Lab 2 - Java Parallel Programming and Sorting Algorithms

- Group AA
- Bon Alexis and Bouziane Abderrahmane

Task 1: Sequential Sort

We chose to implement the quicksort. The algorithm follows the following steps : Choose a Pivot: Select an element from the array as the "pivot". Partition: Rearrange the array so that all elements less than the pivot come before it, and all elements greater come after it. The pivot is now in its final sorted position. Recursively Sort: Apply the same process to the sub-arrays on either side of the pivot.

Source files: - `Sortutils.java` - `SequentialSort.java`

Task 2: Amdahl's Law

Limitations of regular Amdahl's law in our case

Quicksort algorithm recursively divide sorting problem into two subproblems, by cutting initial array into two subarrays. Considering the binary tree of all subarrays, that means at most 2^k threads can execute at the same time in depth k.

Depth 0 corresponds to the first `partition()` call and is our sequential part s.

Depth 1 and below form our parallelizable part. But even with $n > 2$ threads, depth 1 speedup will be only 2 as only 2 threads can work on that part simultaneously.

Similarly, depth k speedup can be at most 2^k as at most 2^k threads can work on it simultaneously. That means we cannot consider as regular Amdahl's law does that the whole parallelizable part is executed in parralel by our n threads.

Our version of Amdahl's law for 2, 4, 8, 16 threads

With $n = 2$ threads, as the whole parallelizable part can be executed with 2 threads, our speedup is the same as the regular Amdahl's law :

$$S_2 = \frac{1}{1 - p + \frac{p}{2}} = \frac{1}{1 - \frac{p}{2}}$$

With $n = 4$ threads, depth 1 speedup will be 2, and depth 2 and below speedup will be 4. The fraction of time spent to compute depth 1 depends on the sorted array size, so we will estimate it to $\frac{p}{2}$, which corresponds to the worst case in term of speedup (array of size 4). Then our speedup is :

$$S_4 = \frac{1}{1 - p + \frac{\frac{p}{2}}{2} + \frac{\frac{p}{2}}{4}} = \frac{1}{1 - \frac{5}{8}p}$$

Similarly, with $n = 8$, we estimate the fraction of time spent to compute depth 1 to $\frac{p}{3}$, depth 2 to $\frac{p}{3}$, and depths 3 and below to $\frac{p}{3}$. Our speedup is :

$$S_8 = \frac{1}{1 - p + \frac{p/3}{2} + \frac{p/3}{4} + \frac{p/3}{8}} = \frac{1}{1 - \frac{17}{24}p}$$

Finally, with $n = 16$, our speedup is :

$$S_{16} = \frac{1}{1 - p + \frac{p/4}{2} + \frac{p/4}{4} + \frac{p/4}{8} + \frac{p/4}{16}} = \frac{1}{1 - \frac{49}{64}p}$$

Plots

Here is a plot of our version of our Amdahl's law

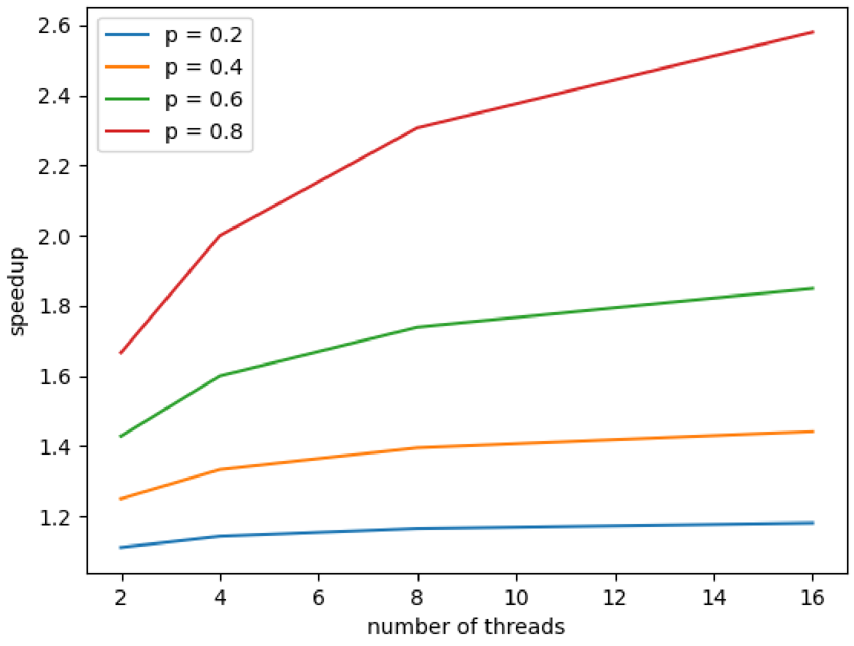

Figure 1: amdahl's law plot

We see that even with a high parallelizable part, speedup doesn't scale that much as n increase, due to the first depths that have to be executed by a few amount of threads.

