Worksheet week 9 – Crash course in OpenMP

1 Purpose

1. Learn the basics of OpenMP.
2. Parallelise some C++ code using OpenMP, including your 2D Ising model code.

2 OpenMP

2.1 What is OpenMP?

OpenMP (Open Multi-Processing) is an application programming interface (API) that acts as an extension to C/C++/Fortran to allow for the parallelisation of existing code. OpenMP is particularly convenient in that it usually only requires a few minor changes to existing code to gain a speedup.

Since the mid-2000s, most CPUs have consisted of multiple cores/processors (modern computers/phones have at least 4 cores). By default, a compiled C++ program will only utilise one of these cores when you run it. OpenMP provides a way of splitting your code over all the cores in your CPU to make use of all the available resources, and (ideally) make the computation faster.

2.2 Important OpenMP concepts:

Threads:
OpenMP parallelises code by creating teams of *threads*. A thread is simply a set of instructions that make up part of your program, and you usually send one thread to each core in your CPU. If you use more threads than you have cores, you will most likely lose your parallel advantage, because the threads get in each other's way.

Master thread:
Your code will begin running on a *master thread*. If you don't include any parallel regions in your code, then no new threads will be created, and the master thread will just run everything in serial (this is equivalent to running without OpenMP).

Parallel regions:
At a given point in your program, you will add a *parallel region* (often around a for loop)—this is where the parallelism happens. Here, the master thread creates multiple additional threads, and the computation is automatically divided among the threads (OpenMP decides how to split things up for you). At the end of the parallel region, the master thread waits for all other threads to finish their work. Once all threads have finished, the additional threads are terminated, and the master thread continues running the next sections of code. See Fig. 1.

Private memory:
Inside a parallel region, each thread has its own *private memory*, where it can store its own variables that are invisible to other threads. For example, if thread 0 writes a variable `A=5` into its private memory, and thread 1 writes `A=10`, then these values are independent of each other. If thread 2 hasn't written the variable A, and tries to read A, it will result in an error, because thread 2 doesn't have access to either of the A values.

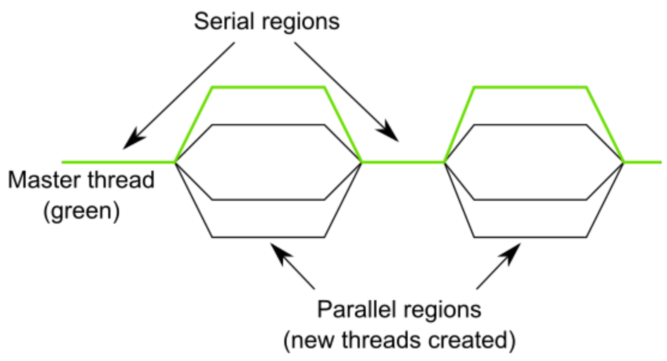

Figure 1: Structure of a code parallelised using OpenMP. Time runs from left to right. In this example, four threads are created in each parallel region.

Shared memory:
Inside a parallel region, all threads have access to a *shared memory*. The threads can't directly communicate with one another, but they can read/write variables to this shared memory as a form of communication. All threads see the same copy of shared variables, and all threads can read/write to these shared variables. For example, if thread 0 writes the variable `A=5` to the shared memory, then thread 1 can come along later and read/update/overwrite that value.

Synchronisation:
The threads are assigned their work on a 'first come first serve' basis, so if one thread finishes an iteration of a parallel loop before any others, it could start the next iteration while the other threads are still working on the previous one. As a result, the threads will very quickly desynchronise. If there needs to be some sort of deterministic communication between the threads (e.g. thread 0 writes a variable to the shared memory and then thread 1 needs to read it afterwards), you need to add synchronisation steps. Similarly, you can run into issues if multiple threads are trying to update a shared variable simultaneously (e.g. if `A = 5` in shared memory, and thread 0 and thread 1 simultaneously try to run `A = A + 1`, then the result will not be deterministic—this a called a 'race' condition).

2.3 Setting up OpenMP

With OpenMP, you write a C++ script 'as normal', except for the following two changes:

1. You need `#include <omp.h>` at the top of your code.
2. You need to include the `-fopenmp` flag when compiling, i.e. `g++ file.cpp -o file -fopenmp`.

Before running the compiled code, you can set the number of threads in the terminal with:

```
export OMP_NUM_THREADS=x
```

where x is the number of threads you want to use (1, 2, 4, 8, etc). Note that if you use one thread, your code will run in serial. `OMP_NUM_THREADS` is an environment variable for the current shell, and the export command just allows us to re-assign its value for use in all future commands in that shell.

