Parallel Sorting Algorithm report

Introduction:

Sorting is a fundamental operation in computer science, essential for optimizing the efficiency of various applications. We have a lot of different algorithm out there such as QuickSort, MergeSort and HeapSort. Those are well known and have been studied for a long period.

But with the advent of multicore systems, it becomes crucial to leverage parallelism to enhance sorting performance on large datasets.

The goal of the project is to compare: **Sequential Sorting and Parallel Sorting** on large datasets and see if we effectively gain an increase in performance:

Definition:

- **Sequential time:** This is the time used by the **sequential sorting algorithm** to process an array of n elements:
- **Parallel time:** This is the time used by the **parallel sorting algorithm** to process an array of n elements:
- **Speedup:** This is the quotient of the sequential time and the parallel time. It's used to measure the effectiveness of the parallel sorting algorithm over the sequential version.

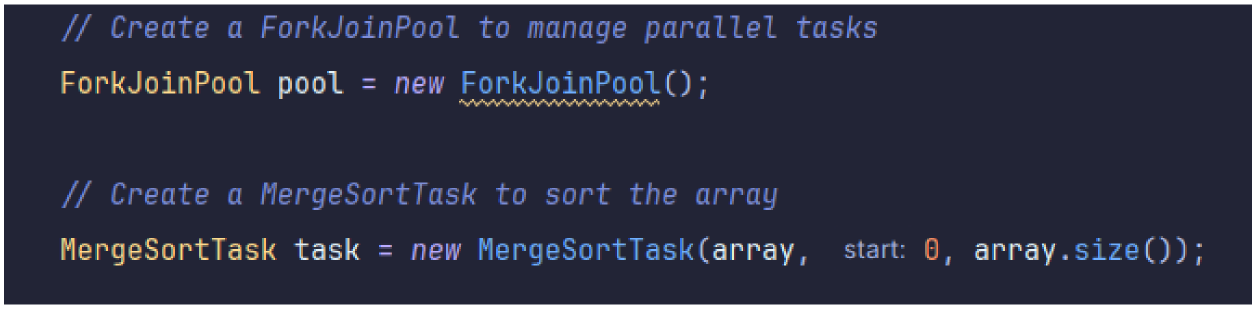
Key Concepts:

- **Divide and Conquer**: Many parallel sorting algorithms are based on the divide-and-conquer paradigm. The dataset is divided into smaller parts, each of which is sorted independently and in parallel. The results are then combined to produce the final sorted array.
- **Data Partitioning**: Efficient partitioning of data is crucial for parallel sorting. The data must be divided in a way that balances the load across all processing units, ensuring that no single unit becomes a bottleneck.

- **Synchronization and Communication**: Managing synchronization and communication between parallel tasks is vital. Efficient algorithms minimize the overhead associated with these operations to maximize the speedup gained from parallelism.

