Università di Pisa
Distributed systems: paradigms and models

Game Of Life - Multithread Implementation
Project Report

Gabriele Marini - 522910

Abstract

The goal of this project is the implementation of "Game Of Life" in Java, both in a sequential and parallel way. For the parallel implementation of the game, Java's threads and a CyclicBarrier, the Skandium framework and the new stream features of Java 8 have been used. The project has been realized using a Map skeleton, and tests have been performed to observe how a parallel implementation of the application could improve the performances. The tests have been performed on two different machines, a Xeon E5-2650 @2.00GHz with 8 cores and 16 threads whose credentials have been provided during the course's classes, and on an Intel i7 4790k @4.00ghz with 4 cores and 8 contexts (Results and graphs of the latter may be found in the Statistiche folder of the project archive).

1 Introduction **2**
2 Design **2**
2.1 Data Structures 2
2.2 Sequential Algorithm 4
2.3 Parallel Implementation 4
2.3.1 The Map model 4
2.3.2 Splitting and merging the board 5
3 Implementation **6**
3.1 Sequential 6
3.2 Multithread 6
3.3 Skandium 7
3.4 Java8 7
3.4 Tests 8
4 Results **8**
5 Conclusions **9**
Appendix **10**
A1 Usage 10

A2 Data Collected and Graphs 12

1 Introduction

The purpose of this report is to show the reasoning behind the design, the implementation and the performances evaluation choices. The parallel version of the software has been implemented in three different ways in Java: By means of the Java threads, by using the Skandium framework and by taking advantage of the new Java 8 stream features.

In the appendix of this report it will be shown how to use the application along with some graphs and test results, performed on a Intel Xeon E5-2650 @2.00GHz with 8 cores and 16 context.

2 Design

The first phase's concern is the design of the main algorithm of the game, with the appropriate data structures to represent the board and the cells inside it. The first designed version of the algorithm was in a sequential form.

The rules of Conway's Game of Life are quite simple, given a matrix of cells, which can be dead or alive. Every cell interact with its eight neighbors, and at each new step, every cell's new state is computed by applying the following rules:

- Any live cell with fewer than two alive neighbors dies.
- Any live cell with two or three alive neighbors stay alive.
- Any live cell with more than three alive neighbors dies.
- Any dead cell with exactly three alive neighbors becomes a live cell.

In this implementation of the Game, the cells located at the corners of the board, will interact with the cells on the other side of the board, as if the board was continuously connected to itself. So for example, if a "glider" (a particular set of alive cells in a 9x9 square such that it moves diagonally as the steps go) get to the lower rightmost corner of the board, at the next iteration it will appear at the top leftmost corner of the board.

2.1 Data Structures

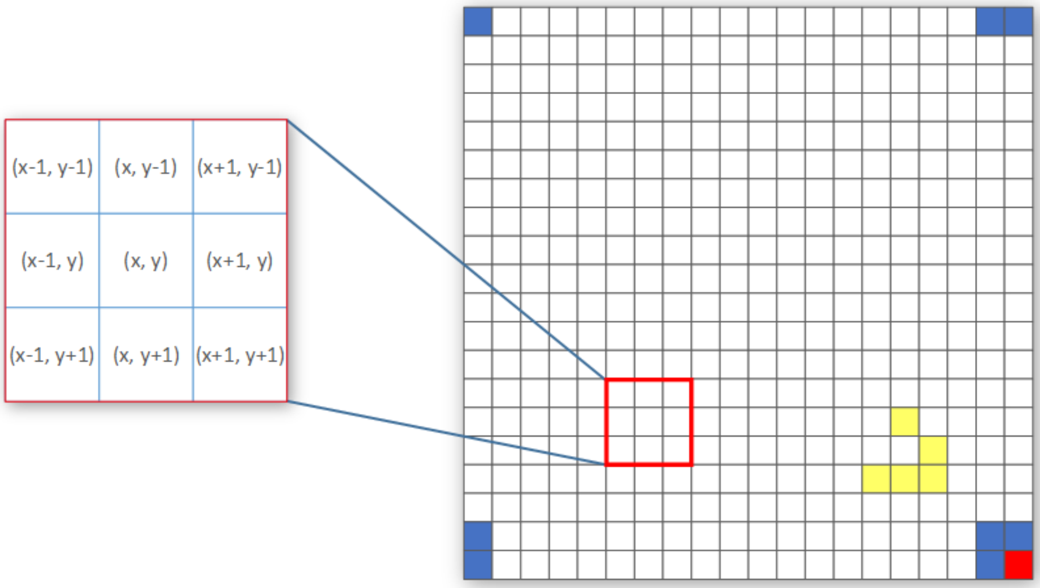
It has been decided to represent the board as a 2-dimensional matrix made of integers numbers represented in bytes, whereas the byte 1 represent an alive cell while the byte 0 is a dead cell. In this way it has been possible to reduce the amount of memory used by the program, using just 8 bit per cell. Besides, it will be quite easier to compute the alive neighbors of a cell by just adding up the byte values around a single cell.

The main classes of the game are located in the package `it.gabrielemarini.spm.gameoflife`:

- `Board:` This class is the most important one, it contains the byte matrices (readingBoard, writingBoard) and all the needed methods to compute the steps of the game, along with functions for initializing (a random or empty board or stored in a text file), storing (to a file or to an image) , comparing or splitting the board.
- `GraphicBoard:` This class contains a graphical representation of the game iterations by using the Java Swing libraries and a Jpanel with a BufferedImage, which observes the board and update the

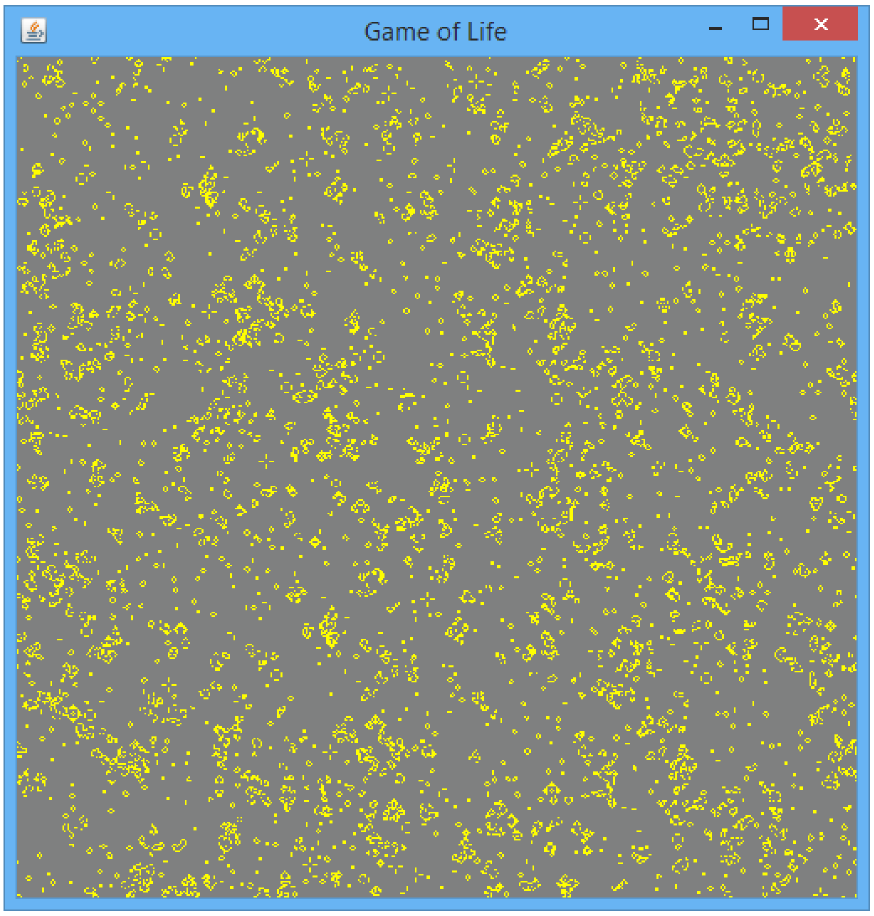
screens whenever the board is modified, generated a binary image from the boards values (1s and 0s). The graphical representation of the game has been disabled by default for testing purposes.

- `BoardChunk`: This class is used for storing the bounds of the ranges of rows for every worker to compute.



As it is evident from the shown image, cells located at the corners will have as neighbors, cells located at the other corners, so the glider shown (the set of yellow cells) will start moving again from the top leftmost corner after hitting the lower rightmost corner.

Here's an example of the graphical representation of the board.



2.2 Sequential Algorithm

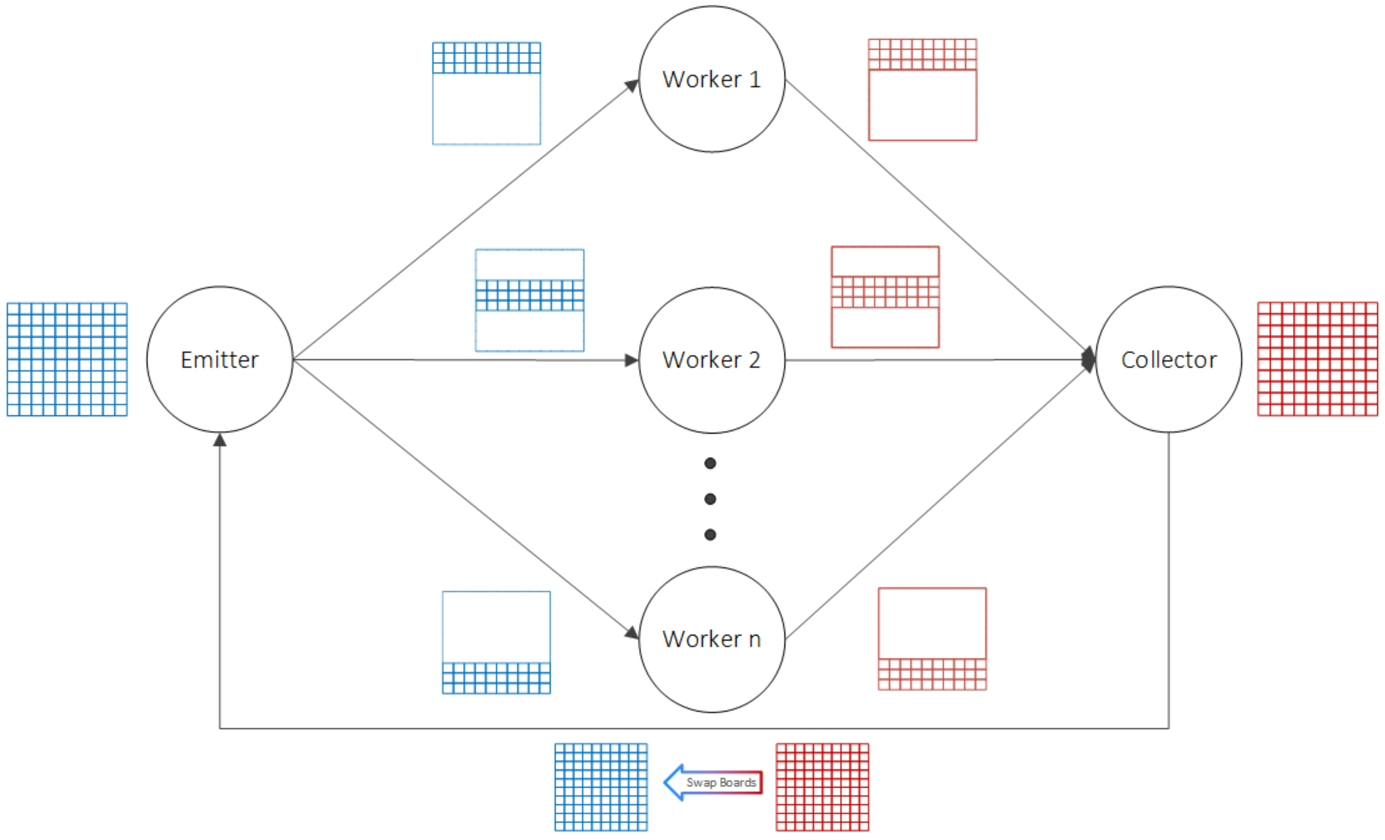
The main stage of the application is fairly easy, as it's just an implementation of the Game of Life rules. I chose to define two identical size-wise matrix, using one of the matrices to read the state of a cell (namely, the `readingBoard`) and compute its neighbors by reading the cells around it. The second identical matrix will be used for storing the new state of the cell (`writingBoard`). I will write down the main steps performed by the algorithm:

1. Iterate through all the cells of the `readingBoard`, for each cell:
    - Compute the alive neighbors around it by reading and adding up their values from the readingBoard
    - Write `the` new state of the cell in the `writingBoard`
2. Swap the pointers of the two matrices
3. Start again from the step 1, this time the algorithm will be executed on the `writingBoard` since the boards pointers have been switched, and since the old values stored in the `readingBoard` will not be used anymore.

