COMP35112 Laboratory Exercise 2: Mergesort
Duration: 2 weeks

Learning outcomes

On successful completion of this exercise, a student will have (1) written a multithreaded mergesort program, and (2) run it on a multicore machine and measured its performance when executing on different numbers of active cores.

Introduction

Sorting is a compute-intensive activity and mergesort is a sorting algorithm that can be fairly easily parallelised. So, this exercise aims to implement a parallel version of mergesort and evaluate its performance on the multicore machine mcore48. All development of code should be done on a normal teaching domain machine, so that usage of the multicore machine is minimised.

Algorithm

Some pseudocode is given below:

```
int[] mergesort(int[] inArray, int start, int end, int numThreads)
{
if (numThreads > 1)
   {
   spawn a new thread to sort the first half ...
     res1 = mergesort(inArray, start, (start+end)/2, numThreads/2);
   meanwhile continue sorting the other half ...
   res2 = mergesort(inArray, (start+end)/2 + 1, end, numThreads/2);
   join with spawned thread;
   generate result by merging res1 with res2;
   }
else
   {
   generate result between start and end (inclusive) using
                        a sequential sort method on inArray;
   }
return result; }
```

Note that the result of the sort should be placed back in `inArray`, that is, the parallel half sorts should modify the same (shared) copy of `inArray`. This is possible because the two half sorts are completely independent of one another. The recommended sequential sort method is that provided by `Java.utils.Arrays`. It is suggested that the numbers to be sorted should be alternately taken from an ascending sequence starting at 1 and a descending sequence starting at n/2, where n is the length of `inArray` Use `System.nanoTime()` to record the times (in nanoseconds) before the sorting commences and after it has been completed, and hence print the time needed to perform the computation.

Fully develop this application on a normal teaching domain machine before running it on the multicore machine.

Speedup

To demonstrate speedup without interference from other students, you will need to use the OGE batch system, as described in the labsheet for Exercise 1. A script `Demo.sct` for OGE that will run the `main` method in a Java class called `Demo` is available on the course unit materials webpage. You should run with a range of numbers of threads (in this case, all powers of 2) to observe how the performance changes. Use `gnuplot` (or some other means) to generate a performance curve from the observed timings showing how performance speeds up with the number of active threads. Make each execution time measurement more than once in order to deal with variation in observations.

Deliverables

The main deliverable for this exercise is the performance graph(s) showing how performance varies with the number of active threads/cores. Ideally, this should show performance increasing proportionally as the number of threads/cores increases (at least up to 8 threads/cores). If the performance curve shows a different trend, you should think of reasons why this might happen. Possible reasons include overheads associated with making the computation parallel and effects due to the parallel algorithm.

Submission

Write a **brief** report on this exercise, commenting on the speedup that your performance curve shows (and the reasons it is not ideal) and stating what you had to do in order to get this speedup. Submit this report together with the performance curve and your source code in a single PDF or ZIP file via the assignment window on the Blackboard course page. The deadline for submission is one week after the final scheduled lab session for this exercise.

Marking Scheme

This exercise is marked out of 20 marks, as follows:
Code – 9 marks
Report – 5 marks
Performance graph(s) – 6 marks