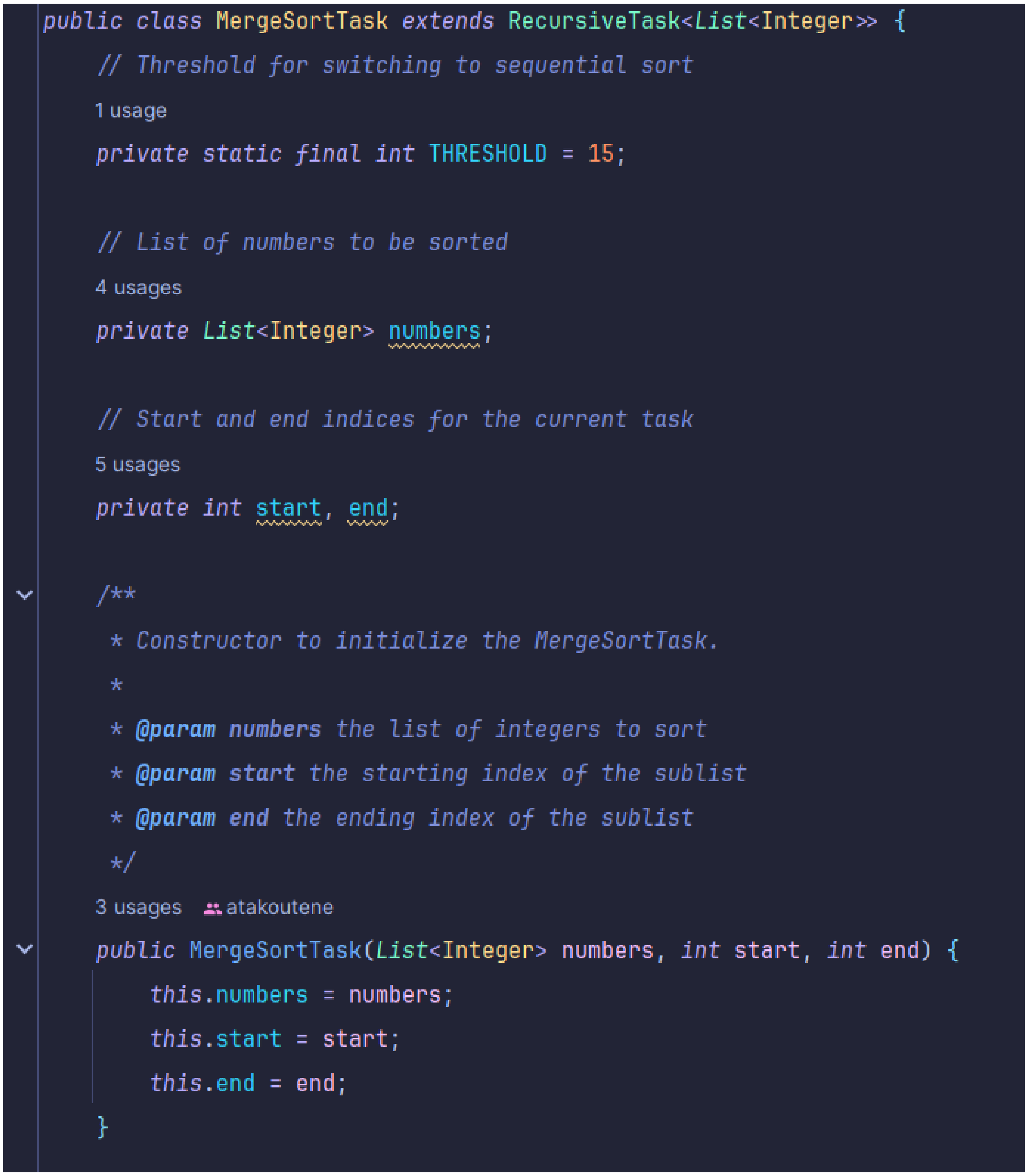
Design:

In order to design the parallel sorting algorithm. We thought first of all about managing the threads by ourself but we quickly figured out that this would be a big overhead. Initially we wanted to create a fixed thread pool where we would have added and removing threads. But due to the recursive nature of the algorithm that we used(merge sort), we decided to use a **ForkJoinPool**. The ForkJoinPool is more suitable compared to a Fixed Thread Pool because it will allocate the right amount of resources to manage all our threads.



```
// Create a ForkJoinPool to manage parallel tasks
ForkJoinPool pool = new ForkJoinPool();

// Create a MergeSortTask to sort the array
MergeSortTask task = new MergeSortTask(array,  start: 0, array.size());
```