2.3 Parallel Implementation

2.3.1 The Map model

The model is structured as a Map, so that the Map only computes a single iteration. I've found different ways of dealing with the generation of the tasks for each worker, which will be explained in the next sub-chapter.



This is a graphical sketch of the parallel pattern used for the parallel implementation.

The pattern used to realize the parallel version of Game of Life is a map and can be described as:

$$\mathit{Map}(\text{board.gameStep(startRow, endRow)}) =$$

$$(\triangleleft_{\mathit{scatter}} \bullet [|\ ((\text{board.gameStep(startRow, endRow)}))\ |]\ n_w \bullet \triangleright_{\mathit{Gatherall}}\ ((\text{board.swapBoards})))$$

By using the RISC-pb formalism.

2.3.2 Splitting and merging the board

To parallelize the application it was necessary to split the board, and for this purpose, three ways have been found during the design phase:

1. To split the board **in range of rows** to compute, calculated according to the selected parallel degree, so that every worker will only compute its number of rows, reading each cell from the readingBoard and writing the computed resulting state of the cell to the new writingBoard, as explained in the sequential algorithm design chapter.

2. To split the board in **squared sub-matrices** according to the number of workers, so that every worker will receive as an input a small squared matrix, compute the state of every cell and write it in the new board.

3. Another possibility was to split the board in a small pack of rows or squared sub-matrices, to create a **stream of tasks**. In this way every worker would ask for a new task to compute, this could potentially lead to a better load-balancing. In this case it would've been necessary to choose the grade of granularity, to ensure the less overhead possible. If the range of rows (or size of the sub-matrices) will be too small, every worker will always have something to compute, but at the price of a bigger management cost. Considering the opposite case, a greater range of rows (or size of the sub-matrices) could lead to the possibility of having workers in an idle state.

In each of these ways it is possible to avoid interferences among the workers, since every worker will be writing on the same matrix but never on the same cell. Besides, given the fact that `readingBoard` is actually read-only, the multithread version of the game is safe.

For my parallel version of the game I chose the **first** option (range of rows), as suggested during the classes.

Another issue to take into account was how to split and merge the board, I actually found two different ways:

1. The first and most intuitive way is to let each worker works on the `readingBoard`, and write on the `writingBoard`, computing the cells on their **range of rows**. Once all the workers completed their task it will be only necessary to swap the pointers of the two matrices

2. Another possibility was to **actually split the board in smaller boards**, and give them as an input to each worker. After all of the workers terminate the execution, it would be necessary to merge the smaller boards together by getting every single cell from the resulting smaller board and writing the new cell state in a new resulting main board.

For this choice I decided, again, to go for the **first** way, as it was easier to implement, and more efficient (less overhead in creating the smaller boards and merging them together).

3 Implementation

The software has been implemented using Java, in four different versions:

- Sequential
- Using Java threads
- Using the Skandium framework
- Using the Java 8 stream feature

3.1 Sequential

The sequential implementation is fairly simple, the main thread execute the algorithm for computing the neighbors and the new state of the cell, reading from the `readingBoard` and writing on the `writingBoard`, at the end of every iteration, it swaps the boards pointers, and start the new iteration. A single game step is repeated for the given amount of iterations.

This version can be found in the package `it.gabrielemarini.spm.gameoflife.sequential`.

3.2 Multithread

The multithread version of the game has been implemented using java threads with a `threadPool` and a `CyclicBarrier`.

The `CyclicBarrier` is a java Class that implements a barrier for synchronizing the execution of a certain number of threads. When all the threads (or worker) complete the execution of their task, they will await inside the barrier for the remaining working threads to arrive. As soon as the last thread arrives, it will trigger a `BrokenBarrierException` exception, allowing each thread to start the new iteration. In order to allow every thread to compute the next iteration, the two boards' pointers are swapped inside the barrier, as soon as the last thread arrives.

Every thread is structured as a `Worker`, and each of them contains a pointer to the current `Board` object, the index of the `startingRow` along with the index of the `endingRow` index, which describe the chunk of the board to compute and, of course, a pointer to the `CyclicBarrier`.

In order for every `Worker` to be able to count the number of iterations already computed, a field iterations has been added to the `Worker` class. So that every `Worker` will count its number iterations and will stop working after reaching the fixed amount.

The Threads are stored in an `ExecutorService`'s `threadPool`, and each one of them is defined and initialized by pre-computing its `startingRow` and `endingRow` according to the total number of threads selected for the execution.

This version can be found in the package `it.gabrielemarini.spm.gameoflife.javathreads`.

3.3 Skandium

Skandium is a Java based framework for parallel computing, it's not supported anymore but it still works properly.
During the implementation of the software I used the jar file found on the website of the course, but some issues showed up during my design and testing phase:

My very first idea was to create a map, and iterate the map a fixed number of times using the **For skeleton provided by the Skandium library**. During my testing phase I realized that the map was being executed in an infinite loop. Therefore I decided to use the Skandium library including the java files found on the GitHub repository into my project as it fixed the mentioned issue.

The Skandium Map has been created as a `Map<Board, Board>` which takes a `Board` element as input and provide the resulting Board as an output. In order to create the Map, three new classes were created, `SkandiumSplitBoard`, `SkandiumExecuteSteps` and `SkandiumMergeBoard`.

- `SkandiumSplitBoard`: This class takes care of splitting the board, by assigning to every worker the index of its `startingRow` and the number of rows to compute on the `readingBoard`. Both values are computed according to the fixed parallel degree. The `startingRow` and the number of rows to compute are then stored into another class:
    - `Interval`: The interval class simply stores the value of the values `startingRow` and the `endRow` that represent the board chunk to compute for every worker.

  The splitting of the board is done by the `splitBoard(nThreads)` method in the `Board` class.
- `ExecuteSteps`: This class will simply call the `gameStep(startingRow, endRow)` method of the Board class that will compute a single step of the game, computing the new state for the cells in the main Board, from the `startingRow` to the `endRow` (both retrieved from the Interval parameter), iterating for the amount of rows computed for each worker.
- `SkandiumMergeBoard`: As already explained in the previous chapter, the "merging" of the newly computed board is very easily achieved by just swapping the two boards pointers inside the `Board` class, calling the `swapBoards()` method of it.

A new For skeleton has then been created from the Map skeleton previously defined, and will be executed for a fixed amount of iterations.

This version can be found in the package `it.gabrielemarini.spm.gameoflife.skandium`.

3.4 Java8

The map implementation takes advantage of the new stream features provided by Java 8. The input board is evenly split according to the selected parallel degree by computing the starting and the ending row indexes for every worker. Splitting the board will generate an array of `J8Interval` objects, which will contain the needed value for each worker to compute its range of the board. A stream of Interval objects is then generated starting from the Intervals array and parallelized using the `parallel()` operator. A `forEach()` operator is then applied to pass each Interval object of the parallel stream as a parameter to the execute method of the `J8ExecuteSteps()` class, which will compute a single iteration of the game on the selected range of rows.

The described lambda is submitted to a `ForkJoinPool`. After the execution of the described function, the board is "merged" by swapping the pointers of the two boards.

The described algorithm is then iterated for the selected number of iterations.

```
ForkJoinPool fjPool = new ForkJoinPool(nThreads);
long startTime = System.currentTimeMillis();

for(int i = 0; i < iterations; i++){
      fjPool.submit(() -> Arrays.stream(new J8SplitBoard(nThreads)
            .split(board))
            .parallel()
            .forEach(interval -> new J8ExecuteSteps().execute(interval))).get();

      J8MergeBoard.merge(board);
}
long endTime = System.currentTimeMillis();

fjPool.shutdown();
```

Four new classes have been created for this version of the game, `J8SplitBoard`, `J8MergeBoard`, `J8Interval`, `J8ExecuteSteps`, and all of them may be found in the package `it.gabrielemarini.spm.gameoflife.java8`

3.4 Tests

In order to execute automatic tests for each version of the application, a Tests class has been implemented that will take care of parsing the input arguments and running all the requested tests. Every test phase need to have at least three parameters:

- A list of board sizes
- Number of iterations for each of the specified board sizes
- Number of times to repeat the whole test

After all of the tests will be executed, the StatsSheetGenerator class will generate an excel file with the resulting service time, computed by calculating the average on all the times obtained by the repetitions of every possible test combination (board sizes, number of threads, version of the game).

The times were measured by using the Java `System.currentTimeMillis()` method.

4 Results

For the performance evaluation, tests are automatically executed on random generated boards, and each version of the game will use the same random generated board of the selected size.

All the implementations have been tested on four different board sizes:

- 500x500
- 1000x1000
- 2000x2000
- 5000x5000

For each board size, **500** iterations have to be performed.

All of the above tests have then been repeated for **10** times to get the most precise and meaningful value for the execution time of each board size and version. After wise, the average among the resulting times is computed by removing the outlier values (the list of the service times is sorted and the first and last time are removed).

The very first intention was to execute the test on five different board sizes, adding a 10000x1000 board, and computing 1000 iterations for each board. But after some estimation on the amount of time needed for completing all the tests, it turned out the testing would take up to more than 24 hours to complete, so I reduced the amount of tests to execute, using just 4 board sizes and 500 iterations for each one.

The Map skeleton service time expected is estimated using the following formula:

$$T_s(n) = T_e + \frac{T_w}{n} + T_c$$

Where T_e, T_w and T_c are the times to split compute and merge. T_w actually corresponds to the `board makeStep()` function, while Tc is actually the `swapBoards()` function. Therefore, assuming the time to split and swap the boards to be negligible with the increasing size of the matrices and being them way smaller than the compute times (the split operation is a very simple linear operation, while the merge operation consists in the swapping of two pointers), and finally assuming no communication overhead, the formula can be approximated to $\frac{T_w}{n}$.

