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

The first 2 sections are essential.

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),
```

```java
                       new MatrixAdd(imed, i2, j1, jmed),
```
```java```
The algorithm is based on the following decomposition:

\[
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
+ \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
= \begin{pmatrix}
A_{11} + B_{11} & A_{12} + B_{12} \\
A_{21} + B_{21} & A_{22} + B_{22}
\end{pmatrix}
\]

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.

```java
double[][] randomMat(int m) {
    double[][] res = new double[m][m];
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++)
            res[i][j] = Math.random();
    return res;
}

double[][] seqAdd(double[][] A, double[][] B) {
    double[][] res = new double[A.length][A[0].length];
    for (int i = 0; i < A.length; i++)
        for (int j = 0; j < A[i].length; j++)
            res[i][j] = A[i][j] + B[i][j];
    return res;
}

double diff(double[][] A, double[][] B) {
    double diff = 0;
    for (int i = 0; i < A.length; i++)
        for (int j = 0; j < A[i].length; j++)
            diff += Math.abs(A[i][j] - B[i][j]);
    return diff;
}

int m = 1000;
var A = randomMat(m);
var B = randomMat(m);

var t = System.nanoTime();
var res = seqAdd(A, B);
(System.nanoTime() - t) / 1e9;

var t = System.nanoTime();
var ma = new MatrixAdd(A, B);
ForkJoinPool.commonPool().invoke(ma);
(System.nanoTime() - t) / 1e9;

diff(res, ma.res);
```
The sum of two matrices does not take enough time to measure a performance gain (most of the
time is spent allocating and initializing the memory). The main issue is that the matrix addition is
memory-bound: there are far less CPU operations than memory ones.
It is even slower, especially with such a low threshold. Increasing its value to 100 provides
comparable performance.

2 Parallel Matrix Multiplication

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

```java
class MatrixMult extends RecursiveAction {
    static final int THRESHOLD = 10;
    static double[][] A, B;
    static double[][] res;
    final int i1, i2;
    final int j1, j2;
    final int k1, k2;
    MatrixMult(double[][] A, double[][] B) {
        this(0, A.length, 0, B[0].length, 0, B.length);
        this.A = A;
        this.B = B;
        res = new double[A.length][B[0].length];
    }
    MatrixMult(int i1, int i2, int j1, int j2, int k1, int k2) {
        this.i1 = i1;
        this.i2 = i2;
        this.j1 = j1;
        this.j2 = j2;
        this.k1 = k1;
        this.k2 = k2;
    }
    protected void compute() {
        if (i1 == i2 || j1 == j2 || k1 == k2)
            return;
        else if ((i2 - i1 < THRESHOLD && j2 - j1 < THRESHOLD && k2 - k1 < THRESHOLD)) {
            for (int i = i1; i < i2; i++)
                for (int j = j1; j < j2; j++)
                    for (int k = k1; k < k2; k++)
                        res[i][j] += A[i][k] * B[k][j];
        } else {
            int imed = (i2 + i1) / 2;
            int jmed = (j2 + j1) / 2;
            int kmed = (k2 + k1) / 2;
            invokeAll(new MatrixMult(i1, imed, j1, jmed, k1, kmed),
                    new MatrixMult(i1, imed, jmed, j2, k1, kmed),
                    new MatrixMult(imed, i2, j1, jmed, k1, kmed),
                    new MatrixMult(imed, i2, jmed, j2, k1, kmed));
            invokeAll(new MatrixMult(i1, imed, j1, jmed, kmed, k2),
                    new MatrixMult(i1, imed, jmed, j2, kmed, k2),
                    new MatrixMult(imed, i2, j1, jmed, kmed, k2),
                    new MatrixMult(imed, i2, jmed, j2, kmed, k2));
        }
    }
}
```
double[][] seqMult(double[][] A, double[][] B) {
    double[][] res = new double[A.length][A[0].length];
    for (int i = 0; i < A.length; i++)
        for (int j = 0; j < B[0].length; j++)
            for (int k = 0; k < B.length; k++)
                res[i][j] += A[i][k] * B[k][j];
    return res;
}

int m = 1000;
var A = randomMat(m);
var B = randomMat(m);

var t = System.nanoTime();
var res = seqMult(A, B);
(System.nanoTime() - t) / 1e9;

var t = System.nanoTime();
var ma = new MatrixMult(A, B);
ForkJoinPool.commonPool().invoke(ma)
(System.nanoTime() - t) / 1e9;

diff(res, ma.res);

The algorithm is based on the following decomposition:

\[
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
= \begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
\]

This implementation is able to achieve a correct speed-up because the application has more CPU
operations than with the addition. Memory accesses are thus negligible for the multiplication.

Study the effect of the threshold on the performance.

The threshold should not be too small (at least 10, or even 20) but below the size of the matrix
divided by the number of threads to have at least one task per thread.

3 Parallel Stream Matrix Multiplication

Implement the matrix multiplication with a stream.

```java
var t = System.nanoTime();
IntStream.range(0, m1.length)
    .parallel()
```
.mapToObj(i ->
    IntStream.range(0, m2[0].length)
    .mapToDouble(j ->
        IntStream.range(0, m2.length)
        .mapToDouble(k -> m1[i][k] * m2[k][j])
        .sum())
    .toArray())
    .toArray(double[][]::new);
(System.nanoTime() - t) / 1e9;

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