Distributed Systems: Paradigms and Models
N-Queens - Project Report

Tsinu Assefa Gezahgn (523652)
Submitted to: Professor Marco Danelutto
January 17, 2017

Contents

1. Introduction

This report describe SPM-course project that is a parallelization of N-queens game[1], an implementation and experiment to validate its design and performance. The project is developed by Java Oracle JDK 1.8 and Java based Algorithmic Skeleton Library called Skandium.

The report consists of design and implementation of three main components:

- A sequential version.
- A parallel version, implemented via Skandium parallel skeletons framework
- A parallel version, implemented by means of the Java 7 Thread mechanisms and by using a new futures of Java 8.

Afterward, the obtained results evaluating compilation time, scalability, speedup and efficiency for the parallel version are discussed. All tests were run on an Intel Xeon E5-2650 (2.00GHz) processor with 8 cores and 16 contexts.

2. Sequential version of N-Queens

The class NQS.java which is used to implement the sequential version of N-queens by instantiating the GameBoard class.

2.1 GameBoard class

The GameBoard class basically implement the sequential algorithm for N-queens. It has the following methods.

- *enumerate*: find the number of possible solutions for placing n-queens in NxN GameBoard . The algorithm is :
  - ✓ Consider one row at a time, within a row consider one column at a time look for *safe column to place a queen.*
  - ✓ If we find, place the queen and make *a recursive call to place queen in the next row*.
  - ✓ If we can't find a safe location in the row, *backtracking by returning from the recursive call*.
  - ✓ If row is equal to game board size then increment number of solution by 1 and *backtracking* by returning from the recursive call.

Example:-

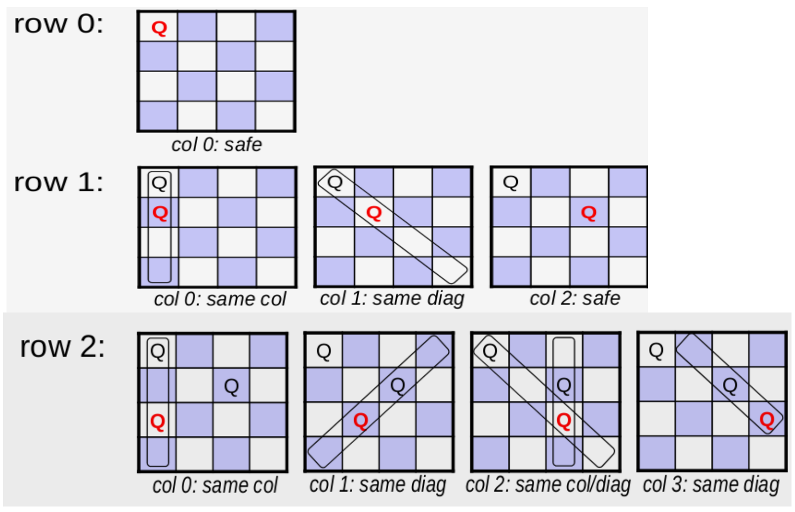

Figure1. 4x4 GameBoard

1-https://en.wikipedia.org/wiki/Eight_queens_puzzle

We've run out of columns in row 2, backtrack to row 1 by returning from the recursive call.

- ✓ pick up where we left off
- ✓ We had already tried columns 0-2, so now we try column 3:

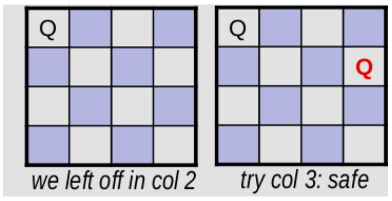

Figure 2. 4x4 GameBoard.

- ✓ Continue the recursion as before.

- *enumerate* (parallel version): slight modification of the sequential version , that execute the sequential version algorithm for a specified number of columns.
- *IsConsistent*: check if the location (row, column) is safe to place a queen or not, by looking for the same diagonal, and column.

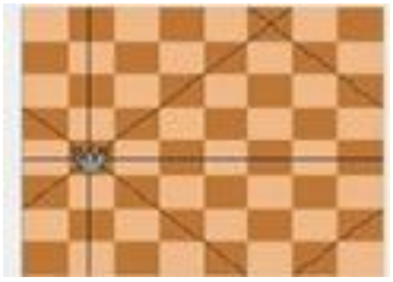

Figure 3. Check for same columns and diagonals.

