Lecture 2: Introduction to OpenMP with application to a simple PDE solver

Mike Giles

Mathematical Institute

Hardware and software

Hardware:

- a processor (CPU) is a single chip – a server often has two
- a processor usually has many cores which operate largely independently of each other

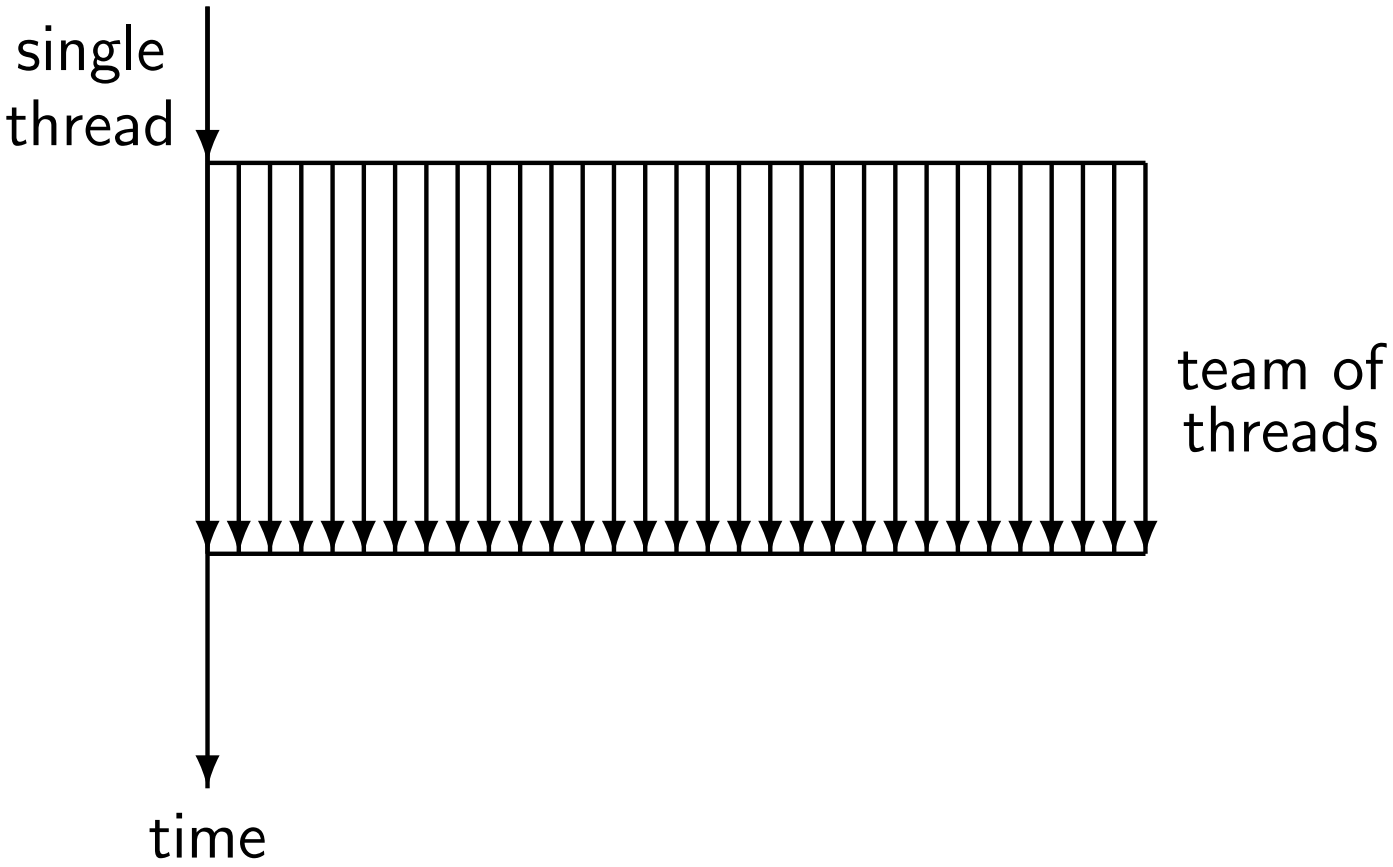
Software:

- a process is a software program which runs on a server: the operating system scheduler is responsible for deciding when/where it executes on the hardware
- a process often uses multiple threads, each running on a core

OpenMP

OpenMP is a way of writing a multi-threaded program which

- uses a single thread some of the time
- uses a set of threads for demanding parts of the code



OpenMP

OpenMP is a standard managed by a non-profit consortium involving all major computer companies.

Wikipedia: `https://en.wikipedia.org/wiki/OpenMP`
gives a good overview of the history:

- Version 1.0: 1997 – initially for Fortran; C/C++ added in 1998
- Version 2.5: 2005 – unified Fortran/C/C++ spec
- Version 3.0: 2008 – added tasks
- Version 4.0: 2013 – added `simd` support for vectorisation
- Version 5.0: 2018 – added support for accelerators
- Version 6.0: 2024 – latest version

In this lecture we will discuss only the C version – the Fortran version is very similar.

