Introduction to OpenMP

Background and Principles

N.M. Maclaren

nmm1@cam.ac.uk

September 2019

1.1 Introduction

1.1.1 Summary

This course is about programming in OpenMP, the shared-memory interface.

CPUs got faster at ≈40% per annum until 2003, but since then have only got larger; they are running at about the same speed they were in 2003. Vendors are increasing the number of CPU cores per chip, initially at the same rate, but now more slowly. So the way to get more performance is to use more CPUs in parallel, and OpenMP is a tool for that on multi-core systems, using a shared memory processing (SMP) model. That is not new, of course, and OpenMP was derived from a several previous mechanisms.

OpenMP is a language extension, and not just a library, because it needs to change the semantics of the language so that the same code has a slightly different meaning. It was designed by a **closed** commercial consortium – "open" just means that they do not charge a fee to use the specification. They did and do accept public comments on the details, but that is about all. It dates from about 1997, and is still active.

The current specification is version 5.0, but the course is based mainly on version 2.5, for portability and sanity. Both have specifications for Fortran, C and C++. Most compilers have some OpenMP support, though the details vary, but everything taught here should be available and behave similarly in all current compilers.

1.1.2 Shared-Memory

Message passing interfaces (e.g. MPI) use multiple processes that run in parallel, and communicate via special I/O channels; their memory is entirely separate and local to where the process runs, which is why that is called *distributed-memory* processing. *Shared-memory* ones have a single process, with mutiple threads of execution that run in parallel, and all of the threads have access to all of the process's memory. This is simpler in some ways, but much more complex in others.

It is very hard to implement this efficiently and automatically on modern systems, for reasons that will be mentioned later, so there needs to be some explicit synchronisation between threads. Programs must follow strict rules to make that work, and OpenMP and this course are about those rules.

OpenMP is **not** generally advised for running separate tasks, and you should use MPI (Message Passing Interface) or a batch scheduler for that, or even or just run multiple

background processes. MPI dominates on clusters and other distributed memory systems, and runs equally well as separate processes on a single multi-core system. But distributing memory is very tricky, both for performance and correctness.

Shared memory means that you can avoid that, to some extent, which is where OpenMP comes in. It is almost always used for more performance of a single program, as in HPC (High Performance Computing), and the objective is genuine parallel execution of a logically serial program. Almost all shared memory HPC programming uses OpenMP, but an increasing amount is done with higher-level toolkits. Things may change in the next decade, because both Fortran and C++ have added parallelism into their new standards and there are several other plausible designs around.

The course teaches the approach taken by most people and libraries to OpenMP design, which is a formal policy of NAG SMP, MKL, ACML and so on. This is where you start with a well-structured serial program – in this context, that means that most of the time is spent in a small number of components, which have clean interfaces and spend their time in actual computation (rather than, say, I/O). You should not even attempt to convert the whole program at once, but do it component by component, where possible. This is the approach used in the examples, and there is more on this topic in the last lecture.

1.1.3 Apologia

The course is not what I would like to give, but that seems to be unavoidable. Some students have complained that it spends too much time telling them what **not** to do, and doesn't tell them **what** to do. Unfortunately, that is the **key** to success in shared-memory programming, not just OpenMP but POSIX and C++ threading as well. Shared memory programming is seriously tricky, and doing the actual coding is the easy part. The hard part is avoiding the '*gotchas*', and that is why you need to know what **not** to do.

The worst problem is data races causing subtly wrong answers; these often escape testing, and can be almost unfindable. Also, there do not seem to be any useful tools for OpenMP. This means that it is easy to 'learn' OpenMP, but not not know enough to use it successfully. That is a very common experience with all forms of shared-memory programming, including people who have been on other courses. Feedback from several users has been that this course has explained *why* they failed previously, and has enabled them to use OpenMP successfully.

The obvious question is why not teach something better, and the answer is unfortunately that there is nothing better at present, for general-purpose shared-memory programming for scientific computing.

More about this is explained in the last two lectures.

1.1.4 Objectives and Prerequisites

This course was originally written as a "transferrable skills" course – i.e. a course given to staff and graduate students, not as part of a degree. So it is practically oriented, and intended for research scientists; it also teaches just the minimum subset of OpenMP needed to write useful OpenMP scientific codes.