- *SplitGameBoard*:-has an argument a number of thread and returns an array of Intervals to divide the Gameboard column-wise into number of threads.
- Data Structures for n-Queens: - by using a one dimensional array of size n we can represent the location of a queens, for instance an array queens [1] =2 implies a queen at row 1 and column 2.

3. Parallel version of N-Queens

This section describes the design of the parallel versions of N Queens. Both the adopted data Parallel pattern and its cost model.

3.1 Map-Reduce Pattern

The natural pattern to parallelize N queens is a map-reduce, which can be described in the figure below.

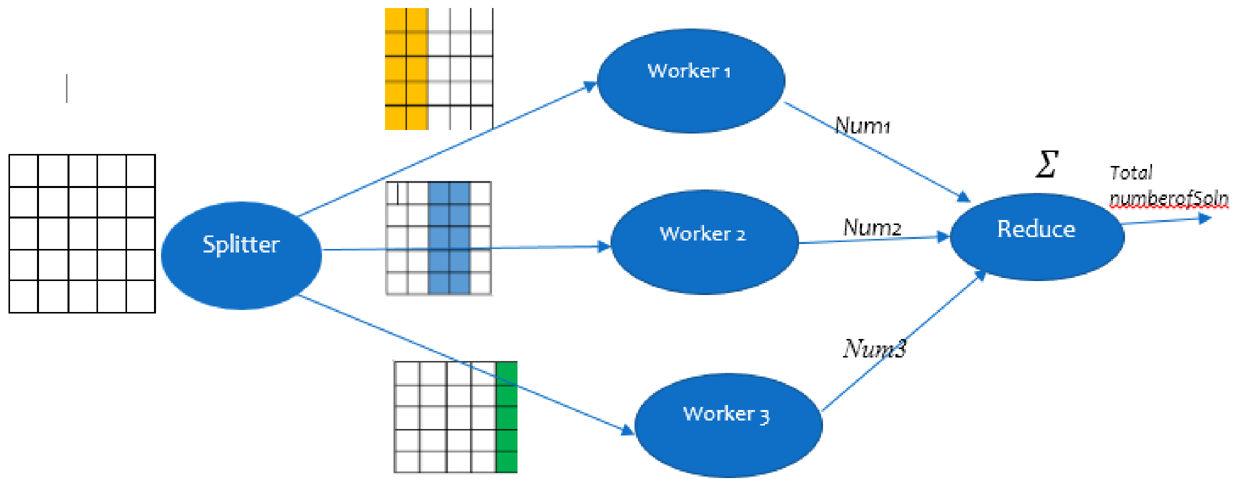

Figure 4. A graphical sketch of the data parallel pattern adopted for GameBoard size 5x5 with three threads.

The concurrent version is done by giving for each thread a chunk of column(s) of a board to find number of possible solutions, in the best case a processor will work on a column. Afterward summing each possible solutions, since each sub solutions can be computed without synchronization of each other the process of finding number of possible solutions in N-queens problem can be done in parallel. The Map-Reduce skeleton is used for the parallel implementation of the algorithm.

3.2 Cost Model

The cost model for the Map-Reduce pattern can be derived by taking into account the completion time for finding number of possible solutions which is:

$$Tc = T_{scatter} + TF + T_{reduce}$$

Where: TF is given as Tseq x G/n

- Tseq is the sequential time
- n is the parallelism degree.
- G is the maximum number of columns from a set of chunks.

$T_{scatter}$ is negligible as it only sends intervals (start index and end index) of a chunk for each worker.
T_{reduce} is also negligible as it only performs summation of the partial results of each worker.
Therefore, the completion time can be approximated as Tc ~ TF.

In the case of number of available processors is greater than the game board size n, we can use a maximum of n-processors, because to compute solutions a processor need at least one column. *Thus the upper bound of required processors is equal to n.*

4. Parallel Implementation of N-queens

Skandium and Java thread implementation are implemented by the class *SkandiumParallel*.java, *Java7MultiThread.java* and *Java8MultiThread*.java respectively, all of this class instantiate Interval.java and GameBoard classes.