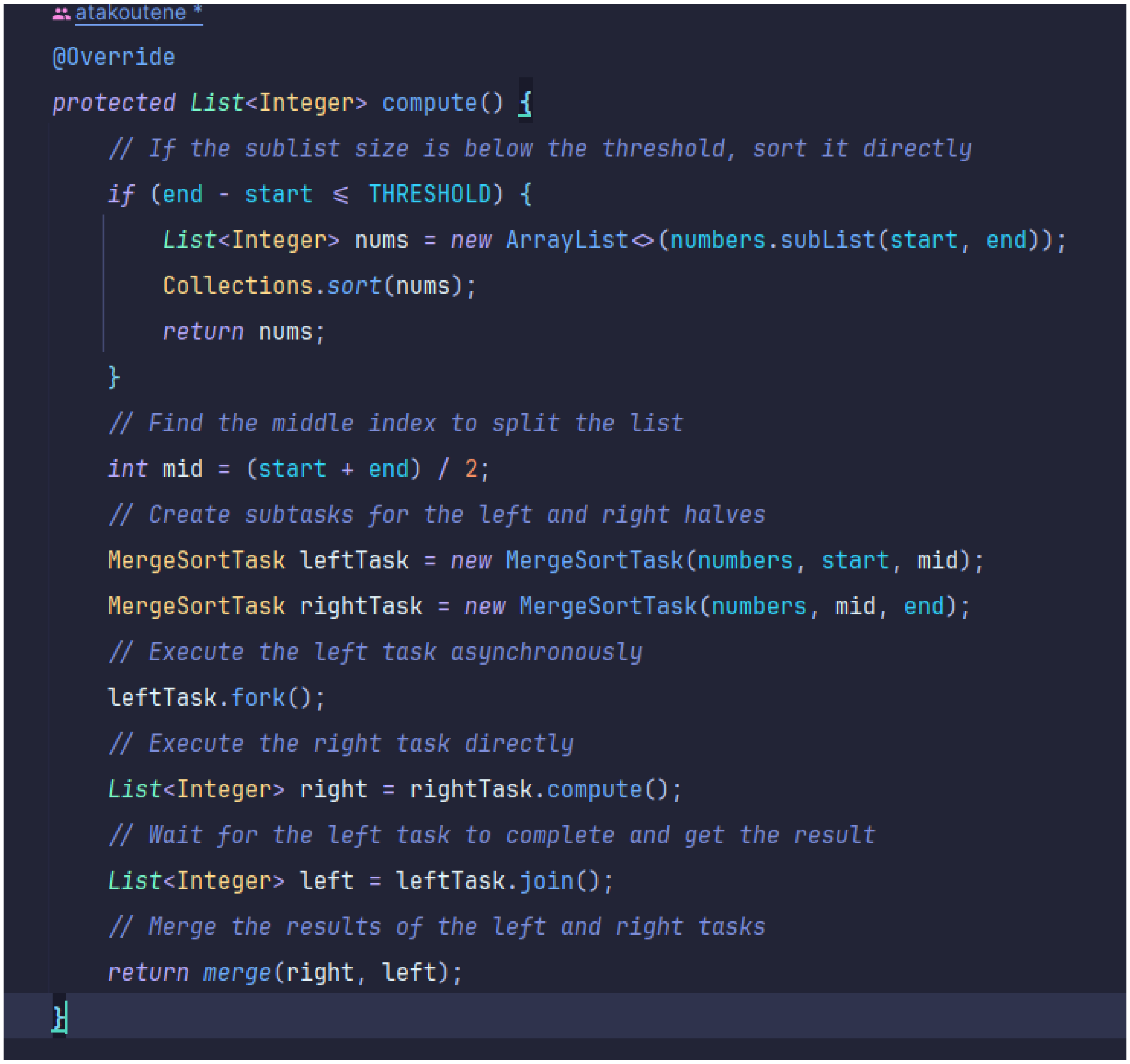
The second part consisted into designing the Tasks that our ForkJoinPool would have proceeded. After researching a lot. We came across the RecursiveTask which is a built in of **Java.concurrent.** It allows to ease the creation of recursive threads and to easily reconcile them.



```java
public class MergeSortTask extends RecursiveTask<List<Integer>> {
    // Threshold for switching to sequential sort
    1 usage
    private static final int THRESHOLD = 15;

    // List of numbers to be sorted
    4 usages
    private List<Integer> numbers;

    // Start and end indices for the current task
    5 usages
    private int start, end;

    /**
     * Constructor to initialize the MergeSortTask.
     *
     * @param numbers the list of integers to sort
     * @param start the starting index of the sublist
     * @param end the ending index of the sublist
     */
    3 usages   atakoutene
    public MergeSortTask(List<Integer> numbers, int start, int end) {
        this.numbers = numbers;
        this.start = start;
        this.end = end;
    }
```