2.4 Using OpenMP

The commands you give to OpenMP are known as *directives.* You signpost these directives in your code by adding what's known as a *sentinel* to the start of the line, which essentially tells the compiler: "this line provides instructions to OpenMP". The sentinel in C/C++ is `#pragma omp`. A directive can have multiple *clauses* after it, which specify additional options for that directive. So the full syntax is:

```
#pragma omp [directive] [clause 1, clause 2, etc.]
{
    [parallelised code]
}
```

The clauses can be comma- or space-separated. There must be a new line at the end of the directive, and braces as shown. Some directives don't require braces, as they often only apply to the following line (e.g. for loops). If you compile and run your code without OpenMP enabled, these directives will be ignored and the code in braces will run in serial.

2.5 Useful OpenMP features

Below is a list of some of the most useful OpenMP features for our purposes. Complete lists can be found online; see e.g.

`https://www.openmp.org/spec-html/5.0/openmpch2.html#openmpse17.html`

A reference card for directives/functions etc. for OpenMP 4.0 has also been provided, with the most relevant features highlighted.

Parallel region:

- To create a parallel region, the syntax is:

  ```
  #pragma omp parallel [clauses]
  ```

- The most important clauses for our purposes will be the "shared" and "default", so your usage will look something like this:

  ```
  #pragma omp parallel default(none) shared(a, b, cout)
  ```

- *Important:* always include `default(none)`!!! The default behaviour in OpenMP is `default(shared)`, which means that, by default, variables will be defined as shared. This can cause a lot of confusion, so it's best avoided.

- Inside the "shared" clause, we list all the variables that we want to be shared amongst our threads (these variables have to be initiated before the parallel region). Note that "cout" is a variable.

- To define private variables, it's best practice to simply initiate them inside the parallel region. You *can* initiate them beforehand, and then pass them into the parallel region using the "private" clause, but this can cause confusion.

For loop:

- Inside a parallel region, we can initiate a parallelised for loop by using the "for" directive. This directive is placed directly before a regular C++ for loop. It's usage will be something like:

  ```
  #pragma omp for [clauses]
  ```

```
for (int i=0; i<N; i++) {
    [code]
}
```

- At the end of a parallel for loop, there is an implicit "barrier", which means that the threads will wait for each other before they proceed. You can prevent this barrier by adding the "nowait" clause.
- The parallel for loop will split up the iterations as equally as possible between the threads. For example, if you run a parallel for loop with 7 iterations over 3 threads, the workload will most likely be split as $3 + 3 + 1$ iterations (although this is compiler dependent).
- Note that if you create a for loop inside a parallel region without the "for" clause, the whole loop will be run independently by each thread.
- To know whether a loop is parallelisable, a good rule of thumb is to ask: will I get the same answer if I run it in reverse order? If yes, it should be parallelisable.

Reduction clause:

- Both the "parallel" and "for" directives can take the "reduction" clause, which has a format like:

  ```
  #pragma omp parallel reduction(op:A)
  ```

- This clause tells OpenMP to create a private version of variable A in each thread (initialised to zero), and at the end of the parallel region, reduce all the private A variables into a single shared A variable, using the operation "op".
- The available reduction operations are:

  ```
  +, -, *, &, |, ^, &&, ||
  ```

- Note that A must be defined before the reduction operation begins, and the initial value is included in the reduction. e.g. the result of the code:

  ```
  int A = 5;
  #pragma omp parallel default(none) reduction(+:A)
  {
      int A = 1;
  }
  ```

  running on four threads will be `A=5+4=9`.

- You can reduce multiple variables simultaneously as well as use multiple types of reduction simultaneously, e.g. `reduction(+:A,B,C) reduction(*:D)`

Synchronisation:

- `#pragma omp barrier`: if you include this line in a parallel region, a "barrier" is created: no threads pass this point until all threads have reached it.
- `#pragma omp master`: anything inside this region will *only* be run by the master thread.
- `#pragma omp critical`: only one thread can enter a critical region at a time. Useful for updating shared variables—avoids race conditions!

