University of Pisa and Scuola Superiore Sant’Anna

Distributed Systems: Paradigms and Models

Game of Life - Project Report

Biruk Bekele Legamo

January 20 ,2017

Abstract

The objective of this report is to describe the Parallel design & implementation of Game of Life which is well known cellular automation with name conway's game of life. the performance evaluation and metrics of the parallel solutions and the sequential results are also included.

Beside the sequential implementation written in Java 8, a multi-thread variants are implemented in Java Thread with standard library and Skandium parallel skeleton framework

I Introduction

In the project more intent is given in in the parallel implementation , where the sequential function implementation is composed into the parallel one. The performance metrics evaluations that is scalability, speedup, completion time and efficiency of all versions have been measured and run on Xeon PHI KNL(Intel(R) Xeon Phi CPU 7210 @ 1.30GHz) with 64 cores , L1d cache: 32K L1i cache: 32K, L2 cache: 1024K and NUMA Architecture.

Describing the document structure, on the first section the sequential Gol and working scenario will be reported , section 2 mainly discuss about the parallel implementation of Java Thread and section 3 about the parlle implementation of skandium briefing details where section 4 shows the metrics evaluation and the performance measures.

2.The Sequential Game of Life

The sequential implementation has three classes Goalboard, Iterval,Gol where the interval class is shard by between parallel implementations. Highlighting the sequential one it consists of finding the state neighbor cell, in order to determine the state of the board on the next generation which more algorithmic than game. The scenarios on each of the class will be discussed below

2.1 Goal Bord class

This class is the main body of the implementation where the sequential and parallel thread access on it. It consists of the Golboard object which receive integer matrix (m) and two matrices arrays *curr grid* and *next grid* that are used to store game life board . They are initialized with the given matrix size plus two in order to copy the ghost cell which is used to check the toroidal structure of the board on each bound.

This means that the Boundary Condition of cells on the edges of the grid will "wrap around" to connect with the opposite edge of the grid.

The northernmost cells are adjacent to the southernmost cells, and the westernmost cells are adjacent to the easternmost cells. As shown in fig. 1 it is simplified the toroidal grid by including "ghost" rows and columns which are copies of the rows and columns on the opposite sides of the grid from the edge rows and columns. A ghost corner is determined by the corner cell that is opposite the ghost corner on the diagonal.

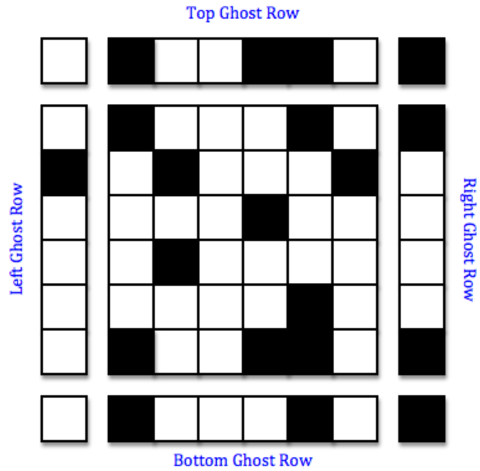


fig. 1 sample grid with the four invisible lines(2 horizontal & 2 vertical) sow that for neighbor calculation not only the adjacent ones are considered but also the Ghost rows.

In calculating the Neighbors the main algorithm is done by new generation(int startrow, in nrow) which returns live or dead cells by calculating the neighbor sand fills to the current grid. For calculating the neighbor the most suggested one is to use the modulo operator (i -1) % n) % m , (i+1)%n. But it is not memory efficient in doing the modulo operation and multiplication. So for checking the neighbor I use sum operation which do summing of the adjacent neighbors checking the sum is either three or two in order to continue for next generation.

```
For (i = startRow; i < startRow+ nRows; i++) {
for (j = 1; j <= matrix_size; j++) {

sum = curr grid[i - 1][j - 1] + curr grid[i - 1][j] + curr grid[i - 1][j + 1]
      + curr_grid[i][j - 1] + curr_grid[i][j + 1]
```

```
                        + curr_grid[i + 1][j - 1] + curr_grid[i + 1][j] + curr_grid[i + 1]
[j + 1];

                if (sum < 2 || sum > 3) {

                  next_grid[i][j] = DEAD;

                } else if (sum == 3) {

                    next_grid[i][j] = ALIVE;

                } else {

                    next_grid[i][j] = curr_grid[i][j];}
```

where in this method startrows+nrows is useful for the parallel implementation of row wise partitioning in order to continue foe each partions of rows.

This method will be called by Play() which is used to display for each iteration the status of the baord. In each iteration the above stated couple of matrices are used, where each matrix is read or written on each step by calling method *swapmatrices()*. Beside the above methods the Golbaord class provides two methods to initialize the Golboard at the beginning of the game either using R*andomintialize (),* which generate random initialization or *seedintialize(long seed)*random generation dependent on inputs seed and useful for obtaining always the same matrix during

Fig 2. sample how generation is done in simple ASCII characters

```
DONE GENERATION:
-----------------------
 | 0 0 0 0 0 0 |
 | 0 0 1 0 0 0 |
 | 0 0 1 1 0 0 |
 | 0 0 1 1 0 0 |
 | 0 0 0 0 0 0 |
 | 0 0 1 0 0 0 |
DONE GENERATION:
-----------------------
 | 0 0 1 1 0 0 |
 | 0 0 1 1 0 0 |
 | 0 1 0 0 0 1 |
 | 0 0 1 1 0 0 |
 | 0 0 1 1 0 0 |
 | 0 0 1 1 0 0 |
```

The last method which is most important for the parallel implementation is splitBoard(int THREADS) returning an array of Intervals dividing the matrix row-wise into balanced no threads.

2.2 Interval Class

this is the container for the details of the intervals during row -wise partitioning. it has two fields (Start and nrows). the Start holds the start row that one worker/thread can update. The nrows holds the no rows to be included or can be said end of row the can be updated by worker.

