Introduction to OpenMP

Background and Principles

Nick Maclaren

nmm1@cam.ac.uk

September 2019

Why Use OpenMP?

This course is about programming in OpenMP

CPUs got faster at 40% per annum until ≈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 5.0
Course is mainly 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

- 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 SMP programming for HPC
But increasing use of higher−level SMP toolkits
That is today (2018) – 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

Apologia (1)

This course is NOT what I would like
And it's unpopular with you MPhil students
Unfortunately ....

Shared memory programming is seriously tricky

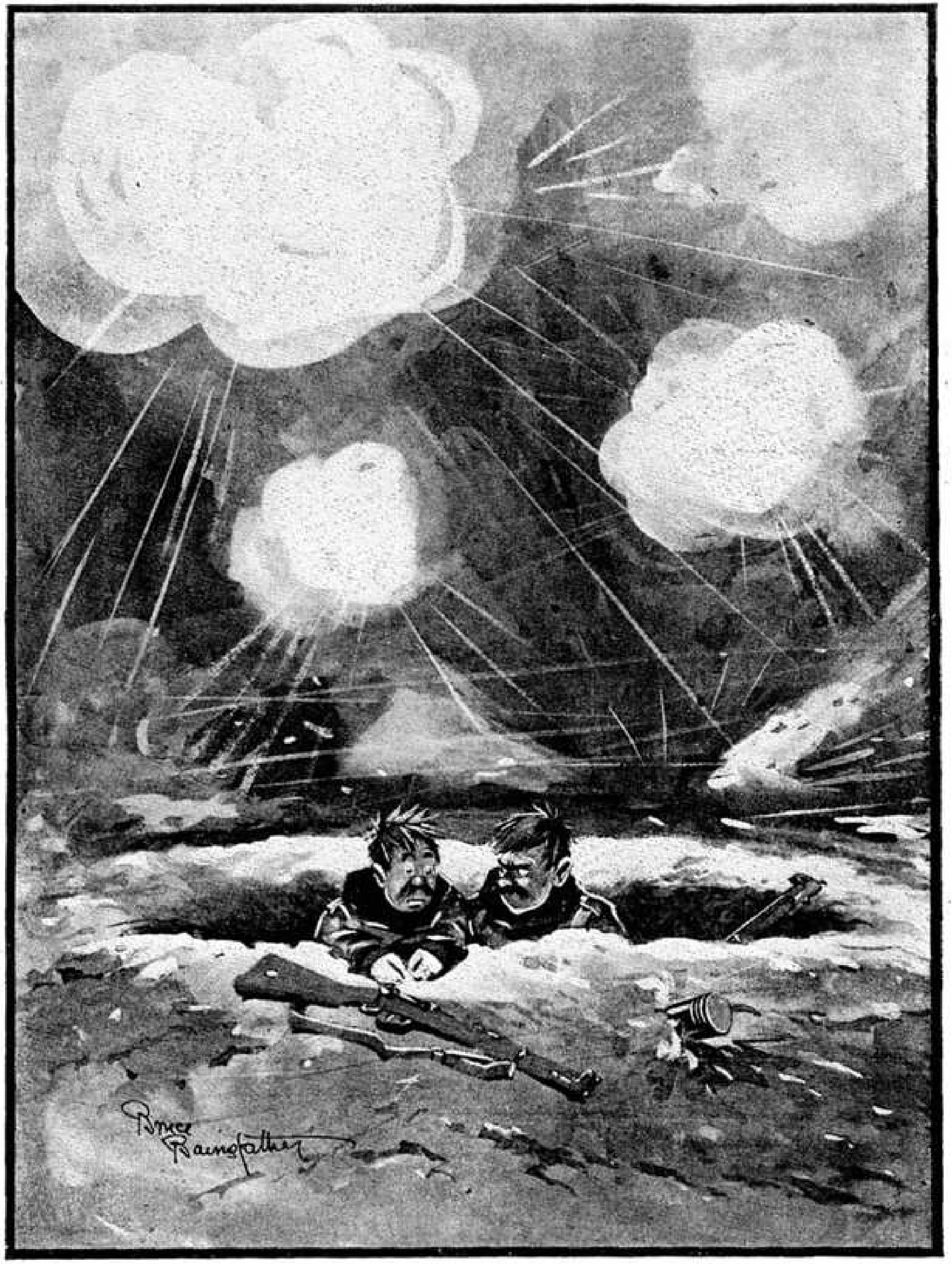
- Doing the actual coding is the easy bit

- Avoiding the 'gotchas' is the hard bit

The key to success is knowing what not to do

This applies to POSIX and C++ threads as well

One of Our Minor Wars



" Well, if you knows of a better 'ole, go to it "

BY CAPT. BRUCE BAIRNSFATHER

Apologia (2)

Why teach it? Because it is best for the purpose
POSIX and C++ threads etc. are even worse

Worst problem is data races causing wrong answers
Often escape testing, and are almost unfindable
Aren't any decent tools to detect them in OpenMP

Easy to 'learn' OpenMP, but not enough to use it
Feedback says that this course explains why
People who follow it usually succeed with OpenMP

I explain more about this in the last two lectures

Beyond the Course (1)

The materials for this course are available from:

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

Several other courses may be relevant to you
Some will be mentioned in passing, but see:

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

Beyond the Course (2)

- Most books and Web pages are unreliable

Far to many just summarise the OpenMP specification

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

Parallel Programming in OpenMP
Chandra, Kohr, Menon, et al.
Morgan Kaufmann, 2001.
ISBN: 1558606718

