that looks accurate but is seriously wrong. The site `http://www.netlib.org` is often reliable, but not always; it is quite a good place to look, but take care.

A few libraries should **not** be included on other grounds, though you are very unlikely to want to use most of them. They are mainly ones that use system facilities in fancy ways, and they may be incompatible with MPI and OpenMP, at least, and possibly GPUs. A specific warning is to avoid anything using the X Windowing System or other GUI interface in a parallel program, because the event handling may well interfere badly. If you need to do that, a much better approach is to use separate processes, just as when mixing C++ and Fortran 95 and described in the course *Building Applications out of Several Programs*.

Algorithm references are a bit of a problem. Data management and related ones are well covered in computer science books, such as:

Cormen, T.H. et al. Introduction to Algorithms
Knuth,D.E. The Art Of Computer Programming

There are a great many other good computer science ones, including by *Sedgewick*, *Ralston*, *Aho et al.* etc., but most reasonably simple good, general numerical ones are very old. The best approach is often to use the NAG library as a reference:

`http://www.nag.co.uk/numeric/FL/FLdocumentation.asp`

† An old joke from *Punch*; look up "*Curate's Egg*" in Wikipedia.

For specialist algorithms, there is little option but to look for an expert in that particular field – note that I am referring to the *mathematical* field of the algorithm, rather than the *scientific* field of the problem. The latter is often a good start.

4.4 Survey of Parallelism

5.4.1 Why Use Parallelism

Moore's Law is that the chip size (strictly, number of transistors per square millimetre) goes up at 40% *per annum*; this is often misquoted in a form that I call *Not-Moore's Law*, which is is that CPU clock rates do, too. Moore's Law is slowing down but has not stopped yet, but Not-Moore's held only until ≈2003 and then broke down. Clock rates now are the same speed as they were then, and there is little prospect of any change. The reason is the power consumption (i.e. watts) due to leakage, which is beyond this course, but see:

```
http://www.spectrum.ieee.org/apr08/6106
```

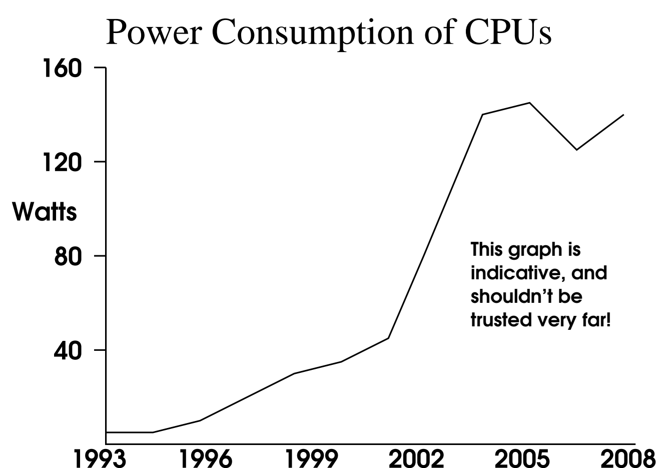

Figure 4.1

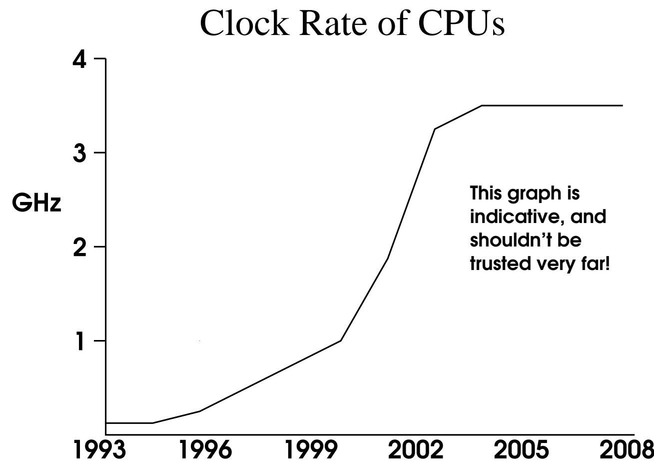

Figure 4.2

The manufacturers' solution has been to use Moore's Law to increase the number of cores, so the total performance still increases at nearly 40% *per annum*, though even that is now slowing down. Specialist CPUs already have lots of cores, and are used in areas like High Performance Computing, video processing and telecommunications, but are currently irrelevant to "general" computing. However, we are interested in HPC, and the following is an indication of how many cores per chip were planned:

2009	typically 4 cores
2014	typically 16–32 cores
2019	typically 128 cores

The figures have slipped a bit, mainly for the 'commodity' CPUs, but a typical high-end desktop now has 16 cores and more in its GPU.

5.4.2 Types of Parallelism

There are hundreds of parallelism models, some purely theoretical, but most with some practical use, but only a few are relevant to this MPhil. The main ones are:

- Message passing, which is currently mainly MPI.

This is the main form for distributed memory (i.e. clusters), but also works well on multi-core, shared-memory systems. The model is that each process executes serially and semi-independently, with communication using I/O-like mechanisms. There is another course on this.

- Small vector units, currently mainly SSE.

These will be described in a moment. Pure vector supercomputers (i.e. ones with large vector units) are essentially dead.

- Attached SIMD units, currently mainly GPUs.

SIMD is *Single Instruction, Multiple Data.* There is another course on GPUs.

