Lab 8: Parallel Sorting

F28SG – Introduction to Data Structures and Algorithms (6 marks)

Plagiarism policy

This lab is not group work, and you are assessed individually.

Therefore, the work you submit for this lab **must be entirely your own**. You must not share your code solution with other students, and you must not copy code solutions from others. The university plagiarism policy is clear:

> *"Plagiarism involves the act of taking the ideas, writings or inventions of another person and using these as if they were one's own, whether intentionally or not."*
> ***Definition 2.1.***

The disciplinary action for plagiarism is an award of an F grade (fail) for the course. Serious instances of plagiarism will result in Compulsory Withdrawal from the university.

About the starter code

In the sorting lectures we looked at two divide-and-conquer algorithms: merge sort and quick sort.

The **SequentialMergeSort.java** file contains a reference sequential version of merge sort. For a visualisation of how merge sort works, see ***Hint1!***

The **ParallelMergeSort.java** file contains template code for a parallel implementation of merge sort. The `ParallelMergeSort` class extends the `RecursiveTask` class, which is needed to make use of the `ForkJoinPool` parallel executor in Java. The `compute()` method currently contains the sequential version of merge sort. Your task is to *parallelise* the `compute()` method by replacing the sequential code with `fork` and `join` calls to decompose merge sort in a divide-and-conquer fashion.

Currently the divide-and-conquer code in `compute()` is sequential:

```
LinkedList<Integer> resultLeft = SequentialMergeSort.mergeSort(left);
LinkedList<Integer> resultRight = SequentialMergeSort.mergeSort(right);
```

Q1) Parallelise Merge Sort

Q1A) Benchmark sequential version (1 point)

The code is **ParallelMergeSort.java** is entirely sequential.

Benchmark the sequential code:

1. Right click ParallelMergeSort.java
2. Click "Run As"
3. Click "Java Application"

At the top of **ParalleMergeSort.java** in **src/sort/parallel/** there's a comment starting "*Merge Sort results*". Enter milliseconds runtime performance reported in the console in Eclipse next to "Before parallelisation:". Write it like this, replacing the `X` with the millisecond runtime:

```
Before parallelisation:

- 1 thread:  Xms

- 2 threads: Xms

- 4 threads: Xms
```

Don't be surprised if they are all quite similar, since your code is not parallel yet!

Q1B) Parallelise Merge Sort (1 point)

We will now *parallelise* merge sort with the goal of achieving shorter runtimes.

You should replace the two lines (above) that use `SequentialMergeSort.mergeSort` , with parallel fork/join code. Implement the following:

1. Comment out the two lines of code above, I.e. the `resultLeft` and `resultRight`.
2. Create two tasks by instantiating the `ParallelMergeSort` class twice, one task to sort the `left` list and another task to sort the `right` list.
3. Fork the task that will sort the left list.
4. Compute the task that sorts the right list in the current thread, and assign that value to a variable `resultRight` of type `LinkedList<Integer>`. See ***Hint2*** for help.
5. Wait for the result from the task forked in step 3, then assign that result value to a `resultLeft` LinkedList. See ***Hint2*** for help.
6. Call the `merge` method with these two sorted lists as the arguments to `merge`. That will create a new list by merging the two lists. That new list will be sorted. Return this list from the `compute()` method.

Run the benchmarks again by running **ParallelMergeSort.java** as a Java Application. If you are using a computer with more than 1 CPU core, using more threads might mean shorter runtimes.

Add the reported parallel runtimes next to "After parallelisation:" in the comment at the top of the file. Write it like this:

```
After parallelisation:

- 1 thread:  Xms

- 2 threads: Xms

- 4 threads: Xms
```

Q2) Optimise Merge Sort with Thresholding (2 points)

Your task is to refactor the code in `ParallelMergeSortThreshold.java` to increase the granularity (size) of parallel tasks generated by your parallel code in question Q1B.