From the tests performed it's evident how with smaller matrix (100x100 or 200x200) the increasing parallel degree will give a small or no improvement at all to the execution time reaching the asymptote at a very low parallel degree, hence decreasing scalability and efficiency at a higher number of threads used. Basically, the overhead due to threads management has a heavier impact on fine grained computations, affecting scalability, speedup and efficiency, especially as the number of used threads increases.

All of the tests and graphs of the results may be found on the folder "Statistiche" found inside the distributed archive, two series of tests have been performed on the Xeon E5-2560 and one series of tests has been executed on the i7 4790k (not needed, but the tests yield some good results).

5 Conclusions

Two different kind of graphs have been drawn:

- The first set of graphs shows how different versions of the game behave on the same board size and number iterations
- The second set of graphs shows how different board sizes and iterations influence the results for each version of the game

As can be seen by the second set of graphs, the parallel implementation of Conway's Game of Life scale pretty well on the number of threads used, increasing the board's size will also improve the efficiency and speedup values, therefore, a bigger matrix usually brings to better results and a better use of the parallel implementation.

By looking at the first set of graphs it's possible to notice how the parallel implementation with the Java Threads got more or less better results in the first two board sizes (500x500, 1000x1000), while on the other two board sizes (2000x2000,5000x5000) the implementation of the game by means of the new Java 8 stream features yield better results in terms of efficiency and scalability.

It would be interesting to execute more tests on bigger matrices (10000x10000) increasing the number of iterations (around 1000) and repeating the test for about 20 or 30 times to see how scalability speedup and efficiency would be affected by the size of the matrix.

Appendix

A1 Usage

The software will be distributed in a tarball archive which will contain the main folder with all the needed files to compile and execute the application. The software can be compiled and packed automatically by means of Apache Ant.

By simply calling the ant command, the whole project will be compiled and packed in a jar file, and stored inside a jar folder.

```
tar –xvzf GameOfLife.tar.gz
Cd GameOfLife
ant
```

After the jar file is created it will be possible to execute the software by using the command:

```
java –jar jar/GameOfLife.jar
```

The above command (or by adding the –h, --help parameter) will show the following menu:

```
        _______  _______  __   __  _______    _______  _______     ___      ___   _______  _______
       |       ||   _   ||  |_|  ||       |  |       ||       |   |   |    |   | |       ||       |
       |    ___||  |_|  ||       ||    ___|  |   _   ||    ___|   |   |    |   | |    ___||    ___|
       |   | __ |       ||       ||   |___   |  | |  ||   |___    |   |    |   | |   |___ |   |___
       |   ||  ||       ||       ||    ___|  |  |_|  ||    ___|   |   |___ |   | |    ___||    ___|
       |   |_| ||   _   || ||_|| ||   |___   |       ||   |       |       ||   | |   |    |   |___
       |_______||__| |__||_|   |_||_______|  |_______||___|       |_______||___| |___|    |_______|

usage: GameOfLife [-g] [-h] -i <1000> -n <30> -s <10x10,50x50,800x800...>
       [-t <4>] [-v <seq, mt, sk, j8>]
 -g,--graphics                          Display graphics, disabled by
                                        default
 -h,--help                              Print help message
 -i,--iterations <1000>                 Number of iterations for every board
 -n,--number <30>                       Number of times to execute the test
 -s,--size <10x10,50x50,800x800...>     Boards sizes, in the form HxW
 -t,--threads <4>                       Maximum number of threads. If not
                                        specified, uses all available
                                        threads
 -v,--version <seq, mt, sk, j8>         Version of the game to execute. If
                                        not specified executes all possible
                                        versions (test) and print a file
                                        with all the results
```

- -g , --graphics: It's an **optional** parameter and it will show a Jpanel window for a graphical representation of the game board

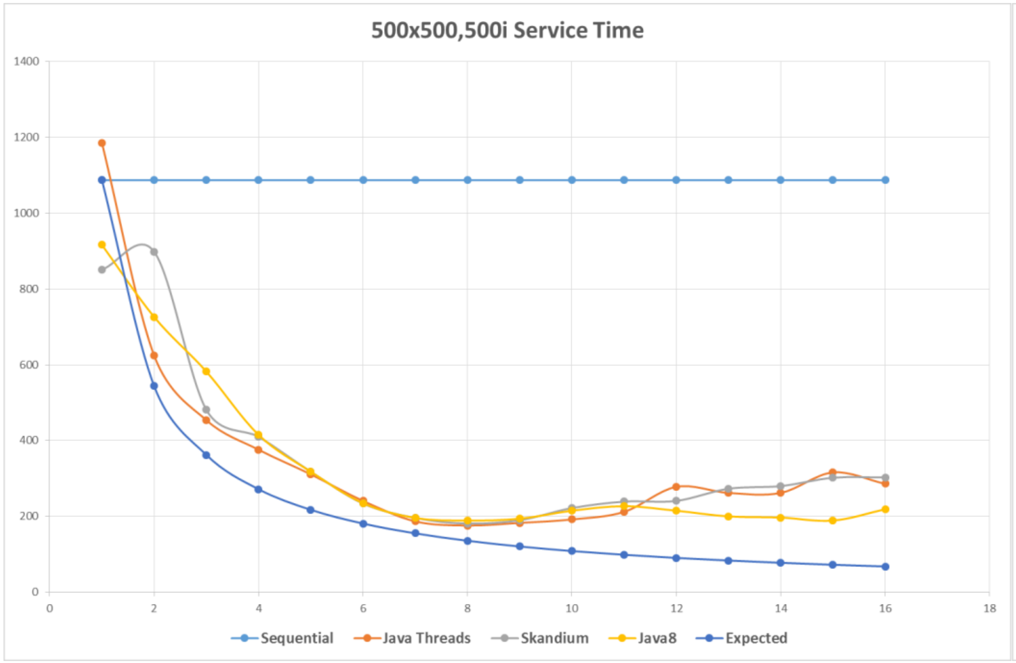
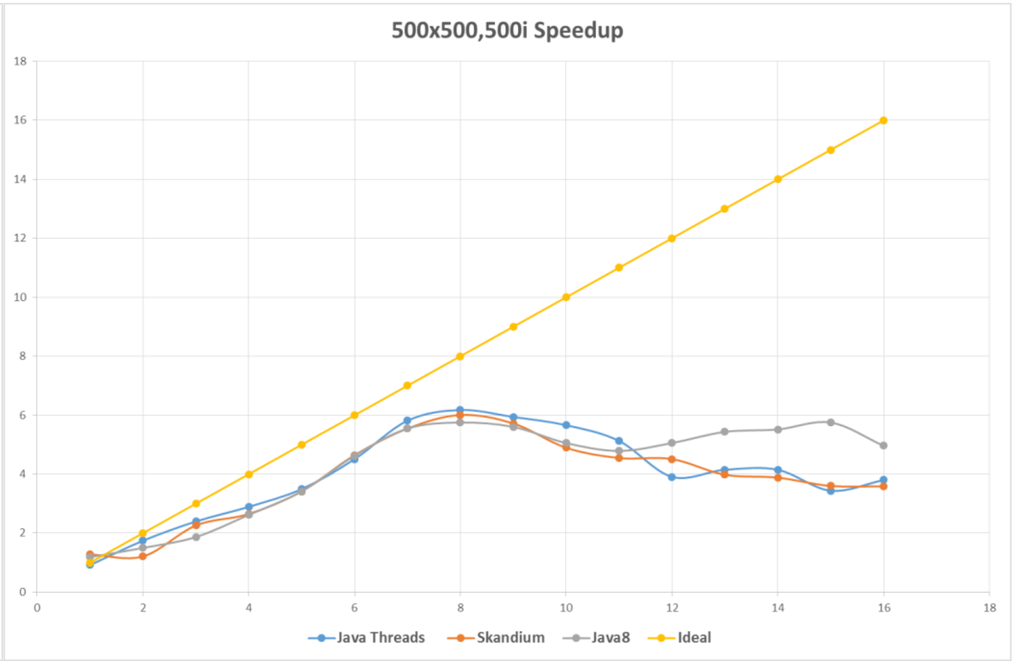
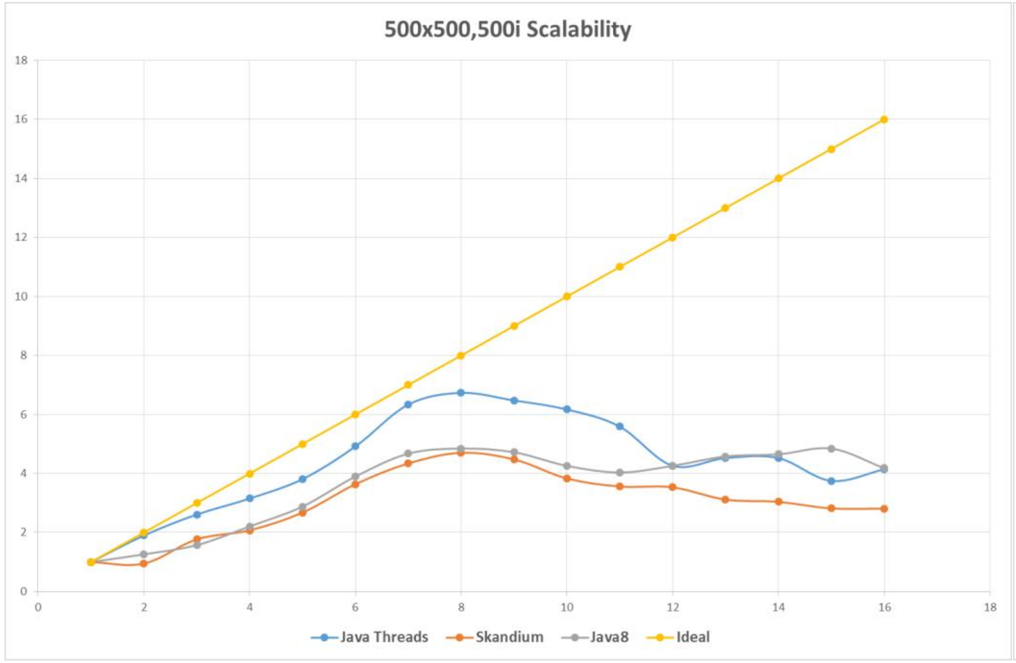
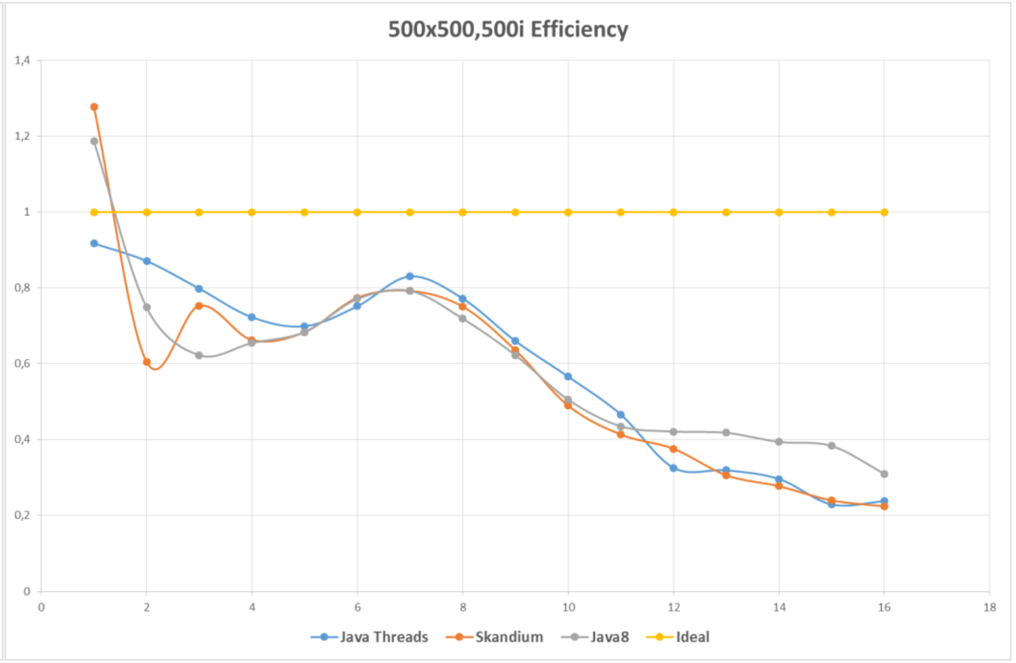
- -i, --iterations: It’s a **required** parameter, it specifies the amount of game steps to execute for each board size
- -n, --number: It’s a **required** parameter and it specifies the amount of times to repeat the whole testing phase, so, each version of the game will compute I (iterations) steps on each of the specified board sizes, this whole process will be then repeated n (number) times, and the final average will be computed among all the obtained times
- -s –size: It’s a **required** parameter and it specifies a list of board sizes on which the tests will be executed
- -t, --threads: it’s an **optional** parameter, and it specifies the **maximum** number of threads to use for the execution. If it’s not specified, all the available threads will be used, starting from a single thread execution to an execution that will take advantage of all of the available threads on the machine. It’s used for debugging or special testing purposes.
- -v, --version: It’s an **optional** parameter and it specifies which version of the game to launch. If it’s not specified, all the possible versions of the game will be executed during the testing phase.

