|epcc|

Shared-Memory Programming with OpenMP
Exercise Notes

Getting started

Copy a tar file containing the code from the course web pages to your account and unpack it with the command

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

Note that you should change directory to the `/work` file system, i.e.

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

Exercise 1: Hello World

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, making sure you use the appropriate flag to enable OpenMP. 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 a call to `omp_get_num_threads()` into the code and print its value within and outside of 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 which 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.

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 the traffic model example altready discussed in the lectures. Things to do include:

- Add timing to the code.
- Parallelise the code that updates the road, being careful how you classify the variables and arrays.
- How does the performance vary if you do / do not parallelise the "copy-back" step?
- use a single parallel region with multiple `!$OMP DO` / `#pragma omp for` directives.
- Put the update step in a function / subroutine and parallelise using orphaned directives.

Exercise 5: Molecular Dynamics

The aim of this exercise is to demonstrate how to use OpenMP critical directives 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.[f|c|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 which will require use of the CRITICAL/critical directive as well as the REDUCTION/reduction clause. The instructions for parallelising the code are as follows:

1. Edit the subroutine/function `forces.[f|c|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 the schedule clause with the option `static,n` to the `DO/for` directive for different values of `n`. Does this have any effect on performance?

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` directive. You can either work with the version you have already written, or start from the version in `*/MolDyn2`.

Orphaning

To reduce the overhead of starting and stopping the threads, you can change the `PARALLEL DO`/`omp parallel for` directive to a `DO/for` directive and start the parallel region outside the main loop of

the program. Edit the `main.[f|c|f90]` file. Start a `PARALLEL`/`parallel` region before the main loop. End the region after the end of the loop. Except for the `forces` routine, all the other work in this loop should be executed by one thread only. Ensure that this is the case using the `SINGLE` or `MASTER` directive. Recall that any reduction variables updated in a parallel region but outside of the `DO`/`for` directive should be 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?

Multiple Arrays

The use of the `CRITICAL` directive is quite expensive. One way round the need for the directive in this molecular dynamics code is to create a new temporary array for the variable `f`. This array can then be used to collect the updated values of `f` from each thread, which can then be "reduced" back into `f`. However, arrays cannot be declared as reduction variables, so this reduction must be coded into the program.

1. Edit the `main.[c|f90]` file. Create an array `ftemp` of size `(npart, 3, 0:MAXPROC-1)` for Fortran, `[3*npart][MAXPROC]` for C, where `MAXPROC` is the maximum number of threads available to the program. This is the array to be used to store the updates on `f` from each thread.
2. Within the parallel region, create a variable `nprocs` and assign to it the number of threads being used.
3. The variables `ftemp` and `nprocs` and the parameter `MAXPROC` need to be passed to the `forces` routine. (*N.B. In C it is easier, but rather sloppy, to declare* `ftemp` *as a global variable*).
4. In the `forces` routine define a variable `my_id` and set it to the thread number). This variable can now be used as the index for the last dimension of the array `ftemp`. Initialise `ftemp` to zero.
5. Within the parallelised loop, remove the `CRITICAL` directive and replace the references to the array `f` with `ftemp`.
6. After the loop, the array `ftemp` must now be reduced into the array `f`. Loop over the number of particles (`npart`) and the number of threads (`nprocs`) and sum the temporary array into `f`

The code should now be ready to try again. Check its still working and see if the performance (especially the scalability) has improved.

Extra exercise (Fortran only)

Use a reduction clause for `f` instead of the temporary arrays. Compare the performance of the two versions.

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. 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 that using a loop directive?

Appendix: OpenMP on ARCHER

Hardware

For this course we will compile and run code on ARCHER, the UK national supercomputer service.

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`

Using a Makefile

The Makefile below is a typical example of the Makefiles used in the exercises. We do not include optimisation flags for the Cray compilers as the default optimisation level is already quite high.

However, you could add `-O3` which is a standard optimisation flag for speeding up execution of the code.

```
## Fortran compiler and options
FC=     ftn
## Object files
OBJ=    main.o \
        sub.o
## Compile
execname:       $(OBJ)
                $(FC) -o $@ $(OBJ)
.f.o:
                $(FC) -c $<
## Clean out object files and the executable.
clean:
        rm *.o execname
```

Job Submission

Batch processing is very important as it is the only way of accessing the back-end systems (the compute nodes).

For doing timing runs you must use the back-end. To do this, you should submit a batch job as follows, for example using 4 threads on `program`:

```
cp ompbatch.pbs program.pbs

qsub -q RXXXXXXX program.pbs
```

where `RXXXXXXX` is the name of the special reserved queue for the course, and `program.pbs` is a shell script containing the line:

```
export OMP_NUM_THREADS=4
```

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