Introduction to OpenMP

Background and Principles

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

June 2011
Why Use OpenMP?

This course is about programming in OpenMP

CPUs got faster at 40% per annum until \( \approx 2003 \)
Since then, they have got larger but not faster
The number of CPU cores per chip is now increasing

- The solution is to use more CPUs in parallel

OpenMP is a tool for that on multi-core systems
It uses a Shared Memory Processing (SMP) model
What is OpenMP?

A language extension, not just a library

Designed by a closed commercial consortium
“Open” just means no fee to use specification
They did/do accept public comments on the details

Dating from about 1997, still active
Current specification is version 3.1
Course is based on version 2.5, for portability

Specifications for Fortran, C and C++
Most compilers have some OpenMP support
Shared-Memory Summary

Message passing (e.g. MPI) uses parallel processes
Each process has separate (“distributed”) memory

SMP has a single process with parallel threads
All threads have access to all the memory

• Simpler in some ways, more complex in others

Hard to implement this efficiently on modern systems
Needs to be synchronisation between threads

• Programs must follow strict rules to make that work
OpenMP’s Role (1)

• Not generally advised for separate tasks
  Use MPI or a batch scheduler for that
  Or just run multiple background processes

• Almost always used for more performance
  As in HPC – High Performance Computing
  Objective is genuine parallel execution

MPI is what most people use for clusters etc.
Also multiple processes on multi-core computers
OpenMP’s Role (2)

• But distributing data is very tricky
  Both for performance and for correctness
  Shared memory means that you don’t have to do that

• OpenMP dominates shared memory for HPC
  That is today (2011) – but may change by 2020

Fortran and C++ standards now have parallelism
  With very different parallel models and objectives

And there are other designs – the area is in flux
OpenMP Design

This is how most people and libraries use it Design policy of NAG SMP, MKL, ACML etc.

- Start with a well-structured serial program
  Most time spent in small number of components
  Must have clean interfaces and be computational

- Don’t even attempt to convert whole program
  Do it component by component, where possible

This is the approach used in the examples
There is more on this topic in the last lecture
Beyond the Course (1)

Email scientific-computing@ucs for advice

The materials for this course are available from:

http://www-uxsup.csx.cam.ac.uk/...
.../courses/OpenMP/

Several other relevant Computing Service courses
Some will be mentioned in passing, but see:

http://www-uxsup.csx.cam.ac.uk/courses/
Beyond the Course (2)

- Most books and Web pages are unreliable
  Far too many just summarise the OpenMP specification

This is listed on the HECToR Web site
Fairly reasonable, but doesn’t warn about problems

Parallel Programming in OpenMP
  Chandra, Kohr, Menon, et al.
  ISBN: 1558606718
OpenMP Specification

• The OpenMP specification is often ambiguous
  Sometimes even inconsistent or nonsensical

Each major version has added lots of new features

• Compilers vary a great deal in important details

http://openmp.org/wp/openmp–specifications/
Course Coverage (1)

It is even harder to test a compiler than user code. Tricky features are likely to be unreliable.

- This course teaches a fairly safe subset. If these features don’t work, the others aren’t likely to.

- It also teaches the simplest useful features. Most likely to actually work in real code.
Course Coverage (2)

• Shared memory programming NOT about syntax
  Far more knowing what to do and what not to do

• This course describes some safe practices
  Most likely to to be got to work in real code

• It includes warnings about potential problems
  Follow its guidelines and avoid problems
  Please ask if you want the why as well as the how

Remember: a problem avoided is not your problem
Portability, RAS, etc. of Code

Specified in standard

Works under conditions A

Works under conditions B

Works under conditions C

Works under conditions D

Safe

Just coding and "testing until it works" may end up here
SIMD Computing (1)

**SIMD** means *Single Instruction, Multiple Data*
A generalisation of the old *vector computing* model
Think about operations on *whole arrays* at once

- Vector hardware is more-or-less defunct
  Modern **SIMD** handled entirely by the compiler
  Fortran *array operations* should do this but often don’t

**SSE/MMX, VMX/Altivec** etc. are **SIMD** instructions

**OpenMP** enables *multiple cores* to be used similarly
Aside: GPUs

⇒ GPUs also use a SIMD model

Using NVIDIA etc. cards for extreme performance
Current language extensions are CUDA and OpenCL

- They need a similar design to OpenMP SIMD
  Most of this lecture applies to them as well

The actual code is completely different, of course
This course will not mention them further
SIMD Computing (2)

A good optimising compiler does all that for you. Fortran ones may even autoparallelise your code.

- The compiler handles the synchronisation. Covers up problems in underlying implementation. E.g. ambiguities in the threading memory model.

- In practice, this implies gang scheduling. All cores operating together, semi-synchronised.

Will cover some of these issues in more detail later.
Why Use OpenMP?

If it’s all automatic, why bother using OpenMP?