3. Parallel Implementation of Game of Life

The Game of Life is inherently a data-parallel problem. Since at a given iteration each cell is updated independently of the others. one can partition the matrix in some way and compute all partitions in parallel. This leads to Data-parallel using Map. That is splitting the board into sub-boards/intervals and applying the whole computation on a sub intervals sequentially and then merging the final result after the termination of the computation on sub intervals. Where also the merger(Gatherall) will check for (!=(cond) of generations).

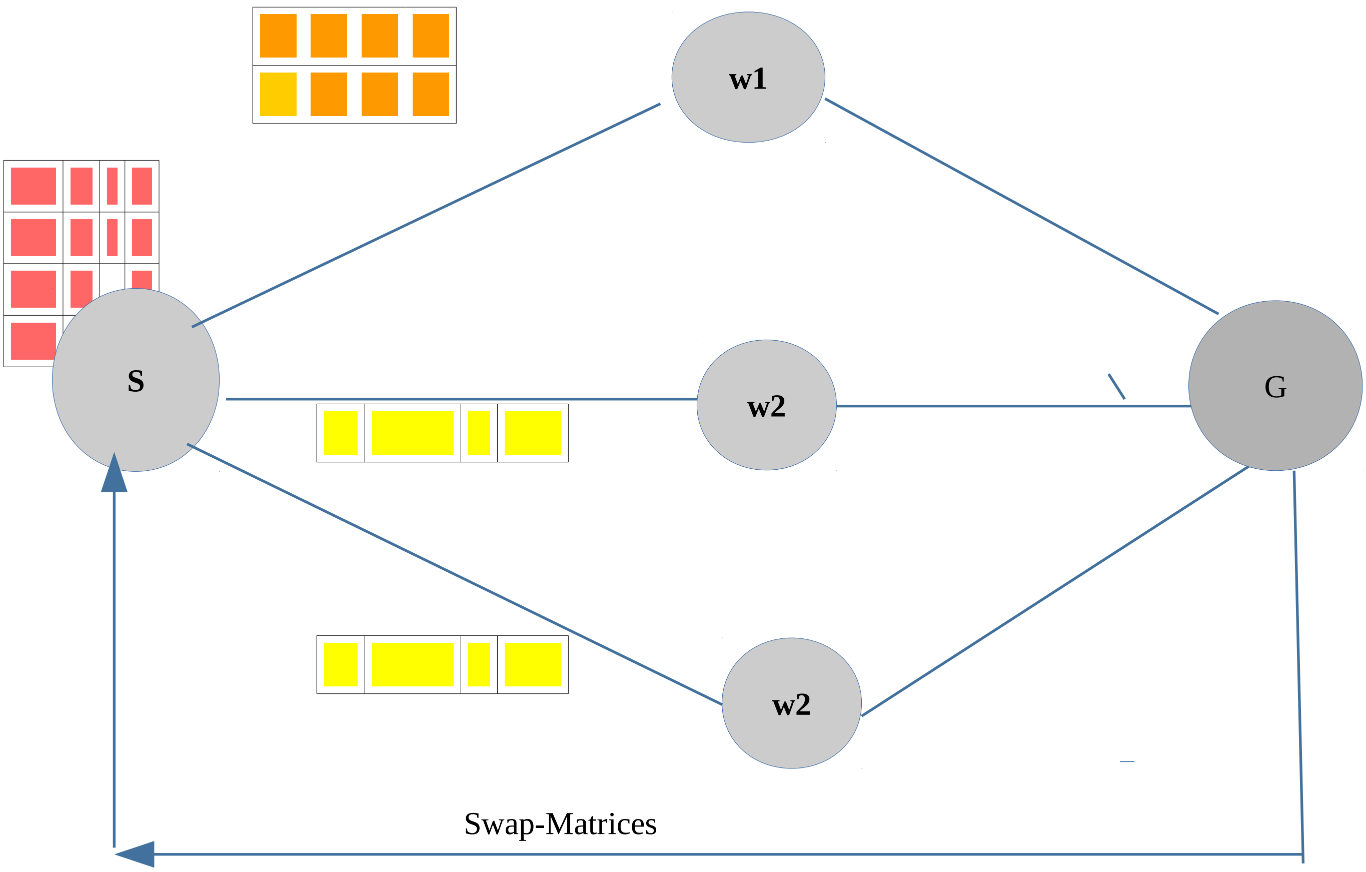


Fig 3 . *4 *4 matrices sketch of data-parallel in Gol implementation*

3.1 The General Cost Model

Taking the computation of one iteration , the map pattern completion time is described

T c = T scatter (n, g) + T map-calc + T gather (n, g)

T c = T scatter (n, g) + T map-calc + T gather (n, g) + Tswap

but the T -swap has been included in the gahterall that makes approximatively

Tmap-calc= Tmap-cacl+tswap

Tmap-calc= g Tf , that is no-generatin(startrows,norows) the time to compute generation of neighbors. For the multhreaded the Tsynch of synchronization of carrier barrier have some overhead. But generally T scatter and T gather is negligible.

Tc= T scatter (n, g) + T map-calc + T gather (n, g) = Tmap-calc = gTf , g=generation
Tf = Tseq/n

3.2 Java Threads

The first parallel implementation of Gol is using javaThread, with thread-pool . There are two class here Golthread and Workerthread. Worker-thread is the one which implements the runnable interface.