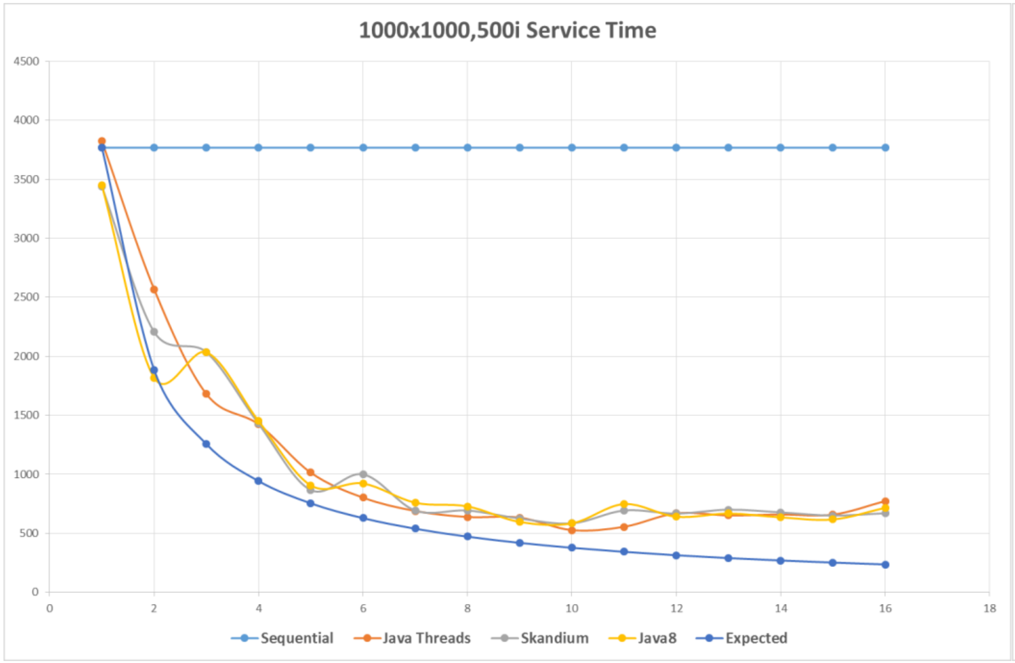
The helper menu at the parsing of the parameters is processed using the Apache Commons Cli library.

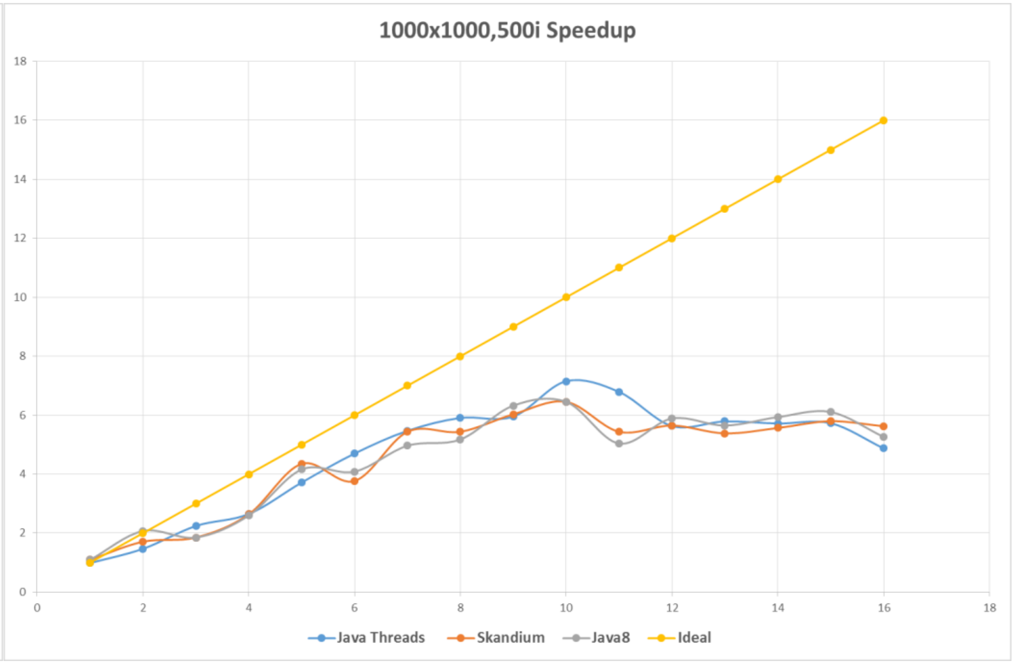
If the software is executed in foreground, a log will be printed on the console, here’s a short example of the console output printed during the execution of the tests:

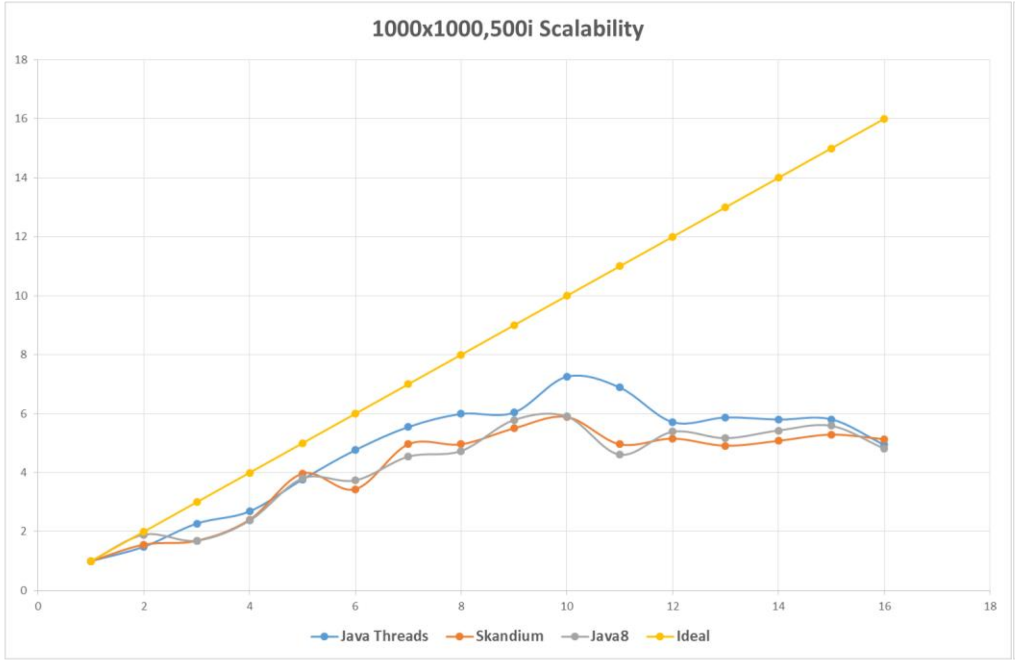
```
**** Run 1/10 - Testing 100x100 Board - 500 iterations ****
Sequential:       Executing...      55 ms
------------  1 Thread(s)  ------------
Multithread:      Executing...      23 ms
Skandium:         Executing...      28 ms
Java8:            Executing...      77 ms
------------  2 Thread(s)  ------------
Multithread:      Executing...      14 ms
Skandium:         Executing...      18 ms
Java8:            Executing...      20 ms
------------  3 Thread(s)  ------------
Multithread:      Executing...      12 ms
Skandium:         Executing...      14 ms
Java8:            Executing...      19 ms
------------  4 Thread(s)  ------------
Multithread:      Executing...      14 ms
Skandium:         Executing...      20 ms
Java8:            Executing...      20 ms
------------  5 Thread(s)  ------------
Multithread:      Executing...      15 ms
Skandium:         Executing...      16 ms
Java8:            Executing...      18 ms
------------  6 Thread(s)  ------------
Multithread:      Executing...      12 ms
Skandium:         Executing...      14 ms
Java8:            Executing...      14 ms
------------  7 Thread(s)  ------------
Multithread:      Executing...      11 ms
Skandium:         Executing...      13 ms
Java8:            Executing...      15 ms
------------  8 Thread(s)  ------------
Multithread:      Executing...      11 ms
Skandium:         Executing...      15 ms
Java8:            Executing...      13 ms
```

