COMP35112 Laboratory Exercise 1: Delivering Speedup
Scheduled duration: 2 weeks

Learning outcomes

On successful completion of this exercise, a student will have: (1) *written* a simple multithreaded program; (2) *predicted* the expected behaviour of the program when run on different numbers of active cores; (3) run the program on a multicore machine and *measured* its performance when executing on different numbers of threads and active cores; and (4) *reflected and acted upon* any differences found between items (2) and (3), above, in order to demonstrate speedup as the number of threads deployed is increased.

Introduction

The purpose of this exercise is to demonstrate that the performance of a suitably designed, simple multithreaded program can be improved by running it on a multicore computer. The multicore machine to be used is called mcore48, details of which appear below. Please take care to follow the instructions given below when running your programs.

All development of code should be done on a normal teaching domain machine, so that usage of the multicore machine is minimised.

The mcore48 multicore platform

The execution platform for these labs is `mcore48.cs.man.ac.uk`, a Dell/AMD quad 12-core system which has a total of 48 cores, up to 12 (i.e. a single 12 core processor) of which will be used to obtain speedup in execution (initially in this lab), and which runs Linux.

You should use `ssh mcore48.cs.man.ac.uk` to login to the multicore platform from a normal teaching domain machine. Please be aware that this will not be accessible to you until the third week of the Semester. When you login, you should use your usual university username and password. You should then see your usual home directory. If anything goes wrong at this stage, please advise the lab supervisor.

Once you are logged in, you need to enable certain software by 'sourcing' a script that sets up the required environment variables. You can achieve this by executing the following command after logging in (or include the command in your `.profile` file) – please take care when typing this command, all of the syntax is significant (ensure the initial "dot space /" is adhered to):

```
. /local/COMP60611/utils/mcore48_vars.sh
```

The multicore platform does have a Java compiler, but you should do all code development on a normal teaching domain machine to minimise activity on that

machine so as not to affect performance results. Only when you have a viable class file do you need to execute it on mcore48. Because you will want to be able to repeat observations of execution times, you need to use a batch queue system for submitting jobs for execution. The provided batch queue system is called OGE (for Open Grid Engine). *Failure to use OGE will result in unreliable timing results, not only for you, but also for anyone else who is currently running a program under OGE*. To submit a program to OGE, use the command `qsub myprog.sct`, where `myprog.sct` is a script which runs your program. **An example script is provided in the course unit materials webpage** (also see below). To see all the jobs in the batch queue, use the command `qstat -u '*'`. Jobs that are waiting will have state `qw`. Jobs that are running will have state `r`. Just before a job runs, it will for a short time be in state `t` (meaning "transfer"); please do not remove a job while it is in (or is about to be in) this state, because it can stall the queue (see below). When the script has finished running, the output of your program will appear in a file named `myprog.sct.oXXXX`, where `XXXX` is a unique job number for each submitted job. Please do not abuse the platform by avoiding the use of OGE.

If you see the batch queue in a stalled state (usually there will be no jobs running, and one job in state `dt` which never disappears), please contact the lab supervisor.

Application

The idea is to implement a relatively simple application which imposes minimal constraints on the parallelism that is available. Vector addition (adding together the corresponding elements of two vectors) has been chosen as a suitable candidate.

Your task is to write a Java program which takes two command line arguments; the first argument is the number of threads to be used, and the second is the length of the vectors to be added together. In order to simplify the code that is executed, **use Java arrays** (not objects of the `Vector` class) to hold the vector data. Initialise the two vectors with arbitrary values and, when creating threads, pass them references to these vectors and to the vector that will contain the result. Each thread should generate a **contiguous** block of the result, indicated by two integer indices (the start index and the end index of the block) that are also passed when the thread is created. Use `System.nanoTime()` to record the times (in nanoseconds) before the threads start and after they all return, and hence print the time needed to perform the computation. When plotting your results, you should plot performance, rather than time, against thread/core count. The simplest definition of performance is the inverse of the execution time (i.e. the number of times the program can be executed in a second). Your code should print this as well. Ensure that results are printed to an appropriate accuracy (3 or 4 decimal places only).

