ITF23019: Parallel and Distributed Programming

Report for Lab 3: Executor Framework

Name: Emil Berglund

GitHub: EmilB04

Exercise 1: Run codes in package *DemoExRunnable*

1.1: Output (2 Screenshots)

```
Parallel-distributed-programming/lab3 on  main [?]
❯ /usr/bin/env /usr/lib/jvm/java-21-openjdk-amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/emilb_linux/.vscode
-server/data/User/workspaceStorage/b3a646436b00f136f681dada5d0ef7fd/redhat.java/jdt_ws/lab3_e7b18db7/bin demoExRunnable.Main
Main: Start
pool-1-thread-1: Executing Task_0 during 5 seconds
pool-1-thread-2: Executing Task_1 during 8 seconds
pool-1-thread-3: Executing Task_2 during 1 seconds
pool-1-thread-4: Executing Task_3 during 3 seconds
pool-1-thread-5: Executing Task_4 during 6 seconds
pool-1-thread-6: Executing Task_5 during 2 seconds
pool-1-thread-7: Executing Task_6 during 5 seconds
pool-1-thread-8: Executing Task_7 during 8 seconds
pool-1-thread-9: Executing Task_8 during 3 seconds
pool-1-thread-10: Executing Task_9 during 6 seconds
pool-1-thread-11: Executing Task_10 during 6 seconds
pool-1-thread-12: Executing Task_11 during 9 seconds
pool-1-thread-13: Executing Task_12 during 9 seconds
pool-1-thread-14: Executing Task_13 during 0 seconds
pool-1-thread-14: Task Task_13: Finished
pool-1-thread-15: Executing Task_14 during 7 seconds
pool-1-thread-16: Executing Task_15 during 4 seconds
Main: Shut down the Executor
pool-1-thread-14: Executing Task_16 during 9 seconds
Main: End
pool-1-thread-3: Task Task_2: Finished
pool-1-thread-3: Executing Task_17 during 1 seconds
pool-1-thread-6: Task Task_5: Finished
pool-1-thread-6: Executing Task_18 during 5 seconds
```

```
pool-1-thread-8: Executing Task_92 during 4 seconds
pool-1-thread-11: Task Task_73: Finished
pool-1-thread-11: Executing Task_93 during 4 seconds
pool-1-thread-15: Task Task_74: Finished
pool-1-thread-15: Executing Task_94 during 1 seconds
pool-1-thread-3: Task Task_90: Finished
pool-1-thread-3: Executing Task_95 during 4 seconds
pool-1-thread-2: Task Task_69: Finished
pool-1-thread-2: Executing Task_96 during 2 seconds
pool-1-thread-15: Task Task_94: Finished
pool-1-thread-15: Executing Task_97 during 4 seconds
pool-1-thread-1: Task Task_91: Finished
pool-1-thread-1: Executing Task_98 during 1 seconds
pool-1-thread-4: Task Task_75: Finished
pool-1-thread-4: Executing Task_99 during 2 seconds
pool-1-thread-6: Task Task_80: Finished
pool-1-thread-13: Task Task_88: Finished
pool-1-thread-7: Task Task_86: Finished
pool-1-thread-1: Task Task_98: Finished
pool-1-thread-2: Task Task_96: Finished
pool-1-thread-5: Task Task_76: Finished
pool-1-thread-12: Task Task_77: Finished
pool-1-thread-16: Task Task_81: Finished
pool-1-thread-4: Task Task_99: Finished
pool-1-thread-8: Task Task_92: Finished
pool-1-thread-10: Task Task_85: Finished
pool-1-thread-14: Task Task_87: Finished
pool-1-thread-11: Task Task_93: Finished
pool-1-thread-3: Task Task_95: Finished
pool-1-thread-9: Task Task_79: Finished
pool-1-thread-15: Task Task_97: Finished
```

1.2: How different tasks are assigned to different threads

Firstly, The Executor creates a thread pool with 16 threads, which matches the number of available processors.

Secondly, the first 16 tasks are assigned to the 16 available threads in sequential order. When a thread completes its task, it becomes available and picks up the next pending task from the queue.

The tasks are assigned from a work queue in FIFO order, and no thread sits idle when tasks remain in the queue.

Lastly the ThreadPoolExecutor automatically balances the workload. Since task durations are random, threads finish at different times and dynamically pick up new work. This ensures that alle threads stay busy until the queue is empty.

1.3: Why messages are still printed after the main thread ends

The reason why messages are still being printed after the main thread ends is because the program is being shutdown using *executor.shutdown()* and not *executor.shutdownNow()*.

The current command initiates an orderly shutdown. This means that already submitted tasks continue to execute. New tasks on the other hand are not accepted. In this case it means that all the 100 tasks will be finished, but tasks defined after shutdown will not be executed.

