Shared-Memory Programming with 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

```
tar -xvf OMP-exercises.tar
```

Exercise 1: HelloWorld

This is a simple exercise to introduce you to the compilation and execution of OpenMP programs. The example code can be found in **`*/HelloWorld/`** where the **`*`** represents the language of your choice, i.e. **`C`** , or **`F`** .

Compile the code. Before running it, set the environment variable **`OMP_NUM_THREADS`** to a number n between 1 and 4 with the command:

```
export OMP_NUM_THREADS=n
```

When run, the code enters a parallel region at the **`!$OMP PARALLEL`** / **`#pragma omp parallel`** command. At this point n threads are spawned, and each thread executes the print command separately. The **`OMP_GET_THREAD_NUM()`** / **`omp_get_thread_num()`** library routine returns a number (between 0 and n-1) which identifies each thread.

Extra Exercise

Incorporate calls to **`omp_get_num_threads()`** into the code and print its value within and outside the parallel region.

Exercise 2: Area of the Mandelbrot Set

The aim of this exercise is to use the OpenMP directives learned so far and apply them to a real problem. It will demonstrate some of the issues that need to be taken into account when adapting serial code to a parallel version.

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 shall 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 3: Mandelbrot again

You can start from the code you have already, or another copy of the sequential code which can be found in `*/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.

Exercise 4: Traffic Modelling

The following exercises, 5 and 6, use a molecular dynamics simulation to illustrate various aspects of OpenMP. If you prefer tackling a simpler code, try parallelising the provided serial version of a traffic model. Things to do include:

- Add timing to the code.
- Parallelise the code that updates the road and the copy-back step, being careful how you classify the variables and arrays.
- Parallelise the update using step using orphaned directives.

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.

Exercise 5: Molecular Dynamics

The aim of this exercise is to demonstrate how to use OpenMP synchronisation constructs to parallelise a molecular dynamics code.

The Code

The code can be found in `*/MolDyn/` . The code is a molecular dynamics (MD) simulation of argon atoms in a box with periodic boundary conditions. The atoms are initially arranged as a face-centred cubic (fcc) lattice and then allowed to melt. The interaction of the particles is calculated using a Lennard-Jones potential. The main loop of the program is in the file `main.c / main.f90` . Once the lattice has been generated and the forces and velocities initialised, the main loop begins. The following steps are undertaken in each iteration of this loop:

1. The particles are moved based on their velocities, and the velocities are partially updated (call to `domove()`)

2. The forces on the particles in their new positions are calculated and the virial and potential energies accumulated (call to **`forces()`**)
3. The forces are scaled, the velocity update is completed and the kinetic energy calculated (call to **`mkekin()`**)
4. The average particle velocity is calculated and the temperature scaled (call to **`velavg()`**)
5. The full potential and virial energies are calculated and printed out (call to **`prnout()`**)

Parallelisation

The parallelisation of this code is a little less straightforward. There are several dependencies within the program that will require use of **`critical`** or **`atomic`** constructs as well as the **`reduction`** clause. The instructions for parallelising the code are as follows:

1. Edit the subroutine/function in **`forces.c`** / **`forces.f90`**. Start a **`!$OMP PARALLEL DO`** **`/#pragma omp parallel for`** for the outer loop in this subroutine, identifying any private or reduction variables. Hint: There are 2 reduction variables.
2. Identify the variable within the loop which must be updated atomically and use **`!$OMP CRITICAL`** / **`#pragma omp critical`** to ensure this is the case.

Once this is done the code should be ready to run in parallel. Compare the output using 2, 3 and 4 threads with the serial output to check that it is working. Try adding a **`schedule (static,`** ***`n`*****`)`** clause to the parallel loop for different values of ***`n`*** . Does this have any effect on performance? Also try using atomic constructs instead of criticals.

Exercise 6: Molecular Dynamics Part II

Following on from the previous exercise, we shall update the molecular dynamics code to take advantage of orphaning and examine the performance issues of using the critical construct. You can either work with the version you have already written, or start from the version in **`*/MolDyn2`** .

Orphaning

You can change the **`PARALLEL DO`** / **`parallel for`** directive to a **`DO`** / **`for`** directive and start the parallel region outside the call to **`forces()`** in in **`main.c`** / **`main.f90`**. Make sure that any reduction variables updated in the parallel region, but outside of the **`DO`** / **`for`** construct, are updated by one thread only. As before, check your code is still working correctly. Is there any difference in performance compared with the code without orphaning?

Extra exercise

In Fortran, you can use a reduction clause for **`f`** instead of critical/atomic. In C, you can achieve the same effect by using a shared temporary array with an extra dimension, indexed by the thread number, to accumulate partial forces on each thread, and then sum these into **`f`** afterwards. Compare the performance with the versions using critical or atomics.

Exercise 7: Mandelbrot with tasks

Redo the Mandelbrot exercise using OpenMP tasks. To begin with, make the computation of each point a task, and use one thread only to generate the tasks. Once this is working, measure the performance. Note that reduction variables cannot be used inside tasks.

Now modify your code so that it treats each row of points as a task. Modify your code again, so that all threads generate tasks. Which version performs best? Is the performance better or worse than using a loop directive?

Appendix: 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`**

Job Submission

Batch processing is important, as it is the only way of accessing the main compute nodes. For doing timing runs you must use these. To do this, you should submit a batch job as follows, for example using 4 threads to run an executable called **`program`**:

You will find a generic batch script in the **`HelloWorld`** 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).