However, the course also teaches some of the computer science that underlies OpenMP, including why things are as they are, and why certain methodologies are good programming practice. It is important to understand the reasons behind its recommendations, and to take note of warnings. You cannot just follow the recipes in these notes.

At the end of this course, you will be able to understand the OpenMP usage of the simpler scientific research applications that use OpenMP, and be able to write fairly efficient, portable and reliable OpenMP programs of your own. Naturally, expertise develops with practice, but you will be familiar with enough of the potential problems to be able to avoid most of them.

It assumes that you are already a reasonably competent programmer in one of C, C++ or Fortran 90, and can use such techniques as adding diagnostic output statements for debugging. It also assumes that you can use Unix for program development. There are some transferrable skills courses on those, as well as on other related aspects, which are referred to in passing. However, they are **not** part of this MPhil, and you will not get credit for them. Some of the courses I used to give may be useful, too:

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

1.1.5 Further information

There is a fair amount of material that goes with this course, including practical examples with specimen answers, to teach the use of specific OpenMP facilities. You are strongly advised to work through most of those, as there is no substitute for actually using a facility to check that you have understood it. There are also some specimen programs. This material is available on:

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

Most books, Web pages and courses on OpenMP are very misleading, for reasons this course will explain; it is called The Web of a Million Lies for a reason – in fact, that is an underestimate! *Parallel Programming in OpenMP* by Chandra, Kohr, Menon, et al. (ISBN: 1558606718) is listed on the HECToR Web site and is fairly reasonable, but does not warn about about even quite serious and common problems.

The OpenMP specification is very often ambiguous, and is even inconsistent with itself or with the languages it extends. Also, each new major version has added lots of major new features. The result is that compilers vary a great deal in important details. Regrettably, this means that the specification is not suitable to use as a reference unless you are both an expert on your programming language and have some low-level implementation experience. However, the URL for the specifications is:

`http://openmp.org/wp/openmp-specifications/`

1.1.6 Course Coverage

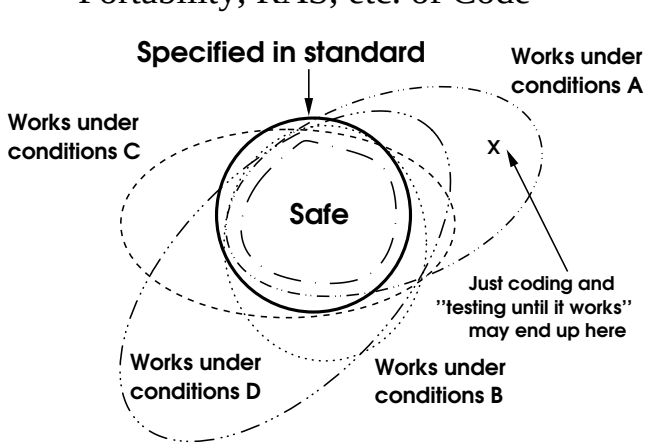

Figure 1.1

In addition to the specification problems, it is even harder to test a compiler than to test user code; the result of both of these is that the trickier features are likely to be unreliable. This course teaches a fairly safe subset; if these features do not work, more advanced ones are not likely to. It also teaches the simplest useful features, which are the most likely for scientific programmers to get to work in real code.

Shared memory programming is generally **not** about the syntax and coding (i.e. getting a program to compile, link and run simple tests), but is far more about knowing what to do and what not to do. This course describes some safe practices for the two most common programming paradigms that use OpenMP-like shared memory interfaces, that are most likely to be got to work in real code. It also includes a large number of warnings about potential problems to avoid and, if you follow its guidelines, you will avoid many of them.

Remember: a problem avoided is not **your** problem.

1.2 Using OpenMP

1.2.1 SIMD Computing

SIMD means Single Instruction, Multiple Data, and is a generalisation of the old vector computing model; it is simplest to think about operations on whole arrays at once, as in modern Fortran. Vector hardware is effectively defunct, though Fujitsu, NEC and Cray still make it, and modern SIMD is handled entirely by the compiler. Fortran array operations should work this way but the compilers often do not optimise them very well.

