|epcc|

Shared Memory Programming
Exercise Notes

Adrian Jackson

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 ~/OpenMP/*/HelloWorld/ where the * represents the language of your choice, i.e. C, Fortran or Fortran90. The code is shown below (you do not need to type it in!).

Fortran:

```
      program helloworld
      implicit none
      integer OMP_GET_THREAD_NUM


!$OMP PARALLEL
      print *,'Hello from thread number',OMP_GET_THREAD_NUM()
!$OMP END PARALLEL

      end
```

Fortran90:

```
program helloworld
  use omp_lib
  implicit none


!$OMP PARALLEL
  print *,'Hello from thread number',OMP_GET_THREAD_NUM()
!$OMP END PARALLEL

end program helloworld
```

C:

```
#include <stdio.h>
#include <omp.h>

int  main(){
    #pragma omp parallel
      printf("hello from thread %d\n",omp_get_thread_num());
  }
```

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

```
setenv 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 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. We will use Monte Carlo sampling to calculate a numerical solution to this problem.

N.B. Complex numbers in C: Complex arithmetic is supported directly in Fortran, but in C it must be done manually. For this exercise the template has a struct complex declared. Then for a given complex number z, `z.creal` *represents the real part and* `z.cimag` *the imaginary part. Then assigning* $z = z^2 + c$ *is given by:*