Interval class

This class provides a way to specify column-wise chunks in the Gameboard. It has two fields: that are indicating a starting columns and the ending columns for the chunk respectively

4.1 Skandium Implementation

The Skandium implementation of N-queens is include the following classes.

4.1.1 Splitter class

This class implements the Split<GameBorad, Interval> interface which is used by the Skandium Map skeleton. It has constructor to initialize number of thread from a user input .The override method split calls a method of GameBoard ,*SplitGameBoard* to obtain an array of Interval objects that are send to the workers.

4.1.2 Worker class

This class implements the Execute<Interval, Long> interface that is needed by Skandium Map skeleton. It receive an interval object which contain the start and end of column index , call GameBorad method *enumerate* and returns number of possible solutions. The override method *execute* is executed in parallel by each worker.

4.1.3 Reduce class

Reduce class implements Merge<Long, Long> interface. This class receive the computed result from each worker, summing up the results and return the total possible number of solutions.

```
GameBorad gameBoard = new GameBorad(BOARD);
                    Skandium skandium = new Skandium(THREADS);
                    Skeleton<GameBorad, Long> map = new Map<GameBorad, Long>(new Splitter(THREADS), new
Worker(), new Reduce());
                    Stream<GameBorad, Long> stream = skandium.newStream(map);
                    long init = System.currentTimeMillis();
                    Future<Long> future = stream.input(gameBoard);
                    Long result = future.get();
                    System.out.println("NQUEENS-Skandium>>Total # of solutions is:" + result + " Size=" + BOARD +
" with Processor:"
                                        + THREADS + " in " + (System.currentTimeMillis() - init) + "[ms]");
                    skandium.shutdown();
```

Listing 1. Core of Skandium implementation --SkandiumParallel.java

4.2 Java thread Implementation

The second parallel version of the n-queens implemented by using Java 7 Thread mechanisms and the new future of Java 8 multicore programming ,however the experiment for both versions are done via oracle jdk 1.8.

4.2.1 *Java7MultiThread* Implementation

This version is composed of two classes: the WorkerThreadPool and Java7MultiThread.
Java7MultiThread.java class manages the thread pool. It initialize a class WorkerThreadPool and assign it into a callable object then submit the callable object to the ExecutorService. ExecutorService return an object of type Future object that is used to check the status of a Callable and to retrieve the result from the Callable.

```
GameBorad gameBoard = new GameBorad(BOARD);
Interval[] bounds = gameBoard.splitGameBoard(THREADS);
ExecutorService threadpool = Executors.newFixedThreadPool(THREADS);
List<Future<Long>> list = new ArrayList<Future<Long>>();

for (int i = 0; i < THREADS; i++) {
        Callable<Long> worker = new WorkerThreadPool(BOARD, bounds[i]);
        Future<Long> submit = threadpool.submit(worker);
        list.add(submit);
}

long init = System.currentTimeMillis();
long result = 0;
for (Future<Long> future : list) {
        try {
                result += future.get();
        } catch (InterruptedException ex) {
                ex.printStackTrace();
        } catch (ExecutionException e) {
                e.printStackTrace();
        }

}
System.out.println("NQUEENS-Java7-Thread>>Total # of solutions is " + result + " Size=" +
BOARD
                + " with Processor:" + THREADS + " in " + (System.currentTimeMillis() - init) +
"[ms]");
```

Listing 2 Core of Java Thread implementation --Java7MultiThread.java

4.2.2 WorkerThreadPool class

Implement the callable *interface*, each instance of this class receives the size of a game Board and an Interval object which specify the starting and ending of a chunk.
The overridden call*()* method of this class , call the GameBorad method *enumerate* that find number of possible solutions and return .

4.3 Java 8 thread Implementation

Lambda expressions and streams are among the most important new features in Java 8 lambda expressions enable the developer to write more concise code. One of the major enhancements is the ability to process streams in parallel. Java 8 also introduce CompletableFuture class, which implements the new CompletionStage interface and extends Future that makes asynchronous operations easier to coordinate.

