

**Esercizio 1**

L'applicazione originale ripete, all'interno di un for loop, per 100'000 volte l'operazione di moltiplicazione tra due matrici. Ogni moltiplicazione esegue su matrici indipendenti fa loro, di conseguenza è possibile eseguire le singole moltiplicazioni in parallelo. Per sfruttare le funzionalità dell'executor framework di Java, è necessario individuare le parti di codice da isolare in light-weight tasks, implementando l'interfaccia Callable o l'interfaccia Runnable. In questo caso specifico, visto che l'operazione di moltiplicazione fra matrici produce un risultato, è conveniente utilizzare l'interfaccia Callable.

In seguito, all'interno del metodo main, utilizzando il metodo statico Executors.newFixedThreadPool(), è possibile istanziare un ExecutorService al quale inviare i tasks responsabili delle singole moltiplicazioni.

```
class MatrixMultiplicationTask implements Callable<int[][]> {
    private final int[][] m0;
    private final int[][] m1;

    public MatrixMultiplicationTask(final int[][] m0, final int[][] m1) {
        this.m0 = m0;
        this.m1 = m1;
    }

    @Override
    public int[][] call() throws Exception {
        final int[][] m2 = new int[m0[0].length][m1.length];
        // Moltiplica matrici
        for (int i = 0; i < m0[0].length; i++)
            for (int j = 0; j < m1.length; j++)
                for (int k = 0; k < m0.length; k++)
                    m2[i][j] += m0[i][k] * m1[k][j];
        return m2;
    }
}

public class S10Esercizio1 {
    public static final int NUM_OPERATIONS = 100_000;
    public static final int MATRIX_SIZE = 64;

    public static void main(final String[] args) {
        final Random rand = new Random();
        final int numThreads = Runtime.getRuntime().availableProcessors();
        final ExecutorService execServ = Executors.newFixedThreadPool(numThreads);
        final S10Es1Timer time = new S10Es1Timer();

        System.out.println("Simulazione iniziata");
        time.start();
        for (int operation = 0; operation < NUM_OPERATIONS; operation++) {
            // Crea matrici
            final int[][] m0 = new int[MATRIX_SIZE][MATRIX_SIZE];
            final int[][] m1 = new int[MATRIX_SIZE][MATRIX_SIZE];

            // Inizializza gli array con numeri random
            for (int i = 0; i < MATRIX_SIZE; i++)
                for (int j = 0; j < MATRIX_SIZE; j++) {
                    m0[i][j] = rand.nextInt(10);
                    m1[i][j] = rand.nextInt(10);
                }
            execServ.submit(new MatrixMultiplicationTask(m0, m1));
        }
        execServ.shutdown();
        while (!execServ.isTerminated()) { /* Busy wait until terminated */ }
        time.stop();
        System.out.println("Simulazione terminata");
        System.out.println("(ExecutorService) Tempo impiegato : " + time.getElapsed() + " ms");
    }
}
```

## Esercizio 2

Come prima cosa, è necessario modificare la classe, sostituendo il tipo di dato di ritorno e aggiungendo la logica per eseguire la somma.

```
class MatrixOperationTask implements Callable<Long> {
    private final int[][] m0;
    private final int[][] m1;

    public MatrixOperationTask (final int[][] m0, final int[][] m1) {
        this.m0 = m0;
        this.m1 = m1;
    }

    @Override
    public Long call() throws Exception {
        final int[][] m2 = new int[m0[0].length][m1.length];

        // Moltiplica matrici
        for (int i = 0; i < m0[0].length; i++)
            for (int j = 0; j < m1.length; j++)
                for (int k = 0; k < m0.length; k++)
                    m2[i][j] += m0[i][k] * m1[k][j];

        // Calcola somma
        long somma = 0;
        for (int x = 0; x < m2.length; x++)
            for (int y = 0; y < m2[0].length; y++)
                somma += m2[x][y];

        return somma;
    }
}
```

