Distributed Systems: Paradigms and Models

Fish and Sharks Project Report

Valdo João (507104)

April, 2016

Contents

1. Introduction

The Fish and Shark application is meant to represent an ecosystem which organism lives in. Given a mesh of M × M cells, the simulation evolves through a sequence of discrete events called generations which are characterized by the following features:

- The space, where cells live, is modeled as a 2D toroidal grid of boxes [1]. Each cell represents a fish, a shark or empty;
- A cell behavior is affected by its state and the states of its neighbors (see Figure 1) according to a 9-point stencil [2] operation taking into account certain rules (Algorithm 1 and 2) ;
- All cells are affected simultaneously in a generation.

Sample cell values are:

- **F_6:** a fish with age equals to 6
- **S_12:** a Shark with age equals to 12
- **Empty:** there is no Fish or Shark in the cell.

F_4	F_3	Empty	F_4	F_3
S_12	Empty	F_6	S_12	Empty
S_7	Empty	F_7	S_7	Empty
F_4	F_3	Empty	F_4	F_3
S_12	Empty	F_6	S_12	Empty

🟦 Ghost corners 🟧 Ghost cells 🟨 Neighbors of F_7

Figure 1: Sample 3x3 Matrix

The goal of the project was to implement a parallel Fish and Shark application targeting on multi-cores. The project was developed in Java starting from a simple sequential solution then using Skandium framework was possible to build the parallel application. All tests were run on an AMD Opteron (2.3GHz) processor with 12 cores and 12 contexts. To conclude the report, a brief user guide to launch the program is given.

Algorithm 1 Fish and Sharks

```

    for (i = 1 to #Rows-1) do {
        for (j = 1 to #Columns-1) do {
        manageMyNeighbors( i, j )
        //code to update ghost cells
        }
  }
```

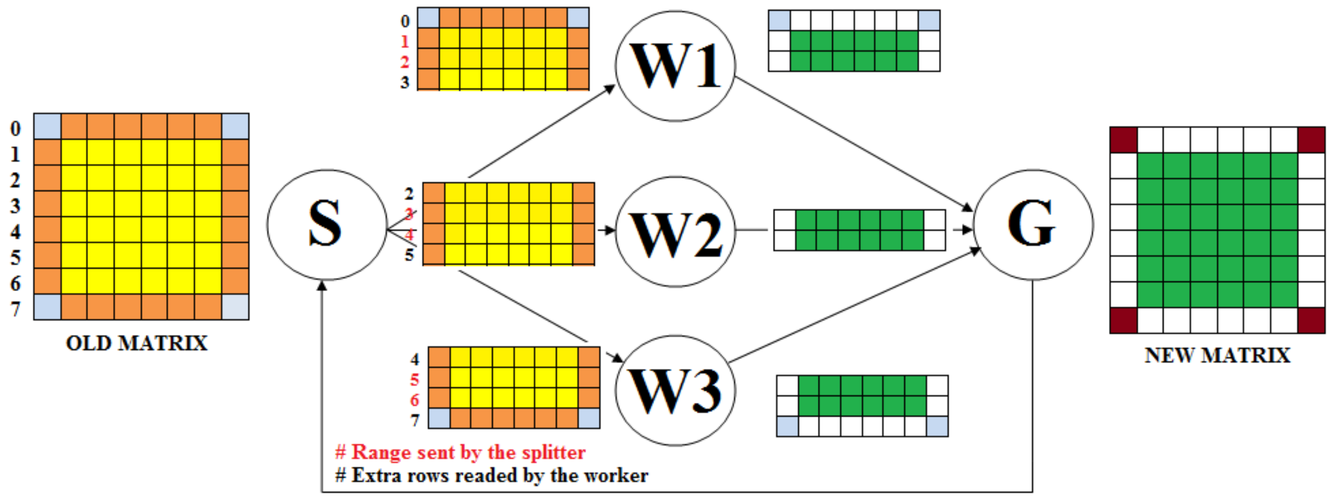
Algorithm 2 manageMyNeighbors

```
1.      //two for loops to find my neighbors
2.      for (row =  i-1 to i+1) do {     //limit row boundaries
3.          for (col = j-1 to j+1) do {  //limit column boundaries
4.              // code to count my fish neighbors and how many are in breeding age
5.              // code to count my shark neighbors and how many are in breeding age
6.          }
7.      }
8.      //manage myself according with my neighbors information
9.      if ( I am fish)   //code to apply fish rules
10.     else
11.         if (I am shark) //code to apply shark rules
12.         else //code to apply empty rules
```