CompletableFutures over parallel stream allow you to configure an Executor in a particular size, that can fit better your application [3] thus we can chose suitable number of threads based on our application.
This version has one class named Java8MultiThread.java

```
GameBorad gb = new GameBorad(BOARD);
Interval[] bounds = gb.splitGameBoard(THREADS);
ExecutorService threadpool = Executors.newFixedThreadPool(THREADS);
List<Interval> bound = Arrays.asList(bounds);

List<CompletableFuture<Long>> numSolution = bound.stream()
                .map(boundIt -> CompletableFuture.supplyAsync(() -> calculateSoultions(boundIt),
threadpool))
                .collect(Collectors.<CompletableFuture<Long>>toList());
                long init = System.currentTimeMillis();
                List<Long> list = numSolution.stream().map(CompletableFuture::join).collect(Collectors.toList());
                long result = list.stream().reduce(0L, (a, b) -> a + b);

                System.out.println("NQUEENS-Java8-Thread>>Total # of solutions is " + result + " Size=" +
BOARD
                          + " with Processor:" + THREADS + " in " + (System.currentTimeMillis() - init) +
"[ms]");
```

Listing 3 Core of Java Thread implementation --Java8MultiThread.java

From a list of Interval objects (bounds) we are concurrently calculating the number of solutions then we apply reduction over a list (List<Long> list) of partial results to calculating total number of solutions.
Elements of the list numSolution <CompletableFuture<Long>> contains a partial solution which is computed by calling supplyAsync. This method takes lambeda expression (() -> calculateSoultions(boundIt), threadpool)) where calculateSoultions is a function to calculate the number of solution and threadpool is an Executor. The join method of CompletableFuture returns the result value when complete it has the same meaning with get method of Future.

5. Performance Analysis

As we know that the aim of parallel programming is to obtain a good performance. Ideally we expect the time to execute parallel program by using n-processing elements is lower by the factor of n with respect to sequential time.
However several factors impair to achieve the ideal time. According to the Amdhal law amount of non-parallelize work in application determine the maximum speedup achieved by parallel implementation. In the context of this application split and merge are considered as non-parallelize work.
Beside this an overhead such as JVM threads management overhead (the time to create and assign resource for threads, destroy threads, fork and wait for join threads) and communication overhead (the time to setup communication, to send from splitter to worker and from worker to merger) degrade the expected performance.

To test the performance the completion time of each solution was measured by invoking the Java method *System.currentTimeMillis().*

Sequential version	Parallel Versions	
Matrix size	Matrix size	Threads
4x4, 8x8, 10x10, 12x12, 14x14, 16x16.	4x4, 8x8, 10x10, 12x12, 14x14, 16x16.	1,4,8,16

The tests were run over different Game board sizes and number of threads. The graphical illustration of the tests is given below

