Introduction to OpenMP

Exercise Notes

Getting started

Change directory to the **/work** file system, i.e.

```
user@archer$ cd /work/y14/y14/guestXX/
```

Copy the tar file containing the code to your account and unpack it with the command

```
cp /home/z01/shared/UKOMP.tar .
tar -xvf UKOMP.tar
```

OpenMP on ARCHER

Compiling using OpenMP

The OpenMP compilers we use are the Cray compilers for Fortran 90 and C, both of which have OpenMP enabled by default. To compile an OpenMP code, simply:

Fortran (ftn): **ftn -o program program.f90**
C (cc): **cc -o program program.c**

Each directory containing the exercise source code also includes a makefile, so you can build the executable simple by typing:

```
make
```

Job Submission

You can compile and run OpenMP programs on the login node, but any timings will be unreliable. For doing timing runs you need to run on the compute nodes via the batch system. To do this, you should submit a batch job as follows, for example to run an executable called **program**:

You will find a generic batch script in the **Mandelbrot** directory called **ompbatch.pbs**

```
cp ompbatch.pbs program.pbs
```

Edit the **export OMP_NUM_THREADS=** line in **program.pbs** to specify the number of threads to use.

Submit the batch job using:

```
qsub -q Rnnnnnnn program.pbs
```

where **Rnnnnnnn** is the name of today's special reserved queue for the course, which the instructor will give you.

You can monitor your jobs status with the command **qstat -u $USER**

When the job has finished, you will find two new files in the directory you submitted the job from, containing the output and error massages (if any).

Exercise 1: Area of the Mandelbrot Set

The example code can be found in **`Mandelbrot/*/HelloWorld/`** where the **`*`** represents the language of your choice, i.e. **`C`** , or **`F`** .

The Mandelbrot Set

The Mandelbrot Set is the set of complex numbers c for which the iteration $z = z^2 + c$ does not diverge, from the initial condition $z = c$. To determine (approximately) whether a point c lies in the set, a finite number of iterations are performed, and if the condition $|z| > 2$ is satisfied then the point is considered to be outside the Set. What we are interested in is calculating the area of the Mandelbrot Set. There is no known theoretical value for this, and estimates are based on a procedure similar to that used here.

The Code

The method we use generates a grid of points in a box of the complex plane containing the upper half of the (symmetric) Mandelbrot Set. Then each point is iterated using the equation above a finite number of times (say 2000). If within that number of iterations the threshold condition $|z| > 2$ is satisfied, then that point is considered to be outside of the Mandelbrot Set. Then counting the number of points within the Set and those outside will give an estimate of the area of the Set.

Parallelise the serial code using the OpenMP directives and library routines that you have learned so far.

The method for doing this is as follows:

1. Start a parallel region before the main loop, nest making sure that any private, shared or reduction variables within the region are correctly declared.
2. Distribute the outermost loop across the threads available so that each thread has an equal number of the points. For this you will need to use some of the OpenMP library routines.

Once you have written the code try it out using 1, 2, 3 and 4 threads. Check that the results are identical in each case, and compare the time taken for the calculations using the different number of threads. Note: to get accurate times, submit a batch job: see the Appendix for how to do this.

Extra Exercise

Try different ways of mapping iterations to threads.

Exercise 2: Mandelbrot again

You can start from the code you have already, or another copy of the sequential code which can be found in **`OMP-exercises/*/Mandelbrot2/`**. This time parallelise the outer loop using a **`PARALLEL DO`** / **`parallel for`** directive. Don't forget to declare the shared, private and reduction variables. Add a **`SCHEDULE`** clause and experiment with the different schedule kinds.

Extra Exercise

Instead of using a reduction variable, try using an atomic update, a critical section, or lock routines to synchronise the accesses to **`numoutside`**.

Exercise 3: Traffic Modelling

Try parallelising the provided serial version of a traffic model. Parallelise the code that updates the road and the copy-back step, being careful how you classify the variables and arrays.

Leave the initialisation step as a serial routine: parallelising random number generators is quite difficult! The advantage of this is that your code should produce exactly the same answer in serial and parallel.