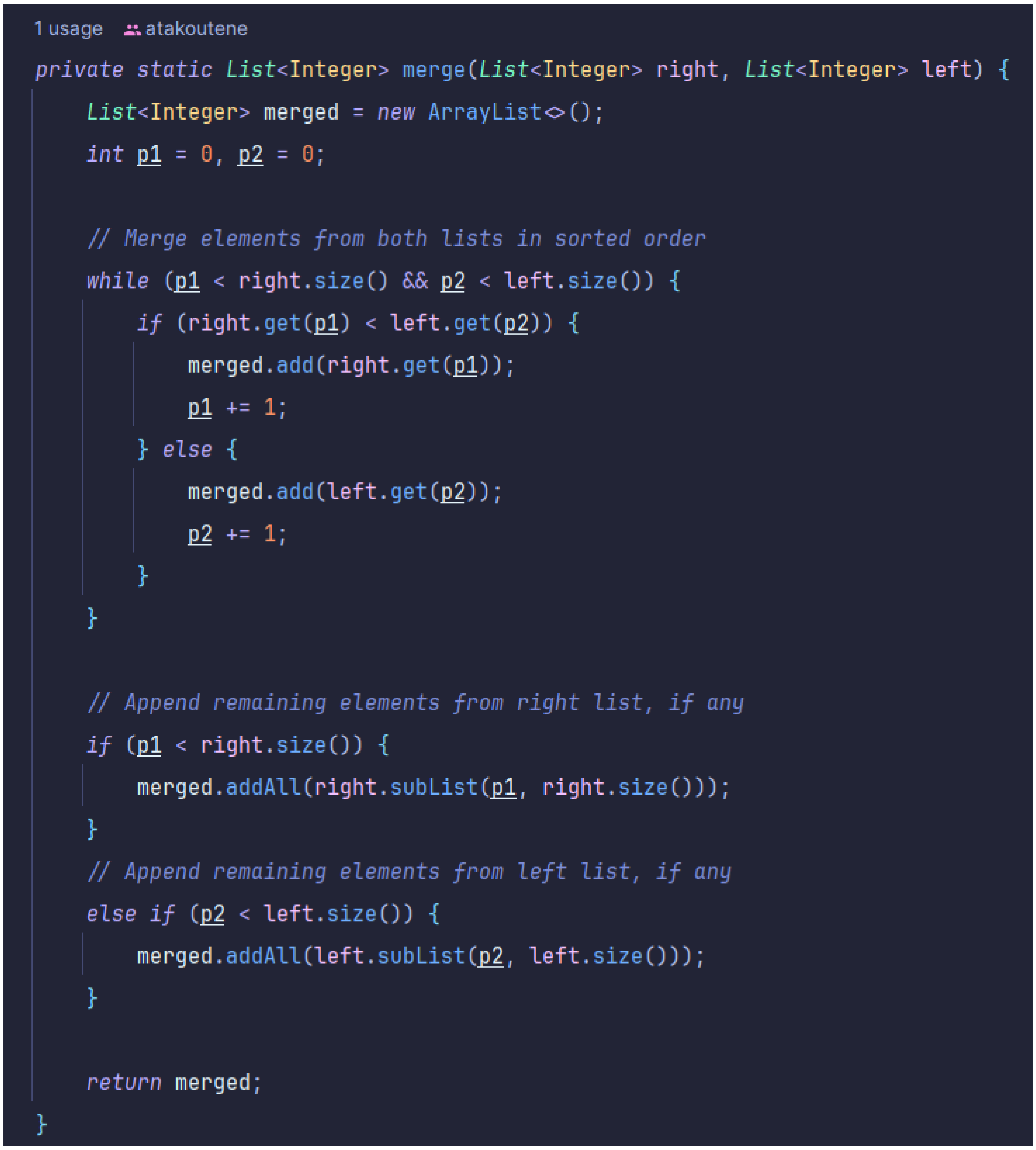
The goal of the task is to recursively divide the tasks until we reach a certain threshold. That is when the length of the processed array is less or equal than 15.