3.2.1 GolThread

This is main class for the Java Thread implementation it is important since it manages the thread pool by initializing the Workerthread and await for termination of each threads. Additionally it defines cyclic-Barrier which is used for synchronization of each thread. In beginning the next iteration, it must wait that all the others threads have computed their partitions. so all threads must wait upon at every loop and it is up to the barrier to execute the method swapmatrices at end of each generation and display the status of the board.

```
CyclicBarrier barrier = new CyclicBarrier(THREADS, board::swapplDsil);
  ExecutorService threadpoo=Executors.newFixedThreadPool(THREADS);

          long init = System.currentTimeMillis();
          for (int j = 0; j < THREADS; j++) {
 threadpool.execute(new Workerthread(board,bounds[j].a,bounds[j].b,ngen,barrier));
          }
           threadpool.shutdown();
```

```
threadpool.awaitTermination(10, TimeUnit.MINUTES);
```

list 2. The main part of Golthread ansdas well core of javathread Impmentation

3.2.2 WorkerThread

This implements the runnable interface. Reaciving the reference to the current Golbooard object, starting (row start) and a number of rows(endofrow) to compute, number of generations to be done and a reference to the *Cyclic-barrier* defined in the GolThread . The override method deceleration is listed as

```
public void run() {
for (int i = 0; i <=ngen; i++) {
  board.new_generation(start,matrix_size);
  try {
    barrier.await();
  } catch (InterruptedException | BrokenBarrierException ex) {
  }
```

3.3 Skandium

another parallel implementation is us using skandium skeleton Framework targeting multi-core architecture. This implementation includes the following classes.

3.3.1 GolSkandium

Implementing as the main method where core of Skandium environment is initialized . Below shows the briefing of this class

```
Skandium skandium = new Skandium( THREADS);
Skeleton<Interval, Interval> map = new Map<Interval, Interval>(new
    Splitter(THREADS), new Worker(),
    new Merger(ngen));

Skeleton<Interval, Interval> whileSkeleton = new While<Interval>(map, new Cond());
Stream<Interval, Interval> stream =
    skandium.newStream(whileSkeleton);

Future<Interval> future = stream.input(input);

result = future.get();
```

3.3.2 Splitter Class

Implements the Split muscle with *Split<Interval, Interval>* interface . It receives an interval of the original mesh and splits the row interval into sub intervals where sub intervals equals to the number of threads. Then the sub intervals are distributed to each worker.

3.3.3 Worker Class

Implementing the *Excute <Interval, Interval>* interface of Skandium muscle. accepts an interval and invoke the methd mat.neighbors2 computing the neighbour of each cell and it will executed by each worker in parallel at each step; the output goes to the Merger.

3.3.4 Matrix Class

This class functionality is some as Goalboard. And copies the current matrix to the next matrix using the constructor Matrix method.

3.3.5 Merger Class

This class is required in making Merge<Interval, Interval> implementing gol skandium interface. merges sub-partions together and returns a single interval with the original size by wrapping matrix swap method and also maximum generation decrement are done here.

3.3.6 Cond class

Represent the `Condition<Interval>` used to check the number of generations for a given worker interval, it returns false if the maximum generation is less than zero else it returns true and the execution continues

4. Performance Analysis and Metrics

The performance of the parallel and sequential computation can be analyzed by measuring the completion time. The absolute time spent in the execution of a given application is measured . Test were run r10 times , for the sequential and each parallel implementation with Gen= 600 of 1-16 workers. Plus of different input matrix size 2048 * 2048 ,1024*2014, 512* 512, 256 * 256. all were initialized with seed no 42450.

Metrics

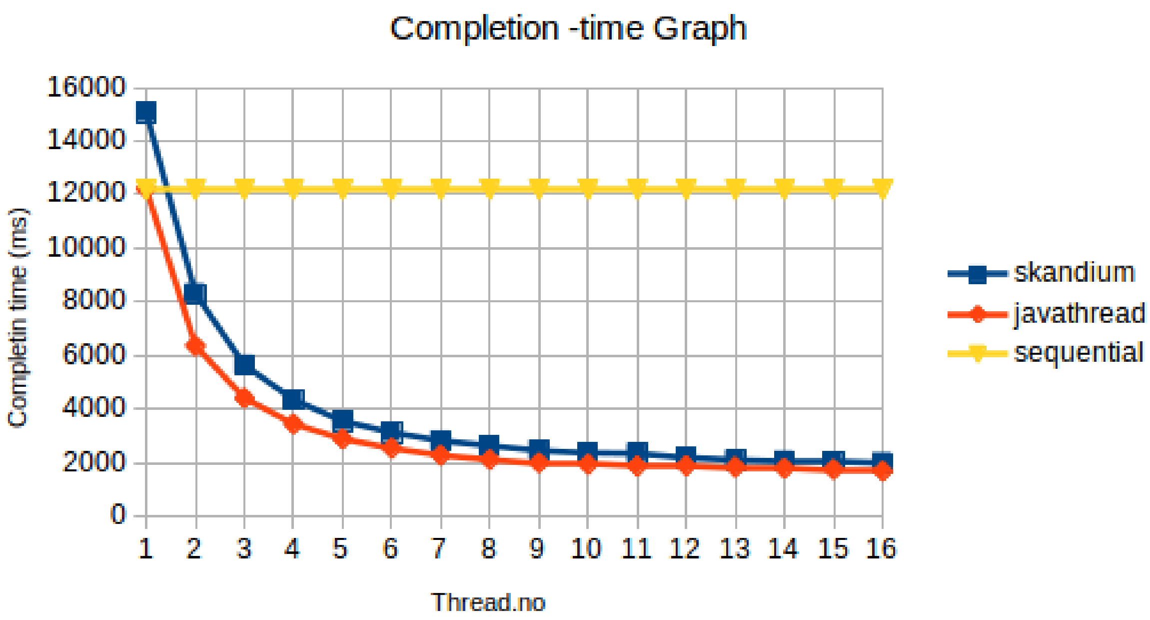
$$\text{speed- up} = \frac{\text{Sequential}}{\text{Parallel(Nw)}}$$