- Shared memory threading, currently mainly OpenMP.

This is effective only on multi-core systems, but those are starting to become widespread. The latest version of the C++ standard supports a form of this, and it also includes POSIX, Microsoft and Java threads. CilkPlus also belongs here, as do several others.

- PGAS (Partitioned Global Array Storage).

In many ways, such designs are intermediate between message passing and and shared memory threading. The latest version of the Fortran standard has coarrays; UPC (Unified Parallel C) is very trendy, but not advised.

I will start with some important special cases. It is important to note that this is **not** the only way to use them, but it is the simplest way to design and debug programs using them. It is probably the way that most people will code their programs, but there are many other approaches. I will then go onto more general parallel models.

Small vector units (SSE, AVX, Altivec etc.) are used as part of serial optimisation; they are tricky to use efficiently in parallel with either MPI or GPUs, though their use can be alternated with few problems. You need a suitable compiler and a high level of optimisation; typically one of Intel's compilers and `-O3`. You need to make your inner loops vectorisable (as for OpenMP), and you can check that using the compiler messages. That is more-or-less all you need to know for simple use; for advanced tuning, check the actual times and possibly use the (hardware) performance registers.

An increasing number of people are using toolkits, usually libraries but sometimes preprocessors. Almost all are field- or model-specific, and they vary from good to utterly ghastly, as you would expect. Most require shared-memory, but some based on MPI. If a good one matches your requirement, especially if it is used by people you will be working with, then it saves a great deal of effort to use it. Such toolkits have not been investigated and are not covered in this course.

5.4.3 Parallelism References

It is very hard to give useful general references, because they all depend on your precise requirements and constraints. This lecture was partly derived from another course that has more information about parallelism:

Parallel Programming: Options and Design
`https://www-internal.lsc.phy.cam.ac.uk/nmm1/Parallel/`

There are a lot of fairly good books around on parallelism, including a few of the computer science textbooks, but most describe a few approaches and give the impression that those are the only ones to consider. There is a very relevant quote by Rudyard Kipling from *In the Neolithic Age*:

> *There are nine and sixty ways of constructing tribal lays,*
> *And every single one of them is right!*

Note that this is frequently misquoted on the Web, and it is not safe to trust the Web on the choice of programming languages or parallelism, either.

This lecture is partly taken from a much longer course, which is dated but still relevant, and goes into much more detail:

https://www-internal.lsc.phy.cam.ac.uk/nmm1/Parallel
Shared-memory programmers (not just Java ones) should also look at
http://docs.oracle.com/javase/tutorial//essential/concurrency/
My guidelines (mainly for OpenMP but more general) are in:

https://www-internal.lsc.phy.cam.ac.uk/nmm1/OpenMP/paper_7.pdf

You are also strongly recommended to look at this link:
http://parlang.pbworks.com/f/programmability.pdf

Ignore the details, which are misleading out of context, but note its summaries. Its associated book has quite a good overview of options, and goes into details I do not (except for dataflow):

Eckel, Bruce (2000, 2003). Thinking in C++, 2nd ed.
http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html

There are two volumes: *Introduction to Standard C++* (800 pages) and *Practical Programming* (500 pages). Note that it still has the restrictions mentioned above, though not as badly as most books on C++.

5.4.4 Key Factors

Distributed-memory parallelism (i.e. over more than a single node) needs MPI, PGAS or one of a few specialist interfaces. You can also use MPI between nodes, and other ways (e.g. OpenMP) inside. This lecture is going to assume MPI, but not describe it, as you have another course on it.

Shared memory parallelism is easy to program, but hard to debug, but can add to serial programs, incrementally, and that is how my course teaches it. However, many people try it, fail to debug or tune it and use MPI instead, because MPI works very well on multiple cores within a single node.

All of MPI, PGAS and GPUs need the programmer to manage the data distribution, and the that must be designed into the program - it cannot easily be added later.

For information on GPUs, see the course *Scientific Programming with GPUs*. This course describes **only** how to mix GPUs with MPI, and that will also be covered in the GPU course. Note that, to some extent, this also applies to mixing SSE and MPI, as well as SSE and GPUs and, under **some** circumstances, OpenMP.

5.4.5 MPI, GPUs and SSE

- Encapsulate your GPU or SSE use in algorithms.

This is so that you can debug and tune those on their own, as much as possible; you should design and test their interfaces in usual way. For example, such algorithm could be a parallel transpose using MPI, or one step of a linear solver using a GPU; that is not a complete algorithm, but is a subsidiary one with a well-defined interface. You can then call these from an MPI process, so a MPI program can also use GPUs and SSE. You can

then design and test these algorithms' interfaces in usual way, and do not need to worry about interactions just yet.

- Use the MPI process as a controller of the GPUs etc.

The simplest approach is to use the MPI process as a controller of the GPUs (and SSE). You must not share GPUs between processes; this is not normally allowed but, even if it is, it is very hard to tune. It is easiest not to overlap MPI calls and other uses, though you can alternate MPI calls, GPU use and SSE use. You should consider separating by suitable barriers, to simplify debugging and tuning.

You could also use OpenMP or threading instead of MPI, on a single shared-memory system, but use it **only** as a controller of the program. Again, do not share GPUs between threads, and do not mix OpenMP or threading and MPI. The reasons for these recommendations are too complicated and messy to go into, but they include arcane details of the MPI progress model and system scheduling.