If we haven’t reached the threshold yet. we separate our task into to subtask. The left one is gonna be executed asynchronously and we’ll finally reconcile the left and the right


atakoutene *

```java
@Override
protected List<Integer> compute() {
    // If the sublist size is below the threshold, sort it directly
    if (end - start ≤ THRESHOLD) {
        List<Integer> nums = new ArrayList<>(numbers.subList(start, end));
        Collections.sort(nums);
        return nums;
    }
    // Find the middle index to split the list
    int mid = (start + end) / 2;
    // Create subtasks for the left and right halves
    MergeSortTask leftTask = new MergeSortTask(numbers, start, mid);
    MergeSortTask rightTask = new MergeSortTask(numbers, mid, end);
    // Execute the left task asynchronously
    leftTask.fork();
    // Execute the right task directly
    List<Integer> right = rightTask.compute();
    // Wait for the left task to complete and get the result
    List<Integer> left = leftTask.join();
    // Merge the results of the left and right tasks
    return merge(right, left);
}
```

Finally, we’ve designed the merge to sort the chunks that individual chunks from our earlier processing. We know that our two lists are sorted. So our goal is just to keep a pointer on each list and repeatedly pick the smallest element between the elements referenced by the two pointers

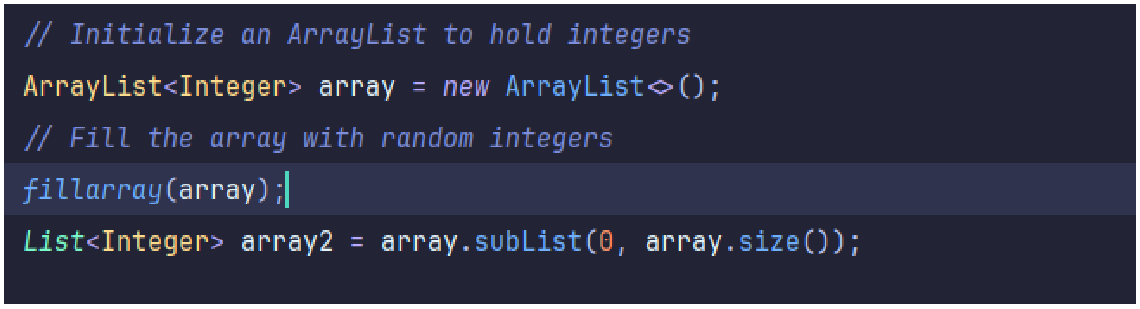


1 usage atakoutene

```
private static List<Integer> merge(List<Integer> right, List<Integer> left) {
    List<Integer> merged = new ArrayList<>();
    int p1 = 0, p2 = 0;

    // Merge elements from both lists in sorted order
    while (p1 < right.size() && p2 < left.size()) {
        if (right.get(p1) < left.get(p2)) {
            merged.add(right.get(p1));
            p1 += 1;
        } else {
            merged.add(left.get(p2));
            p2 += 1;
        }
    }

    // Append remaining elements from right list, if any
    if (p1 < right.size()) {
        merged.addAll(right.subList(p1, right.size()));
    }
    // Append remaining elements from left list, if any
    else if (p2 < left.size()) {
        merged.addAll(left.subList(p2, left.size()));
    }

    return merged;
}
```