SSE/MMX, VMX/AltiVec and so on are SIMD features; OpenMP enables multiple cores to be used similarly, though it is not its only use.

GPUs are also SIMD, but are currently handled in a very different way. You can use NVIDIA or other cards for extreme performance, and the current language extensions to do that are *CUDA* and *OpenCL*. Programs for those need a similar design to OpenMP

SIMD, though the actual code is completely different. Most of this lecture applies to them as well, though the rest of the course does not.

A good optimising compiler does **all** of that for you (except for the GPUs), and a few may even autoparallelise your code, so even apparently serial programs can run in parallel. The compiler handles the synchronisation between cores, and covers up problems in the underlying implementation (such as ambiguities in the threading memory model). In practice, this implies *gang scheduling*, which is all cores operating together in a semi-synchronised fashion. We shall cover some of these issues in more detail later.

You may well ask, if it is all automatic, why bother using OpenMP? Unfortunately, only the simplest cases are automatic and then often only for Fortran. Also, you very often need to write code the compiler will not parallelise.

1.2.2 SPMD Computing

SPMD means Single Program, Multiple Data, where each 'thread' can operate semi-independently; for example, each of them may call a different function. This is much more flexible, but much harder to get right, and this course will cover only the very simplest forms of it. You are strongly advised to be cautious when using SPMD; if you attempt to be 'clever' with OpenMP SPMD, you **will** shoot yourself in the foot. Most books and Web pages do not teach that, unfortunately.

Unlike for SIMD, there is no major advantage in using Fortran, and we will cover SPMD after the simpler SIMD.

You can even add inter-thread communication to SPMD, which makes it almost like separate, communicating processes, but you are **strongly** advised to avoid doing that; I will explain why when we come to it. If you really want to do this, the classic paper is C.A.R. Hoare's *Communicating Sequential Processes*, but it is hard going even for a pure mathematician. You should also look at the memory model in the **new** C++ standard, which is not easy going, either.

The simplest and easiest way of parallelising a serial program is the following:

- Set the compiler options for full optimisation; if it is available (e.g. with Intel compilers) select autoparallelisation.
- Do some fairly high-level profiling of the code on some representative data, and consider just the areas that take the most time. You should add some timing code around the interesting areas to measure how changes improve the performance (or do not).
- Try adding calls to parallel library functions, such as using the LAPACK ones in MKL or ACML. Doing this is a good idea even in serial code, because those are often well optimised.
- Make sure that you link with a parallel version of the library, set appropriate environment variables and use a multi-core system. If the results are good enough, then you have done all you need to, and can go and do something else.
- If not, add some SIMD directives where they make sense and are likely to help. Many compilers, such as Intel's, will tell you if they have parallelised loops and so on. Now look at the performance improvement, if any.

- If that is not enough, change your code to improve its parallelisability or use SPMD directives. Only if you have to go further than that, worry about more advanced parallelism.

Is that too good to be true? Yes, a bit, but it is worth trying; an ancient engineering principle is that the simplest solution that does the job is the best solution.

1.2.3 Basic OpenMP Use

In OpenMP, programs start by running serially, as usual. OpenMP *directives* are used to specify regions that are run in parallel using mechanisms that are hidden from the programmer. It is important to note that a call to a parallel library function is also a parallel region. The regions are executed by some number of serial threads, but simple OpenMP use does not consider the threads explicitly. Directives are are also used to specify properties of variables, such as whether they are shared between threads, private to a thread, and so on.

Writing the OpenMP directives is the easy part of this; the real problem is using them correctly and efficiently. We shall divert slightly before we start to consider that, and see how to tune for SMP without doing any special coding. You can and should use the same techniques for real OpenMP coding. This is **always** how to start OpenMP programming, or almost any other shared-memory programming. After all, why keep a dog and bark yourself?