Task 3: ExecutorServiceSort

Source files: - `SortUtils.java` - `ExecutorServiceSort.java`

In this task, we implemented a parallel version of quicksort using Java's `ExecutorService` with a fixed-size thread pool. The goal was to exploit parallelism by submitting sorting tasks to the pool as the algorithm recurses.

At each recursive step, after partitioning the array around a pivot, we submit new sorting tasks for the left and right subarrays to the thread pool. This allows multiple parts of the array to be sorted in parallel, up to the number of available threads.

However, this approach introduces a challenge: the number of tasks grows exponentially with the depth of recursion, since each partition creates two new tasks. If all threads in the pool are busy waiting for subtasks to complete (for example, if each thread submits more tasks and waits for their completion), we can encounter a deadlock. This happens when all threads are blocked, waiting for tasks that cannot start because there are no free threads left in the pool.

To prevent this issue, we use an atomic counter to track the number of active tasks. Before submitting a new task, we increment the counter; after completion, we decrement it. By ensuring the number of active tasks never exceeds the thread pool size, we avoid deadlocks and keep the algorithm lock-free.

Task 4: ForkJoinPoolSort

Source files: - `SortUtils.java` - `ForkJoinPoolSort.java`

In this approach, we implemented parallel quicksort using Java's `ForkJoinPool`. The sorting logic is encapsulated in a `RecursiveAction` subclass, where the `compute()` method recursively partitions the array and invokes subtasks for the left and right segments. To avoid excessive overhead from creating too many small tasks, we introduce a threshold: if the subarray size falls below this threshold, we sort it sequentially instead of further splitting. `ForkJoinPool` efficiently manages task splitting and work-stealing, allowing threads to execute available subtasks and maximizing parallelism without manual thread management.

Task 5: ParallelStreamSort

Source files:

- `ForkJoinPoolSort.java`

In this task, we used java parallel streams to independantly apply recursion in parallel.

The algorithm structure is pretty close to the sequential one. Because java parallel streams uses all available cores on the machine, we used a global `ForkJoinPool` to limit the amount of threads working at the same time.

Then, after `partition()` cuts the array in two, we creates two `Interval` objects, which contains begin and end indexes of each sub-arrays, and we create a parallel stream with the two intervals. Each `Interval` will lead to a recursive call of `parallelSort` as well as in the sequential algorithm, but here two differents threads will independantly execute it and recursively gives a part of the work to more and more threads, as long as some are available.

Task 6: Performance measurements with PDC

Source file :

- `task6.sh`

For our performance measurements, we used `MeasureMain` class. For each parallel algorithm, we measured its mean execution time using 1, 2, 4, 8, 16, 32, 48, 64 and 96 threads.

Each measurement contained 10 warmup rounds, and 1000 measure rounds. Each round consisted of sorting an array of 10,000,000 integers.

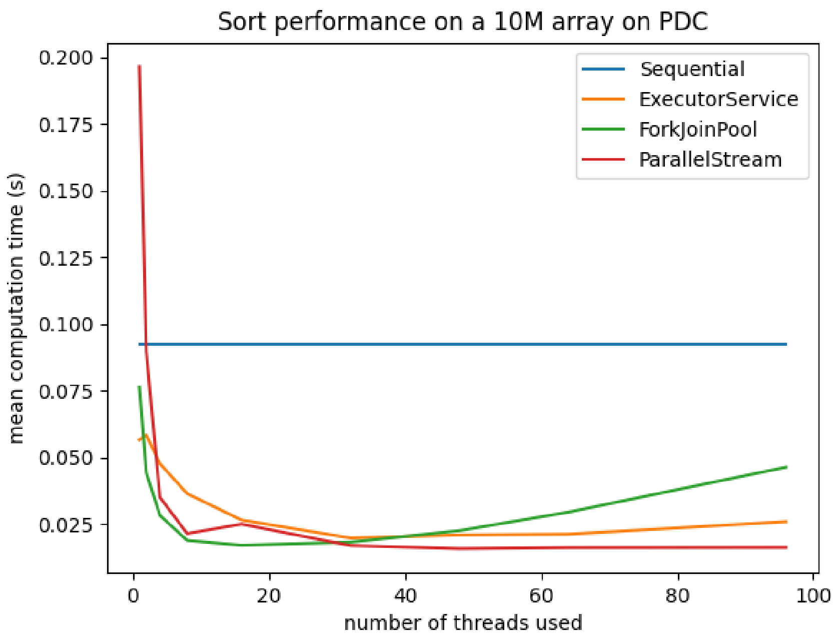

Figure 2: pdc plot

We see that when we use several threads, every parallel algorithm becomes faster than the sequential algorithm.

When the number of threads increases, `ExecutorService` and `ParallelStream` algorithms seems to converge quickly.

On the other hand, `ForkJoinPool` mean execution time increases after a certain point, showing that parallel objects used are costing more time than it gives when we use too many threads.