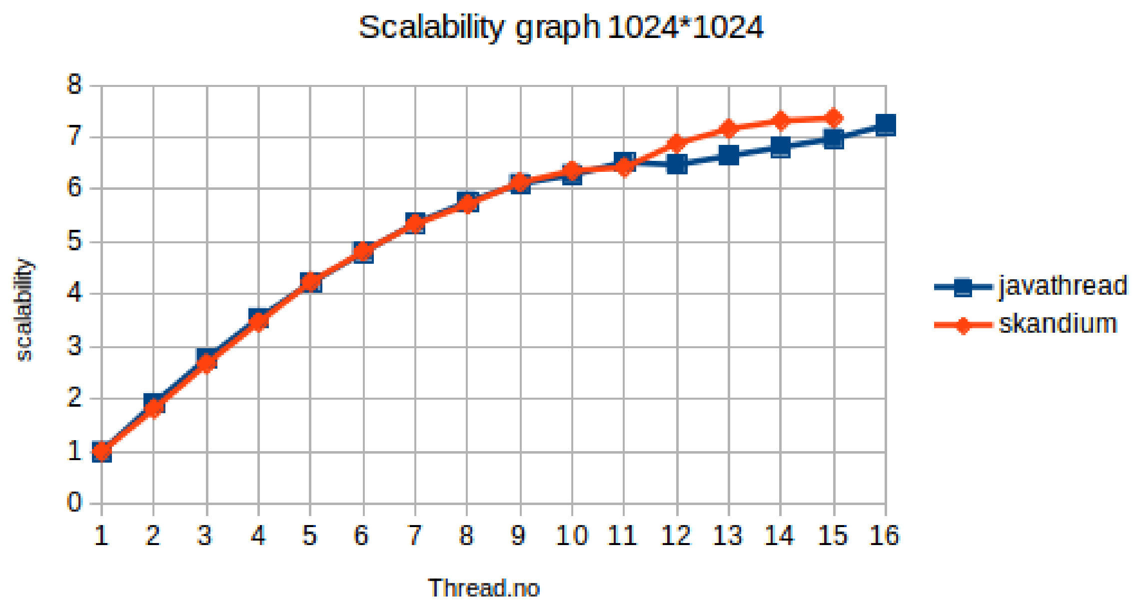
$$\text{Scalability} = \frac{\text{Parallel(1)}}{\text{Parallel(Nw)}}$$

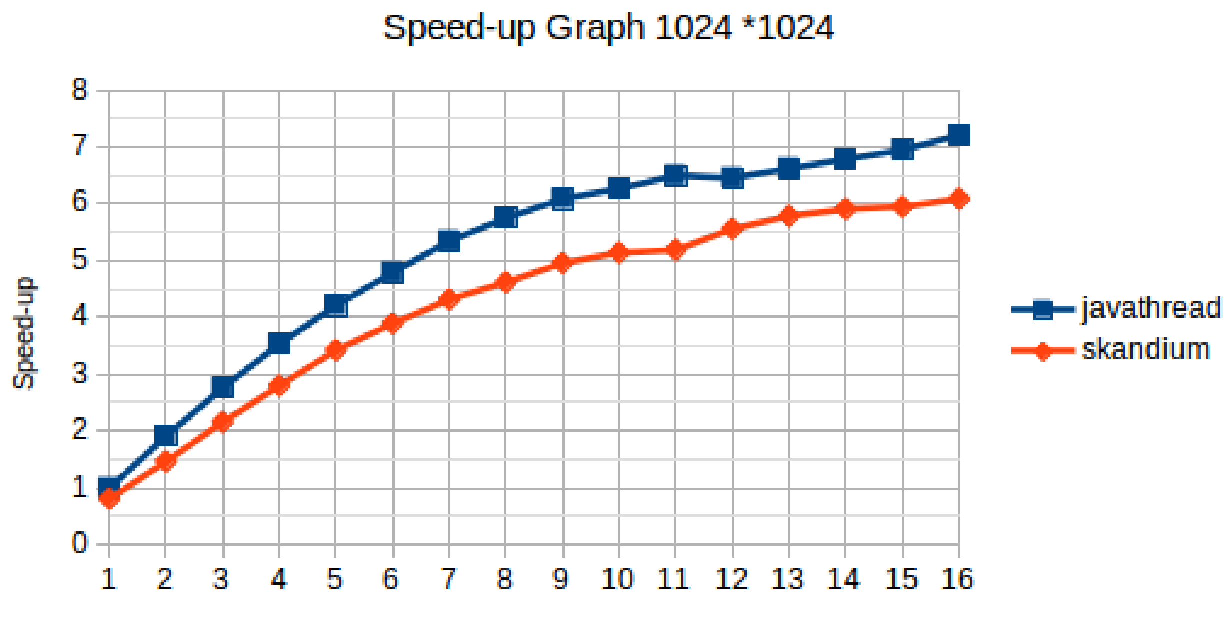
$$\text{efficiency} = \frac{\text{Sequential/T parallel(Nw)}}{\text{Nw}}$$

Results

According set of the metrics the evaluation has been tested. And seen below in for Java-Thread implementation for fine grained 512 × 512,256× 256 computations suffer more from the overhead due to threads management and sequential operations, leads to worst scalability, speedup and efficiency as the number of threads increases. This is also true for skandium implementation, but the scandium implementation as seen below has low efficiency, speedup compared to the Java-thread one this is because of the scalability issue of the While skeleton. Specially with increasing no generation,the checking the condition using he while skeleton, that makes high the computation time for single thread,




Scalability graph 1024*1024
scalability
Thread.no
javathread
skandium
0
1
2
3
4
5
6
7
8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16


Speed-up Graph 1024 *1024
Speed-up
javathread
skandium
0
1
2
3
4
5
6
7
8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