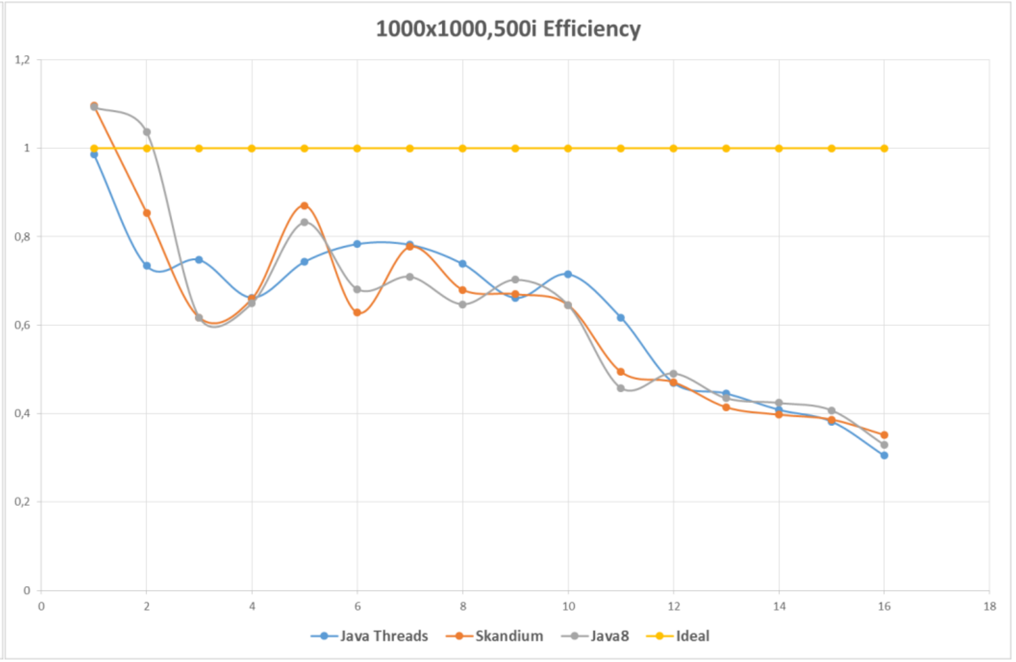
A2 Data Collected and Graphs

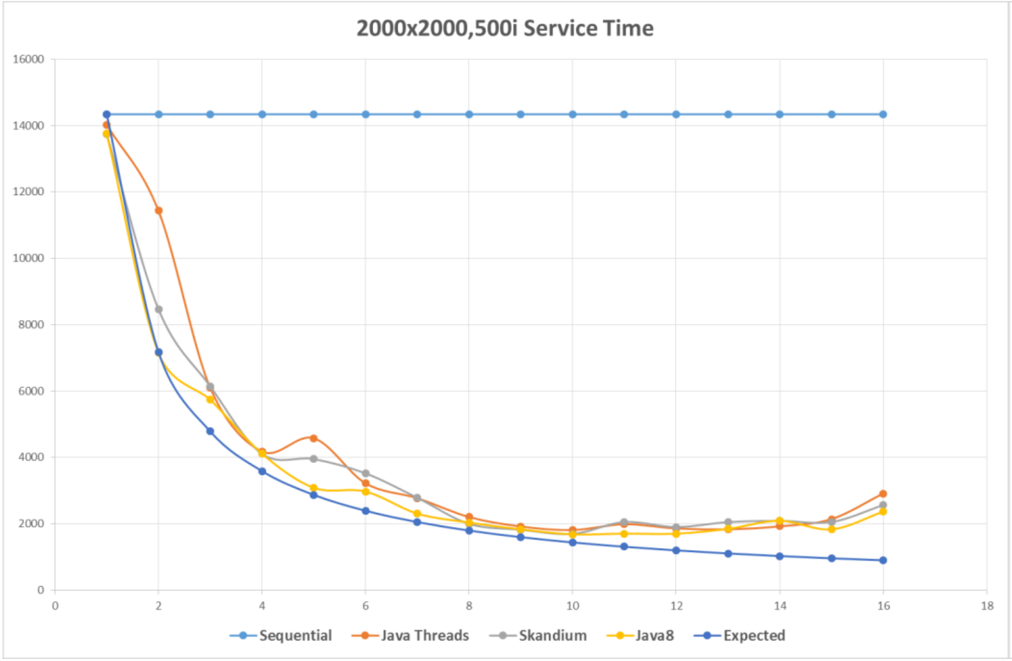
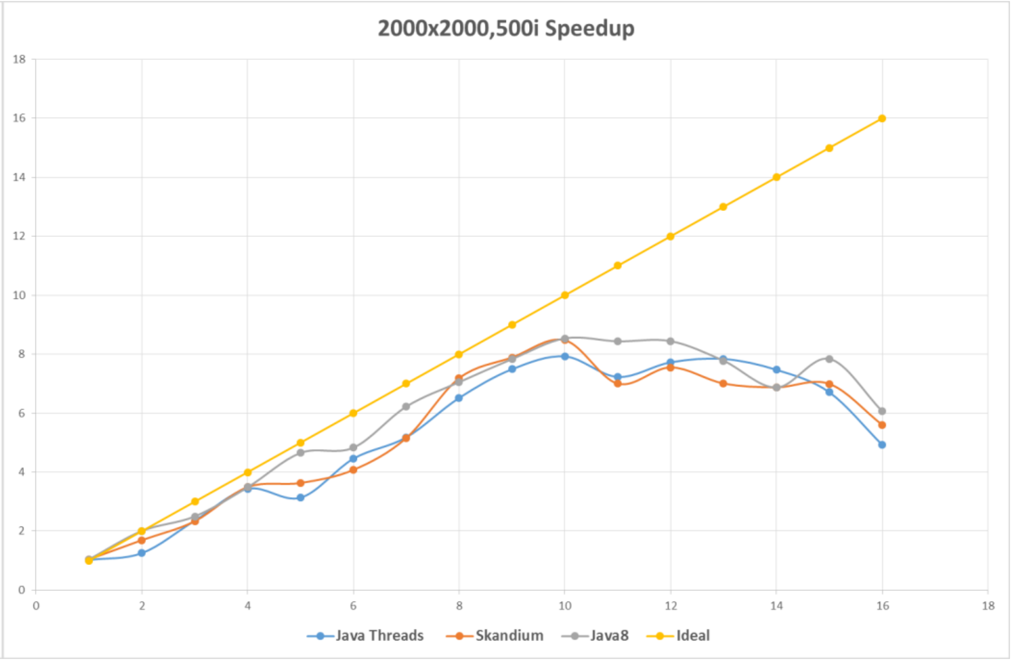
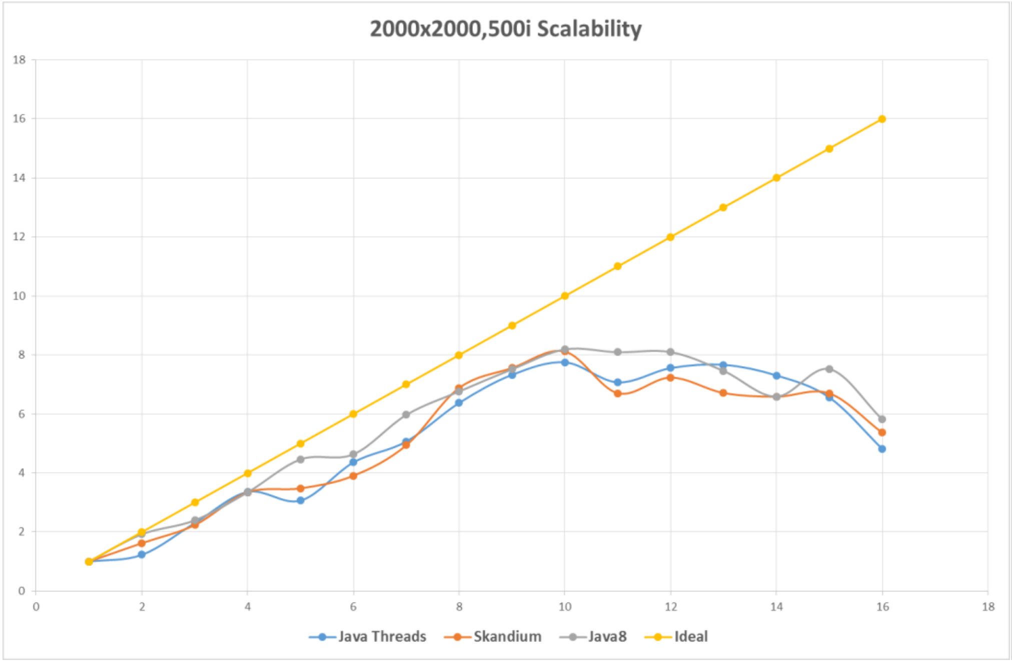
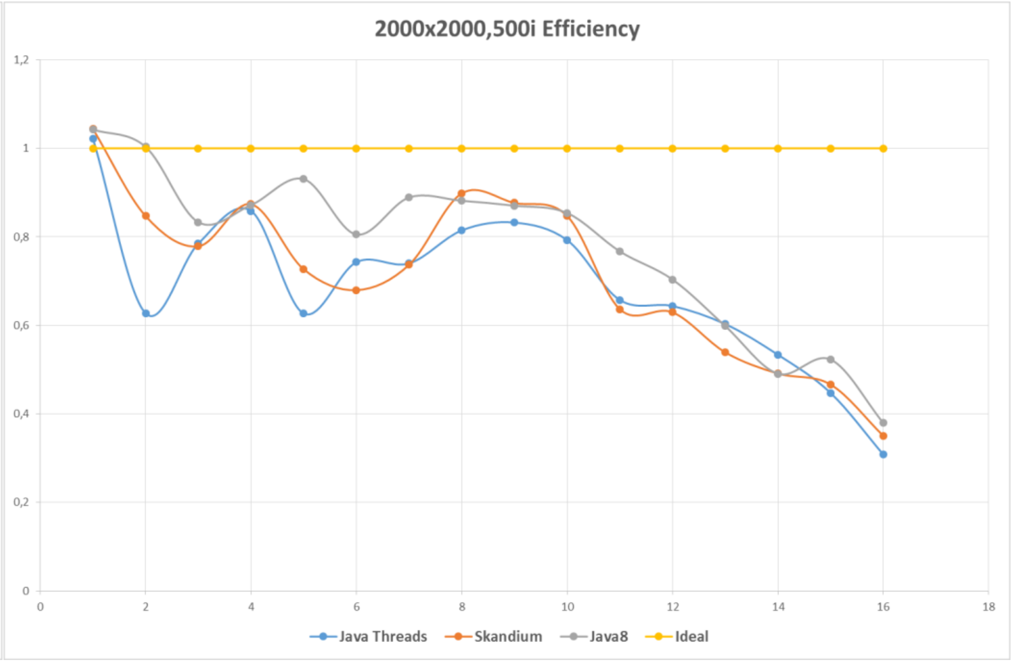

500x500,500i Service Time
1400
1200
1000
800
600
400
200
0
0
2
4
6
8
10
12
14
16
18
Sequential
Java Threads
Skandium
Java8
Expected

500x500,500i Speedup
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
Java Threads
Skandium
Java8
Ideal

500x500,500i Scalability
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
Java Threads
Skandium
Java8
Ideal

500x500,500i Efficiency
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0
2
4
6
8
10
12
14
16
18
Java Threads
Skandium
Java8
Ideal