OpenMP Specification

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

Each major version has added lots of new features
- Many of those features are unlikely to work
Many others have really arcane gotchas

- Compilers vary a great deal in important details
And fancy features if often broken in some compilers

http://openmp.org/wp/openmp–specifications/

The Bright Side

- For HPC, it is vastly easier than threading

This course teaches using it simply and safely

- But 30% of is warning about gotchas

Avoiding traps is the key to success with OpenMP

Most of that is left to the lecture Critical Guidelines

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

Don’t trust the Web of a Million Lies or even books

Course Coverage (2)

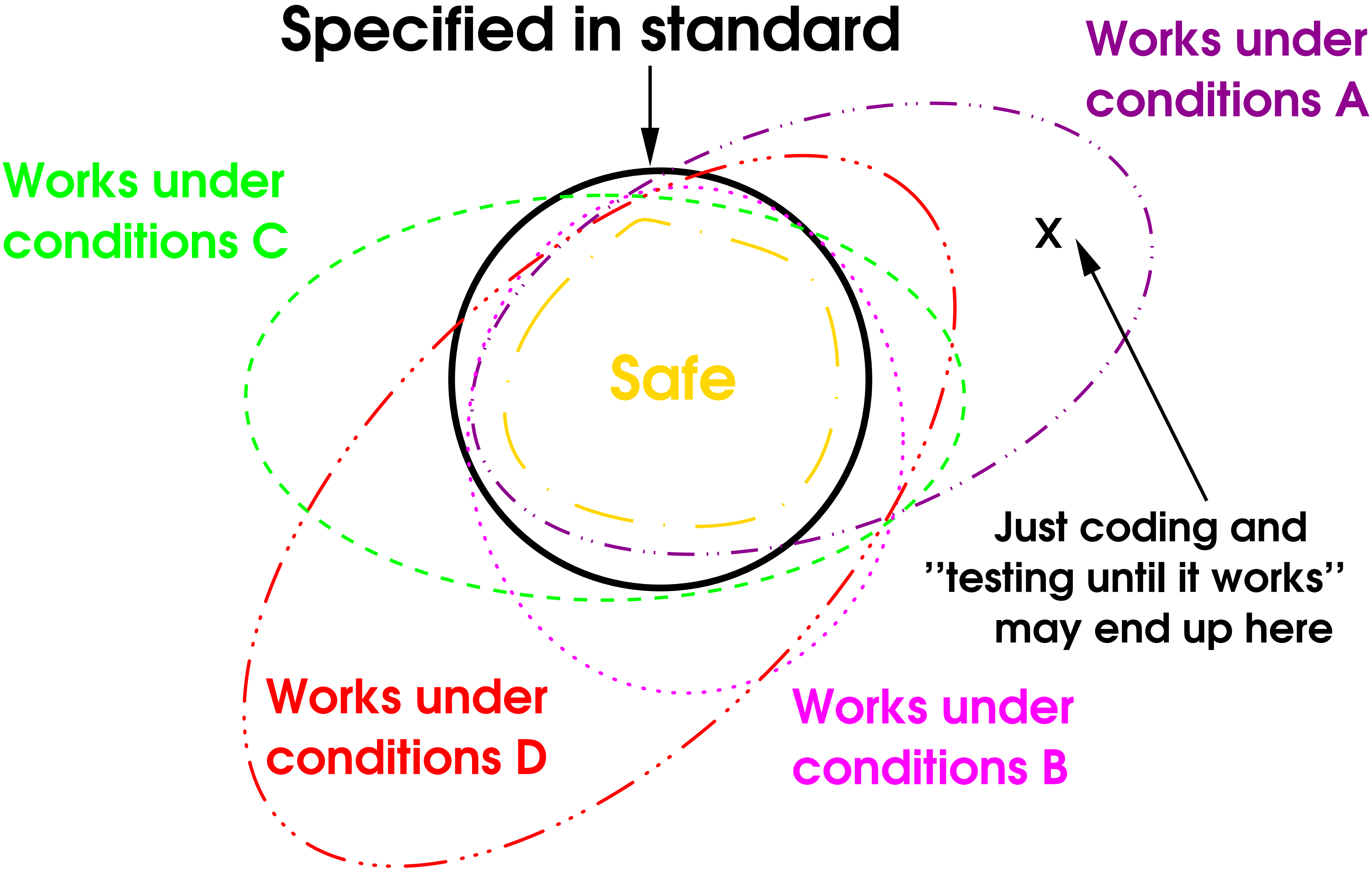
- Shared memory programming NOT about syntax
Far more knowing what to do and what not to do

- 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



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
A few ones may even autoparallelise your code
This is much easier and more effective for Fortran

- 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

If available (e.g. Intel), 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

Basic OpenMP Model

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 mainly for Fortran

Details too compiler and version−dependent to cover
−Ofast, −O3, −ipo, −fopenmp, −openmp, −mp etc.

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

Profile Your Code (2)

The following timing functions are available

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

time() is very imprecise (whole seconds)
I show a more precise alternative in a moment
clock() and CLOCK are often 0.01 seconds

High-Precision Timestamp

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. 4×4) 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

There are often C++ template libraries, too

- Little use for very small (e.g. 4×4) 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

Skip the first examples for this MPhil
The systems don't have suitable libaries installed
Instructions are in the notes if you want to try

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_rt
Parallel: –acml_mp or –mkl_rt

- To try the effects of thread count

  export OMP_NUM_THREADS=1
  export OMP_NUM_THREADS=4

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