

Parallellization of the Coin Change Problem

Beatriz Rosa (55313)

3rd of November of 2023

1 Parallelization strategy chosen

The 3 approaches chosen for this assignment were based on the *Fork/Join* Framework: Fork/Join without using a ForkJoinPool; Fork/Join with *ForkJoinPool* and the last one with *CompletableFuture*, since they handle relatively well **Irregular Parallelism** - which is the case for the sequential algorithm of the Coin problem in which are present two recursive calls at each step: one excluding the current coin, and another that includes it, so its reasonable to think that the workload required to determine the best combination varies from one branch of recursion to another, and given the level of independence of each, each recursive branch can be explored concurrently. The reason why this specific approaches were chosen is the following:

1.1 1: *ForkJoin without ForkJoinPool*

First of all, the Fork/Join framework was chosen because it uses a *work-stealing* algorithm: tasks are distributed by the framework to worker threads and worker threads that completed their assigned work, can steal from others that are still busy, which fits the problem in question where we have slightly different workloads. To accomplish this approach the a subclass of *RecursiveTask*, *CoinTask* was created and forked in the main class *Coin.java*, after that, *compute()* function was called, performing a divide-and-conquer strategy to solve the coin change problem in parallel by: checking the base cases of recursion; checking if the depth of the current computation exceeds or equals to 15 (this will be explained further in the fine-tuning section) executes the sequential algorithm, otherwise the current computation is split into two tasks:

- *includeNextCoin* - includes the current coin, is forked to run in parallel;
- *excludeNextCoin* - excludes the current coin, is computed in a blocking manner, which means that the current thread has to wait for this task to be completed.

Back in the *Coin.java* class the *CoinTask* calls *join()* , waiting for the computation to complete and return the result.

1.2 2: *ForkJoinPool*

In this approach, a pool of threads is created to manage the tasks distribution automatically, leveraging multiple threads according to the number of threads *nCores* of the machine. The *CoinTask* is submitted to the thread pool and then waits for its completion, returning the result.

1.3 3: *ForkJoinPool with CompletableFuture*

Using *CompletableFuture* with a thread pool allows for finer-grained control over task creation and execution by associating each task with a *CompletableFuture*, enabling better tracking and coordination of tasks, and resources management which is introduced by the element of asynchronicity into the parallelization process. The creation of this promise enables the creation of task pipelines, where one task can trigger the execution of another as soon as it's completed. This pipelining can further enhance the overall throughput of the parallelization process.

2 Specifications of the processor

12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz with: Total Cores: 14; Performance-Cores: 6; Efficient-cores: 8; Total threads: 20.

3 Results achieved

The approach that provided the best performance was the Fork/Join without the thread pool and the best thread count that provides the best performance is 20 which is the value returned by *Runtime.getRuntime().availableProcessors()* of this machine, this approach was better for every thread count tested, but the second place is contested by the other 2 approaches, that have a very similar performance but the third approach seems to overcome the second one for the optimal thread count, as it is possible to see in figure 1, and in bigger detail in figure 2:



Figure 1: Evolution of Execution times Sequential VS Parallelization approaches through 50 iterations using the full parallelization potential of the machine (20 threads) and with recursion depth fine tuning



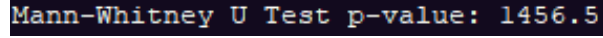
Figure 2: Boxplot for the execution times of each parallelization approach using different thread counts

Although the first approach requires manually handling task execution and result retrieval, the second and

third approaches being capable of abstracting and simplifying those tasks by delegating those responsibilities to the *ForkJoinPool*, the act of creating and forking tasks explicitly for this Coin change problem makes it possible to have a more fine-tuned thread management for this specific task, which makes only forking and joining tasks faster than doing it with a *ForkJoinPool*.

For the optimal thread count, comparing the *ForkJoinPool* version without *CompletableFuture*'s and the approach that uses them, the combination of both leverages the capabilities to schedule and execute tasks, performing better, because as mentioned before the *CompletableFuture* provides a more fine-grained control over scheduling and load balancing.

The following figure is the result of the Mann-Whitney U Test for the datasets of the second and third approaches, which had a similar performance:



```
Mann-Whitney U Test p-value: 1456.5
```

Figure 3: Mann-Whitney U Test p-value for the Second and third approach execution time datasets using 20 threads

This non-parametric test compared the distributions of the two groups without making assumptions about the shape of the underlying data distribution. In this case, the execution times are in seconds. This large p-value observed suggests that the test did not find a significant difference between the two groups, a conclusion that is in line with what is expected since both second and third approaches have close execution times.

4 Fine-tuning for performance improvement

To improve the performance of the parallelization techniques some conditions were introduced to the *CoinTask*:

- The condition *if (depth >= 15)* means that when the recursion depth reaches 15 the task will be computed sequentially and no more parallelism will be involved in that branch for *CoinTask*. Adding to that, it prevents excessively **deep recursion**, that as a consequence would cause an increase of the overhead of task creation and synchronization. This specific value was the best found (Figure 1), swapping it for bigger or smaller values leads to the benefits of fine tuning being cancelled, as it is possible to see in the many line charts in the package *src/plots* of the project .

- The condition *if (RecursiveTask.getSurplusQueuedTaskCount() > 2)* checks the number of tasks queued for execution in the pool, if it exceeds a threshold, the task can be executed sequentially. This condition is related to the **workload balancing** among threads, if there are many tasks queued for execution, it may suggest that some threads are idle, therefore distributing more tasks could help balance the workload. The value used is the standard recommended by the Java library. Overall, only using this condition in the *CoinTask* led to the poorest performance (Figure 5).

- When combining both conditions (Figure 4), the performance obtained still did not overcome the one that uses the depth-based condition alone as the only fine-tuning decision.

The reason why the depth-based condition improves performance even better than the combination of both conditions or using load-balancing alone is because it is a more assertive and direct way to control when to switch to sequential execution, on the other hand, surplus condition might lead to more parallel execution even when it's not beneficial for small problem sizes.



Figure 4: Execution times evolution with both fine tuning conditions



Figure 5: Execution times evolution with load balancing-based condition