1.2.4 Basic Tuning

The key principles of tuning are:

- You should use the compiler, rather than trying to bypass it (as far too many Web pages and books on C recommend). You cannot hand-optimise properly for modern CPUs, so do not handicap the compiler when it tries to do so.
- Optimise for memory access, and not for calculation. Also, memory latency is nowadays the main bottleneck rather than bandwidth, except in a few codes. Modern CPUs can execute somewhere between 100 and 1,000 instructions for every main memory access.
- Modern CPUs rely on caching for performance, so ensure that your code is cache-friendly. Cache problems will not cause wrong answers, but will cause OpenMP to run slower than serial code.

Lastly, keep the scheduling really, really trivial. There is some more on this later, but it is a horribly complicated area; for now, simply ignore this.

The key technique for helping the compiler is to keep your code as clean and and as simple as possible. It is not possible to overstress the importance of this for optimisation, but there is no time in this course to mention details except in passing. This is as important for serial optimisation as it is for OpenMP, and is also a massive help when debugging your code. The most important single example is to make your `DO`- or `for`-loops, clean, simple and long; we shall describe some aspects of this later.

1.3 Programming in OpenMP

1.3.1 Terminology and Correctness

Aliasing refers to the case when two variables overlap, either in whole or in part. The most common form is two names for the same location, but there are many others once one uses compound objects like Fortran arrays or derived types, C structures and C++ classes. Aliasing bugs are illegal in serial code, but often show up only when the code is run in parallel.

Atomic means that an action does not overlap with another atomic action; it does not always imply consistency, and the rules for how atomic actions interact with non-atomic ones are complicated. This is described later.

A *data race* is caused when two non-atomic actions overlap, or often when an atomic one overlaps with a non-atomic one. The effect is completely undefined, and is often often complete chaos.

Synchronisation refers to coding to prevent data races, and a lot of this course is about precisely doing that.

The number one approach for ensuring correctness to avoid even correct aliasing as much as possible – i.e. avoid two threads accessing the same location, except when all of those accesses are read-only. To do this, minimise the update of global objects, which includes anything not passed as arguments; this includes Fortran modules, C extern, C++ static members, any pointers and so on.

And, most of all, never access anything both globally and via arguments unless you can guarantee both accesses are read-only to the whole object. Doing this is required by Fortran, but OpenMP requires much the same for C and C++.

1.3.2 Compiler Options

Use reasonably aggressive optimisation but, as with many parallel interfaces, sometimes the absolute maximum causes problems. You should also use inter-procedural optimisation and possibly inlining, which are almost essential for effective optimisation of C and C++. Obviously, you need to enable OpenMP, and perhaps automatic parallelisation; only a few compilers have the latter, and typically only for Fortran.

Unfortunately, the details are too compiler- and version-dependent to cover here. It is generally a bad idea to use optimisations that are specific to the particular machine you are running on (including profile-guided optimisations), or even the very specific ones. The ones to consider first are called things like `-Ofast`, `-O3`, `-ipo`, `-fopenmp`, `-openmp`, `-mp`, `-parallel` and so on.

1.3.3 Profiling

It is always worth starting with this, but all you need to know for most OpenMP tuning is where the time goes – i.e. the percentage of the wall-clock (real, physical) time spent in different regions of code. Using CPU time can be better on shared systems, but is much

harder to use to profile and tune. Function-level profiling, such as `-pg` and `gprof`, is quite adequate. Alternatively, writing your own profiling is very easy, and you can print out just the times you need.

Start by looking to see where most of the time goes; this can sometimes be in a commonly used auxiliary function, which may not be suitable for parallelising; if so, you need to look for a higher level. Tune the most important area, and then try again. You should leave any fancy profiling until very much later, and probably not do it at all!

The following timing functions are available:

	CPU time	Wall-clock time
OpenMP		`omp_get_wtime`
C/C++	`clock()`	`time()`
Fortran	`CLOCK`	`SYSTEM_CLOCK`

C and C++ `time()` is very imprecise (it returns the time in whole seconds). `clock()` and `CLOCK` often return a resolution of 0.01 seconds, though that is not guaranteed. I use the POSIX `gettimeofday()` function for a reliable high-precision timestamp; it is callable from Fortran, and works equally well in serial code:

```
/* Return high-precision timestamp. */
#include <stddef.h>
#include <sys/time.h>
double gettime_ (void) {
        struct timeval timer;
        if (gettimeofday(&timer,NULL)) return -1.0;
        return timer.tv_sec+1.0e-6*timer.tv_usec;
}
```

For now, do not use `Omp_get_wtime`, because we need to declare it; that is easy to do, but I would rather leave it for now. Use the language's built-in timers (or the above timer, if you prefer). The examples will use them, for simplicity, but the C examples actually use the above code. It really does not matter which timers you use; that applies to all tuning, and not just to OpenMP.

