CSC2002S Tutorial 2016

Parallel Programing with the Java Fork/Join framework:

Parallel Sorting

In this assignment you will implement two parallel algorithms for sorting arrays of integers and compare their performance. Note that parallel programs need to be both **correct** and **faster** than the serial versions. Therefore, you need to demonstrate both correctness and speedup for your assignment.
You will:

- Use the Java Fork/Join framework to implement parallel versions of Mergesort and Quicksort for sorting integer arrays. There is also an option to implement an additional parallel sort, such as Radixsort.

- Validate the correctness of your serial and parallel implementations using JUnit tests.

- Evaluate the performance of your parallel sorting algorithms experimentally:

    - using high-precision timing to evaluate the run times of your parallel methods, across a range of input sizes and architectures.

    - calculating the speedup of the code with respect to the builtin Java Arrays.sort() method;

    - experimenting with different parameters to establish the optimal number of threads for each architecture (i.e. the limits at which sequential processing should begin).

- Write a report that clearly documents and explains your findings.

Assignment details and requirements

1.1 Input and Output

Your program must take the following command-line parameters (in order):

```
<sort> <arraySizeMin> <arraySizeMax> <arraySizeIncr> <outFileName>
```

where
`<sort>` is a word that specifies the sorting algorithm, either `Mergesort` or `Quicksort` or, optionally, `Altsort;`
`<arraySizeMin>` size of the smallest (random) array of integers to be sorted
`<arraySizeMax>` size of the largest (random) array of integers to be sorted
`<arraySizeIncr>` the step size for increasing the array of integers to be sorted
`<outFileName>` is the name of the file to which the output data will be written.

The output file, `<outFileName>,` should consist of successive lines:

```
<arraySizeMin> <optimalNumThreads> <bestTime> <bestSpeedup>
 <arraySizeMin+arraySizeIncr> <optimalNumThreads> <bestTime> <bestSpeedup>
…
<arraySizeMax> <optimalNumThreads> <bestTime> <bestSpeedup>
```

Note that the number of threads is determined by the size of the array and the sequential cutoff. Also, the integer arrays must be randomized between each sort (remember that the performance of Quicksort is worse with sorted arrays).

1.2 Code structure

You must construct a separate Java class for each of your sorting implementations, extending the `RecursiveAction<V>` class from `java.util.concurrent.ForkJoinTask`: naming the classes `DriverSort, ergesortParallel, QuicksortParallel` and, if implemented, `AltSortParallel`.

1.3 Software Testing with JUnit classes

You are required to use JUnit to test your code. The source code for the JUnit test classes that you create for the assignment should be placed within a sub directory called 'test'. You must construct a JUnit class to test each of your sorting implementations against the output from the serial : `TestMergesortParallel, TestQuicksortParallel` etc.

1.4 Benchmarking

You must time the execution of your parallel sorts across a range of data array **sizes** and **number of threads** (sequential cutoffs) and report the speedup relative to the serial implementation in `Arrays.sort`. The code should be tested on at least **two different machine architectures** (at least one of which must be a multi-CPU/multi-core machine).

Timing should be done at least 5 times.
Use the `System.currentTimeMillis` method to do the measurements.
Timing must be restricted to the sorting of the code and must exclude the time to read in the files and other unnecessary operations, as discussed in class. Call `System.gc()` to minimize the likelihood that the garbage collector will run during your execution (but do not call this within your timing block!).

1.5 Report

You must submit an assignment report **in pdf format**. Your clear and **concise** report should contain the following:

- An *Introduction*, comprising a short description of the aim of the project, the parallel algorithms and their expected speedup/performance.
- A *Methods* section, giving a description of your approach to the solution, with details on the parallelization. This section must explain how you validated your algorithm (showed that it was correct), as well as how you timed your algorithms with different input, how you measured speedup, the machine architectures you tested the code on and interesting problems/difficulties you encountered.
- A *Results and Discussion* section, demonstrating the effect of data sizes and numbers of threads and different architectures on parallel speedup for both Mergesort and Quicksort. This section should **include speedup graphs** and **a discussion**. Graphs should be clear and labelled (title and axes). In the discussion, we expect you to address the following questions:
    - For what range of data set sizes does each parallel sort perform well?
    - What is the maximum speedup obtainable with your parallel approach? How close is this speedup to the ideal expected?
    - How do the parallel sorting algorithms compare with each other?
    - How do different architectures influence speedup?
    - What is an optimal number of threads on each architecture?
    - Is it worth using parallelization (multithreading) to tackle this problem in Java?
- A *Conclusions* (note the plural) section listing the conclusions that you have drawn from this project. What do your results tell you and how significant or reliable are they?

Please do NOT ask the lecturer for the recommended numbers of pages for this report. Say what you need to say: no more, no less.

1.3 Assignment submission requirements

- You will need to create, **regularly update**, and submit a GIT archive (see the instructions on Vula on how to do this).
- Your submission archive must consist a technical report (**pdf format**) and your solution code (including a **Makefile** for compilation).
- Upload the file and **then check that it is uploaded.** It is your responsibility to check that the uploaded file is correct, as mistakes cannot be corrected after the due date.
- The usual late penalties of 10% a day (or part thereof) apply to this assignment.

Due date:

9AM ON 8TH AUGUST 2016

- **Late** submissions will be **penalized at 10%** (of the total mark) per day, or part thereof.
- The deadline for marking **queries** on your assignment is **one week after the return of your mark.** After this time, you may not query your mark.
- You may not ask your tutor for extensions or concessions – **all queries must be directed to the course lecturer.**

1.4 Additional work

If you finish your assignment with time to spare, you can attempt one of the following for extra credit:

- Run on the HAL HPC cluster (contact Michelle Kuttel at least one week before the assignment is due for details).
- Implement and alterative parallel sorts, such as the odd-even sort or Radix sort.

1.5 Assignment marking

Your mark will be based primarily on your analysis and investigation, as detailed in the report.

MARKING GUIDE: CSC2002S 2016 ASSIGNMENT 1 (PARALLEL PROGRAMMING)

CATEGORY		Section subtotal/
1. Code:	**conforms to specifications (input, output etc), code correctness (Junit), timing, style and comments.**	**40**
2. Documentation:	**all aspects required in the assignment brief are covered, e.g. Introduction, Methods, Results (with graphs and analysis) Conclusions.**	**50**
3. Going beyond specification - extensions and improvements.		**10**
	Successfully implementing and benchmarking an additional sort, with reporting **OR** Additional architectures (e.g. Hal) tested and reported thoroughly/correctly.	
	Total	**100**
Penalties		
	Code does not execute/run.	**-50**
	Code executes, but no GIT repository.	**-20**
	GIT repository, but no regular updates (at least on 5 separate days).	**-15**
	No makefile.	**-10**
	Late penalty (10% per day or part thereof)	**10 x d**

Note well: submitted code that does not run or does not pass standard test cases will result in a mark of zero. **Any plagiarism, academic dishonesty, or falsifying of results reported will get a mark of 0 and be submitted to the university court**.