Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

Lab Handout (worth ≈1.3% of overall mark)

The aim of this activity is to implement a *parallel quicksort* of integer data. This exploits multithreading in order to do the quicksort computation more efficiently, particularly on machines with more than one processor (CPU).

A small library for controlling parallel code is provided in the swen221.concurrent package. This is a simpler version of the tools found in java.util.concurrent. You can download the quicksort.jar file from the SWEN221 homepage.

1 ParSum — Parallel Sum

To illustrate how the swen221.concurrent library works, a small example called ParSum is provided. This reads a file containing integers and sums them to produce a single result. You can run ParSum from the command-line like so:

```
% java ParSum large.dat
Executing on machine with 4 processor(s).
Read 10000000 data items.
Performing a SEQUENTIAL sum...
Summed data in 1200ms
Sum was 49999995000000
Data was summed correctly!
```

The above executes a standard sequential version of the sum operation (**NOTE:** you can execute this directly from Eclipse by providing command-line arguments to the run configuration). To execute the sum operation in parallel, you add the "−−parallel" option:

```
% java ParSum --parallel large.dat
Executing on machine with 4 processor(s).
Read 10000000 data items.
Performing a PARALLEL sum...
Summed data in 870ms
Sum was 49999995000000
Data was summed correctly!
```

You should notice that the parallel version takes less time to execute on a machine with more than one processor. The amount of speed up will vary and (unfortunately) will not be linear in the number of processors. You will find the method ParSum.parallelSum() helpful in completing this lab.

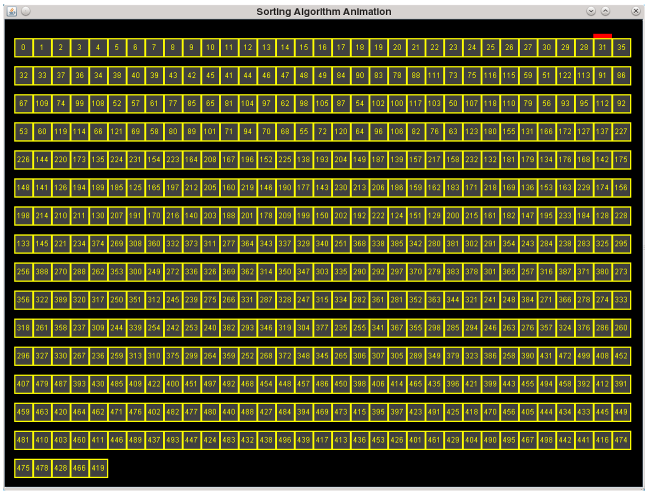

Figure 1: Illustrating the Sort GUI

2 ParSort — Parallel Quick Sort

An initial implementation of the quick sort code has been provided for you in the class ParSort. In particular, this contains the method ParSort.sequentialSort() which implements a *sequential* quick sort. You can run the ParSort code like so:

```
% java ParSort large.dat
Executing on machine with 4 processor(s).
Read 10000000 data items.
Performing a SEQUENTIAL quicksort...
Sorted data in 10756ms
Data was sorted correctly!
```

A special GUI is also provided which animates the quick sort. You can view this like so:

```
% java ParSort --gui tiny.dat
Executing on machine with 4 processor(s).
Read 500 data items.
Running in Display Mode (so ignore timings).
Performing a SEQUENTIAL quicksort...
```

You may find the GUI mode helpful in debugging your code (**NOTE:** you should use only the tiny.dat file, as others are too big). The GUI window is shown in Figure 1.

Finally, to execute the quick sort in parallel, you add the "−−parallel" option:

```
% java ParSort --parallel large.dat
Executing on machine with 4 processor(s).
Read 10000000 data items.
Performing a PARALLEL quicksort...
Sorted data in 0ms
!!! ERROR: data not sorted correctly
```

NOTE: the result from the parallel sort is currently incorrect because the parallel implementation is not finished. Specifically, a method ParSort.parallelSort() is provided without any implementation and your task is to implement this!

2.1 HINTS

You should find the method ParSum.parallelSum() helpful in completing the implementation of ParSort.parallelSort(). However, there are some important differences in the problems and you will need to consider carefully how to parallelise the quick sort. There are two approaches you can take:

1. **Division based on Job Size**. This approach follows ParSum.parallelSum() quite closely. The input data list is split into a roughly even number of chunks, based on the number of processors available. A job is created for each chunk, and then the code waits for each Job to finished. *However, at this point, the results from each chunk must still be recombined to produce the final sorted list.*

2. **Division based on Depth.** This approach is differs from ParSum.parallelSum(), but is easier to implement. In this case, sequentialSort() is modified to count the depth of the computation tree. Initially, the computation proceeds sequentially as before. But, at a certain depth (e.g. 2), a Job is created to complete the remainder of the chunk being sorted. This way, you end up creating 2^n Jobs at a depth n. *The advantage of this approach is that when the Jobs are finished, the data is sorted correctly.*

When your implementation of ParSort.parallelSort() is working correctly, you should find that it outperforms ParSort.sequentialSort() on large data sets.

Marking Guide

Each lab is worth just under 1% of your overall mark for SWEN221. The lab should be marked during the lab sessions, according to the following grade scale:

- **0**: Student didn't attend lab.
- **E**: Student did not really participate in the lab.
- **D**: Student's participation was *poor*. For example, he/she made some attempt to work on the lab, but did not complete any activities.
- **C**: Student's participation was *satisfactory*. That is, he/she attempted the lab but didn't achieve any parallelisation.
- **B**: Student's participation was *good*. That is, he/she got parallelisation working, but result produced was incorrect.
- **A**: Student's participation was *excellent*. That is, he/she computed the correct answer, and it was faster than sequential version on `large.dat`.