The learning outcome of this practical session is to parallelize the matrix multiplication, a CPU and memory-bound application.

While jshell provides a convenient environment for prototyping and testing, memory allocations are inefficient. Therefore, all performance measurements must be performed with a compiled executable (through an IDE or directly javac).

1 Parallel Matrix Addition

The following code allows adding two matrices using a fork-join pool:

```java
class MatrixAdd extends RecursiveAction {
    static final int THRESHOLD = 10;
    static double[][] A, B;
    static double[][] res;
    final int i1, i2;
    final int j1, j2;
    MatrixAdd(double[][] A, double[][] B) {
        this(0, A.length, 0, A[0].length);
        this.A = A;
        this.B = B;
        res = new double[A.length][A[0].length];
    }
    MatrixAdd(int i1, int i2, int j1, int j2) {
        this.i1 = i1;
        this.i2 = i2;
        this.j1 = j1;
        this.j2 = j2;
    }
    protected void compute() {
        if (i1 == i2 || j1 == j2)
            return;
        else if ((i2 - i1 < THRESHOLD && j2 - j1 < THRESHOLD) {
            for (int i = i1; i < i2; i++)
                for (int j = j1; j < j2; j++)
                    res[i][j] = A[i][j] + B[i][j];
        } else {
            int imed = (i2 + i1) / 2;
            int jmed = (j2 + j1) / 2;
            invokeAll(new MatrixAdd(i1, imed, j1, jmed),
                new MatrixAdd(i1, imed, jmed, j2),
                new MatrixAdd(imed, i2, j1, jmed),
                new MatrixAdd(imed, i2, jmed, j2));
        }
    }
}
```
The algorithm is based on the following decomposition:

\[
\sum_{i=1}^{n} \left( \begin{array}{cc}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{array} \right) + \sum_{j=1}^{n} \left( \begin{array}{cc}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{array} \right) = \left( \begin{array}{cc}
A_{11} + B_{11} & A_{12} + B_{12} \\
A_{21} + B_{21} & A_{22} + B_{22}
\end{array} \right)
\]

To assess the correctness and the performance of this parallel algorithm, propose a sequential implementation of the matrix addition. We will check the result by using random square matrices of size \( n = 1000 \). Write a function that generates a random square matrix. Finally, write a function that measures the difference between two matrices. We can now validate the parallel algorithm and measure its performance.

2 Parallel Matrix Multiplication

Extends the previous solution to perform the matrix multiplication in parallel with the fork-join pool.

Study the effect of the threshold on the performance.

3 Parallel Stream Matrix Multiplication

Implement the matrix multiplication with a stream.

Compare the result and the performance for the sequential and parallel stream versions (measure the effect of parallelizing the most inner and outer streams).