School of Computing, University of Leeds

COMP3221
Parallel computation

Worksheet 1: Shared memory parallelism with OpenMP

This worksheet will provide you with an opportunity to practice the key concepts in shared memory parallelism (SMP), relevant to multi-core CPUs, using OpenMP and C. Most of the questions closely follow the worked examples used in the lectures.

Compilation

To compile C code with OpenMP using the `gcc` compiler on the School machines:

```
gcc -fopenmp -Wall -o <executable_name> <source_code.c>
```

See the end of Lecture 2 for further instructions.

Questions

1. *[Lecture 3]* `saxpy` is a numerical vector operation somewhat similar to vector addition. It stands for '*a times* **x** *plus* **y**,' and the **s** denotes all arrays and variables are single-precision (*i.e.* `float`). Given two n-vectors $\mathbf{x}$ and $\mathbf{y}$, and a scalar a, the `saxpy` operation is

   $$a\mathbf{x} + \mathbf{y} \to \mathbf{y}$$

   In serial C, this would be written: `for(i=0;i<n;i++) y[i] = a*x[i] + y[i];`

   Write a parallel code for `saxpy` using OpenMP. You may like to base your solution on the `vectorAddition_parallel.c` code that accompanied Lecture 3, but your code should use dynamically allocated vectors using `malloc` and `free`. Include a test in your program. Also include some text messages for small n using `omp_get_thread_num()` and `omp_get_max_threads()`, to convince yourself it really is running in parallel.

2. *[Lecture 3]* Download `Mandelbrot.c` and `makefile` from Lecture 3 and try parallelising first the inner then the outer loop as described in that lecture. Now trying parallelising **both** loops. Do you see any increase in speed? Add a `printf` statement to show `omp_get_thread_num()` during the loop. Is this what you expect?

   This example demonstrates that OpenMP does not support nested `#pragma omp parallel for` *directives, so the nested loop is left in serial. If you wanted to parallelise both loops, you would have to use other OpenMP features, such as the* `collapse` *clause.*

3. *[Lecture 4]* For the following numbers of processing units p and non-parallelisable fraction f, predict the parallel speedup and efficiency according to (a) Amdahl's law, (b) the Gustafson-Barsis law.

(i) $p = 2$ and $f = 0.1$.

(ii) $p = 17$ and $f = 0.2$.

(iii) $p = \infty$ and $f = 0.05$.

4. *[Lecture 5]* Download `work1_q4.c` from the Minerva page for this worksheet. This serial code is somewhat similar to the `shiftDependency.c` discussed in the lecture, except (a) the dependency is reversed (so `a[i]=a[i-1]`), and (b) it is cyclic (so `a[0]=a[n-1]`). Convert this code to parallel and check your answer agrees with the serial version.

5. *[Lecture 6]* Download the code `work1_q5.c` from Minerva. This is a serial singly-linked list similar to that covered in the lecture, except that it also supports removal of items. Parallelise the loop that adds items to the list, and add critical regions until it is thread safe, following the lecture notes. Now parallelise the removal loop, and note that for large lists this can cause memory leaks. Enclose the code in `popFromList()` within a critical region to ensure this does not happen (do not try to break it down into smaller critical regions, unless you are feeling brave . . .). You are not expected to ensure thread safety under simultaneous addition *and* removal of items.

6. *[Lecture 6]* Download the code `work1_q6.c` from Minerva, and understand how it counts the number of integers in a given range that has a certain divisor. Parallelise the main loop, and see that it now gives the wrong answer. Use a critical region to correct this problem, but observe how much this slows it down. Replace the critical region with a single atomic, and check this still gives the correct answer while being faster than the critical region version.

7. *[Lecture 7]* Download the code `nestedCriticalRegions.c` that was covered in Lecture 7, in which the same lock employed for nested critical regions produces deadlock. Change the code to (a) use different locks for each level of nesting, and (b) use named critical regions, and show that both approaches solve the deadlock problem. For (b), show that using named critical regions *with the same name* compiles but deadlocks.