2. Implementation of Parallel Fish and Shark

In order to parallelize the application it was used the Map pattern in which the Scatter sends to each worker a certain range (interval) of the matrix. Workers should also need two extra rows in order to compute the next state, i.e. the row immediately before the assigned sub-matrix and the one immediately after. The workers also update the ghost cells of the sub-matrix.

The matrix is logically split by row-wise giving to each worker a range of the matrix to process. This can be done safely by using two matrixes, the old one which is read-only and the new one which is write-only. The merger updates the ghost corners and swaps the new and old matrix.



0
1
2
3
4
5
6
7
OLD MATRIX
S
W1
W2
W3
G
NEW MATRIX
# Range sent by the splitter
# Extra rows readed by the worker

Figure 2: sample 6x6 matrix with three threads

2.2 The Cost Model

The cost model for the Map pattern can be derived by taking into account the completion time for computing one generation which is:

$$Tc = Tscatter + TF + Tgather$$

Where TF is given as Tseq/n. Tseq is the sequential time and n is the parallelism degree.

Tscatter is negligible as it only sends intervals (start index and end index) of each sub-matrix to each worker. Tgather is also negligible as it only performs update of the four ghost corners and swaps the two pointers of the old and new matrix. Therefore, the completion time can be approximated as:

$$Tc \sim TF$$

The completion time for all the generations can be approximated as:

$$Tc \sim G \times TF$$

Where G is the total number of generations.

2.3 Implementation with Skandium

The parallel implementation of fish and sharks with Skandium followed the approach suggested in [3] and [4]. The application consists of the following classes:

- **Matrix:** this class is used generate the original matrix using a method to instantiate the matrix with random values based on the indexes computed by the random index x and random index y methods.

- **Splitter:** this class implements the Split<Range, Range> interface required by the Skandium Map skeleton. It receives an interval of the original matrix and splits the row interval into sub intervals where sub-intervals are equal to the number of threads. Then the sub-intervals are distributed to each worker. The size of each sub interval is determined by dividing the total matrix size with the number of threads (reminder operation is used to distribute balanced row size intervals).

- **Worker:** implements Execute<Range, Range>. The method getN*eighbors*, accepts an interval and computes the neighbors of each cell and writes the result value onto the new matrix in the given interval, based on the algorithm 2 stated above.

- **Merger:** implements Merge<Range, Range>. This class writes the values of the 4 Ghost corners onto the new matrix, and then swaps the new and old matrix.

- **Range:** this class is the container for the details of the intervals; it has two fields (Start and end). The Start holds the start row that one worker can update. The end holds the end of the row that one worker can update.

- **Cond:** this class represent the condition used to check the number of generations, it returns false if the number of generations is less than zero else it returns true and the execution continues.

- **Run:** this class implements the main method and initializes the Skandium environment.

Initialization of Skandium environment

```
Skandium scandium = new Skandium( threads );
Skeleton<Range, Range> map =   new Map(
                          new Splitter( threads),
                          new Worker(),
                          new Merger());

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

Future<Range> future = stream.input(new Range(1, matrixSize));
Range result = future.get();
```

3. Performance Analysis

In order to test the performance of the fish and shark application, the completion time of all the modules (scatter, workers and gather) was measured by the Java System.nanoTime for extremely precise measurements of elapsed time [5].

The tests were run over different sizes of the input matrix: 500x500, 1000x1000, 1500x1500 and 2000x2000. All of them with 50 generations.

