Shared Memory Programming with OpenMP - Exercise Notes

OpenMP at EPCC

Hardware

For this course you will be compiling and running the code on Cirrus: you should connect to **`login.cirrus.ac.uk`**

Linux and Mac users can use the ssh command:

`ssh -Y username@login.cirrus.ac.uk`

Windows users should install MobaXterm – please see the course web page for instructions.

Compiling using OpenMP

The Intel compilers (icc/ifort) are available on Cirrus. To access them type

`module load intel-compilers-17`

To compile an OpenMP code, simply add the flag **`-qopenmp`**

You can also use the GNU compilers (gcc/gfortran) instead: to access an up-to-date version that supports the latest OpenMP features, type

`module load gcc`

For the GNU compilers the OpenMP flag is **`-fopenmp`**

Using a Makefile

The Makefile below is a typical example of the Makefiles used in the exercises. The option **`-O3`** is a standard optimisation flag for speeding up execution of the code.

```
## Fortran compiler and options
FC= ifort -O3 -qopenmp
## 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
```

To build the code, just type **make**, and to remove all object and executable files, type **make clean**

Job Submission

Batch processing is very important as it is the only way of accessing the compute nodes on Cirrus. Interactive access is not allowed. For doing accurate timing runs you *must* use the compute nodes. To do this, you should submit a batch job as follows.

```
qsub myjob.pbs
```

where **myjob.pbs** is the shell script supplied with each exercise. You will need to modify the line

```
#PBS -A tc002-username
```

in the script to use the budget code corresponding to your own username on Cirrus.

To change the number of threads, edit the script and change the value assigned to the **OMP_NUM_THREADS** variable. To run on different numbers of threads you can reassign this variable and re-run the executable multiple times in the same batch script. You can monitor your job's status with the **qstat** command and jobs can be deleted with **qdel**.

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 **Fortran90**.

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** directive. 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 inside 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 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 Mandelbrot 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 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 lead to 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. Modify the bounds of the outermost loop so that each thread has an equal number of the points. To calculate the loop bounds for each thread you will need to use **`omp_get_thread_num()`** and **`omp_get_num_threads()`**

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 different number of threads.

Extra Exercise

Is your solution well load balanced? 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 use **default(none)** and declare the shared, private and reduction variables. Add a schedule clause and experiment with the different schedule kinds.

Exercise 4: Molecular Dynamics

The aim of this exercise is to demonstrate how to use OpenMP critical 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 which will require use of the critical construct 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**. Add a **!$OMP PARALLEL DO / #pragma omp parallel for** directive to the outermost loop in this subroutine, identifying any private or reduction variables.
Hint: There are 2 reduction variables.
2. Identify the variable within the loop which must be protected from possible race conditions and use **!$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 kind **`static,n`** to the **`DO/for`** directive for different values of n . Does this have any effect on performance?

Exercise 5: Molecular Dynamics Part II

Following on from the previous exercise, we will update the molecular dynamics code to take advantage of orphaning and examine the performance issues of using critical regions.

Orphaning

To reduce the overhead of starting and stopping the threads, you can change the **`PARALLEL DO`** / **`parallel for`** directive to a **`DO/for`** directive and start the parallel region outside the main loop of the program. In **`main.c`** / **`main.f90`** enclose the main loop in a parallel region. 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 construct. Recall that any reduction variables updated in a parallel region but outside of the **`DO/for`** construct 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?

Per-thread Arrays

The critical region becomes a bottleneck for sufficiently large thread counts. One way round the need for the directive in this molecular dynamics code is to create a new per-thread temporary array for the variable **`f`** . This array can then be used to store the updates to **`f`** from each thread, which can then be added back into **`f`** itself. In Fortran **`f`** can be declared as a reduction variable.
However, in C, arrays cannot be declared as reduction variables, so this reduction must be coded by hand:

1. Skilled C programmers can dynamically allocate an array **`ftemp`** with dimensions **`[3*npart][omp_get_max_threads()]`** and pass this into the forces routine. An easier and dirtier method is to declare a global array of dimensions **`[3*npart][MAXTHREADS]`**, where **`MAXTHREADS`** is a **`#define`**d constant.
2. Within the parallel region, create a variable **`nthreads`** and assign to it the number of threads being used.
3. 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`** .
4. Initialise **`ftemp`** to zero.
5. Within the parallelised loop, remove the critical regions 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 (**`nthreads`**) 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

Try using atomic directives or lock routines (with one lock variable per particle) instead of critical regions.

Exercise 6: Nested parallelism

Now parallelise *both* outer loops in the Mandelbrot example using nested parallel regions. To enable nested parallelism use

```
export OMP_NESTED=true
```

and set the number of threads at each level with, for example,

```
export OMP_NUM_THREADS=2,4
```

You will need to think carefully about the data scoping (i.e. shared/private/reduction). Experiment with different thread numbers – is the performance ever better than with one level of parallelism?

You can also try parallelising both loops using a parallel loop directive on the outer loop and a collapse clause.