Per recuperare i risultati, s'introduce una lista di Futures, che viene popolata con le Futures restituite all'invocazione del metodo submit. In seguito, dopo aver accodato tutte le operazioni, si possono recuperare i risultati, invocando il metodo get() di ogni Future. L'applicazione termina quando sono stati recuperati tutti i risultati.

```
public class S10Esercizio2 {
    public static final int NUM_OPERATIONS = 100_000;
    public static final int MATRIX_SIZE = 64;

    public static void main(final String[] args) {
        final Random rand = new Random();
        final S10Es2Timer time = new S10Es2Timer();
        final int numThreads = Runtime.getRuntime().availableProcessors();
        final ExecutorService execServ = Executors.newFixedThreadPool(numThreads);
        final List<Future<Long>> futureResults = new ArrayList<Future<Long>>();

        time.start();
        for (int operation = 0; operation < NUM_OPERATIONS; operation++) {
            final int[][] m0 = new int[MATRIX_SIZE][MATRIX_SIZE];
            final int[][] m1 = new int[MATRIX_SIZE][MATRIX_SIZE];

            // Inizializza gli array con numeri random
            for (int i = 0; i < MATRIX_SIZE; i++)
                for (int j = 0; j < MATRIX_SIZE; j++) {
                    m0[i][j] = rand.nextInt(10);
                    m1[i][j] = rand.nextInt(10);
                }
            final Future<Long> futureResult = execServ.submit(new MatrixOperationTask(m0, m1));
            futureResults.add(futureResult);
        }

        execServ.shutdown();

        long max = -1;
        for (final Future<Long> future : futureResults) {
            try {
                final long result = future.get().longValue();
                if (max < result)
                    max = result;
            }
        }
    }
}
```

```
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}

if (!execServ.isTerminated()) {
    try {
        // Wait a while for existing tasks to terminate
        if (!execServ.awaitTermination(60, TimeUnit.SECONDS)) {
            execServ.shutdownNow(); // Cancel currently executing tasks
            // Wait a while for tasks to respond to being cancelled
            if (!execServ.awaitTermination(60, TimeUnit.SECONDS))
                System.err.println("Pool did not terminate");
        }
    } catch (InterruptedException ie) {
        // (Re-)Cancel if current thread also interrupted
        execServ.shutdownNow();
        // Preserve interrupt status
        Thread.currentThread().interrupt();
    }
}
time.stop();
System.out.println("Elapsed time : " + time.getElapsed() + " ms");
System.out.println("Max sum found: " + max);
}
```

### Esercizio 3

Sfruttando il `CompletionService` si può eliminare la lista di `Futures`, introducendo un oggetto di tipo `ExecutorCompletionService` al quale, in fase di costruzione, va passato l'executor. I tasks da eseguire vengono inviati al `CompletionService` che si occuperà, sia di inoltrare i tasks all'executor, sia di gestire le `Futures` risultanti. Infine, per recuperare le `Futures` si usa il metodo `take()` del `CompletionService`.

```
public class S10Esercizio3 {
    public static final int NUM_OPERATIONS = 100_000;
    public static final int MATRIX_SIZE = 64;

    public static void main(final String[] args) {
        final Random rand = new Random();
        final S10Es3Timer time = new S10Es3Timer();
        final int numThreads = Runtime.getRuntime().availableProcessors();
        final ExecutorService execServ = Executors.newFixedThreadPool(numThreads);
        final CompletionService<Long> completionService = new ExecutorCompletionService<Long>(execServ);

        time.start();
        for (int operation = 0; operation < NUM_OPERATIONS; operation++) {
            final int[][] m0 = new int[MATRIX_SIZE][MATRIX_SIZE];
            final int[][] m1 = new int[MATRIX_SIZE][MATRIX_SIZE];

            // Inizializza gli array con numeri random
            for (int i = 0; i < MATRIX_SIZE; i++)
                for (int j = 0; j < MATRIX_SIZE; j++) {
                    m0[i][j] = rand.nextInt(10);
                    m1[i][j] = rand.nextInt(10);
                }
            completionService.submit(new MatrixOperationTask(m0, m1));
        }

        execServ.shutdown();

        long max = -1;
        for (int operation = 0; operation < NUM_OPERATIONS; operation++) {
            try {
                final Future<Long> take = completionService.take();
                final Long result = take.get();
                if (max < result)
                    max = result;
            } catch (final InterruptedException e) {
                e.printStackTrace();
            } catch (final ExecutionException e) {
                e.printStackTrace();
            }
        }

        while (!execServ.isTerminated()) {
            // Busywait until terminated
        }
        time.stop();
        System.out.println("Elapsed time : " + time.getElapsed() + " ms");
        System.out.println("Max sum found: " + max);
    }
}
```

