

# Introduction to Parallel and Distributed Computing Exercise 3 (June 6)

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above via the Moodle interface as a .zip or .tgz file which contains

- a single PDF (.pdf) file with
  - a cover page with the title of the course, your name, Matrikelnummer, and email-address,
  - a section with the source code of the program benchmarked, the output of the parallelizing compiler, and an explanation of the output,
  - a section with the raw data of the benchmarks,
  - a section with a summary table and graphical diagrams of the benchmarks.
- the source (.java) file(s) of the programs.

### Exercise 3: Multi-Threaded/Network Programming in Java

The goal of this exercise is to develop a multi-threaded client/server version of the “all pairs shortest paths” problem presented in Exercise 1; the solution shall be implemented in the programming language Java using Java’s concurrency and networking API. Use for this exercise the most recent version of Java available (e.g., module `load jdk/11.0.1+13`, see `module avail jdk` for all installed Java versions).

First, create a sequential Java solution for the problem; you may use the provided sample program `MatMult.java` for matrix multiplication as a starting point of your solution. Benchmark the program with two appropriate values for  $N$  (not necessarily the same as in Exercises 1/2, at least one value  $N$  shall let the program run for at least one minute).

Next, develop a multi-threaded version of the program. Use the high-level concurrency API to manage a fixed size pool of  $T$  threads among which tasks are scheduled each of which processes a block  $B$  of iterations of the squaring algorithm (generate the tasks as instances of `Callable` and use for task submission the method `invokeAll()` which blocks until all tasks have been processed); experiment to find a suitable value for  $B$ . Please note that the pool is to be created only *once* before the algorithm is started and subsequently *reused* for every “squaring” operation.

Write the program such that it can be started in one of two ways:

1. With the command line parameter `-server`: in this case the program is executed as a server which repeatedly waits (on some designated port) for the request of a client to create a random matrix of dimension  $N$  with seed  $R$  for the random number generator and solves the problem with  $T$  threads; the server sends back to the client the number  $M$  of milliseconds that the solution of the equation system took.
2. With the command line parameter `-client N B R T`: in this case, the program is started as a client that contacts the server on the designated port, sends the parameters  $N$ ,  $B$ ,  $R$ , and  $T$  to the server, waits for the result  $M$ , and prints  $M$  to the standard output.

Both server and clients may be run on the same machine. Please note that for the Java solution you may use the programs `MatMultPool.java` and `MatMultNet.java` posted on the course site as a pattern for your own solution.

For generating random numbers, use the class `java.util.Random`<sup>1</sup> of the Java standard library. For instance, assuming the declaration `import java.util.*`; the code

```
Random r = new Random(R);
for (int i=0; i<100; i++)
    System.out.println(r.nextDouble());
```

prints 100 floating point numbers generated by a random number generator with seed  $R$ .

For benchmarking Java programs, you may use the function

```
System.currentTimeMillis()
```

which returns the current wall clock time in milliseconds.

Make sure that threads are pinned to freely available cores by executing a command like

<sup>1</sup><https://docs.oracle.com/javase/8/docs/api/java/util/Random.html>

```
dplace -c 64-91 program ...
```

which pins all threads to 32 physical cores (the numbers refer to the cpu partition in the upper half of the machine). Use `top` to verify the applied thread/core mapping and the thread's share of CPU time (which should be close to 100%).

Report the results as in Exercise 2 (state the version of Java that you used).

**Alternative** If you prefer, you may elaborate this exercise in C/C++ using Posix threads and Unix sockets (also using `dplace` for pinning threads to cores). In that case, you may simply split the  $N$  rows into  $N/T$  blocks each of which is processed by one thread. Use `srand()` and `rand()` for random number generation and measure times with `clock_gettime` (as in Exercise 1).