Parallel and Distributed Programming
Lab 7

In this lab our task was to perform both trivial and using the Karatsuba algorithm multiplications of polynomials of arbitrary sizes using the **MPI protocol**. I will firstly present the main idea of the algorithms and then show a benchmark of comparative running time analysis.

Algorithms:

1. **Trivial Parallel**

In this I enhanced the algorithm described at 1. by parallelizing using an ExecutorService with a variable number of threads one of the loops described above. That is, I parallely did the calculations for each of the coefficients of the first polynomial and after all I just added up the results in a sequential manner.

2. **Karatsuba Parallel**

Here I implemented the divide-and-conquer algorithm developed by Karatsuba and described here: https://en.wikipedia.org/wiki/Karatsuba_algorithm. This starts from the idea that it's faster to perform a little bit more additions than do many multiplications, and, subsequently, does the following: splits each polynomial in 3 parts, the first half, the second half and one overlapping part, that sums up the 2 mentioned before. If we are in the base case when both polynomials are of degree one, do the trivial computation. Else, do recursively down the tree the same operation and then, having obtained the results, we combine them in the following manner: shift the high part to its place by adding the needed number of zeroes, remove what is above the correct result (the overlapping part) and then add the low part. We parallelize the algorithm described at 3. by using an ExecutorService for the recursion, so that each is done parallely. The catch is that we need a new Executor for each iteration since the threads go one way and the recursion in the other. In order to buy efficiency, when the polynomials are small enough, we go to sequential again, and also when the depth is over a given threshold. The rest of the algorithm is the same.

Communication:

The MPJ (MPI for Java) framework provides a handy way to distribute work between tasks. Thus, it creates the given number of processes and then provides a protocol for inter-process communication.

Sending data to a process is done like this:

```
MPI.COMM_WORLD.Send(new Object[]{p}, 0, 1, MPI.OBJECT, i, 0);
```

The relevant part for us is the first argument, which is the passed data, and the penultimate, which represents the ID of the receiver process.

To receive data we do:

```
MPI.COMM_WORLD.Recv(p, 0, 1, MPI.OBJECT, 0, 0);
```

Where the first argument is the received data and the penultimate is the ID from which process' is received from.

The hack created by Mr. Alex Popa is that we can zero-out all the positions we do not want and do the computations only for the relevant part for the respective thread and then combine them together again.

Benchmark:

Algorithm	Size	Time
Trivial Parallel classic	10	0.24s
Trivial Parallel classic	50	0.22s
Trivial Parallel classic	100	0.03s
Karatsuba Parallel classic	10	0.067s
Karatsuba Parallel classic	50	0.91s
Karatsuba Parallel classic	100	1.98s

Trivial Parallel MPI	10	0.291s
Trivial Parallel MPI	50	0.125s
Trivial Parallel MPI	100	0.187s
Karatsuba Parallel MPI	10	0.093s
Karatsuba Parallel MPI	50	0.233s
Karatsuba Parallel MPI	100	0.374s