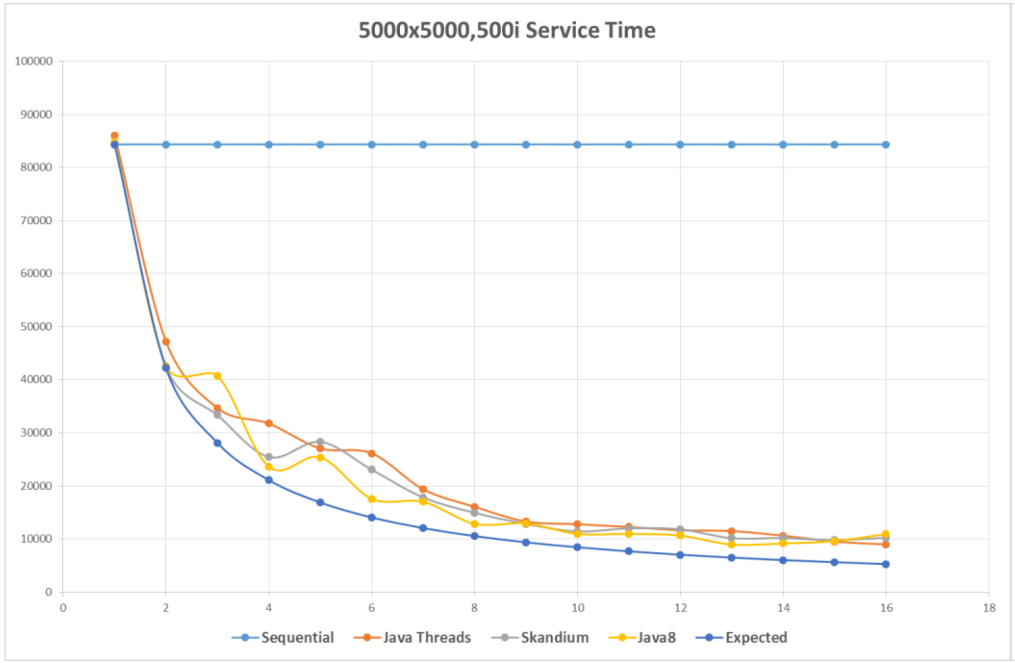

1000x1000,500i Service Time
Sequential
Java Threads
Skandium
Java8
Expected

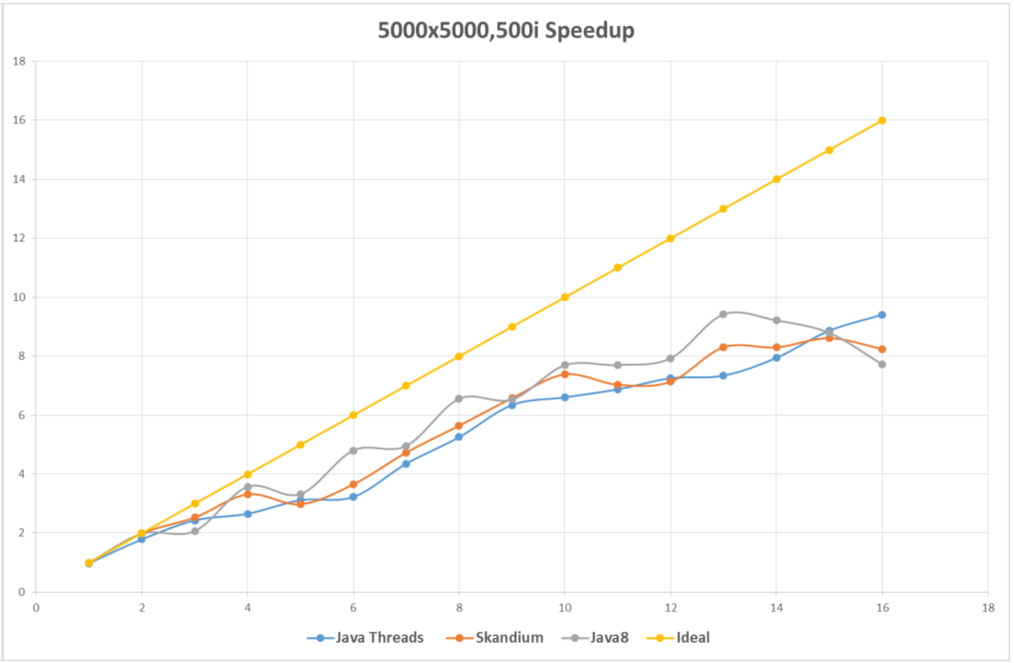

1000x1000,500i Speedup
Java Threads
Skandium
Java8
Ideal

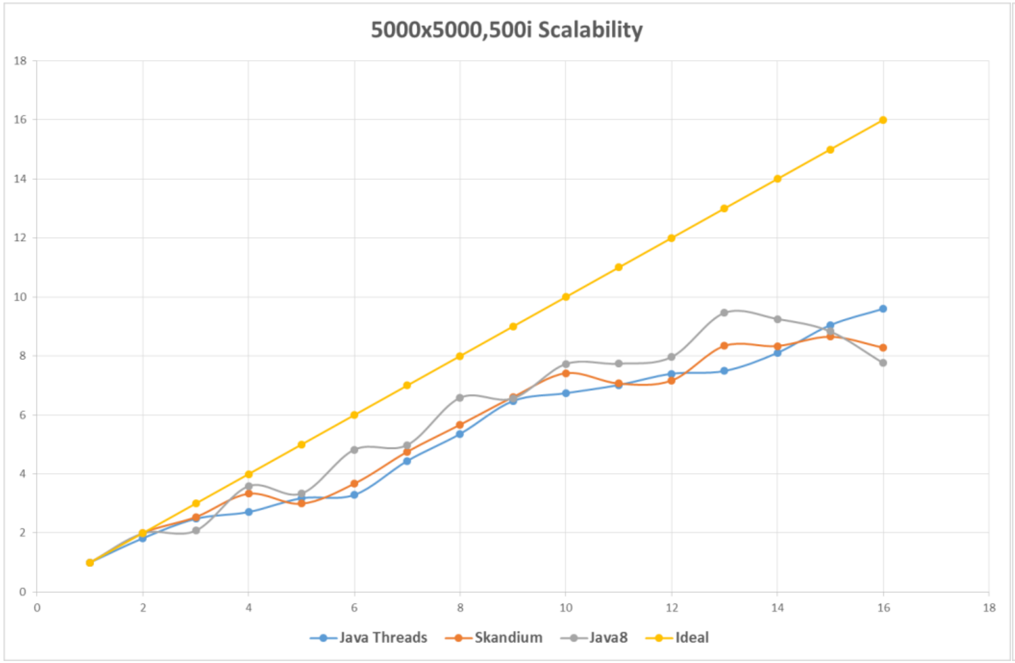

1000x1000,500i Scalability
Java Threads
Skandium
Java8
Ideal

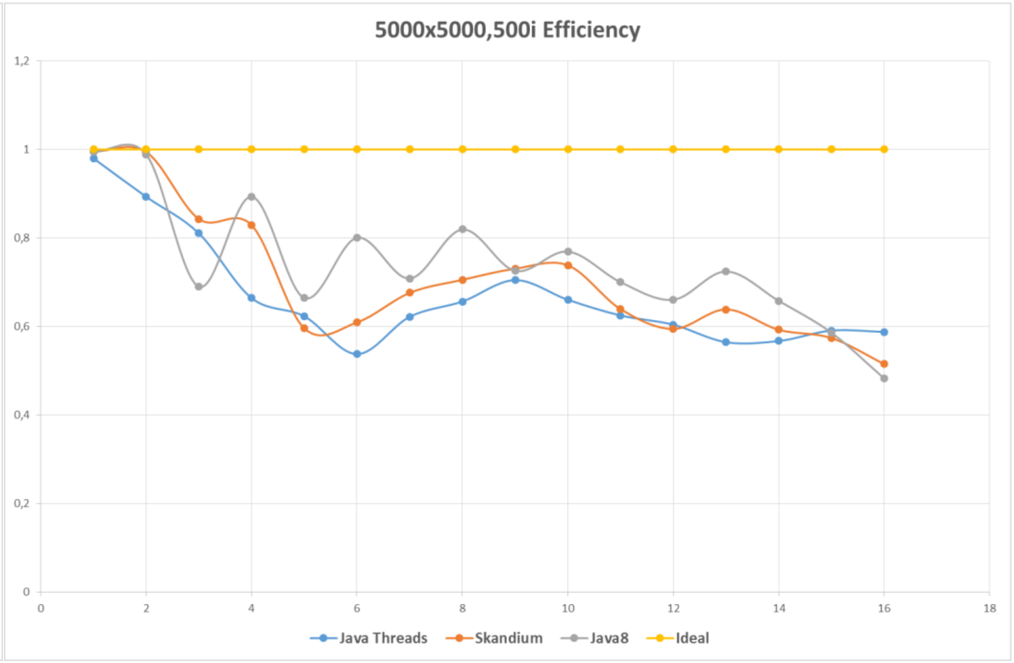

1000x1000,500i Efficiency
Java Threads
Skandium
Java8
Ideal

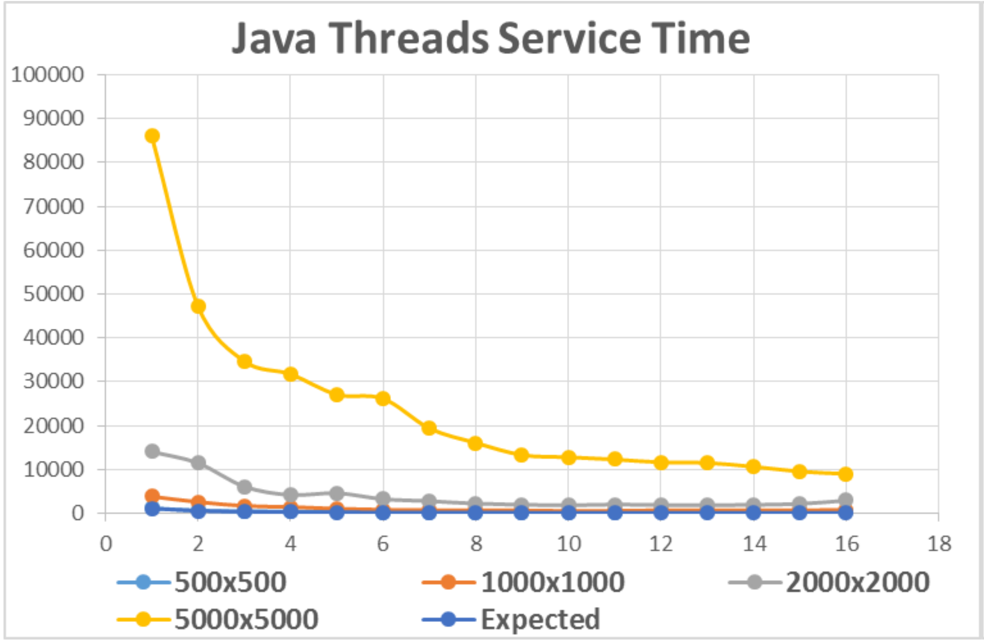

2000x2000,500i Service Time
16000
14000
12000
10000
8000
6000
4000
2000
0
0
2
4
6
8
10
12
14
16
18
Sequential
Java Threads
Skandium
Java8
Expected

2000x2000,500i Speedup
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
Java Threads
Skandium
Java8
Ideal

2000x2000,500i Scalability
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
Java Threads
Skandium
Java8
Ideal

2000x2000,500i Efficiency
1,2
1
0,8
0,6
0,4
0,2
0
0
2
4
6
8
10
12
14
16
18
Java Threads
Skandium
Java8
Ideal