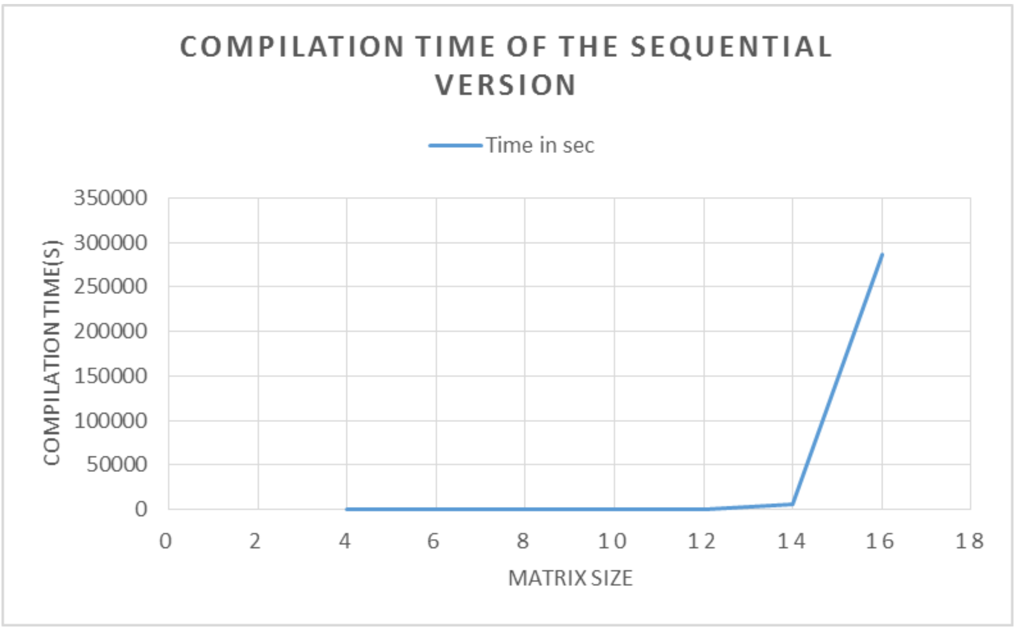

Figure 5. Compilation time of a sequential program

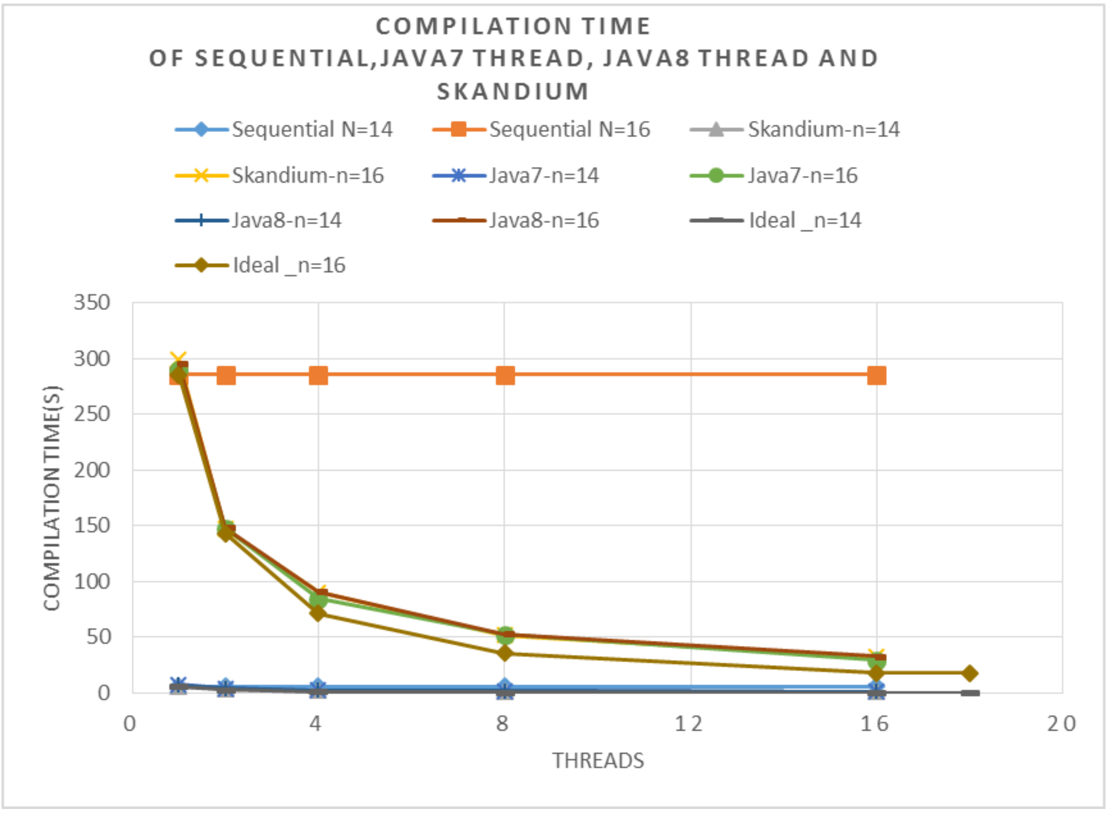

Figure 6. Compilation time of Sequential, Skandium, Java7 Java8 Thread version

Scalability: is the ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to n.