1.3.4 Using Libraries

Most systems have some libraries tuned for OpenMP, and the easiest tuning is to change your code to use them. Suitable ones include AMD's ACML, Intel's MKL, the NAG SMP library and FFTW. These include all of the BLAS and LAPACK, and more; generally, the most useful procedures are the ones for dense linear algebra and fast Fourier transforms. You may need to restructure your code to use them, but this will also help for serial execution, and really does pay if your arrays are fairly large. This is especially true for C and C++, where the optimisability of the language is poor.

For Fortran, try changing uses of `MATMUL` to calls to `Z/DGEMM` (`gfortran` has an option to do this); matrix×vector operations (i.e. `Z/DGEMV`) usually deliver less benefit, and look for anywhere else you can call libraries. This is of little use for very small (e.g. 4×4) arrays, though. You also need to watch out for array copying problems; these are covered in

another course: *Introduction to Modern Fortran*, especially in *Advanced Use Of Procedures* and *Advanced Array Concepts*.

For C and C++, libraries can be used to provide array operations, and emulate Fortran's whole array operations; this can make the code clearer and a lot faster. But, again, this is of little use for very small (e.g. 4×4) arrays. Issues with the C++ Standard Template Library (STL) will be covered later.

1.3.5 Environment

By **far** the most important 'mode' is the number of threads. You should always start tuning by trying different values and, later, try varying the number of threads in combination with other optimisations.

For SIMD, you should **never** exceed the number of CPU cores, which should **not** count hyperthreading or other *SMT* mechanisms. The main reason is that you need to optimise memory access, and not calculation, but there are also scheduling and other reasons.

It is well worth considering the use of **fewer** threads than cores, and this is especially important if system used for anything else. The reasons for this are covered in the course: *Parallel Programming: Options and Design*.

All of OpenMP's environment variables are in upper-case and start `OMP_`, and there is only one that is critical:

```
export OMP_NUM_THREADS=<n>
```

Two others can be useful for SIMD programming, but are less important and you need not understand what they do for now:

```
export OMP_SCHEDULE=static
export OMP_DYNAMIC=false
```

There are a few others that can be useful, and we shall cover them as we need them.

1.3.6 Library Examples

This first lecture actually gains enough performance for many people, and those people could stop now, but this is a course on actually using OpenMP! It is worth trying the simple approach first, though, just to see what happens.

There are two simple linear algebra examples:

`Programs/Multiply.f90` and `Programs/Multiply.c` is standard matrix multiplication using the obvious code.

`Programs/Cholesky.f90` and `Programs/Cholesky.c` solves some positive definite of symmetric linear equations using Cholesky's method; the code was taken from LAPACK, but simplified and modernised.

You should start by looking at them and seeing what they do; for now, just look at the main program. They do the calculation in two different ways: calling the BLAS or LAPACK routines, and using the example code in the relevant language. In addition, the Fortran also uses its intrinsic procedures where relevant (i.e. `MATMUL`).

The objective is to try the effects of optimisation (`-O3`) versus no optimisation, to try the effects of different libraries

Basic: `-lblas` and `-llapack`
Tuned: `-acml` or `-mkl_rt`
Parallel: `-acml_mp` or `-mkl_rt`

and to try the effects of thread count

```
export OMP_NUM_THREADS=1
export OMP_NUM_THREADS=4
```

All of methods and libraries should give the same answer, so you are looking for how to reduce the time. You should look at both the wall clock time and CPU time; in the parallel context, it is the former you optimise and the latter is likely to increase. Where the wall clock time and CPU time are the same, the execution is serial, and level of parallelisation is essentially the ratio; the improvement due to that is the factor by which the wall clock time has been reduced.