An example of the easiest design may be useful:

start:	Use MPI to initialise
	[Consider calling `MPI_Barrier`]
loop:	Use GPUs to do calculation
	Use SSE to do calculation
	Use GPUs to do calculation
	Use SSE to do calculation
	[Consider calling `MPI_Barrier`]
	Use MPI to synchronise data
	[Consider calling `MPI_Barrier`]
	Repeat from loop
stop:	Use MPI to finalise

You can add suitable barriers between the GPUs and SSE usages if it helps, though calling `MPI_Barrier` will not help. And, of course, every reasonable variation on that design is worth considering – obviously, few people will use MPI, GPUs and SSE! There is a little more on asynchronous use later.

5.4.6 Shared Memory Parallelism

Many people use one MPI process per core, which has the advantage that the same code runs on both multi-core systems and clusters.

Currently, almost the only alternative is OpenMP. Sometimes, using OpenMP is easy and efficient; at others, it is evil to debug and tune. There is only one simple use: calling an existing threaded library. Such libraries include the NAG SMP library, Intel's MKL and AMD's ACML. This use is most effective when the time is dominated by a few calculations, and some library already has a SMP solver for them; you can then just call it, and your problem is solved!

- You can call such code from MPI but, in that case, use one MPI process per system and leave it to the SMP library to use all of the cores.

This is a critical point, as using both multiple MPI processes per system and an SMP library is extremely advanced use, and needs both programmer and administrator skills. You can alternate calls to an SMP library with using GPUs, as described above for GPUs and SSE, at least when you have only one GPU per system. As with that, each type of use can be debugged and tuned separately.

Combining MPI and OpenMP is possible, but is definitely advanced use and is not generally recommended; as with using SMP libraries, you should use one MPI process per system.

You are **not** advised to use POSIX or Microsoft threads, but the reasons are considerably outside this course, and foul almost beyond belief. You are not advised to use C++11 threads, either, because they are not realistically usable by any normal person, at all safely.

4.5 Parallelism Models

This considers how you structure your application for parallelism, and it is largely independent of the parallel technology - e.g. you can do anything in either MPI or OpenMP. The model you use changes how you approach your problem, especially as regards its design and debugging. This lecture only summarises the main issues, and is merely intended to point you in the right direction – you will have to do further study to learn how to use the techniques effectively.

5.5.1 Farmable Problems

I shall describe farmable problems first, to get them out of the way. These are when the requirement is divided into independent tasks; it is fairly common, and very easy to parallelise. Common examples include:

- Parameter space searching.

This is finding the best choice of parameters, and includes many forms of global optimisation – any such problem where brute force is effectively the only solution. Brute force should be avoided whenever possible, but sometimes it is unavoidable.

- Monte-Carlo simulation.

Parallelism means that you get a bigger sample, faster, which improves the accuracy. When doing this, remember to change the random number sequence between each simulation! It is also a good idea to check for possible dependence between the sequences, but that topic is again outside this course.

The simplest approach to these is to code a task as a simple, serial program – you can then debug and test it as an ordinary, serial program, using an ordinary, serial debugger if you want to.

- Then wrap it up in a a parallel harness.

The harness simply runs the program N times, giving it a separate task to perform each time. Remember to keep the original serial form in case you hit a new problem. Sometimes, you need make no changes whatsoever to the serial program, and you usually need to make only a very few, localised changes. It is very easy to do.

- Parallelise using processes and not threads.

Using processes looks more complicated, but is actually easier, mainly because the problems are far better understood. You should normally use pipes or files for input and output, and most program changes will be to do this. The controller (or harness) creates the serial program's input and merges its output, and all of the code to handle parallelism is in the controller.

5.5.2 Basic Master-Worker Design

The parent application runs as controller, and manages several jobs in parallel. Each task gets a CPU from a pool when one becomes free. The controller's tasks are to:

- Create a suitable job to run the task
- Create suitable input
- Run the job and wait until it finishes
- Collect its output and store or analyse it

It may run multiple jobs at once and, after it has finished a job, may run further jobs, perhaps indefinitely. There are a great many ways in which this can be done, and many are often almost trivial. Examples of easy implementations include:

- Using a batch scheduler to run serial jobs

This is essentially just a general-purpose controller written and debugged by someone else. It is usually best to script the submission and collation of the results, to save effort, and it is generally the most flexible and easiest solution.

- Writing a controller using MPI.

This is partially covered in my MPI course and is probably the easiest practical use of MPI. It requires MPI to be installed, of course.

- Writing a simple controller in Python.

This is a little harder to do, but not very much, especially for a single multi-core system; on clusters, it requires being able to run commands on remote systems.

There are some common bad solutions that are quite strongly **not** recommended:

+ Writing a controller in Perl, C or C++.

This is a significantly harder task, and there is some details of why in the extra information in the course *Building Applications out of Several Programs*.

+ Writing a controller as a shell script.

Shell scripts are unsuitable for any non-trivial task, as it is almost impossible to implement any useful level of error handling.

+ Writing a controller using OpenMP or threads.

This is easy until something goes wrong, when chaos ensues. One thread can compromise others too easily, there is far too much changeable state per process, and there is no clean way to kill a stuck thread.