Useful functions:

- `omp_get_num_threads()` returns the number of threads that are currently in existence.
- `omp_get_thread_num()` returns the index of the current thread (0, 1, 2, etc.).
- `omp_get_wtime()` returns the current time. This is good for timing your code, e.g.:

```
// Start of code
double t1 = omp_get_wtime();
...
// End of code
double t2 = omp_get_wtime();
cout << "Elapsed time (seconds): " << (t2 - t1) << endl;
```

2.6 Random number generation using OpenMP

When generating random numbers with OpenMP enabled, we have to be a bit careful to avoid race conditions. Pseudorandom number generators (PRNGs) are inherently serial—each time they are called, they generate the next number in their 'random' sequence. This means that if multiple threads call the PRNG at the same time, the PRNG will be at the same point in the sequence, and the threads will all be given the same random number from the generator.

The solution is to define a `thread_local` instance of the PRNG (i.e. a separate instance on each thread, each with its own seed).

An example script, `randomOpenMP.cpp`, is provided to demonstrate the issue, as well as how to define a `thread_local` random number generator avoid it. The script goes through four examples:

1. Incorrect: generate random numbers in parallel, with the race condition.
2. Incorrect: Initiate a `thread_local` PRNG, incorrectly. (Because the PRNG is initiated outside of a parallel region, it is only defined on the master thread.)
3. Correct: initiate a `thread_local` PRNG correctly (inside a parallel region).
4. Correct, cleanest method: define a function 'my_generator' that can be called within the parallel region. The first time it is called, it initiates a `thread_local` PRNG, and then outputs a random number. Each subsequent time it is called, it just outputs a random number.

3 Tasks

Exercise – Hello world
An example script `helloOpenMP.cpp` is provided. You can run this on `smp-teaching`.

- Compile with

  ```
  g++ helloOpenMP.cpp -o helloOpenMP -fopenmp
  ```

- Before running, set the number of threads to 4 in the terminal with `export OMP_NUM_THREADS=4`.
- Have a look at the code—there are comments explaining what each line does.
- Run the code again a few times, and look at the output. You should see that you get a different result every time. This is because the thread scheduling will turn out differently each time.
- You can also try running the code sequentially by setting `export OMP_NUM_THREADS=1` in the terminal, and running again. Alternatively, just compile without the `-fopenmp` flag.

Exercise – Hello world, with synchronisation
We can clean up the output of our "hello world" code by adding a few synchronisation steps. Set the number of threads to 4 again.

- Firstly, we notice that the statement "`if (ntid == 0)`" in the code is equivalent to using the directive 'master'. Replace this if statement with a master directive, and run again. You should see the same output. Be careful to put the braces, { and }, in the right place!
- Now, we want to prevent our threads from interfering with each other when they're trying to print "hello world from thread...". To do this, add a 'critical' directive around the printing line to ensure that only one thread runs this line of code at a time. Compile and run again a few times—you should see that the output looks a bit cleaner now.
- But our thread 0 will usually print the "Number of threads..." line before our other threads have finished printing their "hello world..." lines. To fix this, add a 'barrier' directive after the "hello world..." line so that all threads wait for each other to synchronise before any threads continue beyond that point. (Important: don't put the barrier inside the master directive!)

You should now see that the output is a lot cleaner, and only the order in which the threads print their "hello world" is non-deterministic.

Exercise – Parallel for loops
The script "sumOpenMP.cpp" provides an example of a for loop that has been parallelised using OpenMP. The script counts the number of iterations that are being performed by each thread, and prints out the result. If you compile and run it, you should see that the workload has been fairly evenly divided between among the threads—OpenMP has done this for us!

- Have a look at the behaviour when you use a number of iterations that is not a multiple of the thread number.
- In this script, you should see four lines that involve the variable "A" that have been commented out. Un-comment them, and run the script a few times. What happens to the value of A? Can you explain the behaviour? (If you don't see anything strange happening, try increasing the number of iterations.)
- There are a few ways to fix this problem. We could e.g. wrap the line `A+=1` in a critical region, but this is not a good solution, because it will be very slow for large numbers of iterations (your

threads will constantly be waiting for each other, so you lose a lot of parallelism!) Instead, add a 'reduction' clause to the for loop for the variable A, using addition as the operation.

- If you like, compare how long the code takes to run with a critical region around A, instead of the reduction clause—you might need a few million iterations to see a significant difference between the two.

Exercise – Computing a 2D integral
The script "intOpenMP.cpp" contains a simple script for numerically calculating the integral

$$\int_0^1 \int_0^1 \frac{1}{x^2 + y^2 + 1} dxdy \approx 0.63951, \tag{1}$$

using $N = 10,000$ sampled points in each direction.