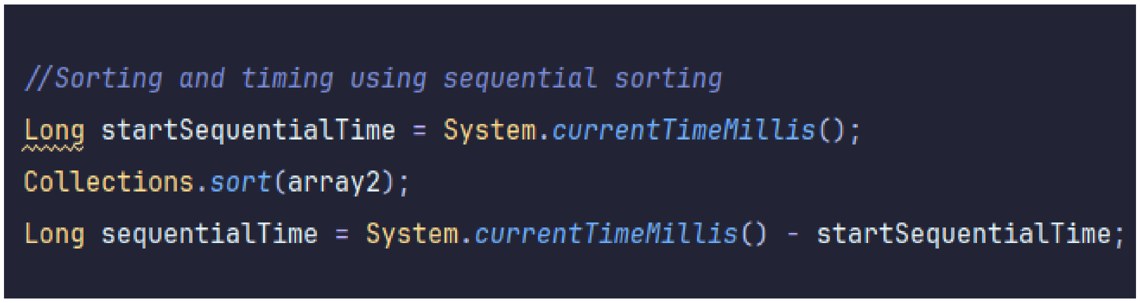
Approach:

- **Initialization:** First of all initialize two arrays. We'll incrementally increase their size and check the speedup time to draw a conclusion



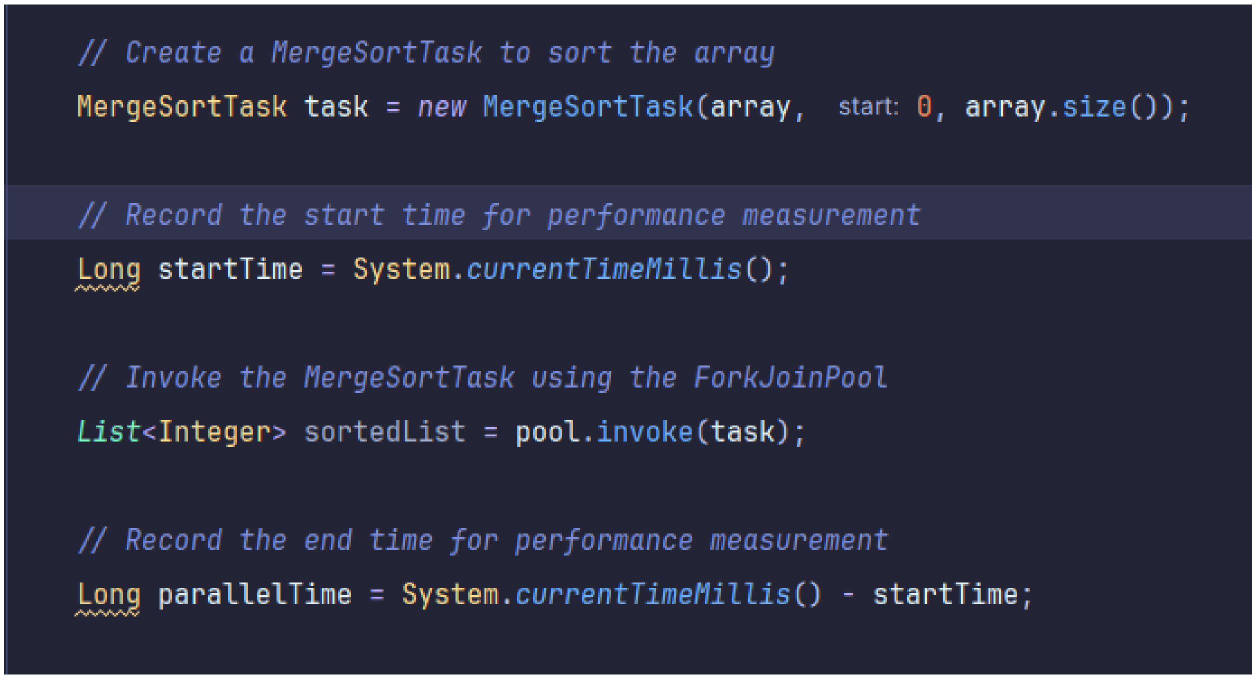
```java
// Initialize an ArrayList to hold integers
ArrayList<Integer> array = new ArrayList<>();
// Fill the array with random integers
fillarray(array);
List<Integer> array2 = array.subList(0, array.size());
```