5.5.3 Obtaining Parallelism

In general you have to introduce parallelism into your program, by diving it up into tasks, and that needs communication between the tasks. The first rule is to use the most natural design (i.e. the one that fits the mathematics of your program most closely), and the second rule is to choose the one with least communication. These rules maximise debuggability and help tunability.

- Do **not** rush towards the coding!

Careful design is **essential** for success, more here than in almost any other aspect of programming. If you are uncertain where the time goes in your program, and how much interaction between tasks would be needed, consider running a prototype to get timing and communication data.

You also need to consider the consequences of *Amdahl's Law*. Let us assume that the program takes time T on one core, and spends a proportion P of its time in parallelisable code. Then the theoretical minimum time on N cores is:

$$T * (1 - P * (N - -1)/N)$$

This means that you cannot ever reduce the time below $T * (1 - P)$ and the gain drops off very rapidly above $1/(1 - P)$ cores. You should use this to decide how many cores are worth using, which in turn can affect whether to use multi-core systems or clusters – and, in some cases, whether the project is worthwhile at all. But you also need to note a practical warning, which someone quoted in this form on the net:

The difference between theory and practice
Is less in theory than it is in practice.

Amdahl's Law is a theoretical limit and, in practice, parallelism introduces inefficiency – especially if the parallelism is fine-grained, or there is frequent communication between threads.

- Allow at least a factor of 2 for overheads.

This means that you usually need a potential gain of 4 to be worth the effort of parallelising a program, and at least 8–16 if your program need to be redesigned to introduce parallelism (i.e. more than simply changing one algorithm for a more parallelisable one). You can save a lot of time by doing rough calculations first, and avoiding approaches that cannot be made to work.

5.5.4 Practical Parallelism

A very non-theoretical point is that parallelism for performance (usually called HPC – *High Performance Computing*) uses a SPMD model (*Single Program, Multiple Data*). Exactly the same program runs on all cores (or systems, on clusters) are allowed to have data-dependent logic, so each thread (or process) may execute different code.

The simplest is *master-worker*, which has already been covered, but it can be extended to *lock-free SPMD*, and still remains reasonably easy to debug. That is a very ill-defined term, but here is roughly what it means. Workers communicate only with the master, or by atomic access to global variables, which are often implemented by a call to the master, anyway! This includes using *reductions* in MPI, OpenMP and similar interfaces.

- The key is to avoid any execution-order dependencies (except as controlled by the master).

This includes logic where one worker has to wait for another to finish a task and, especially, workers never take out a *lock* on access to any data, which is why it is called lock-free. If you think about it, this is not very logical, as you can get exactly the same problems with sufficiently complicated communication with the master, but the point is that this centralises and encapsulates the problems.

- In practice, HPC implies gang scheduling.

This is when all cores operate together, in a semi-synchronised fashion; the details of how systems arrange this are very complicated and outside the scope of this course, but it is usually fairly easy provided that the system is not being used for any other purpose at the time you are running your program. There is no theoretical reason for this, but it is so today; whether it will be true in some years time is less clear.

- Do **not** try to use dynamic core counts.

This is when the number of cores (i.e. active threads or processes) varies during the execution of a single program. You may see facilities to support this, and possibly some recommendations to use it, but it is best called an open research problem in computer science – i.e. nobody is sure how to make it work, in general.

5.5.5 Asynchronism

- You can overlap communication and computation but, unfortunately, this is much more in theory than in practice.

The main reason is that synchronism at any level (and that includes in the operating system and even hardware) 'poisons' asynchronism and makes it effectively synchronous. This aspect is closely related to MPI's concept of *progress*, but that is too complicated to cover either here or in the MPI course; it is covered in extra information for the MPI course.

A specific point is that the network operates independently of the CPUs on the systems attached to it, but the TCP/IP protocol is synchronous and needs a CPU to handle data. Ethernet itself is similar, but is becoming less so (e.g. 100 Mbit 'Fast Ethernet' was entirely synchronous, but Gbit Ethernet is only partly so). InfiniBand (used on specialist HPC clusters) is much more asynchronous, but its drivers are often synchronous.

- However, modern CPUs are almost all multi-core, so we can reserve some cores for communication.

This is very often worthwhile, and it is common to get faster times by using only some of the cores on a multi-core system; try varying the number and seeing what happens. Also GPUs can usually execute independently of the CPU; if their program is using only their own memory, there is no problem.

- The memory controller is usually a bottleneck.

Most apparently CPU-bound codes are actually memory- bound, though this can be due to limited bandwidth, the finite latency or conflict between accesses – the situation is

too complicated to describe here. Many books and Web pages get this one wrong, and some of them describe what used to be the situation, but is not how modern systems are designed. This is simplest to show in figures:

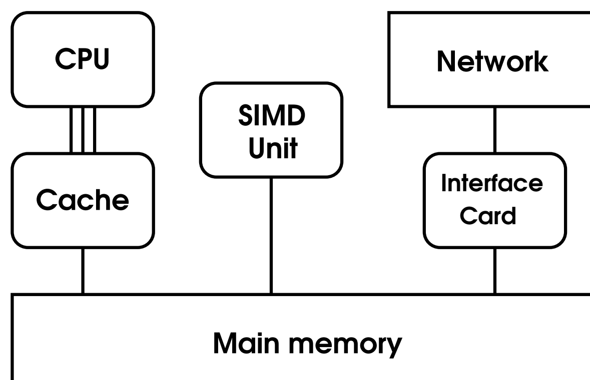

Figure 5.1

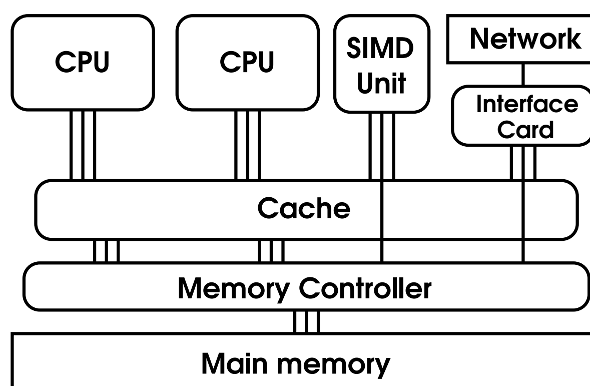

Figure 5.2

- Do **not** rush into coding asynchronous programs.

They can be a great deal harder to debug, though careful design is the key to success, as usual. The old mainframes were much more asynchronous than modern systems, and the relevant skills were more widespread, but it was regarded as difficult, even then.

- GPUs are the best bet for making this work.

This applies especially to GPUs and MPI communication, but do watch out, as the situation is complicated. Remember that the memory controller is a bottleneck, and all of transfer to and from the GPUs, the CPU and the network need it. Overlapping memory access often causes conflict, and problems with that can be very hard to track down.

5.5.6 HPC Models

Sometimes the problem has a natural model of parallelism, arising out of either the basic science or the algorithm used. If there is a suitable implementation of some parallel mechanism that provides that model, then use it. If not, you must map the natural model to another parallel model.

- This is too complicated an area for this course.

I will describe three of the most important HPC models, which are the only ones that I have seen used in production code; there are almost others in use, but these are probably the most common. There are some more details on this in the MPI course. But, as always, remember that careful design is **critical**.

5.5.7 The Vector/Matrix/SIMD Model

This is the basis of Matlab, Fortran 90 and so on, and is where the basic primitives are matrix operations like `mat1 = mat2 + mat3*mat4`. Typically, it assumes that the vectors and matrices are very large. This is very close to the mathematics of many areas, and often highly parallelisable – I have seen 99.5% in a real program.

- The main problem arises with access to memory.

Vector hardware had truly **massive** bandwidth compared with its CPU performance, and all locations were equally accessible. That is completely unlike the situation with modern cache-based, SMP CPUs.

- Main memory has affinity to a particular CPU.

Only local accesses are fast, and memory conflict is bad; this is why LAPACK and other high-quality libraries use blocking algorithms. Unfortunately, some vector codes run like drains even if they do use blocking algorithms.

- Regard tuning this model as **all** about memory access.

On a modern system, the CPU can be regarded as infinitely fast compared to access to main memory; a typical ratio between the time for a double precision multiplication and a main memory access is 1:100. To a great extent, the same applies to using MPI and to some extent GPUs, and the main cost is for the non-local accesses. The hardest part of the design is minimising those.

5.5.8 Problem Partioning

This is not really a specific parallel model, but more a class of such models. It is based around the idea of dividing the problem up into sections, and assigning each section to a thread. There are three main objectives:

- KISS – keep it simple and stupid.
- Equalise the CPU requirements for each thread, so that each runs for about the same length of time.
- Minimise communication between threads, especially when one thread has to wait for others before proceeding.

Sometimes, partioning is natural and easy, and may arise directly from the scientific problem. For example, separating by component in a motor, or by compound in a composite material, or by species in a ecological simulation. You may need to group tasks together if you have more tasks than threads, and you should use the objectives described above when doing that.

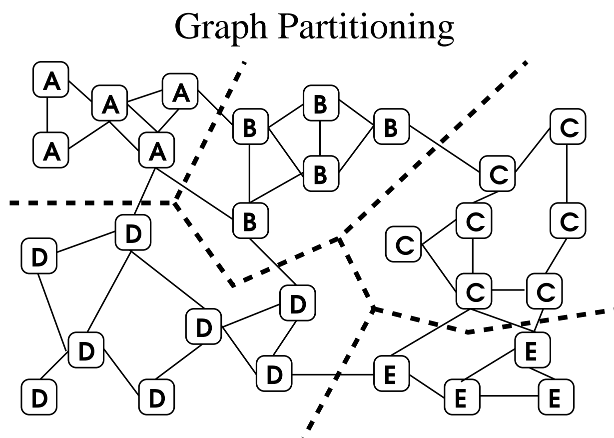

Figure 5.3

The partioning is very often done using spatial dimensions, and the simplest use is a rectangular grid. You can assign indices by blocks or cyclicly; the former is more common.