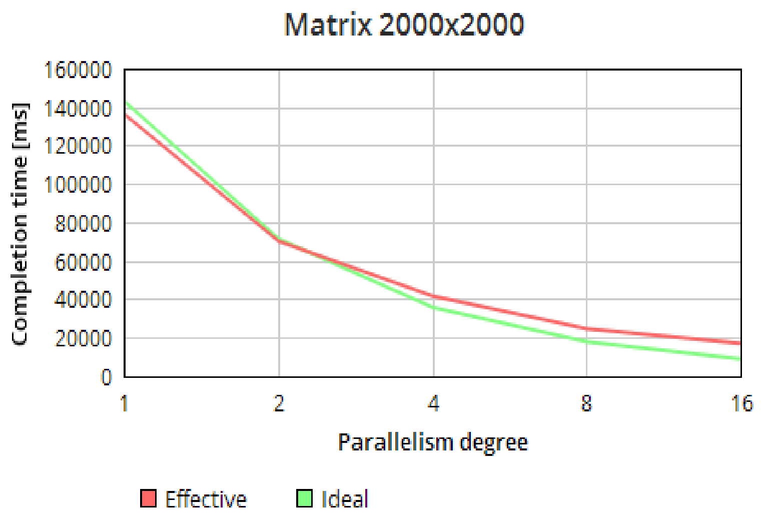


Figure 3: Completion time for computation of size 2000

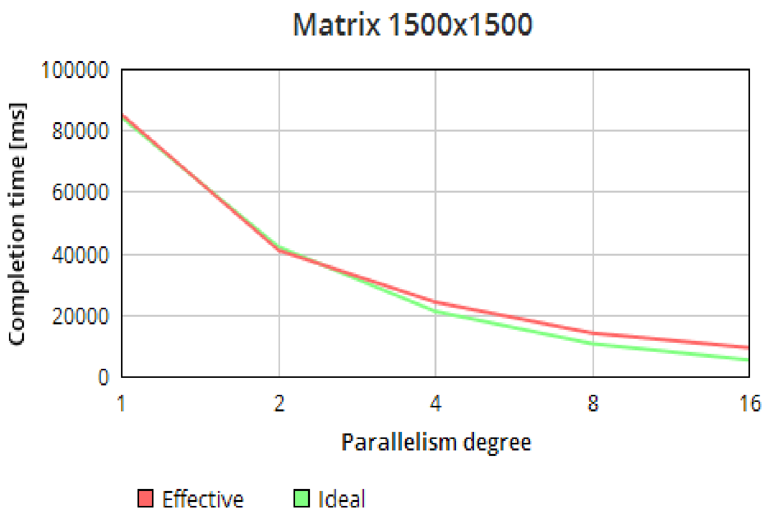


Figure 4: Completion time for computation of size 1500

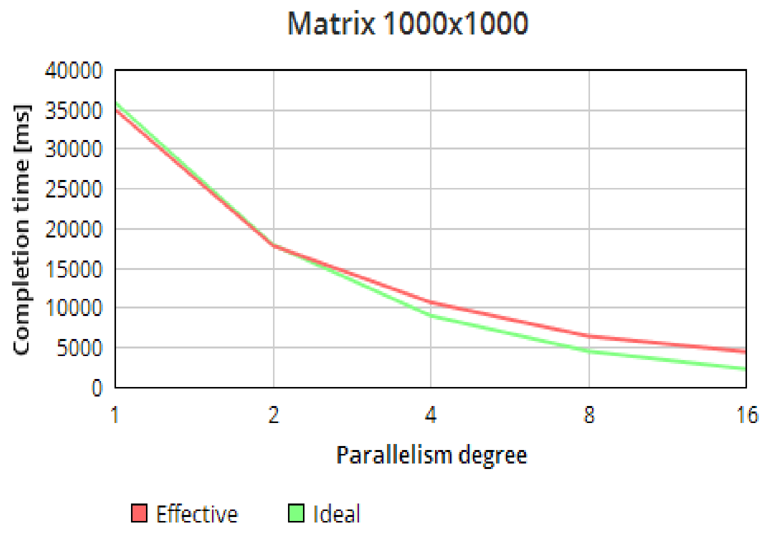


Figure 5: Completion time for computation of size 1000

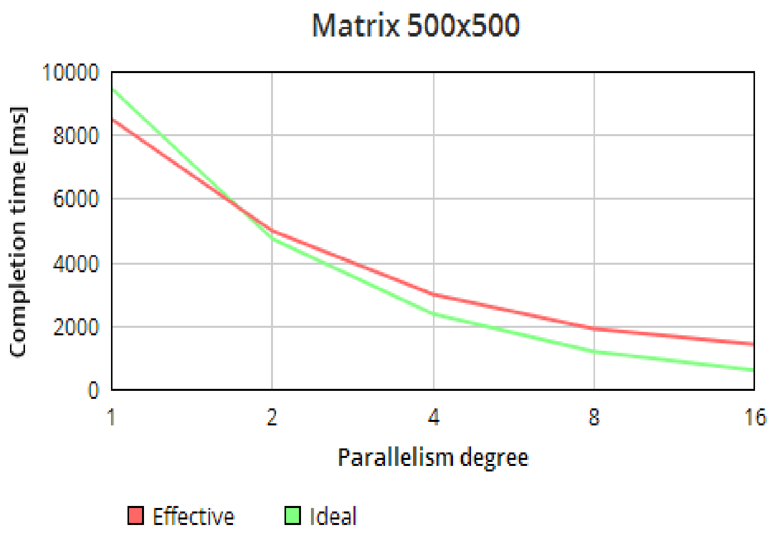


Figure 6: Completion time for computation of size 500

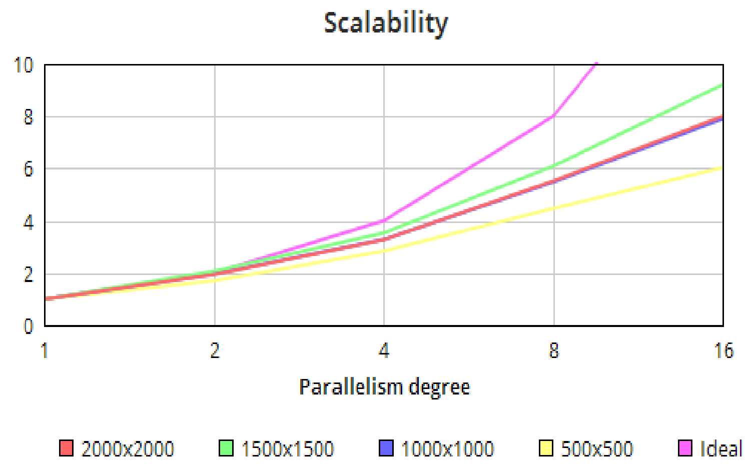


Figure 7: Scalability

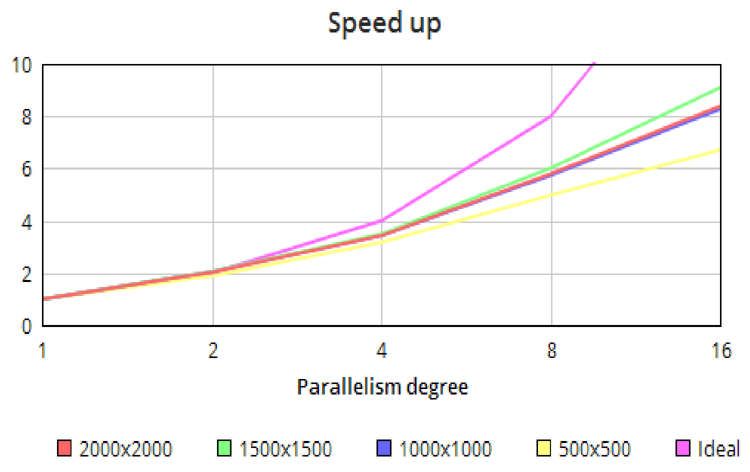


Figure 8: Speed up

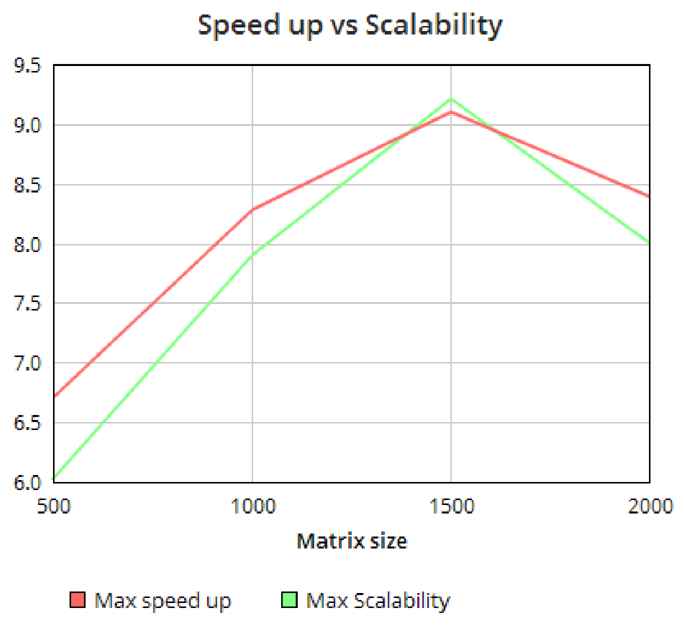


Figure 9: Speed up vs Scalability with 16 threads

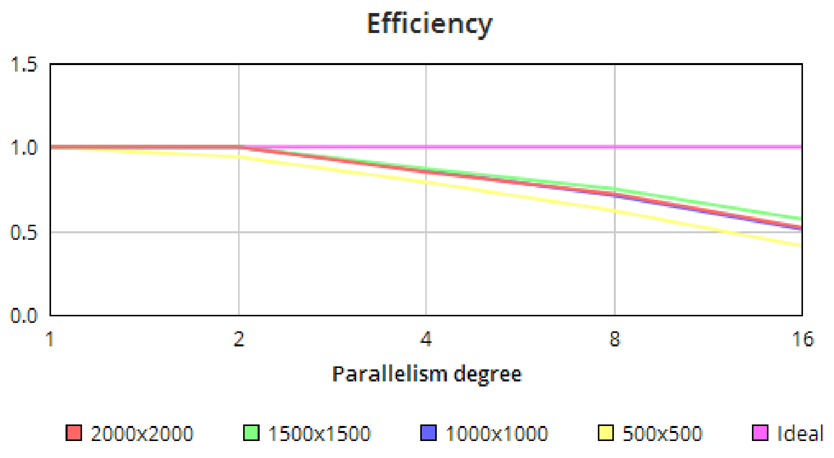


Figure 10: Efficiency

4. Conclusions

The tests of the fish and shark application were made in different scenarios. From the results obtained from the completion time it is clearly noticeable that it depends on the computation grain and the number of generations.