- **Sequential Time:** We try to evaluate the required time for the sequential operation. We should note that behind the scenes the **Collections.sort** is using a synchronous Merge Sort.



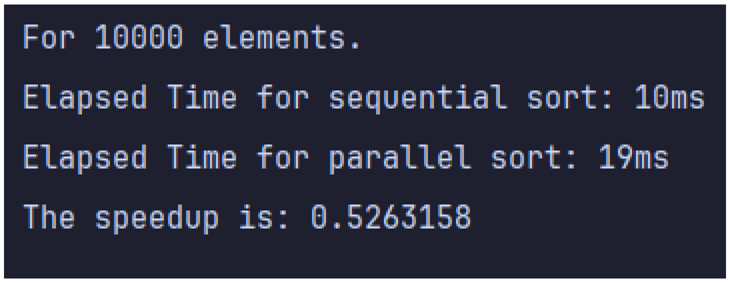
```java
//Sorting and timing using sequential sorting
Long startSequentialTime = System.currentTimeMillis();
Collections.sort(array2);
Long sequentialTime = System.currentTimeMillis() - startSequentialTime;
```

- **Parallel Time:** We try to evaluate the required time for the parallel operation. The ForkJoinPool is gonna be used to manage the Recursive Task that we are creating. It's gonna control and manage threads allocation

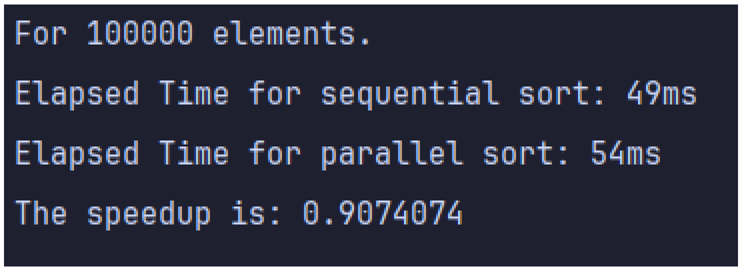


```java
// Create a MergeSortTask to sort the array
MergeSortTask task = new MergeSortTask(array,  start: 0, array.size());

// Record the start time for performance measurement
Long startTime = System.currentTimeMillis();

// Invoke the MergeSortTask using the ForkJoinPool
List<Integer> sortedList = pool.invoke(task);

// Record the end time for performance measurement
Long parallelTime = System.currentTimeMillis() - startTime;
```

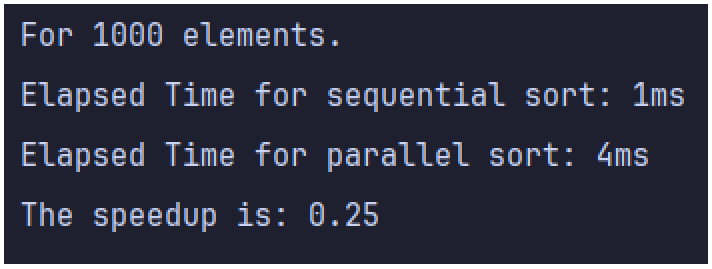
- **Evaluation**



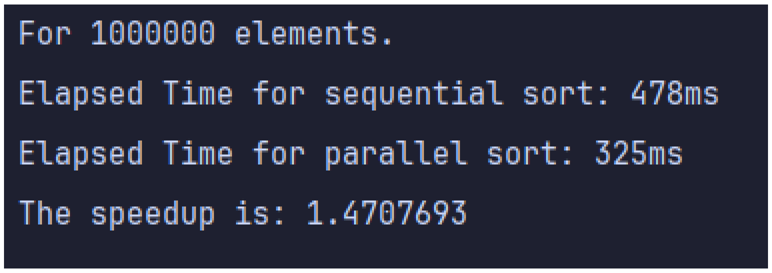
```
For 10000 elements.
Elapsed Time for sequential sort: 10ms
Elapsed Time for parallel sort: 19ms
The speedup is: 0.5263158
```