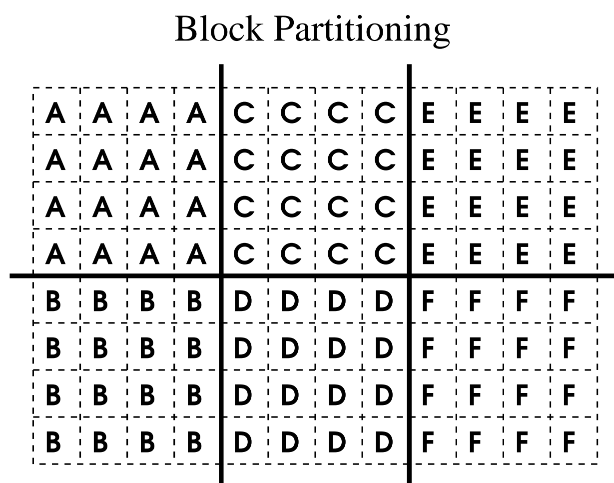

Figure 5.4

Unfortunately, it is common for some areas to take much longer than others (e.g. the flow of a fluid round a point), and in others the communication is often non-uniform.

- Irregular divisions are often more efficient, but they are more tedious and more error-prone to program. There are many different ways of doing this, including multi-grid methods, mesh refinement and transforming the coordinates of the grid corners.

Irregular Partitioning

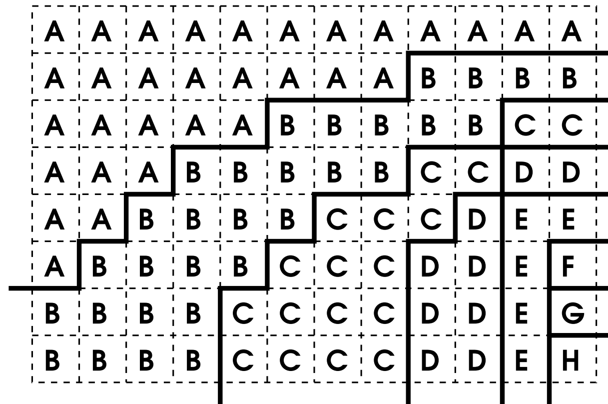

Figure 5.5

Mesh Refinement

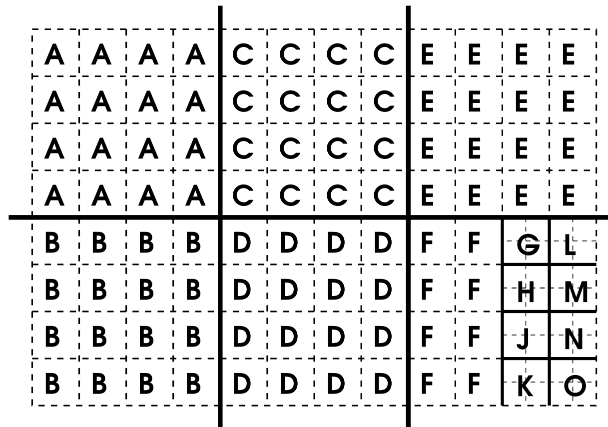

Figure 5.6

Transformed Mesh

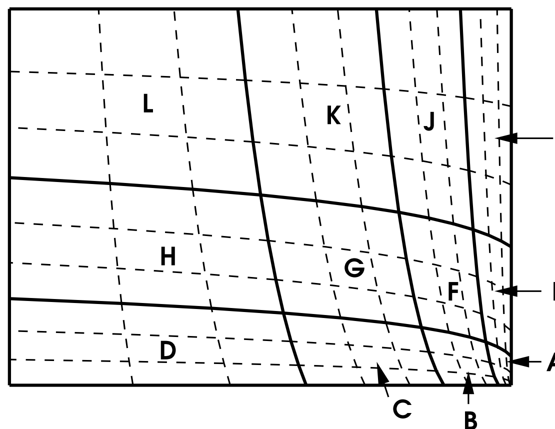

Figure 5.7

There are also many forms of cyclic partitioning, and different forms can be nested or combined in other ways, but the above are the most common. Partitioning can also be done using *Voronoi diagrams* (a.k.a. *Dirichlet tesselation*, a.k.a. *Delaunay triangulation*), which is mentioned in the MPI course and may be covered in the mesh generation one.

5.5.9 Dataflow Models

These can be useful for irregular problems, but it seems that some people find it easy to 'think dataflow' and others find it very hard.

- If you find it an unnatural way of thinking, do **not** use it unless you have to, as you will make too many errors.

The structure of the program is made up of actions on units of data, and defines how these depend on each other. The data are filtered through the actions – it is very like the object-oriented model in this respect. Actions run when all of their input is ready, and input can get stacked up several deep on one path, while the action waits for another. You can tag input with a 'transaction tag' if all of the different inputs to an action must match each other.