OpenMP

OpenMP consists of 3 sets of components:

- compiler directives ("pragmas")

  these are comments in the code which give instructions to the compiler if it is run with the appropriate OpenMP flag

- run-time library routines

  these are functions which are called by the program and defined in a header file `omp.h`

- environment variables

  these are set by the user before running the program, and give directions to both the program (e.g. number of threads to use) and the operating system scheduler (e.g. where to run them)

There are also compiler flags to consider, but these are not part of the standard.

A simple loop

```
      int i, m;
      m = 8000;

#pragma omp parallel for \
  implicit(none) shared(a,b,c,m) private(i)
      for (i=0; i<m; i++) {
        c[i] = a[i] + b[i];
      }
```

If this is executed by 4 threads, then the first thread does 0 – 1999, the second does 2000 – 3999, etc.

Also, in addition to thread parallelism, each thread will use vectorisation (if the right compiler flags are given)

A simple loop

Shared variables: these are variables which are referenced by all threads – i.e. the threads all reference the same single copy

Private variables: each thread has its own (uninitialised) copy of the variable – this is clearly needed for the loop counter.

(By default, the values are forgotten once the loop ends.)

The `implicit(none)` avoids default assumptions, so everything has to be declared shared or private – this is strongly recommended

These are all examples of an OpenMP "clause" which provides additional information

A simple loop

Variables defined within the loop are automatically private, so I prefer to use

```
      int m = 8000;

#pragma omp parallel for \
  implicit(none) shared(a,b,c,m)
      for (int i=0; i<m; i++) {
        c[i] = a[i] + b[i];
      }
```

A simple loop

What happens if we want to sum all of the elements in an array?

```
int    m = 8000;
double sum = 0.0;

for (int i=0; i<m; i++) {
  c[i] = a[i] + b[i];
  sum = sum + c[i];
}
```

`sum` can't be a private variable, but the calculation won't work correctly if it is a shared variable – we discussed this in the first lecture

A simple loop

Fortunately, this is so important that OpenMP has a special solution:

```
      int    m = 8000;
      double sum = 0.0;

#pragma omp parallel for \
  implicit(none) shared(a,b,c,m) reduction(+:sum)
      for (int i=0; i<m; i++) {
        c[i] = a[i] + b[i];
        sum = sum + c[i];
      }
```

The compiler creates a private copy of `sum` for each thread, initialised to 0.0, and then afterwards adds them onto the `sum` for the main thread

Reductions

As well as addition, the reduction clause can handle $*$, $-$, max, min and various logical operators.

It is even possible to define your own reduction operator.

OpenMP version 4.5 extended the syntax to arrays so that

`reduction(+:array[:10])`

will apply the reduction operation to an array of length 10.

i.e. each loop element is creating an array of length 10, and these are all being added to together to create a overall sum array of length 10.

Nested loops

What about nested loops?

```
#pragma omp parallel for ...
      for (int i=0; i<m; i++) {
        for (int j=0; j<m; j++) {
          ....
        }
      }
```

Starting and stopping teams of threads is expensive, so usually best to parallelise outer loop.

The compiler will probably vectorise the inner loop.

Nested loops

Sometimes the outer loop doesn't offer enough parallelism – can use another "clause" to collapse 2 or more loops

```
#pragma omp parallel for ...  collapse(2)
      for (int i=0; i<I; i++) {
        for (int j=0; j<J; j++) {
          ....
        }
      }
```

which I think this is equivalent to

```
#pragma omp parallel for ...
      for (int k=0; k<I*J; k++) {
        i = k / J;
        j = k % J;
        ....
      }
```

Nested loops

A final pragma to mention is the tile pragma which has the effect of converting

```
void func1(int A[100][128])
{
   #pragma omp parallel for
   #pragma omp tile sizes(5,16)
   for (int i = 0; i < 100; ++i)
       for (int j = 0; j < 128; ++j)
           A[i][j] = i*1000 + j;
}
```

Nested loops

into

```
void func2(int A[100][128])
{
   #pragma omp parallel for
   for (int i1 = 0; i1 < 100; i1+=5)
       for (int j1 = 0; j1 < 128; j1+=16)
           for (int i2 = i1; i2 < i1+5; ++i2)
               for (int j2 = j1; j2 < j1+16; ++j2)
                   A[i2][j2] = i2*1000 + j2;
}
```

Breaking a big grid into tiles/blocks like this can lead to better cache reuse – tiling/blocking is a standard optimisation for matrix-matrix multiplication

OpenMP

You can go a long way in many applications using just `#pragma omp parallel for`

We turn attention now to the other elements:

- run-time functions
- environment variables
- compiler flags

OpenMP RTL

These are some of the more useful run-time library (RTL) functions:

- `int omp_get_max_threads()` – gets number of threads
- `void omp_set_num_threads(int num_threads)` – sets number of threads to be used (but I prefer to use an environment variable for this)
- `double omp_get_wtime()` – gets wall time in seconds from some arbitrary fixed time

All of these are usually called in the sequential part of the code which is executed by the main thread.

Remember: the program must include the OpenMP header file:

```
#include <omp.h>
```

OpenMP RTL

One slight problem with the RTL is if you want to compile the code without OpenMP.