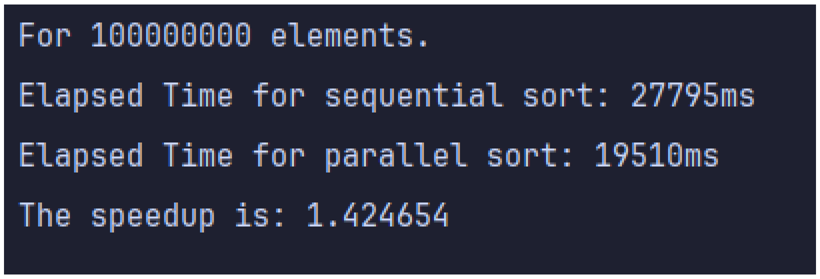
```
For 100000 elements.
Elapsed Time for sequential sort: 49ms
Elapsed Time for parallel sort: 54ms
The speedup is: 0.9074074
```



```
For 1000 elements.
Elapsed Time for sequential sort: 1ms
Elapsed Time for parallel sort: 4ms
The speedup is: 0.25
```



```
For 1000000 elements.
Elapsed Time for sequential sort: 478ms
Elapsed Time for parallel sort: 325ms
The speedup is: 1.4707693
```



```
For 100000000 elements.
Elapsed Time for sequential sort: 27795ms
Elapsed Time for parallel sort: 19510ms
The speedup is: 1.424654
```

Analysis:

1. **1,000 Elements**:
    - Sequential Sort: 1 ms
    - Parallel Sort: 4 ms
    - Speedup: 0.25

    For small data sizes (1000 elements), parallel sorting performs worse than sequential sorting. This is likely due to the overhead of parallel processing outweighing the benefits.

2. **10,000 Elements**:
    - Sequential Sort: 10 ms

- Parallel Sort: 19 ms
- Speedup: 0.5263158

For 10000 elements, parallel sorting still performs worse than sequential sorting, with the speedup being less than 1. The overhead continues to have a significant impact.

3. **100,000 Elements**:
    - Sequential Sort: 49 ms
    - Parallel Sort: 54 ms
    - Speedup: 0.9074074

    As the data size increases to 100000 elements, the performance of parallel sorting starts to get closer to sequential sorting, though it is still slightly slower. The speedup is approaching 1, indicating that the benefits of parallel processing are starting to be realized.

4. **1,000,000 Elements**:
    - Sequential Sort: 478 ms
    - Parallel Sort: 325 ms
    - Speedup: 1.4707693

    For large data sizes (1000000 elements), parallel sorting outperforms sequential sorting, with a speedup greater than 1. This shows that parallel sorting is more efficient for larger datasets, as the overhead becomes negligible compared to the sorting work itself.

Conclusion

- **Small Data Sizes (1000 to 10000 elements)**: Parallel sorting is less efficient due to the overhead of managing parallel tasks.
- **Medium Data Sizes (100000 elements)**: Parallel sorting becomes nearly as efficient as sequential sorting, with overhead starting to be compensated by the benefits of parallel processing.
- **Large Data Sizes (1000000 elements)**: Parallel sorting is significantly more efficient than sequential sorting, with a clear performance improvement and

speedup greater than 1.

Overall, parallel sorting demonstrates its strength in handling large datasets effectively, whereas sequential sorting remains more efficient for smaller datasets due to lower overhead.