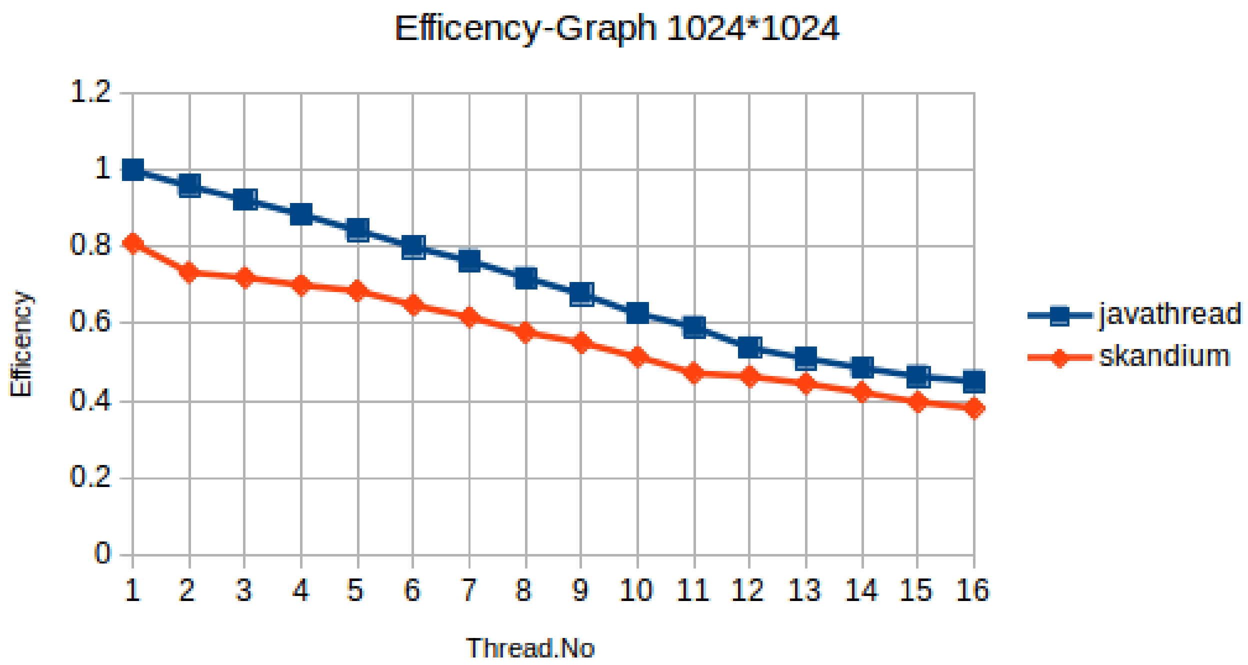
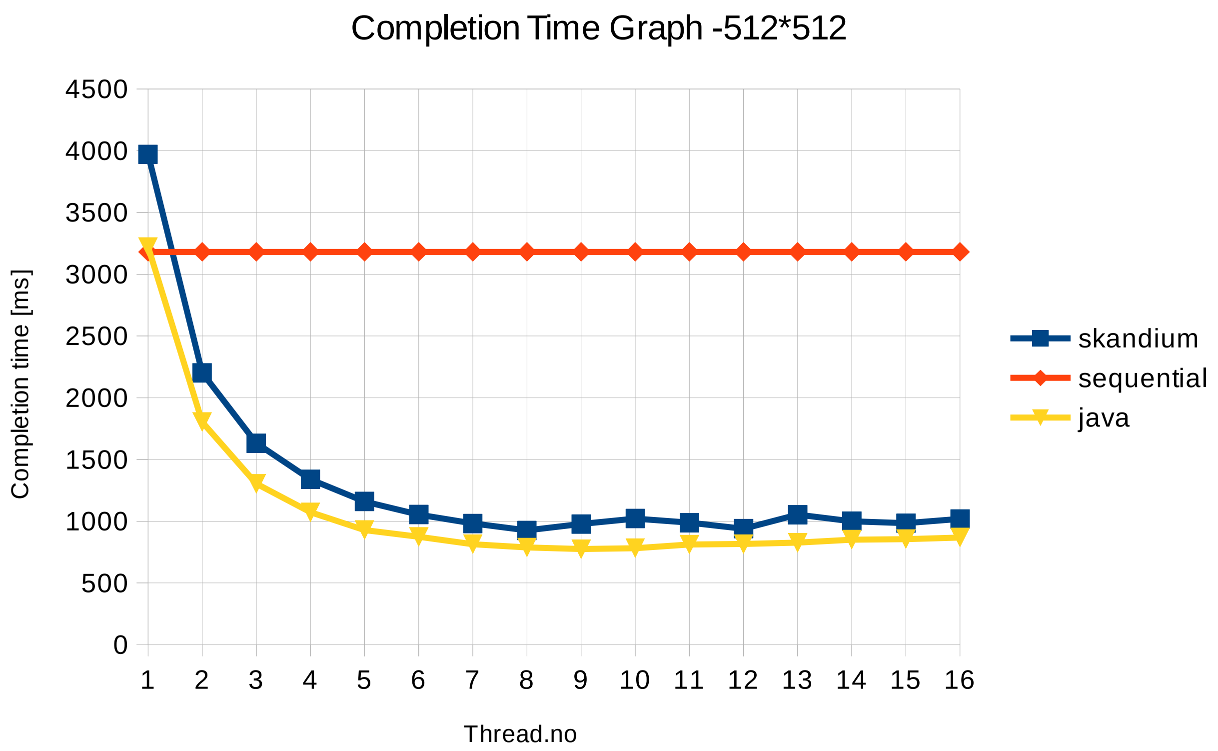
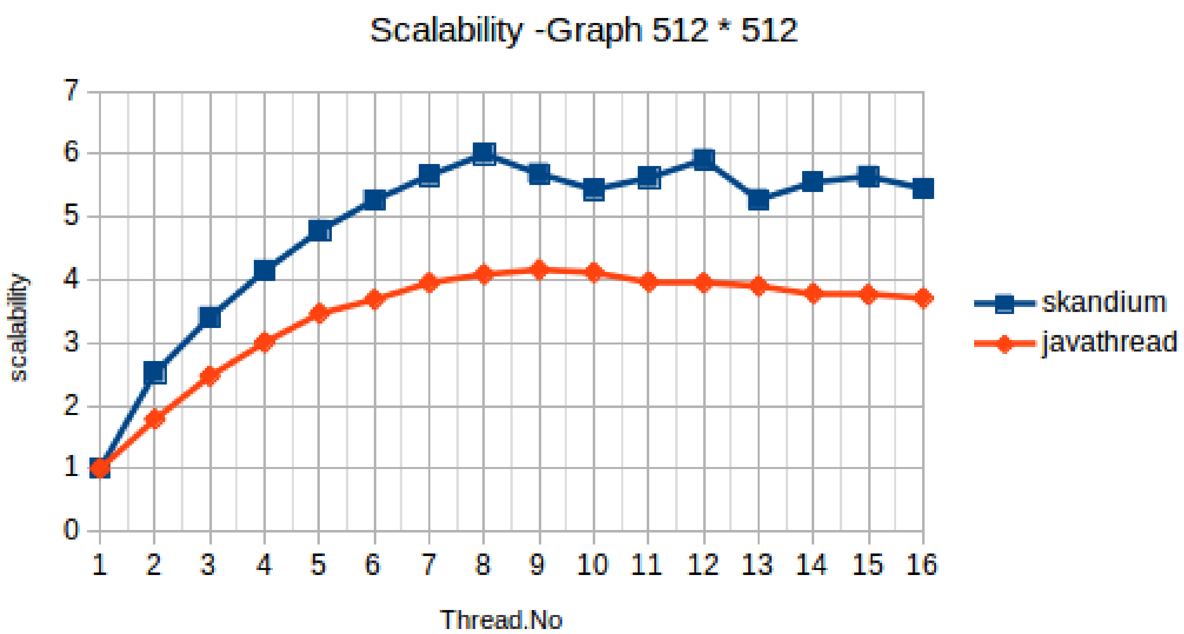
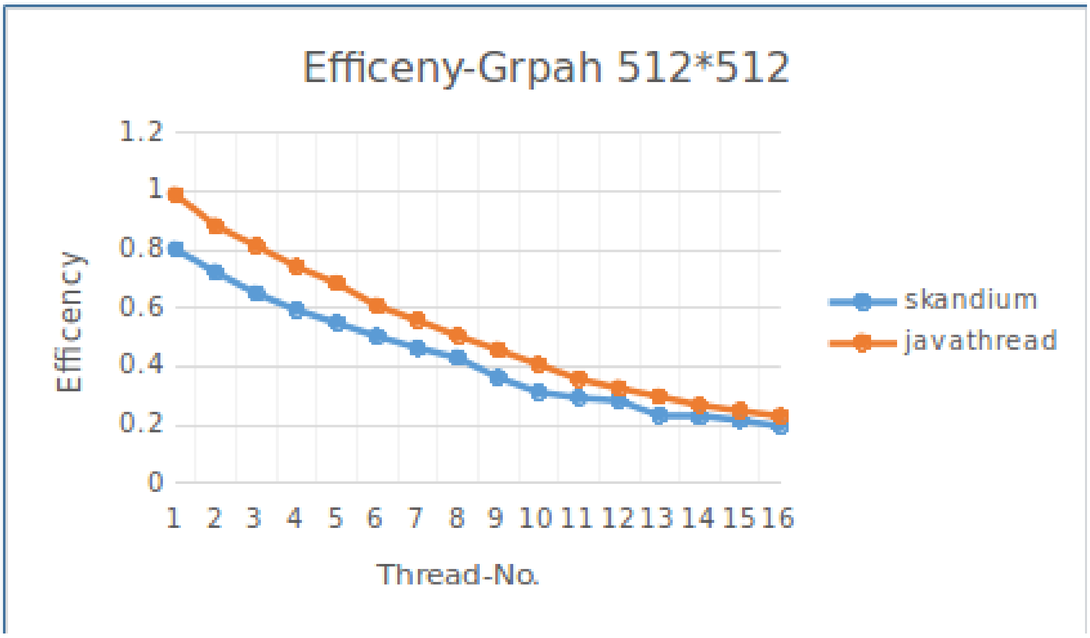