Currently your code generates parallel tasks by splitting sub-lists into two lists: left and right. The base case is when a sub-list has 1 or 2 elements in it. That is, currently the base case is:

```
int length = arr.size();
// base case
if (length < 2) {
    return arr;
}

// step case
else {
```

There are two performance issues with a `length < 2` base case, which are:

1. For large input lists of integers to be sorted, this creates a huge number of parallel tasks because a list of length 1000 is split into two lists of length 500, each of those split into two of length, 250, then 125, and so on until a length of 1 at the leaves of the divide-and-conquer tree.
2. Each parallel task at the leaves of the divide-and-conquer tree are tiny. The only computation to performed is `return arr`, when the length of the integer list is 0 or 1.

Your task is to change this code so that fewer parallel tasks are created so that each task has more work to do. In the merge sort implementation in question Q1, we had a *base* case and *recursive* step case. This time, you will have a ***sequential*** case and a ***parallel*** case. To understand this task granularity performance issue, see ***Hint3!***

The sequential case performs divide-and-conquer merge sort of a sub-list within 1 task. The parallel case performs divide-and-conquer merge sort of a sub-list using multiple parallel tasks.

- Sequential case
    - **Condition**: if the length of the input list is smaller than the threshold value.
    - **Action**: Use the sequential version of merge sort in `SequentialMergeSort.mergeSort` to sort the input sub-list.
- Parallel case
    - **Condition**: if the length of the input list is larger than the threshold value.
    - **Action**: Split the input list into two sub-lists and sort them in parallel. This code should be the same fork/join parallelism technique that you used for question Q1B, but instead use the `ParallelMergeSortThreshold` class to create tasks rather than `ParallelMergeSort`.

The `ParallelMergeSortThreshold` constructor takes an addition argument, `threshold`. This should be used to decide whether to execute the sequential case, or the parallel case.

After you have parallelised the `compute()` method in **ParallelMergeSortThreshold.java** using this thresholding mechanism, run the benchmarks and add the reported runtime results in the comment at the top of the Java file. Write it like this, replacing `X` with the reported runtime in milliseconds:

```
After parallelisation:

- 1 thread

  - no threshold: Xms

  - threshold=128: Xms

  - threshold=512: Xms

  - threshold=2048: Xms

  - threshold=8192: Xms



- 2 threads

  - no threshold: Xms

  - threshold=128: Xms

  - threshold=512: Xms

  - threshold=2048: Xms

  - threshold=8192: Xms
```

If you are lucky enough to have more than 2 CPU cores, also include the runtimes for those numbers of threads too. The benchmarks runs on up to 1, 2, 4 .. N CPU cores that it finds on your computer.

If you are disappointed by only modest improvement with shorter runtimes, see ***Hint4!***

Q3) Measuring Parallel Speedup and Parallel Efficiency (1 point)

In the comment at the top of **ParallelMergeSortThreshold.java** file, specify which parameters resulted in the shortest runtime that you were able to achieve. For the shortest runtime of all those reported in the console window in Eclipse, state for that run: (1) how many threads were used and (2) what the threshold value was. Write it like this, replacing `X` as appropriate:

```
Parameters of the shortest runtime:
* - runtime: Xms
```

```
* - how many threads: X
* - threshold value: X
```

Now compute the parallel speedup by comparing this runtime with reported runtime **using 1 thread** and there a threshold is not used ("**no threshold**"). Parallel speedup is sequential runtime divided by the parallel runtime. Add it here:

```
Best parallel speedup: X
```

Finally, calculate the parallelism efficiency: speedup divided by the number of threads used. Add it here:

```
Parallelism efficiency: XX%
```

This algorithm involves a sequential phase (the merge method), which limits the theoretical parallel speedups, and parallelism efficiency. This is due to Amdahl's Law, see ***Hint4!*** Parallelism efficiency of between 30-50% would be doing well for your threshold-based parallel merge code implementation.