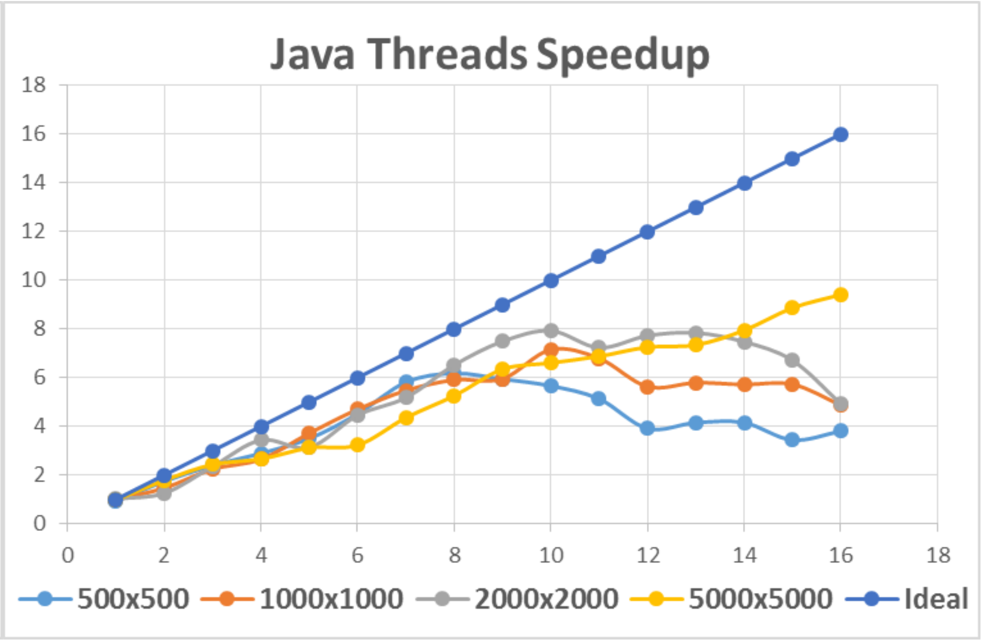

5000x5000,500i Service Time
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
2
4
6
8
10
12
14
16
18
Sequential
Java Threads
Skandium
Java8
Expected

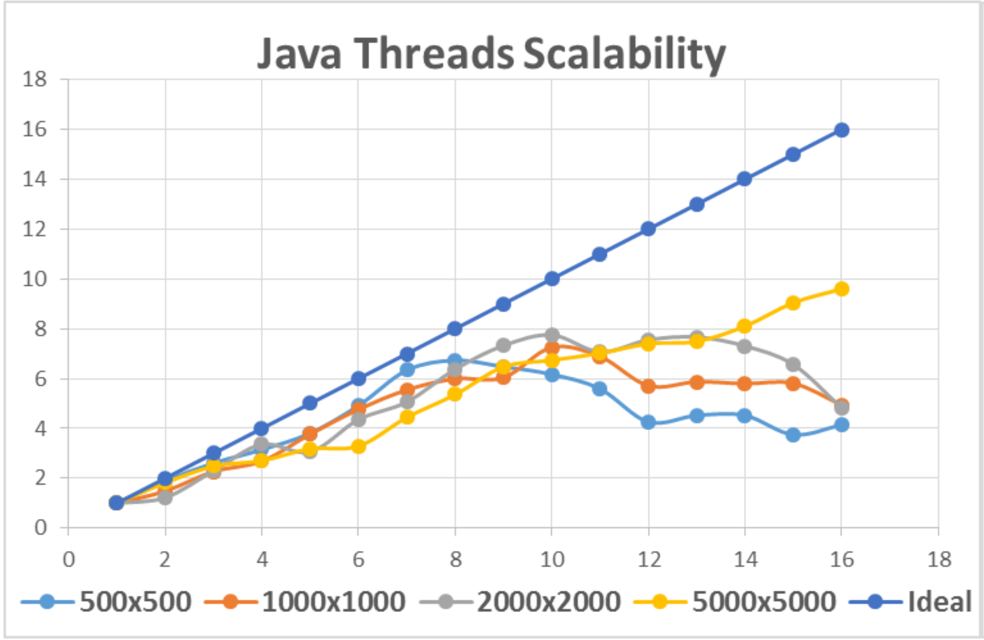

5000x5000,500i Speedup
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
Java Threads
Skandium
Java8
Ideal

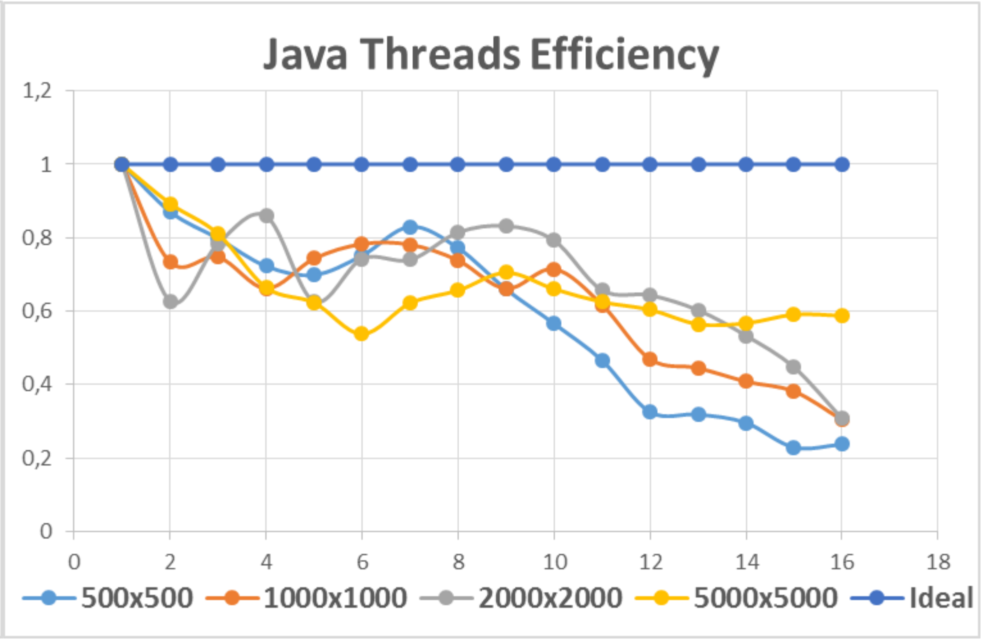

5000x5000,500i Scalability
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
Java Threads
Skandium
Java8
Ideal

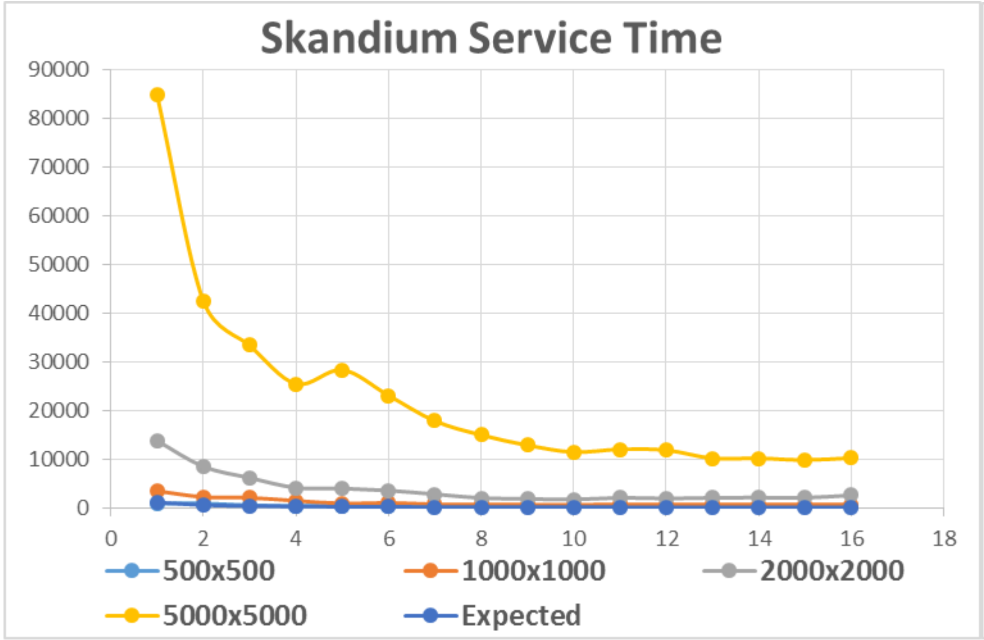

5000x5000,500i Efficiency
1,2
1
0,8
0,6
0,4
0,2
0
0
2
4
6
8
10
12
14
16
18
Java Threads
Skandium
Java8
Ideal

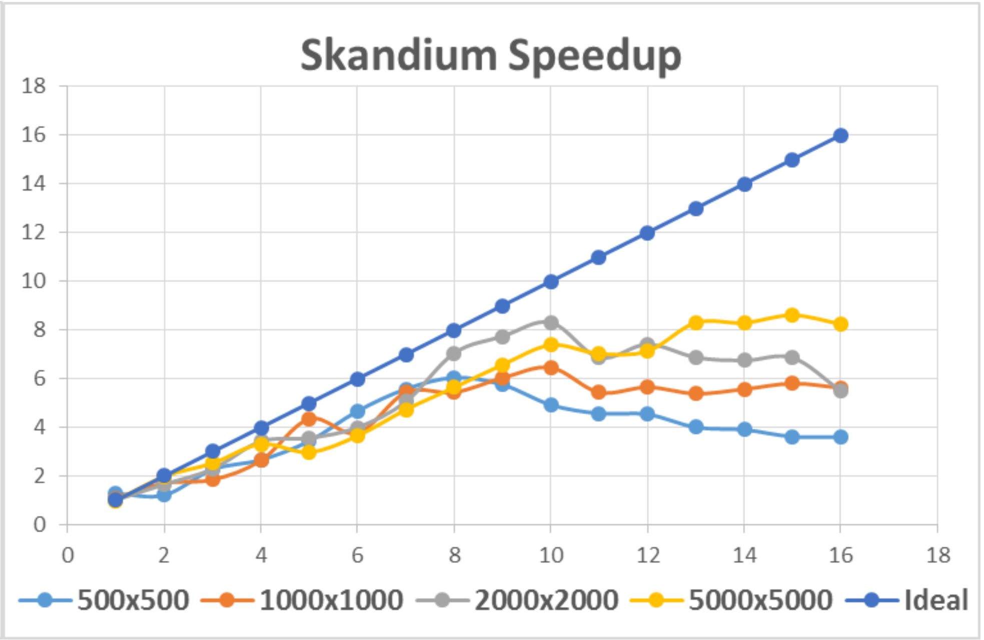

Java Threads Service Time
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Expected