- Only the simplest cases are automatic
  Often need things the compiler won’t parallelise
SPMD Computing (1)

SPMD means Single Program, Multiple Data
Each ‘thread’ can operate semi–independently
E.g. each of them calls a different function

Much more flexible, but much harder to get right
We will cover only the very simplest forms of this

You are strongly advised to be cautious
• Be ‘clever’ and you will shoot yourself in the foot

Most books and Web pages do not teach that
SPMD Computing (2)

There is no major Fortran advantage for SPMD
• We will cover it after the simpler SIMD

Lastly, you can add inter-thread communication
Almost like separate, communicating processes

• But you are strongly advised to avoid that
I will explain why when we come to it

See Hoare’s Communicating Sequential Processes!
Also the memory model in the new C++ standard
Simplistic OpenMP (1)

Easiest way of parallelising a serial program is:

- Set compiler options for full optimisation
  Fortran users should select autoparallelisation

- Do some fairly high-level profiling of it
  Now consider just the areas that take the most time
  Add some timing code around the interesting areas

- Try adding calls to parallel library functions
  For example, LAPACK in MKL or ACML
  It’s a good idea to use those even in serial code
Simplistic OpenMP (2)

- Make sure you link with a parallel library. Set environment variables and use multi-core system. If good enough, then you have done all you need.

- Next, add SIMD directives where possible. Use compilers (e.g., Intel’s) to tell you if they work. Look at the performance improvement, if any.

- Then change your code or use SPMD directives. Finally, worry about more advanced parallelism.

Too good to be true? A bit, but it’s worth trying.
Programs start by running **serially**, as usual. **Directives** specify **parallel regions**. These are run **automagically** in parallel.

- A parallel **library call** also a parallel region.

This is done by some number of serial **threads**. **Simple use** doesn’t consider the threads explicitly.

**Directives** also specify **variable** properties. They can be **shared**, **thread-private** etc.
Diversion

- Writing the OpenMP directives is the easy bit
  Problem is using them correctly and efficiently

Will divert before we start to consider that
- See how to tune for SMP without coding
  Use same techniques for real OpenMP coding, too

- Always how to start OpenMP programming
  Or almost any other shared-memory programming

- Why keep a dog and bark yourself?
Principles of Tuning

- **Use** the compiler, don’t **bypass** it.
  Can’t **hand-optimise** properly, so don’t **handicap** it.

- **Optimise memory access**, not **calculation**.
  Memory **latency** is nowadays the main bottleneck.

- Modern CPUs rely on **caching** for performance.
  Problems will cause OpenMP to run **slower** than serial.

- Keep the **scheduling** really, really **trivial**.
  There is some more on this later.
Helping the Compiler

- Keep your code **clean and simple**

- Can’t overstress the importance for **optimisation**
  No time to mention **details** except in passing

- Important for both **serial optimisation** and OpenMP
  And is a massive help when **debugging** your code

- Make **DO/for-loops**, **clean, simple and long**
  Will describe some aspects of this later
Terminology

- **Aliasing** is when two variables overlap. Most common form is two names for same location. Bugs often show up only when run in parallel.

  Atomic doesn’t overlap with another atomic action. Doesn’t always imply consistency (see later).

  A data race is when two non-atomic actions overlap. The effect is completely undefined – often chaos.

  Synchronisation is coding to prevent data races. A lot of this course is about precisely that.
Ensuring Correctness

- **Number one** approach is avoiding aliasing Two threads accessing the same location (except when all accesses are read-only)

- Minimise the **update** of global objects Generally, anything not passed as arguments In modules, static/extern, via pointers etc.

- Never access both **globally** and via arguments Unless you can guarantee both are read-only
Compiler Options

• Use reasonably aggressive optimisation
Sometimes the absolute maximum causes problems

• Use inter-procedural optimisation and inlining
This is almost essential for C and C++

• Enable OpenMP, maybe automatic parallelisation
Few compilers have the latter, and only for Fortran

Details too compiler and version-dependent to cover
Profile Your Code (1)

- All you want to know is where the time goes
  I.e. percentage of wall-clock time in regions of code
  Using CPU time can be better on shared systems

Function-level profiling (e.g. -pg and gprof) is fine
Alternatively, writing your own is very easy

- Look to see where most of the time goes
  Sometimes in a commonly used auxiliary function

Tune the most important area, and try again
Leave any fancy profiling until much later
The following **timing functions** are available

<table>
<thead>
<tr>
<th></th>
<th>CPU time</th>
<th>Wall–clock time</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td></td>
<td><strong>omp_get_wtime</strong></td>
</tr>
<tr>
<td>C/C++</td>
<td><strong>clock()</strong></td>
<td><strong>time()</strong></td>
</tr>
<tr>
<td>Fortran</td>
<td><strong>CLOCK</strong></td>
<td><strong>SYSTEM_CLOCK</strong></td>
</tr>
</tbody>
</table>

**time()** is very **imprecise** (whole seconds)
I show a more precise **alternative** in a moment
**clock()** and **CLOCK** are often 0.01 seconds
I use this if I need to – it’s callable from Fortran

/* 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; 
}
Omp_get_wtime

- Don’t use it yet, because we need to declare it. Easy to do, but I would rather leave it for now.