```
ztemp=(z.creal*z.creal)-(z.cimag*z.cimag)+c.creal;
z.cimag=z.creal*z.cimag*2+c.cimag;
z.creal=ztemp;
```

using `double ztemp` *as temporary storage for the updated real part of* z*. Also, for the threshold test* $|z| > 2.0$*, use* `abs(z)>2.0` *in Fortran, and in C use:*
`z.creal*z.creal+z.cimag*z.cimag>4.0`.

The Code : Sequential

The Monte Carlo method we shall use generates a number of random points in the range $[(-2.0, 0), (0.5, 1.125)]$ of the complex plane. Then each point will be iterated using the equation above a finite number of times (say 100000). 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 random points within the Set and those outside will reveal a good approximation of the area of the Set.

The template for this code can be found in `~/OpenMP/*/Mandelbrot/`. Implement the Monte Carlo sampling, looping over all `npoints` points. For each point, assign `z=c(i)` and iterate as defined in the earlier equation, testing for the threshold condition at each iteration. If the threshold condition is not satisfied, i.e. $|z| \leq 2$, then repeat the iteration (up to a maximum number of iterations, say 100,000). If after the maximum number of iterations the condition is still not satisfied then add one to the total number of points inside the Set. If the threshold condition is satisfied, i.e. $|z| > 2$ then stop iterating and move on to the next point.

Parallelisation

Now 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 Monte Carlo iteration loop, making sure that any private, shared or reduction variables within the region are correctly declared.
2. Distribute the `npoints` random points across the n 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.

   N.B.: there is no ceiling function in Fortran, so to calculate the number of points on each thread (`nlocal`*) compute* `(npoints+n-1)/n` *instead of* `npoints/n`*. Then for each thread the iteration bounds are given by:*

   ```
   ilower = nlocal * myid + 1
   iupper = min(npoints, (myid+1)*nlocal)
   ```

   where `myid` *stores the thread identification number.*
3. Rewrite the iteration loop to take into account the new distribution of points.
4. End the parallel region after the iteration loop.

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

Write out which iterations are being performed by each thread to make sure all the points are being iterated.

Exercise 3: Image

The aim of this exercise is to use the OpenMP worksharing directives, specifically the `PARALLEL DO`/`parallel for` directives.

This code reconstructs an image from a version which has been passed through a a simple edge detection filter. The edge pixel values are constructed from the image using

$$edge_{i,j} = image_{i-1,j} + image_{i+1,j} + image_{1,j-1} + image_{i,j+1} - 4\, image_{i,j}$$

If an image pixel has the same value as its four surrounding neighbours (i.e. no edge) then the value of $edge_{i,j}$ will be zero. If the pixel is very different from its four neighbours (i.e. a possible edge) then $edge_{i,j}$ will be large in magnitude. We will always consider i and j to lie in the range $1, 2, \ldots M$ and $1, 2, \ldots N$ respectively. Pixels that lie outside this range (e.g. $image_{i,0}$ or $image_{M+1,j}$) are simply considered to be set to zero.

Many more sophisticated methods for edge detection exist, but this is a nice simple approach.

The exercise is actually to do the reverse operation and construct the initial image given the edges. This is a slightly artificial thing to do, and is only possible given the very simple approach used to detect the edges in the first place. However, it turns out that the reverse calculation is iterative, requiring many successive operations each very similar to the edge detection calculation itself. The fact that calculating the image from the edges requires a large amount of computation, makes it a much more suitable program than edge detection itself for the purposes of timing and parallel scaling studies.

As an aside, this inverse operation is also very similar to a large number of real scientific HPC calculations that solve partial differential equations using iterative algorithms such as Jacobi or Gauss-Seidel.

Parallelisation

The code for this exercise can be found in `~/OpenMP/*/Image/`.
The file `notlac.dat` contains the initial data.

The code can be parallelised by adding `PARALLEL DO`/`parallel for` directives to the three routines `new2old`, `jacobistep` and `residue`, taking care to correctly identify shared, private and reduction variables. *Hint*: use `default(none)`. Remember to add the `-mp` compiler flag. You can view the final image using `display`, though this is a poor test for correctness: better to compare the final residue value to the sequential code.

Use the script `image.sct` to submit to the batch queue and record the execution time on up to 8 threads, and calculate the speedup and efficiency achieved. Try to answer the following questions:

- Where is synchronisation taking place?
- Where is communication taking place?

Extra exercise

Redo Exercise 2 using worksharing directives.

Extra exercise (Fortran only)

Implement the routine `new2old` using Fortran 90 array syntax, and parallelise with a `PARALLEL WORKSHARE` directive. Compare the performance of the two versions.

Exercise 4: Load imbalance

This exercise will demonstrate the use of the `SCHEDULE`/`schedule` clause to improve performance of a parallelised code.

The Goldbach Conjecture

In the world of number theory, the Goldbach Conjecture states that:

Every even number greater than two is the sum of two primes.

In this exercise you are going to write a code to test this conjecture. The code will find the number of Goldbach pairs (that is, number of pairs of primes p_1, p_2 such that $p_1 + p_2 = i$) for $i = 2, \ldots, 8000$. The computational cost is proportional to $i^{3/2}$, so for optimal performance on multiple threads the work load must be intelligently distributed using the OpenMP `SCHEDULE`/`schedule` clause.

The Code

The template for the code can be found in ∼/OpenMP/*/Goldbach/.

1. Initialise the array `numpairs` to zero.
2. Create a `do/for` loop which loops over `i` and sets each element of the array `numpairs` to the returned value of a function `goldbach(2*i)`. This array will contain the number of Goldbach pairs for each even number up to 8000.
3. Once the number of Goldbach pairs for each even number has been calculated, test that the Goldbach conjecture holds true for these numbers, that is, that for each even number greater than two there is at least one Goldbach pair.

Your output should match these results:

Even number	Goldbach pairs
2	0
802	16
1602	53
2402	37
3202	40
4002	106
4802	64
5602	64
6402	156
7202	78

Table 1: Example output for Goldbach code

Parallelisation

Parallelise the code using OpenMP directives around the do/for loop which calls the `goldbach` function. Validate that the code is still working when using more than one thread, and then experiment with the schedule clause to try and improve performance.

Extra Exercise

1. Print out which thread is doing each iteration so that you can see directly the effect of the different scheduling clauses.
2. Try running the loop in reverse order, and using the `GUIDED` schedule.