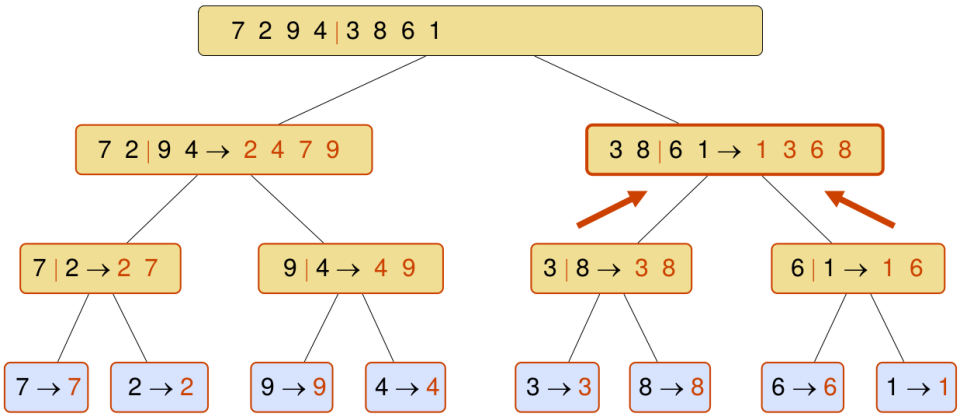
Q4) Code Quality (1 point)

Code quality is vitally important for so many reasons. Not least, for readability and maintainability, not just for yourself but for others too since in industry, software engineering is almost always a group exercise. Real world software engineering is mostly about refactoring and testing code, rather than writing new code (more code means higher maintenance costs!).

An additional mark is awarded if your code is deemed to be of high quality:

- **Code simplicity**
  - Is the code as short as it can be?
  - Is the Big-Oh complexity as small as it can possibly be?
  - Is the control flow (loops, while statements, if statements, etc.) simple to follow?
- **Documentation**
  - Is Javadoc syntax used for documenting the code? ***Hint!*** In Eclipse move your cursor to the text definition of a field, method or a class, then use the following keyboard shortcut: ***Alt + Shift + j*** , and then fill in the generated Javadoc template. If you are using MacOS, the shortcut is: ***⌘ + Alt + j***
- **Conventions**
  - Are sensible Java code conventions used, e.g. for code indentation, declarations, statements, etc? See Sections 4, 5, 6, 7 and 9 of *Java Code Conventions*. ***Hint!*** *Use the keyboard shortcut in Eclipse:* ***Control + Shift + f*** , or on MacOS: ***⌘ + Shift + f***

Hint1: Here is a visualisation of the divide-and-conquer merge sort algorithm:



Hint2! The `fork` method creates a new parallel task. The `join` method is a blocking call that waits for the result from a forked task. The `compute` method computes a task in the current thread without turning the task into a parallel task.

Hint3! When writing parallel software, it is important to consider the parallelism limits of the underlying CPU processor. For example a 64 core CPU can handle more parallelism compared with a 4 core CPU. A CPU with 4 cores is unlikely to benefit from more than 4 threads. Parallel tasks in a program will often be of unequal sizes. In Java, the `ForkJoinPool` executor uses work stealing for load balancing tasks of different sizes between execution threads from overloaded CPU cores to idle does. Generating more parallel tasks than CPU cores enables load balancing but finding the ideal ratio between (1) the number of parallel tasks created, (2) parallel task sizes, and (3) CPU cores, is an inexact science. Too many parallel tasks overload runtime systems with overheads of synchronisation, coordination and scheduling. The only way to know what the "ideal" task granularity (task size) for a particular algorithm is to benchmark and profile your code.

Hint4! Parallel programming is getting easier thanks to easy to use high level programming abstractions, but ***obtaining shorter runtimes*** by parallelising code is hard! There are many factor that impact the parallelism efficiency of parallelised code, e.g. scheduling overheads, ineffective load balancing, too-fine granularity, too-coarse granularity, and Amdahl's Law limiting theoretical speedup. Amdahl's Law says says that if you have even a very small sequential component of your code, this severely limits theoretical speedups (and the `merge` method is sequential code). If you can achieve 2x speedups (I.e. twice as fast with a parallelism efficiency of 50%) with 4 CPU cores, you are doing very well. As you add more CPU cores, the parallelism efficiency will inevitably drop because of scheduling overheads and memory contention.