Efficency-Graph 1024*1024
Efficency
Thread.No
javathread
skandium
0
0.2
0.4
0.6
0.8
1
1.2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

fig 4 above.average completion time, speed up, scalability and efficiency with Gen = 600 and a board 1024 x 1024







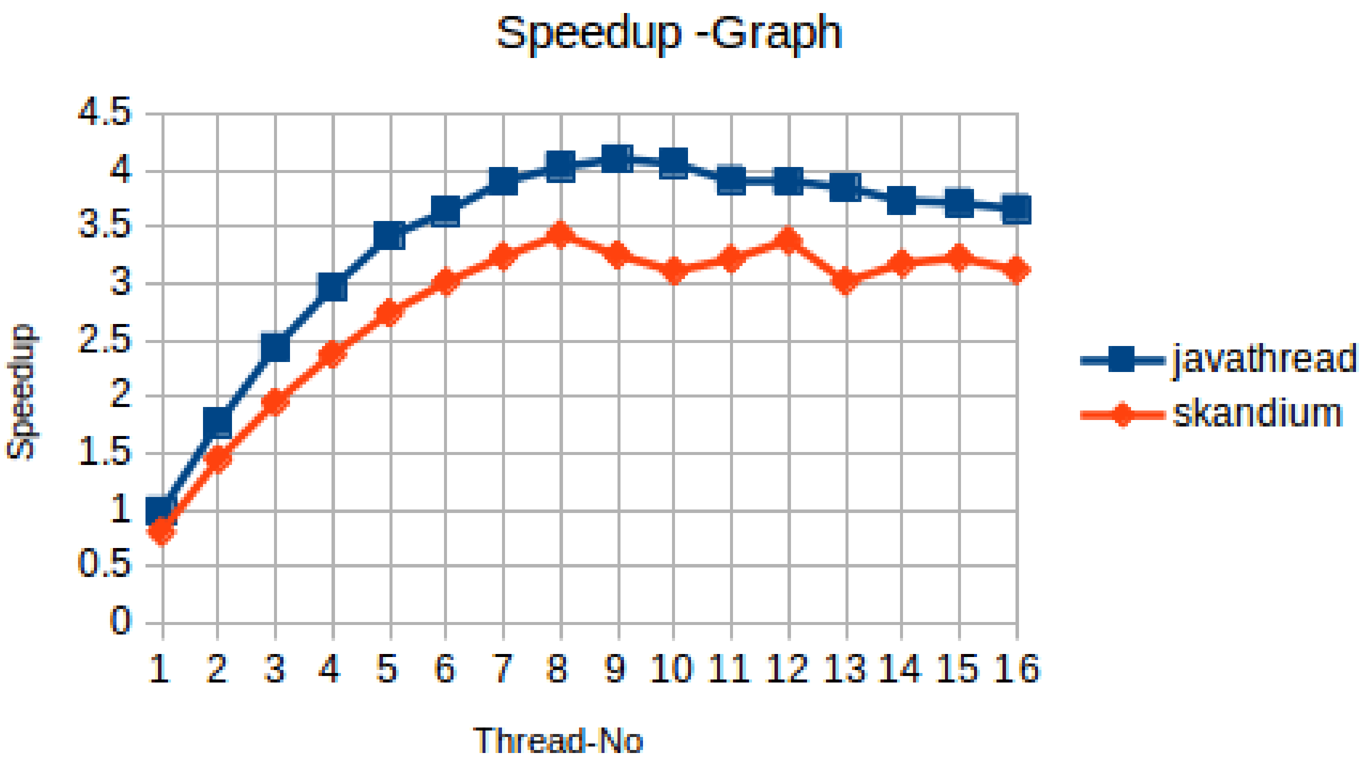
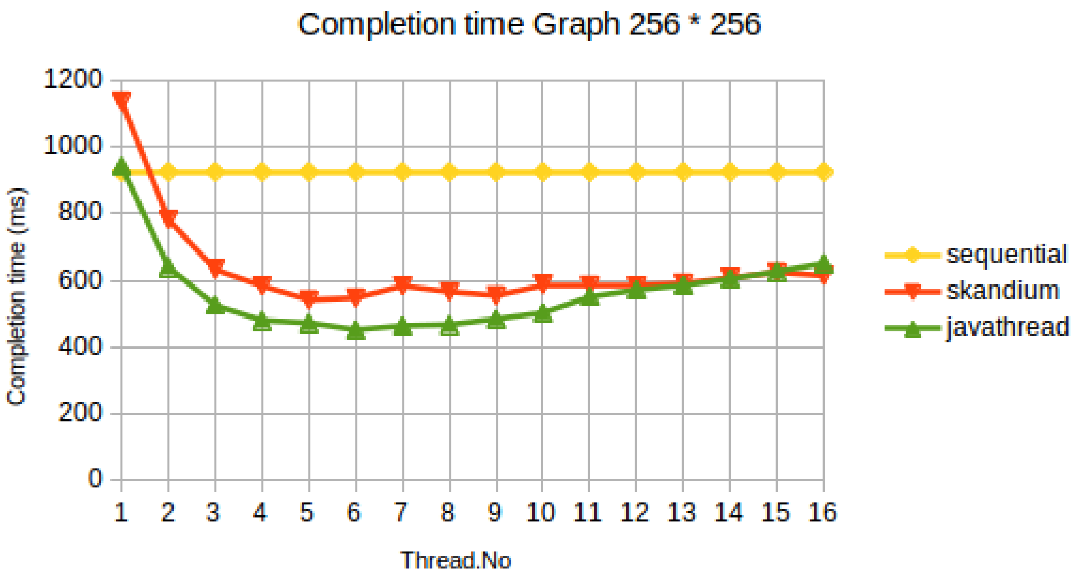
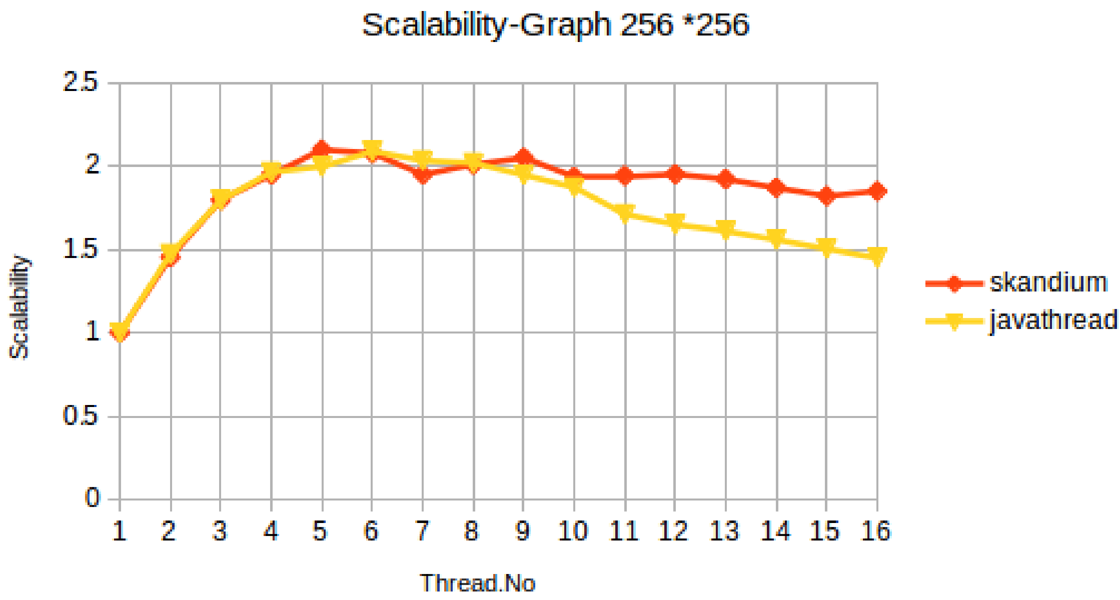
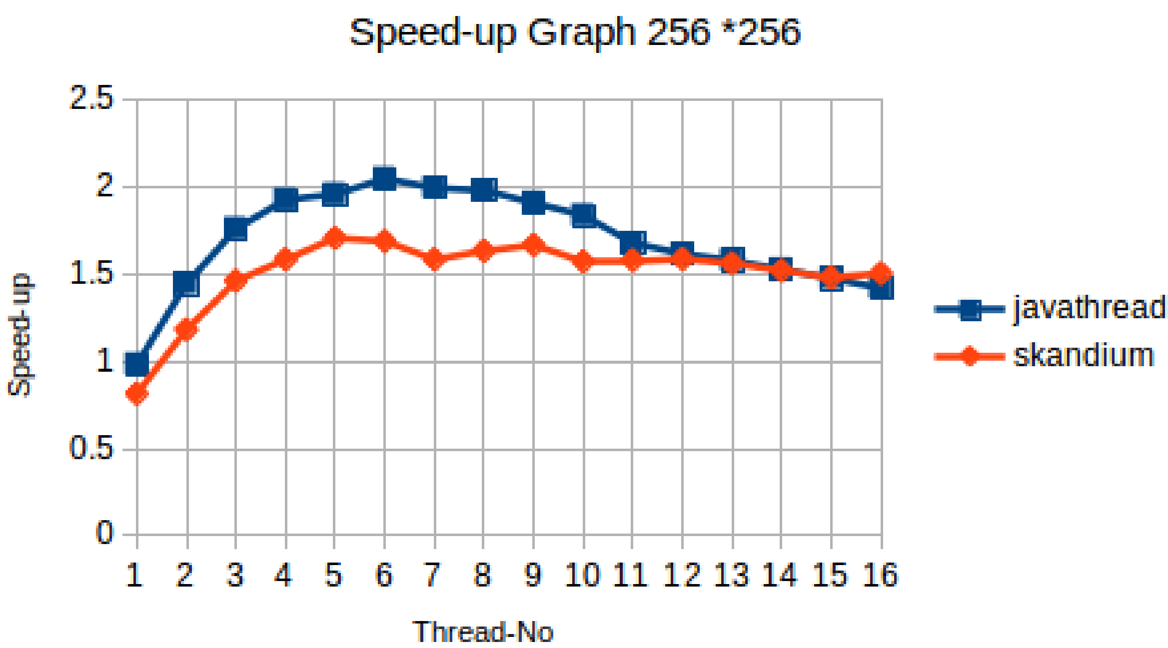