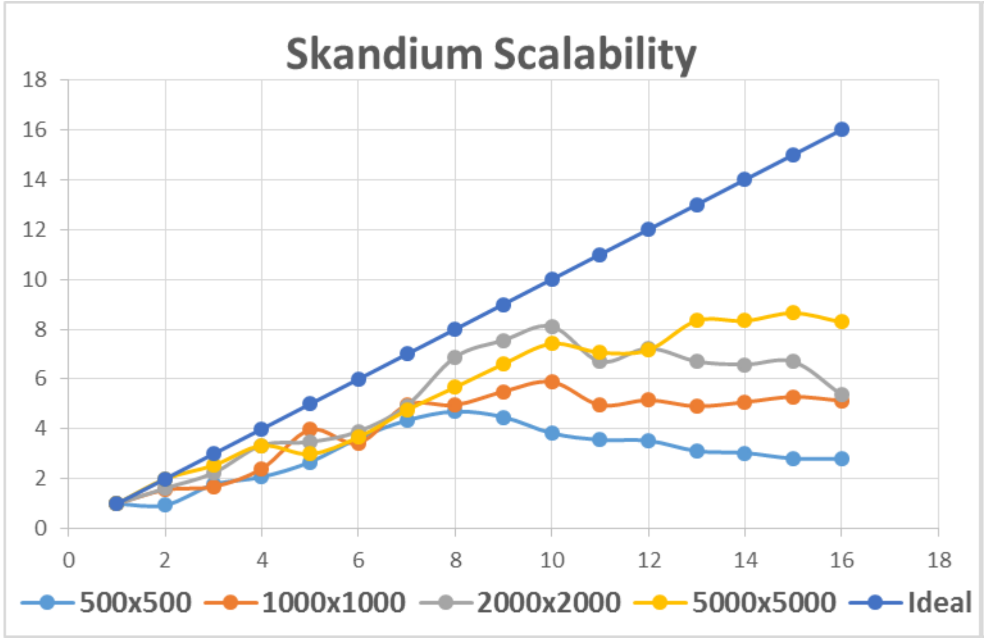

Java Threads Speedup
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Ideal

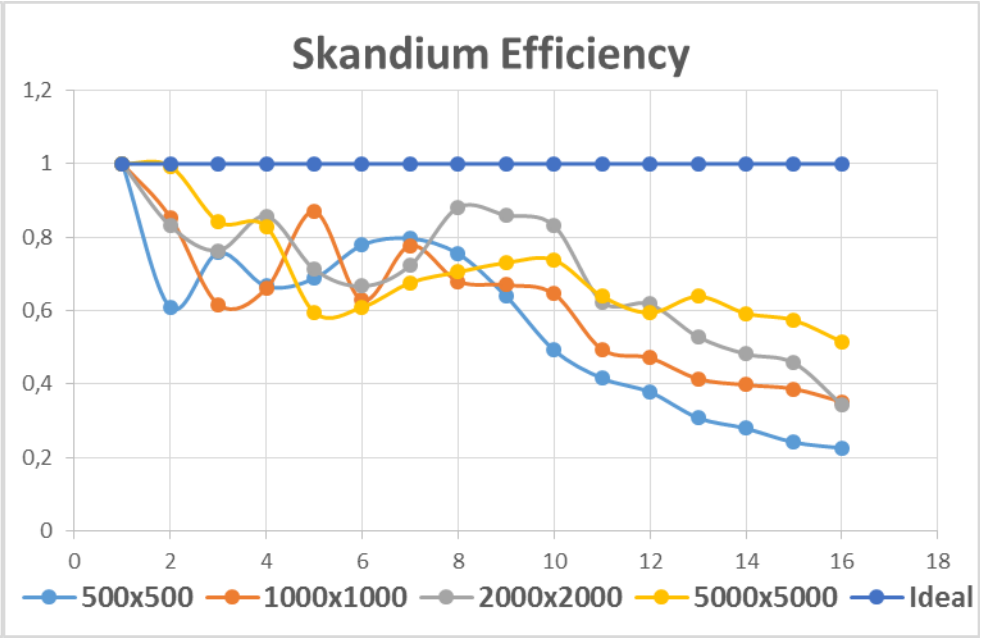

Java Threads Scalability
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Ideal

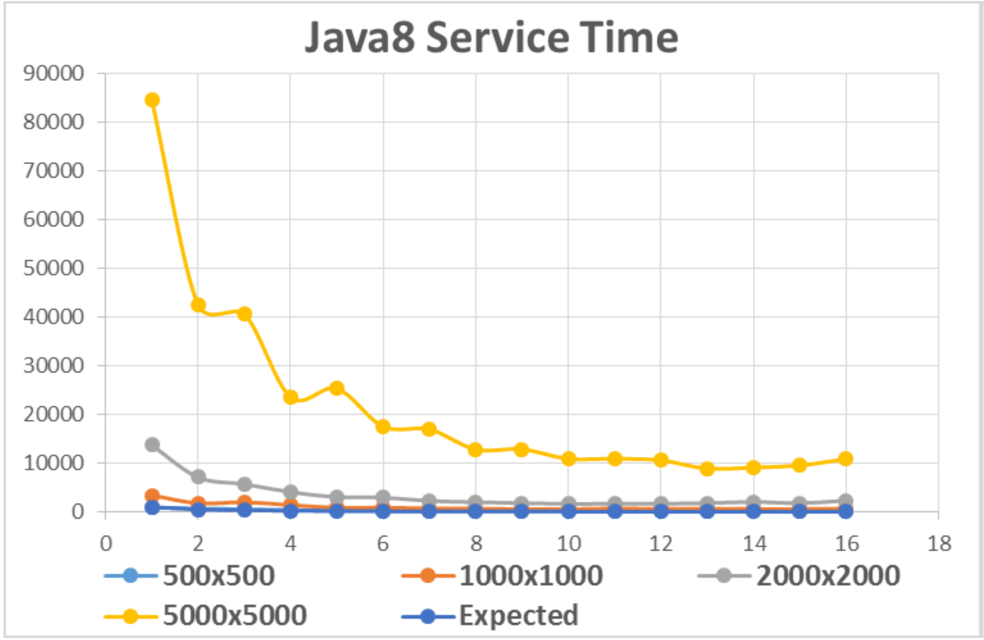

Java Threads Efficiency
1,2
1
0,8
0,6
0,4
0,2
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Ideal

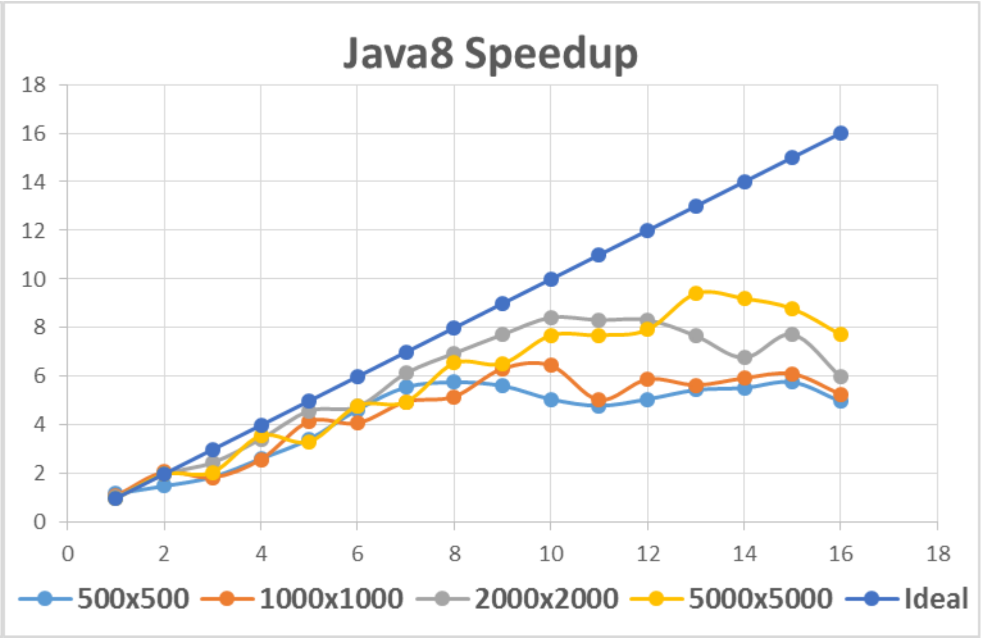

Skandium Service Time
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Expected

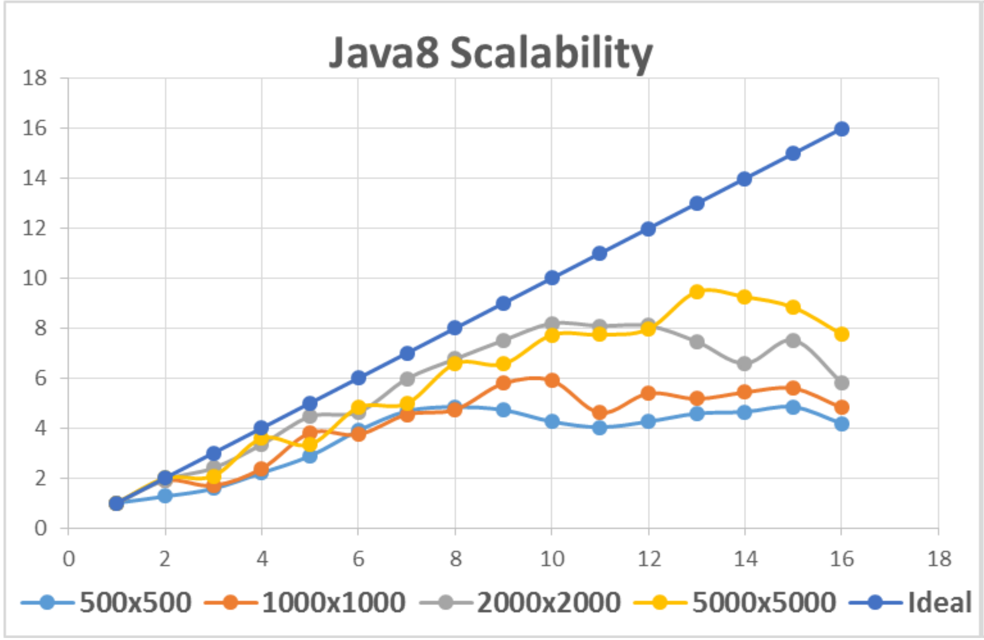

Skandium Speedup
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Ideal

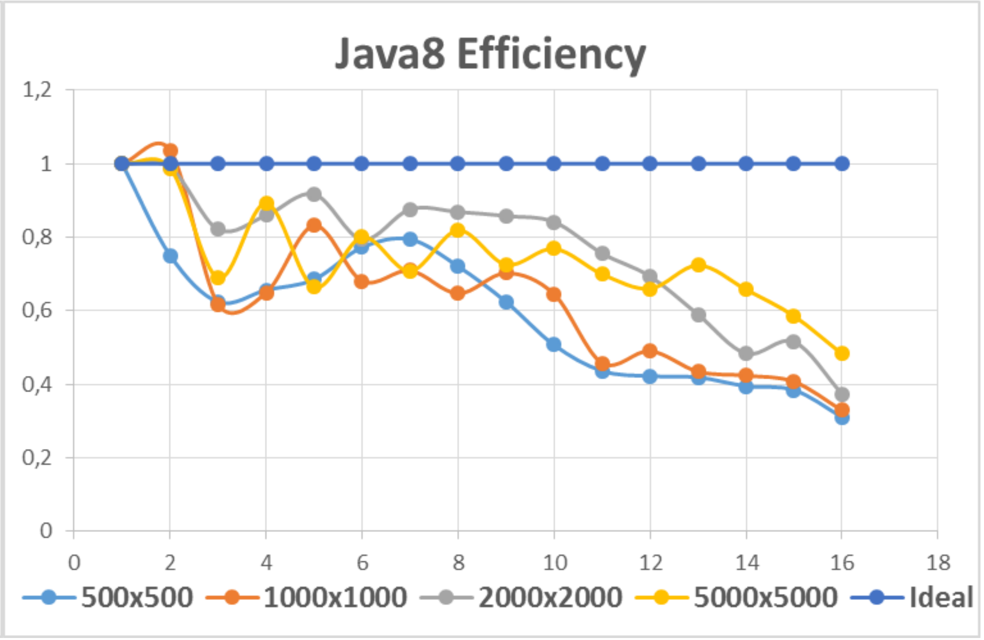

Skandium Scalability
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Ideal


Skandium Efficiency
1,2
1
0,8
0,6
0,4
0,2
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Ideal


Java8 Service Time
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Expected


Java8 Speedup
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Ideal


Java8 Scalability
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Ideal


Java8 Efficiency
1,2
1
0,8
0,6
0,4
0,2
0
0
2
4
6
8
10
12
14
16
18
500x500
1000x1000
2000x2000
5000x5000
Ideal