In that case, you can avoid the compilation of the RTL functions by using CPP (the C Pre-Processor) to perform conditional compilation based on a special variable `_OPENMP` defined by the compiler.

```
#ifdef _OPENMP
     int nthreads = omp_get_max_threads();
     printf("#threads = %d \n",mthreads);
#endif
```

(https://en.wikipedia.org/wiki/C_preprocessor)

Environment variables

There are several environment variables which can affect the execution of an OpenMP program

The most important two are:

- `OMP_PROC_BIND` which specifies whether threads are pinned
- `OMP_NUM_THREADS` which specifies the number of threads

I set both of these in my `.bashrc-user` and `.bash_profile-user` configuration files:

```
export OMP_PROC_BIND=true
export OMP_NUM_THREADS=36
```

Thread pinning

Thread pinning means that a thread is pinned to a particular core.

If the operating system scheduler suspends the thread for a moment to allow another process to run, then when it starts it again it does so on the original core.

The benefit of this is that most of the data the thread was using is still in the caches of that core.

Compiler directives

Finally we come to the compilation commands.

Using Intel's `icx` compiler, I use something like

```
icx -O3 -qopenmp -xHost prog.c -o prog -lm
```

- `-O3` forces a high degree of optimisation
- `-qopenmp` turns on the processing of OpenMP pragmas
- `-xHost` generates code aimed at the system on which it is compiled; this turns on vectorisation on modern CPUs

Compiler directives

For the gcc compiler, the corresponding command is

```
gcc -O3 -fopenmp -march=native prog.c -o prog -lm
```

- -O3 forces a high degree of optimisation
- -fopenmp turns on the processing of OpenMP pragmas
- -march=native generates code aimed at the system on which it is compiled

Practical 1

Practical 1 concerns an approximation to the 2D parabolic PDE

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

on a unit square with homogeneous b.c.'s.

Using a regular grid with spacing of h in each direction, and a timestep of size k, a forward-time central-space approximation leads to the discrete equations

$$U_{i,j}^{n+1} = U_{i,j}^n + \nu \left(U_{i+1,j}^n + U_{i-1,j}^n + U_{i,j+1}^n + U_{i,j-1}^n - 4\, U_{i,j}^n \right),$$

where $\nu \equiv k/h^2$ and we will keep $\nu \leq 1/4$ for stability.

Practical 1

Observations:

- this is a naturally parallel mathematical application, i.e. all of the $U_{i,j}^{n+1}$ can be evaluated at the same time.
- if the grid size (i.e. total number of unknowns) is big enough, there should be enough work in each timestep to keep busy up to 64 cores, each executing vectors of length 8 or 16.
- there's not a lot of compute per memory reference, so the execution performance may be limited by data bandwidth rather than compute capability.

Practical 1

Some programmers prefer to use a 2D array `u[i][j]` for this kind of application. In this case, `u[i]` is a pointer to a contiguous block of memory in which the values `u[i][j]` are stored for all values of `j`.

i.e. `u[i][j+1]` is stored next to `u[i][j]`, but `u[i+1][j]` is not.

I prefer to use simple 1-dimensional arrays for storage, and map (i, j) indices to a 1-dimensional memory index: `u[i+j*I]` where `I` is the grid size in the `i` direction.

In this case, `u[i+1+j*I]` is stored next to `u[i+j*I]`, but `u[i+(j+1)*I]` is not.

The choice is a matter of personal preference, but it is important to understand the layout and which pairs of indices (i, j) are neighbours.

Practical 1

The central part of the C implementation in `fd.c` can be written as

```
for(int n=0; n<N; n++) {

  for(int i=0; i<I; i++) {
    for(int j=0; j<J; j++) {
      int ind=i+j*I;
      u2[ind] = (1.0-4.0*nu)*u1[ind] +
          nu*(u1[ind+1]+u1[ind-1]+u1[ind+I]+u1[ind-I]);
    }
  }

  double *tmp=u1; u1=u2; u2=tmp;
}
```

Practical 1

Notes:

- old U^n array is `u1`, new U^{n+1} is `u2`
- the pointer swap is an efficient way to swap the arrays to prepare for the next timestep – much cheaper than copying from `u2` to `u1`
- the `i` and `j` loops could be swapped – which order is best?
- where should the OpenMP pragma go?

Practical 1

Notes:

- a second code `fd3.c` does a similar calculation on a 3D grid
- this gives an opportunity to experiment with the `collapse` clause and the `tile` pragma
- a **BIG** difference: with the 2D code, the amount of data is small enough to be held in the L2 caches, collectively; with the 3D code there's too much to be held in the L2/L3 caches, so each timestep it all has to be loaded from DRAM – this has big consequences for the performance

Final comments

- using OpenMP can be relatively easy
- however, Practical 1 will show that performance can be poor if you don't understand what is going on
- Practical 1 also illustrates the use of timing to work out the effective GFlop/s performance and GB/s bandwidth

  For important calculations I want to know what fraction of the peak capability I am achieving – my target is usually 10% of either compute capability or bandwidth, whichever is the limit on overall performance