To put another way would be to think of this as a restaurant:

- shutdown() = "Kitchen is closing, but we'll finish all orders already placed"
- Orders already in the queue: Will be made
- New orders after closing: Rejected

Exercise 2: Run code from demoExCallable

2.1: Output (2 Screenshots)

```
Parallel-distributed-programming/lab3 on  main [?]
❯ cd /home/emilb_linux/dev/V2026/Parallel-distributed-programming/lab3 ; /usr/bin/env /usr/lib/jvm/
java-21-openjdk-amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/emilb_linux/.vscode
-server/data/User/workspaceStorage/b3a646436b00f136f681dada5d0ef7fd/redhat.java/jdt_ws/lab3_e7b18db7
/bin demoExCallable.Main
Main: Starting.
pool-1-thread-1: Task Task_0: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-2: Task Task_1: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-2: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_1
pool-1-thread-5: Task Task_4: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-5: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_4
pool-1-thread-4: Task Task_3: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-4: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_3
pool-1-thread-3: Task Task_2: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-3: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_2
pool-1-thread-6: Task Task_5: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-6: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_5
pool-1-thread-7: Task Task_6: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-7: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_6
pool-1-thread-8: Task Task_7: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-8: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_7
pool-1-thread-9: Task Task_8: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-9: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_8
pool-1-thread-10: Task Task_9: Executed on: Thu Jan 22 15:43:58 CET 2026
pool-1-thread-10: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_9
Main: Number of completed tasks is 0
pool-1-thread-1: Task Thu Jan 22 15:43:58 CET 2026: Finished on: Task_0
Returned value is 2
Main: Task 0 isDone true
Returned value is 1
Main: Task 1 isDone true
Returned value is 3
Main: Task 2 isDone true
Returned value is 0
Main: Task 3 isDone true
Returned value is 9
Main: Task 4 isDone true
Returned value is 2
Main: Task 5 isDone true
```

```
Main: Task 5 isDone true
Returned value is 6
Main: Task 6 isDone true
Returned value is 1
Main: Task 7 isDone true
Returned value is 8
Main: Task 8 isDone true
Returned value is 1
Main: Task 9 isDone true
Main: Results
Core: 0 Task 2
Core: 1 Task 1
Core: 2 Task 3
Core: 3 Task 0
Core: 4 Task 9
Core: 5 Task 2
Core: 6 Task 6
Core: 7 Task 1
Core: 8 Task 8
Core: 9 Task 1
Main: Shut down the Executor
Main: End

Parallel-distributed-programming/lab3 on  main [?]
❯
```

2.2: What happens when calling *get()* method from a task that is not done

If we call the *get()* method while a task is not finished, the calling thread will wait until the task is finished. When finished, it will return the computed value (Long in this case) or an exception if it task threw and exception.

Exercise 3: Write multithread program – 100 tasks

Code is under attached files in folder “exercise3”

Below are two screenshots from the output, one from the first 10, and one from the last 10.