Fully develop this application on a normal teaching domain machine before running it on the multicore machine. 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. If you edit this script, *do not remove the apparent comments at the start*; these are necessary OGE commands that set the job up in the proper fashion.

Correctness

You should take steps to convince yourself that your program is properly doing the job it was supposed to. One thing that is easy to get wrong is to fail to sum together *all* the pairs of elements in the vectors. Typically this can go wrong when the number of elements in the vectors is not exactly divisible by the number of threads being used. Other mistakes have been seen.

Speedup

Before you do any experiments, think about what you are expecting to see happen to performance when you change the number of threads being used to execute your program. In other words, explain what evidence of 'speedup' you are expecting to see. Record this information at the start of your report on this lab exercise.

To demonstrate speedup without interference from other students, you will need to use the OGE batch system, as described above. You should run with a range of numbers of threads to observe how the performance changes. Do not use more than 12 threads (initially) in order to avoid having more than one thread execute on one core. As with all experiments of this kind, there will be errors in your measurements, so take each measurement several times and show the resulting variation in your plots (or otherwise plot appropriate error bars). Use `gnuplot` or Excel to plot a 'performance curve' showing how performance actually changes with the number of active threads. Remember to use good scientific practice when plotting graphs.

Please do not abuse the OGE batch queue. Run only jobs that you know will take a short time to complete. If you see your job executing for longer than a minute or two, please remove the job so that you do not hold up the other students. Do not put large numbers of jobs into the queue at the same time. The OGE command for removing a job from the queue is `qdel XXXX`, where `XXXX` is the unique job number for the submitted job. Please try to avoid removing a job just before it is about to run (see above).

When you first run your program, the performance curve will probably show a different trend to what you were expecting. This is likely due to one of two reasons: (1) the program is not parallelising the work well (for example, each thread is doing more work than it strictly needs to, or else different cores are given significantly different numbers of additions to perform), or (2) the sizes of the arrays being added together are too small for speedup to be visible (due to the overheads incurred when creating and joining the parallel threads). Remember that the time to launch a Java thread is of the order of a millisecond, so any parallel computation you are measuring needs to take significantly longer than this to execute. Simply changing the size of the arrays may not be the best way to increase the computational work per thread (can you think of any reason why this might be the case?), thus, you might try increasing the work per thread by having threads repeat their computation of their part of the result array. You may have to experiment to find a suitable combination of array size and repeat count (you should aim for an execution time on one thread of less than 1sec.)

Java is an interpreted language with Just-in-Time (JiT) compiler support. It may be that your code will run sufficiently long enough for the JiT to "kick-in". Can you think of a change to your code which would enable you to see if this is happening? Finally, you may find it useful to put a timer around the actual operation loop performing the vector addition (including around any repeat loop). This would provide insight into the time taken by each thread. It would be sensible to restrict the number of threads to a maximum of 12 in this case.

Record in your report how you reacted to your initial experimental observations in order to find a speedup for your code that more closely fitted with what you were expecting to happen. If you try a number of approaches, plot one performance curve for each attempt (you may submit more than one performance graph, if required).

Deliverables

There are three deliverables for this exercise. The first deliverable is the final version of the program you have written. You should deploy good software engineering practice during code development to ensure that the marker can readily understand what you were trying to achieve and, for example, to advise the user what may have gone wrong if they use parameters that are incompatible with the constraints given above. The second deliverable is a *brief* report stating what speedup you were expecting to see from your program, what you did in order to find a scenario that shows something like the expected speedup, and what, if anything, you think maybe causing the observed speedup to be different from what you expected. The third deliverable is a performance graph showing how the performance of your final program varies with the number of active threads/cores (the graph may show more than one performance curve if you wish to report more than one attempt to achieve speed up, and you may submit more than one graph if that makes your result easier to comprehend).

Submission

Submit your brief report together with the performance graph(s) and your source code in a single PDF or ZIP file via the assignment window on the Blackboard site for this course unit. The deadline for submission is one week after the final scheduled lab session for this exercise. Marks will be deducted for late submission.

Marking Scheme

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