fig 5 above.average completion time, speed up, scalability and efficiency with Gen = 600 and a board 512 x 512


Completion time Graph 256 * 256
Completion time (ms)
1200
1000
800
600
400
200
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Thread.No
sequential
skandium
javathread


Scalability-Graph 256 *256
Scalability
2.5
2
1.5
1
0.5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Thread.No
skandium
javathread


Speed-up Graph 256 *256
Speed-up
2.5
2
1.5
1
0.5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Thread-No
javathread
skandium

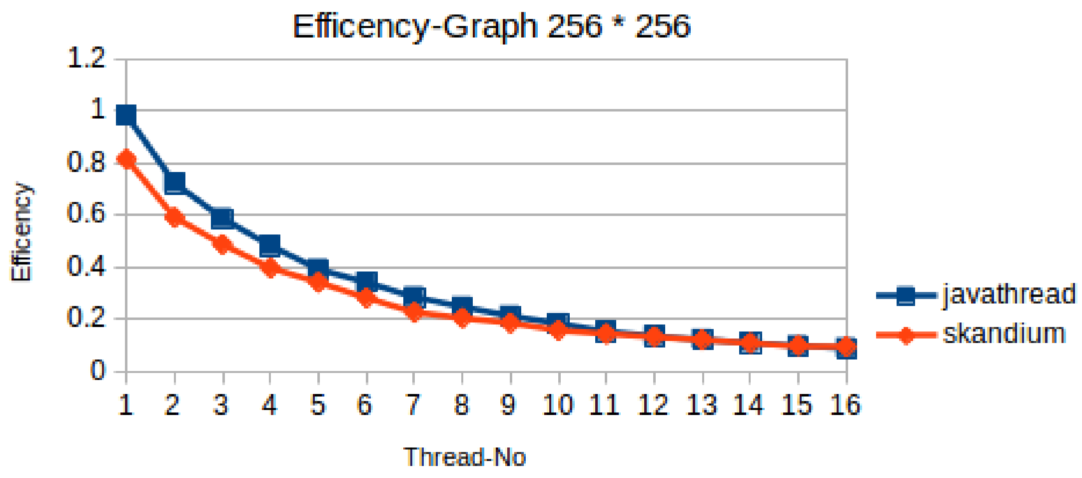


fig 6 above average completion time, speed up, scalability and efficiency with Gen = 600 and a board 256 x 256

Conclusions

when I come to the conclusion ,The project has been implemented for the parallel part by Java thread library and the skandium skeleton library framework. When compared together the skandium suffer some overhead in efficiency, speedup compared to Java thread this is because completion time depends on the communication grain, for the fine grain computation of P =1 both Java-thread versions and the sequential version have almost identical completion time, but the skandium takes more than completion time of sequential computation this due to skandium has many classes , and have to define classes for simple application, and overhead of the while skeleton. however for coarse grain computation of both parallel versions compilation time exponentially decreases as the number of thread increases and the Java Thread implementation as seen on average is the one with the better performance.

Future work includes testing the performance with large matrix size and larger no of threads with random initialization with out user inputs and see the changes in the performance.