## Esercizio 4

Per sfruttare l'ExecutorFramework, bisogna sostituire l'array di threads con la variabile *executorService* di tipo *Executor* e scambiare la creazione dell'array di threads con la creazione di un oggetto *Executor*. In questo caso, è stato scelto un *FixedThreadPool*, creandolo attraverso il factory-method *newFixedThreadPool* della classe *Executors*. Invece di creare nuovi threads, bisogna creare dei *Runnable*s (in questo caso sfruttando la lambda expression), che in seguito vengono inoltrati all'executor framework tramite il metodo *execute*.

```
public class S9PerformanceAnalysis extends JPanel {
    private static final long serialVersionUID = -765326845524613343L;

    private Executor executorService;

    [...]

    /**
     * This method is called when the user clicks the Start button, while no
     * computation is in progress. It starts as many new threads as the user has
     * specified, and assigns a different part of the image to each thread. The
     * threads are run at lower priority than the event-handling thread, in
     * order to keep the GUI responsive.
     */
    void start() {
        [...]
        executorService = Executors.newFixedThreadPool(threadCount);

        for (int i = 0; i < threadCount; i++) {
            [...]
            executorService.execute(() -> {
                try {
                    // Compute one row of pixels.
                    for (int row = startRow; row <= endRow; row++) {
                        final int[] rgbRow = fractal.computeRow(row);
                        // Check for the signal to abort the computation.
                        if (!running)
                            return;
                        imagePanel.setRowAndUpdate(rgbRow, row);
                    }
                } finally {
                    // make sure this is called when the thread finishes for
                    // any reason.
                    threadFinished();
                }
            });
        }
    }

    /**
     * Called by each thread upon completing it's work
     */
    synchronized void threadFinished() {
        final int numThreads = 1 + threadCountSelect.getSelectedIndex();
        threadsCompleted++;
        if (threadsCompleted == numThreads) {
            // all threads have finished
            startButton.setText("Start");
            startButton.setEnabled(true);
            // Make sure running is false after the thread ends.
            running = false;

            executorService = null;
            threadCountSelect.setEnabled(true); // re-enable pop-up menu
        }
    }
}
```

**Esercizio 5**

Per misurare i tempi d'esecuzione è possibile utilizzare la classe *Timer* più sotto. Per utilizzarla è sufficiente introdurre una chiamata al metodo *start* di un oggetto *Timer* all'interno del metodo *start* della classe *S10Mandelbrot*, rispettivamente una chiamata al metodo *stop* quando tutte le thread hanno completato il proprio lavoro. In seguito, è possibile stampare in console il tempo trascorso.

```
class Timer {  
    private long start = -1, stop = -1;  
  
    public void start () {  
        this.start = System.currentTimeMillis();  
    }  
  
    public void stop() {  
        this.stop = System.currentTimeMillis();  
    }  
  
    public long getElapsed() {  
        if (start < 0 || stop < 0)  
            return 0;  
        return stop - start;  
    }  
}
```

Con le misure ottenute è possibile calcolare la progressione di speed-up fra la versione seriale (1 solo thread) e le versioni parallele (2 o più threads), ottenendo, di conseguenza, il grafico di scalabilità dell'applicazione. Più sotto vengono riportate le misure ed il grafico di scalabilità ottenuti eseguendo il programma su un computer con 4 cores fisici e hyperthreading:

Numero Threads	Tempo [ms]	Speedup	Numero Threads	Tempo [ms]	Speedup
1	13204	1.00	9	2225	5.94
2	6882	1.92	10	2253	5.86
3	5189	2.54	11	2172	6.08
4	3978	3.32	12	2165	6.10
5	3450	3.83	13	2084	6.34
6	2862	4.62	14	2049	6.44
7	2554	5.17	15	2144	6.16
8	2397	5.51	16	2231	5.92