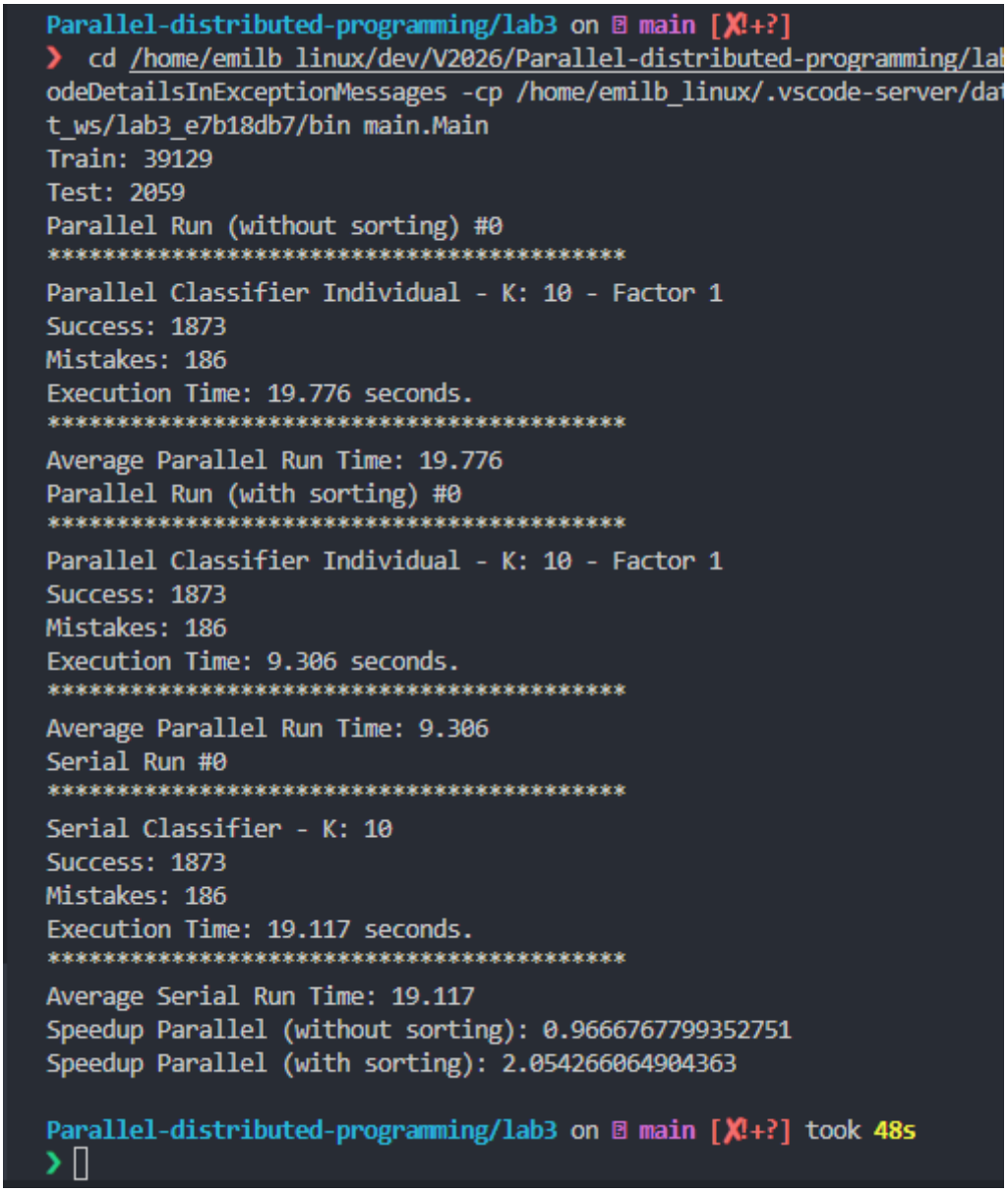
```
Parallel-distributed-programming/lab3/exercise3_4 on  main [!?]
❯ cd /home/emilb_linux/dev/V2026/Parallel-distributed-programming/lab3/exerci
/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/emilb_linux/dev/V2026/
Main: Starting.
Task 1: Sum from 1 to 1 = 1
Task 2: Sum from 1 to 2 = 3
Task 3: Sum from 1 to 3 = 6
Task 4: Sum from 1 to 4 = 10
Task 5: Sum from 1 to 5 = 15
Task 21: Sum from 1 to 21 = 231
Task 22: Sum from 1 to 22 = 253
Task 23: Sum from 1 to 23 = 276
Task 24: Sum from 1 to 24 = 300
Task 6: Sum from 1 to 6 = 21
Task 7: Sum from 1 to 7 = 28
Task 8: Sum from 1 to 8 = 36
Task 9: Sum from 1 to 9 = 45
Task 10: Sum from 1 to 10 = 55
```

```
Task 90: Sum from 1 to 90 = 4095
Task 91: Sum from 1 to 91 = 4186
Task 92: Sum from 1 to 92 = 4278
Task 93: Sum from 1 to 93 = 4371
Task 94: Sum from 1 to 94 = 4465
Task 95: Sum from 1 to 95 = 4560
Task 96: Sum from 1 to 96 = 4656
Task 97: Sum from 1 to 97 = 4753
Task 98: Sum from 1 to 98 = 4851
Task 99: Sum from 1 to 99 = 4950
Task 100: Sum from 1 to 100 = 5050
Main: All tasks completed.

Parallel-distributed-programming/lab3/exercise3_4 on  main [!?]
❯
```

Exercise 4: Parallel program with Speedup

4.1: Complete the implementation of the Main.java



4.2: How Parallel versions are better than serial version:

Parallel versions are better than serial versions mostly because it can divide the work on multiple CPU cores. This allows tasks to run simultaneously instead of one after another. Furthermore, this reduces the total execution time and increases throughput, especially in larger datasets.

4.3: Which parallel version is better and why:

The parallel version with sorting is clearly better. The execution time (9.3 seconds) and speedup (2.05 times) are significantly higher than the “without sorting” version (19.8 seconds, 0.97 times).

Sorting helps the threads process data more efficiently by improving data organization and cache locality, reducing synchronization overhead, and ensuring faster access patterns. The “without sorting” version likely suffers from unbalanced workloads or inefficient memory access, offsetting the benefits of parallelism.

4.4: Changing the size of the dataset

Case	K	RUNS	Parallel (no sort)	Parallel (sort)	Serial	Speedup (no sort)	Speedup (sort)
1	10	1	19.81 s	12.56 s	25.23 s	1.27×	2.01×
2	40	2	20.29 s	12.57 s	22.59 s	1.11×	1.80×
3	40	5	19.22 s	10.12 s	22.58 s	1.17×	2.23×

- Parallel (with sorting) consistently performs better than both the serial version and the unsorted parallel version.
- As K increases (meaning more computation per test sample), both parallel versions slow down, but sorting still gives a higher speedup.
- Increasing RUNS (number of repetitions) gives more stable timing and slightly improves the consistency of the speedup result.