- Create a parallel region outside the for loops, and parallelise one of the for loops within the parallel region, using a reduction clause for the integral. Which loop should you parallelise? Does it matter?
- You can actually "collapse" the two loops into a single parallelised loop. This is done by adding a "collapse" clause to the parallel for loop (presuming you've parallelised the outer loop): `#pragma omp for collapse(n)`, where n is the number of loops being collapsed. The syntax to collapse two loops is:

  ```
  #pragma omp for collapse(2)
  for (i...) {
      for (j...) {
      }
  }
  ```

  Note that the loops have to be directly inside of each other for this to work. Think about the difference in parallelism with and without the "collapse" clause. Which do you expect to be faster? (Note: you won't need 'collapse' for the 2D Ising model.)

Exercise – Parallelising the 2D Ising model
Part 2.3 of the assignment asks you to parallelise your 2D Ising model code using OpenMP. This exercise will walk you through how to do this. This exercise assumes that you already have a working 2D Ising model code—if not, you are strongly encouraged to get that working before attempting this!

There are many ways in which the Monte Carlo code could be parallelised, so this is just one (tested) method. With any method, though, there's one really important thing to keep in mind: we want to avoid the possibility of having two (or more) different threads proposing Metropolis updates to neighbouring spins simultaneously. If this happens, we'll get race conditions, and the Metropolis rule for accepting/rejecting proposed spin flips will break down!

We can avoid this problem by having our threads sweep through the lattice in a particular pattern. One common method is to use a "checkerboard" update pattern, where threads all work in parallel on updating spins on the "black" squares of the checkerboard, before synchronising (with a barrier), and then starting to work on the "white" squares of the checkerboard. This pattern ensures that no two neighbouring spins can ever be flipped simultaneously.

However, the following method seems to give a more reliable speedup with OpenMP, so it's suggested you try this for the assignment:

- We start by generating an initial state, and calculating the energy and magnetisation.
- We first do a *parallel* loop over all even rows of the lattice ($i = 0, 2, 4, \ldots, L-2$). Each thread is assigned one of these single rows, and it (privately) loops over the spins in that row in sequence ($j = 0, 1, 2, \ldots, L-1$), performing the Metropolis update on each spin as it goes. For each accepted Metropolis step, the changes in energy and magnetisation ($\Delta\mathcal{E}$ and $\Delta\mathcal{M}$) must be calculated, and added to a (private) running total. When a thread finishes its row, it is assigned another row, etc. until the parallel loop over i is finished.
- All threads synchronise, i.e. they wait for each other to finish the even rows before continuing (this is automatically achieved by OpenMP's implicit barrier at the end of a parallel for loop).
- We then repeat the process with the odd rows ($i = 1, 3, 5, \ldots, L-1$).
- After a single sweep completed in this fashion, we combine the energy/magnetisation measurements that each thread has privately been keeping track of, and add them to shared values.

We then repeat the whole process a large number of times. (Think about why this method avoids race conditions when updating neighbouring spins.)

Below is some pseudo-code for this method:

```
[Generate your initial state]
[Calculate initial E, M - these should be shared]

#pragma omp parallel [clauses]
{
[Initiate thread_local PRNG (if using method 3 from sec 2.6)]
[Loop over lattice sweeps] {
    [Initiate private dE=0,dM=0]

    [Parallel loop over even rows, perform Metropolis updates,
    keep track of private dE, dM]

    [Parallel loop over odd rows, perform Metropolis updates,
    keep track of private dE, dM]

    [Critical region where each thread updates shared E, M
    with their private dE, dM]

    [Barrier to prevent any thread from starting the next sweep
    while other threads are still working on the current one]

    [Master directive for data output]
}
```

Some things to note:

- Each thread needs to have its own random stream, with its own initial seed. If all threads start with the same seed, they will all generate the same sequence of random numbers, which you definitely don't want!
- Your private E and shared E need different variable names (the same goes for M).

- You will definitely want some efficient functions/methods for calculating the ΔE and ΔM from a given accepted flip. See last week's worksheet if you missed how to do this.
- You can also reduce the Metropolis condition to a single 'if' statement for more efficiency.
- The workload will most efficiently divide among your threads if your lattice size divides evenly into your number of threads. If both numbers are powers of 2, you should get the maximum efficiency.
- To time your code, use the `omp_get_wtime()` function introduced above.
- Try running your code with different numbers of threads—you might find that there's an 'optimal' number (and it's not necessarily the largest!)
- You will probably see a better (relative) speedup for larger grids.