- For now, use the language’s built-in timers. The examples will use them, for simplicity. Actually, the C uses gettimeofday().

It really doesn’t matter which timers you use. That applies to all tuning, and not just OpenMP.
Using Libraries

Most systems have libraries tuned for OpenMP etc.
- Easiest tuning is to change code to use them
- Suitable ones include ACML, MKL and NAG SMP
  These include all of BLAS and LAPACK, and more
  Most useful are dense linear algebra and FFTs

May need to restructure your code to use them
- Really does pay, if your arrays are fairly large
  Especially for C/C++, where optimisability is poor
Fortran and Libraries

Try changing `MATMUL` to `Z/DGEMM`  
Matrix $\times$ vector usually less benefit (`Z/DGEMV`)  
And look for anywhere else you can call libraries

- Little use for very small (e.g. $4 \times 4$) arrays, though
- Also watch out for array copying problems

See *Introduction to Modern Fortran: Advanced Use Of Procedures Advanced Array Concepts*
C/C++ and Libraries

Can be used to provide array operations
Emulates Fortran’s whole array operations
• Code can be clearer and a lot faster

• Little use for very small (e.g. $4 \times 4$) arrays, though

Issues with the C++ STL will be covered later
Number of Threads

By far the most important ‘mode’
• Always start tuning by trying different values
  And try that in combination with others

• For SIMD, never exceed number of CPU cores
  And don’t count Hyperthreading or other SMT
  Reason: optimise memory access, not calculation

• Consider using fewer threads than cores
  Especially important if system used for anything else
  See Parallel Programming: Options and Design
Environment Variables

They are in **upper-case** and start **OMP_**

There is only one that is critical:

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

Two can be useful for **SIMD** programming:

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

There are a few others that can be useful

*We shall cover them as we need them*
Library Examples

- That actually gains enough for many people
But this is a course on using OpenMP...

There are two simple linear algebra examples

Programs/Multiply.f90 and Programs/Multiply.c
Standard matrix multiplication using the obvious code

Programs/Cholesky.f90 and Programs/Cholesky.c
Solution of positive definite linear equations
These are LAPACK code, simplified and modernised
What They Do

Start by looking at them and seeing what they do. For now, just look at the main program.

They do the calculation two different ways:

- Calling the BLAS or LAPACK routines
- Using the example code, in the relevant language

Plus, for Fortran only, of course:

- Using Fortran’s intrinsic procedure MATMUL
Example Objective

• To try the effects of **optimisation** (--O3)

• To try the effects of different libraries
  Basic: --lblas and --llapack
  Tuned: --acml or --mkl_r
  Parallel: --acml_mp or --mkl_r

• To try the effects of **thread count**
  export OMP_NUM_THREADS=1
  export OMP_NUM_THREADS=4

Not on the **PWF**, as only single–core systems
What To Look For

All methods and libraries give the same answer
So you are looking for how to reduce the time

Look at both the *wall clock* time and *CPU* time
In the parallel context, it’s the *former* you optimise

Where they are the *same*, the execution is *serial*
Level of *parallelisation* is essentially the *ratio*
- The *improvement* is reduction in *wall clock time*
OpenMP Programming Model. OpenMP API Overview. Compiling OpenMP Programs. OpenMP Directives. Directive Format. C/C++ Directive Format. OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file. If every thread conducts I/O to a different file, the issues are not as significant. It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program. Introduction to OpenMP Programming. NERSC Staff. Agenda. Basic information. An selective introduction to the programming model. Directives for work parallelization and synchronization. Some hints on usage. Hands-On Lab. Writing compiling and executing simple OpenMP programs. Presentations available at. module load training! cp $EXAMPLES/NUG/Presentations/. After a long thirst for parallelizing highly regular loops in matrix-oriented numerical programming, OpenMP was introduced by OpenMP Architecture Review Board (ARB) on 1997. In the subsequent releases, the enthusiastic OpenMP team added many features to it including the task parallelizing, support for accelerators, user-defined reductions and lot more. The latest OpenMP 5.0 release was made in 2018 November. Open Multi-processing (OpenMP) is a technique of parallelizing a section(s) of C/C++/Fortran code. Module 1: Introduction to parallel programming Module 2: The boring bits: Using an OpenMP compiler (hello world) Discussion 1: Hello world and how threads work. Unit 2: The core features of OpenMP Module 3: Creating Threads (the Pi program) Discussion 2: The simple Pi program and why it sucks Module 4: Synchronization (Pi program revisited) Discussion 3: Synchronization overhead and eliminating false sharing Module 5: Parallel Loops (making the Pi program simple) Discussion 4: Pi program wrap-up. OpenMP is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism in C/C++ programs. It is. OpenMP is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism in C/C++ programs. It is not intrusive on the original serial code in that the OpenMP instructions are made in pragmas interpreted by the compiler. OpenMP uses the fork-join model of parallel execution.