Parallel Sum Euler (Optional)

The goal of this optional task is to introduce a chunking mechanism to a parallel implementation of an algorithm called Sum Euler. This algorithm computes the *sum of Euler totient computations* over a range of integer values. Two numbers **m** and **n** are relatively prime (**⊥)**, if the only integer number that divides both is 1. The *Euler Totient* function computes, for a given integer **n**, the number of positive integers smaller than **n** and relatively prime to **n**.

The Euler function is:

$$\Phi(n) \equiv \mid \{m \mid m \in \{1, \ldots, n-1\} \wedge m \perp n\} \mid$$

The Sum Euler function from 1 to **n** is therefore:

$$\sum_{i=1}^{n} \Phi(n)$$

See ***Hint5!*** for a video that explains this algorithm with an example.

A parallel implementation of the Σ above for summing the Euler calculations is in **ParallelSumEuler.java** in **src/euler/parallel**. The approach to parallelism in the `compute()` method is:

1. Loop from 1 to **n**, and `fork` a Euler task that sequentially computes Φ(n).
2. Loop from 1 to **n**, and wait for their results with `join`.

This approach generates **n** parallel tasks.

The size of each task is quite small (computing relative primes), but a chunking mechanism could increase task granularity to increase parallelism efficiency, and with that possibly shorten runtimes.

Profile Parallel Sum Euler

To run the parallel Sum Euler benchmarks:

1. Right click **ParallelSumEuler.java**
2. Click "Run As"
3. Click "Java Application"

Document your parallel runtime results in the *"Sum Euler results"* comment at the top of **ParallelSumEuler.java**.

Chunking Sum Euler Parallelism

The main issue with the previous version is that it creates too many parallel Euler tasks.

We can fix this by increasing the granularity of parallel tasks:

- **Before**
  - 1 parallel task per Φ(n) Euler tasks using `EulerTask.java`
  - Sequentially sum all Euler task results.
- **After chunking**

- Parallel tasks to compute and sum Φ Euler tasks **for a range** between two integers using `SumEulerTask.java`
- Sequentially sum each partial Sum Euler task results.

Your task is to complete **ParallelSumEulerChunked.java**, in particular `compute()`.

1. Loop through chunks of integer ranges:

```
for (long i = lower; i <= upper; i += chunkSize) { ... }
```

2. In each loop iteration create a new task by instantiating `SumEulerTask`, between a lower value and the upper value. ***Hint***: `Math.min` ensures you don't exceed the upper value **n**. Fork each task and it to the `tasks` lists.
3. Again, iterate over the chunks of Sum Euler tasks:

```
for (long i = lower; i <= upper; i += chunkSize) { ... }
```

   In which you wait for the results of parallel tasks with join, and for each result update the v intermediate variable `sum`.

4. Return the `sum` value from the `compute()` method.

Profile Chunking Performance

To run the chunking-based parallel Sum Euler benchmarks:

4. Right click **ParallelSumEulerChunked.java**
5. Click "Run As"
6. Click "Java Application"

In the comment at the top of **ParallelSumEulerChunked.java** file beneath "*Chunked based Sum Euler results*", specify which parameters resulted in the shortest runtime. For the shortest runtime of all those reported in the console window in Eclipse, state for that run: (1) how many threads were used and (2) the Sum Euler chunk size.

To test you parallel Sum Euler implementations, uncomment the tests in `SumEulerTest.java`.

Hint5! The Sum Euler algorithm is described in a video used in the 4th year F20DP *Distributed and Parallel Technologies* course. The video also hints on how to efficiently parallelise the algorithm using chunking. Watch the video from 44 seconds to 4 minutes 39 seconds. This is a link to the video.