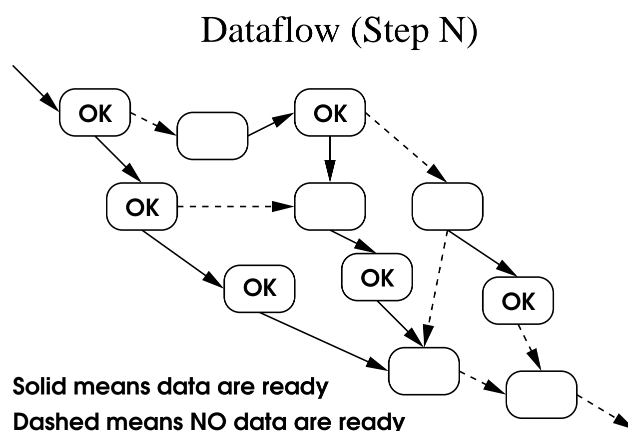

Figure 5.8

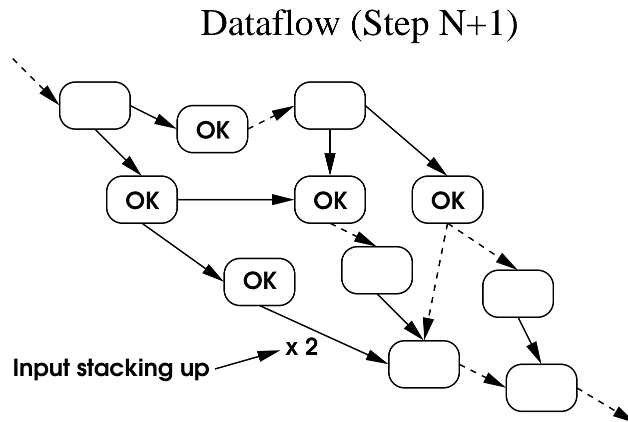

Figure 5.9

The usual implementation is that each 'data packet' is stored in some queue, and is associated with the action it is for – in the simple case being described here, each data packet is assigned to a unique action, though that action can then pass it onto another one.

- The program chooses the next action to run.

The way that it prioritises the data matters for efficiency, and most of the tuning of dataflow models is concerned with this, but it is separate from correct operation (because any execution order will eventually complete). This is a gross over-simplification, of course, but dataflow models are a complete subject in themselves.

- The approach can make the program's design a lot simpler.

The reason is that there is no communication except by an action passing data packets to other actions, and the only communication failures that can happen are deadlock and livelock – erroneous results cannot arise because two actions run simultaneously. This gives a much higher chance of successful debugging.

5.5.10 Designing for Distribution

A good rule of thumb is the following:

- Design for SIMD if it makes sense.
- Design for lock-free SPMD if possible.
- Design as independent, communicating processes otherwise.

That is for correctness – i.e. the order of increasing difficulty. It is **not** about performance, and **not** about shared versus distributed memory. If you think you can design your own structure, then you should be aware of the seminal paper on the topic: *Communicating Sequential Processes* by *C.A.R Hoare*; see:

`http://www.usingcsp.com/cspbook.pdf`

I am referencing that paper mainly to put you off from being too clever – you are **not** being recommended to read it – it is 260 pages long, and hard going even for mathematicians. But that fact may explain why I keep stressing the need to keep your design simple. Note that the order of increasing performance may be the converse to the order given above: *There Ain't No Such Thing As A Free Lunch*, especially in parallelism.

- The next stage is to design the data distribution.

SIMD is usually easy, because you just chop the data into sections, but other models are trickier. The best approach is very dependent on the details of your problem.

- Then you work out the need for communication.

This is which threads need which data and when. At this point, you can usually do a back of the envelope efficiency estimate; if this is too slow, then you need to redesign the data distribution, and this is often the stage where simple SIMD models are rejected.

- Do **not** skimp on this design process, because data distribution is the key to success.
- You may need to use new data structures (compared to the serial code) and, of course, different and more parallelisable algorithms.
- Above all, *KISS* – Keep It Simple and Stupid. Not doing that was the main failure of *ScaLAPACK*, and is why most people find it very hard to use and worse to debug.

Annex: C++ Notes

I taught a course using Bjarne Stroustrup's "Programming – Principles and Practice Using C++ – which needed 200+ hours' work. This annex is some points and additions I made on important practical issues that are rarely mentioned, some of which are more detail on points that he made. For the details, refer to **my** version of the course:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++`

Some of them are also in his original:

`http://www.stroustrup.com/Programming/`

.

C++ has a large number of open modes, but many of them have serious *gotchas*. The following are generally safe:

Open options	Purpose
`in`	for reading only
`out|trunc`	write to a file, replacing it
`out|app`	extend a file at the end
`in|out`	update an existing file, starting with a read
`in|out|ate`	update an existing file, starting by extending it
`in|out|trunc`	update or create a file, starting with a write

`ifstream` includes `in` and `ofstream` includes `out`, and all can be used with or without `binary`. The above is the **only** safe use of `app`.

Repositioning is also tricky, though rewinding is a lot safer, and you can seek only on ordinary disk files, not on anything 'special', sometimes including remote files located on a file server. You must **never** reposition or rewind a file if opened for `app`. The following uses are generally safe:

Action	Purpose
`seek(0)`	to reread or overwrite from the start
`seek(0,trunc)`	to clear the file and rewrite
`seek(0,ate)`	to extend the file at end

You **must** separate reading and writing by a `seek`. You can seek by byte count safely only if it is an ordinary disk file on a Unix-derived system, and it was opened with `binary`.

Some files are **not** just arrays of bytes. Some may be structured but, more often some can be opened only once – *be warned*! FIFOs (sockets, TTYs, etc.) are like that, as are many files on some non-Unix system. Simplex stream I/O is only reliable way of using them: that is input-only or output-only, with **no** repositioning. Remote files (e.g. ones on NFS etc.) also have restrictions. If they may be accessed in parallel, open them for input-only. Use a library like HDF (or MPI I/O) if you need to update them.