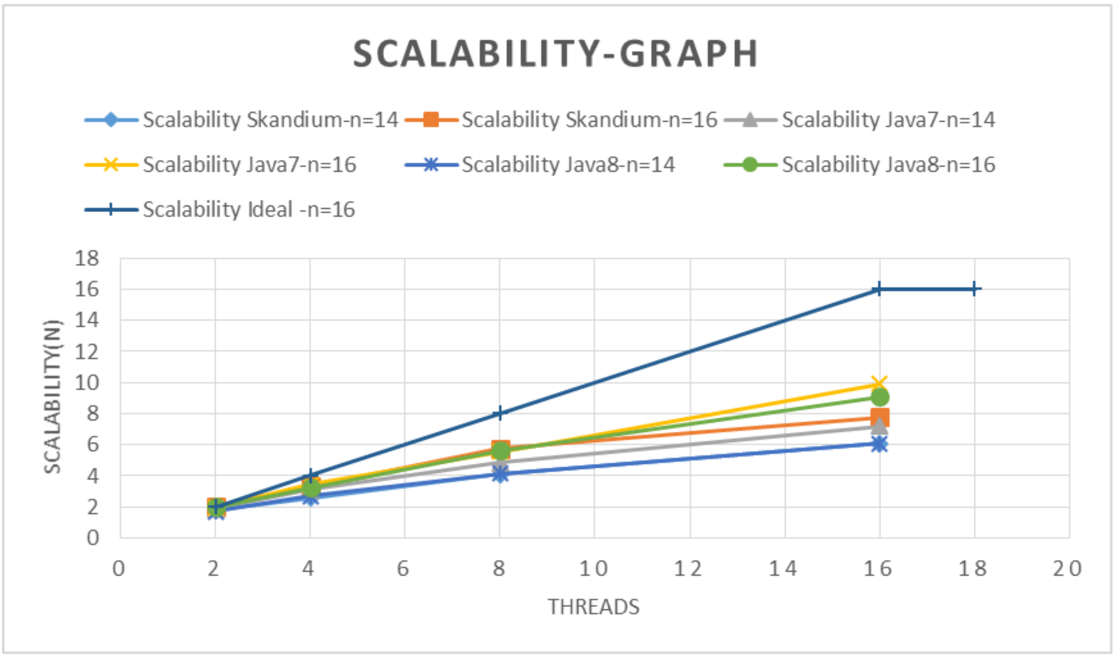

Figure 7. Scalability of Skandium, Java7 and Java8 Thread version

Speed Up: - is the ratio between the best known sequential execution and parallel execution time.

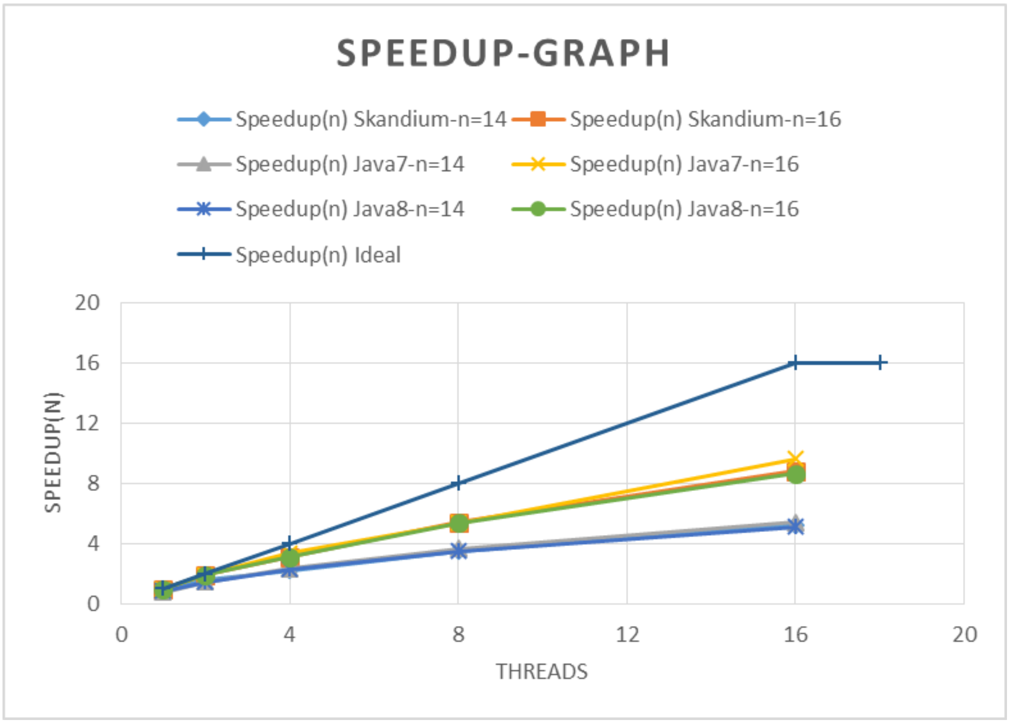

Figure 8. Speedup of Skandium, Java7 and Java8 Thread version

Efficiency*:* - measure the ability of the parallel application in making a good usage of available resources.