When C++ gets an I/O error, it just sets a flag bit and ignores further calls on that file until the program clears it; that is very dangerous, but better than C. But **never** use `clear()` on the `bad()` bit; you can set up a stream to throw on bad errors, which is better; see slide 21 of my course (slide 19 of his) `10_iostreams.odp`. C is even worse, continues regardless, and has no separation of recoverable and catastrophic errors. This leads to undefined behaviour and chaos.

Under Unix-derived systems (including Microsoft ones), system errors on output are rarely detected, and system errors on input often look like end of file. This area is **completely broken** in modern systems, and all you can do is to watch out and use files defensively.

Many people dislike C++'s formatted I/O facilities, for good reason. C's are easier to use and more flexible but unsafe. There are several alternative approaches described in:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++/11a_other_io.odp`

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++//24a_more_numerics.odp`

I am going to mention only a few points here, mainly ones that are relevant to other courses.

Those lectures say why I do not like the STL's design much, describes a lot of *gotchas* to avoid, and describes approaches that I regard as cleaner and simpler. Vastly the most useful standard containers are `<vector>` and `<list>`, followed by `<array>`, `<map>` and `<set>`. Do not bother with `<valarray>`, `<stack>` etc., which merely complicate your code for no benefit; `<algorithms>` is not useful, either – code them yourself. Generally, use `<vector>` unless need a fixed size, when you must use `<array>` – it is cleaner than built-in C++ arrays, but no more functional.

Watch out for shared-memory parallelism; the standard is hopelessly ill-defined, but this is the way it is interpreted. Separate container objects are independent, and information methods are read-only on container. Separate elements are independent if left in place (i.e. they may be updated but not replaced). In general, even element assignment may update the whole container,the rules for iterators are full of serious *gotchas*, and the data are not contiguous (i.e. unlike C). The exceptions are `<vector>`, `<deque>`, `<array>` and `<string>`, where you can also replace elements (but not append them, insert or erase

them; because the data **are** contiguous), you can create a C pointer to the data, and pass to MPI etc.

It is critical to use a pure data class (not a C++ term) when passing data to MPI, binary I/O etc.; this is slightly stronger than a standard-layout class, and not quite the same as as a POD. In simple terms, pure data classes must not contain any of:

Any reference or pointer, or
any container except `<array>`, or
any class except a pure data class, or
any virtual functions

There are also arcane restrictions on derived classes, and I suggest you avoid assuming anything about those. Some standard classes are likely to be pure data classes, but the area is too complicated to describe; the good news is that `<complex>`, `<tuple>` and `<bitset>` are. `<exception>` is definitely not, despite appearances. Almost none of this is actually specified.

Class layout is, in general, a can of worms. The alignment and padding may vary considerably, according to hardware, system, compiler and compiler options; check this carefully when reading in binary files. Be **very** careful when using any library class, whether C++ or external (e.g. Boost), because their exact properties (including whether they can be moved) are very rarely defined; e.g. what constraints are there on `<mutex>`?

C++ 2011 changed its base from C90 to C99. C90 required `errno` be set for errors in `math.h`, and C99 broke that. The 'replacement' IEEE 754 error handling is solid with *gotchas*, and is worse in C++. C++ 2011 included C99's library calls but not its pragmas, so using those library calls is **necessarily** undefined! Compilers, libraries and options will all differ on their interpretation of this whole hopeless mess, so you need to check yourself, as taught in the previous lecture.

For some information on precision and accuracy, look at exercises `1a`, `1b` and `1c` in:
`https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++/15_graphing.odp`
they show how to solve some common problems. There is a great deal more, including information on Kahan summation, precision extension etc., in:
`https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++/Exercises/Chapter_24`
My high-precision
accumulator code is there, and you are welcome to use it (e.g. on GPUs).

Almost all of the Web and **most** books are erroneous on random numbers. Simple rules include not to use `rand()` in serious code, because it is truly awful, Numerical Recipes and `Boost::random` are unreliable, and C++ 2011 supersedes the latter, anyway. In C++, only the `Ranlux` and `Mersenne` genererators are any good, though `Knuth_b` is tolerable for occasional use. On the Web, Marsaglia's generators are very variable in quality; I have a good, highly portable generator that people are welcome to.

A godd rule is to always recheck important results with different generators, because interactions of the generator withthe program can cause spurious effects, and use ones based on different mathematical principles for safety.

Parallelism is a major problem – you are welcome to ask me for advice – thread quasi-

independence is a **very tricky** problem. If initialising separate per thread or per process generators, you **must** use a very high-quality generator, with a very long-period, and randomise the creation of your initial seeds (preferably using another generator).

Many scientific libraries have suitable matrix classes, but I tried using the STL and Boost, and the code was longer, more complicated and much harder to debug than doing it myself. It is much easier to write your own, as described in:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++/24a_more_numerics.odp`

Exercises 15–18 help you to learn how. Also, note that Fortran storage order can be faster than Algol (C/C++) order, due to the common use of the right solution (rather than left solution) of equations. You can often make gains by storing both a matrix and its transpose, but be warned that writing an efficient transpose function needs care. There is example code (both Bjarne Stroustrup's and mine) in:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/C++/Exercises/Chapter_24`

You are welcome to use them, but please give us credit for them.

Note that Algol 68 and Fortran handle subsections properly - i.e. you can pass them as an argument to a function as a normal array. To do that, you must use a lower bound, size and stride for each dimension, but the example code above does **not** do that.