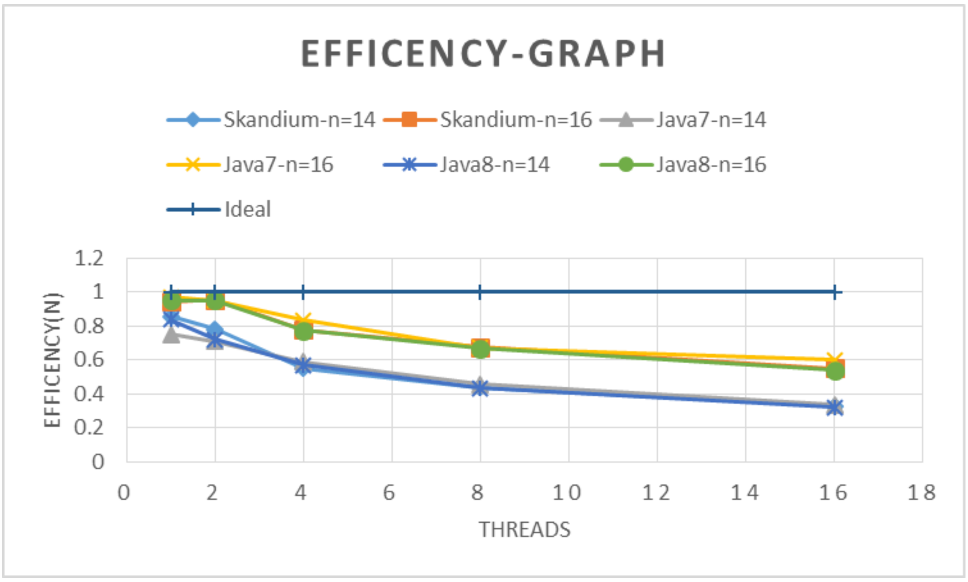

Figure 9. Efficiency of Skandium, Java7 and Java8 Thread version

Figure 7, 8 and 9 show the that for fine grained (matrix size 14 × 14) computations suffer more from the overhead due to threads management and has the worst scalability, speedup and efficiency.

6. Conclusions and Future Work

This project shows Parallel Implementation of N-queens via Skandium, Java7 and Java 8 Thread and an experiment for N-queens problem, whereas all have very close results.

I did an experiment by alternating the matrix size and the number of processors. It is clear that the completion time depends on the commutation grain, for the fine grain computation both parallel versions and the sequential version have almost identical completion time, however for coarse grain computation the parallel versions compilation time decrees as the number of thread increase.

Beside this, both scalability and speed up close to the ideal one for coarse grain computation. At all the parallel implementation does not give the ideal values because of the reasons given in the above. Regarding to the efficiency, the ideal one is achieved for a sequential version .For parallel versions performance decrease for small matrix size when a number of thread increases.

Future work would include repeat the tests without other users on the processor and for higher matrix size starting from 17.
 Another approach to solve N-queens problems is permutation rather than backtracking. This can be easily done by exploiting the futures of java 8 stream class. Providing board size n and computing all possible permutation (from 0 to n-1) to place the queen, then apply filler to identify safe positions to place queens ,lastly count the number of possible solutions. The parallel version can be easily implement by using parallelStream method.

7. References

[1] Danelutto, M. Distributed systems: paradigms and models. Teaching notes, 2013.
[2] Java Platform Standard Edition 8 Documentation. https://docs.oracle.com/javase/8/docs/
Accessed on: Oct-2-2016.
[3] Raoul gabriel Urma,Java 8 In Action.Manning publications
[4] N-queens http://introcs.cs.princeton.edu/java/23recursion/Queens.java.html
Accessed on: Oct-2-2016.
[5] Recursion and Recursive Backtracking.
http://*www.fas.harvard.edu/~cscie119/lectures/recursion.pdf Accessed on: Oct-2-2016.*
[6] Skandium on Github. https://github.com/mleyton/Skandium. Accessed on: Oct-2-2016.
[7] Skandium website. http://backus.di.unipi.it/m̃arcod/SkandiumClone/skandium.niclabs.cl/index.html . Accessed on: Oct-2-2016.
[8] Vanneschi, M. High performance computing: parallel processing models and architectures.