Based on the tests shown in the figures above, the computation has better performance (close to the ideal one) for matrix of size 1500, while the performance measures for matrix of size 1000 and of size 2000 are very similar. The worst performance was achieved for fine grain computation as the matrix with size 500.

It’s also noticeable from figure 9 that both maximum scalability and maximum speed increases with the size of the matrix until reaches it’s peak with matrix size equals to 1500, then it starts decreasing. With that saying, it can be concluded that with a matrix of size 3000 or even 4000 the performance also tends to decrease.

As expected from the derived cost model Tc ~ TF. This can be proven with the example of 2000x2000 matrix with 16 threads and 50 generations in which:

- Tc = 19085.234 [ms]
- TF = 19084.564 [ms]
- Tscatter = 0.37 [ms]
- Tmerge = 0.292 [ms]

[2] Suggests that it’s possible to achieve stencil and cache optimizations by assigning “strips” to each core in a technique known as Strip-mining (also known as loop tiling or loop blocking) which groups elements in a way that avoids redundant memory accesses and aligns memory accesses with cache lines. Future work includes apply this technique to measure how much it improves the performance.

5. References

[1] Arvind Krishnamurthy. Shared Memory Programming. 2004. http://homes.cs.washington.edu/~arvind/cs424/notes/l5-6.pdf

[2]Department of Computer and Information Science of University of Oregon. Stencil Pattern. http://ipcc.cs.uoregon.edu/lectures/lecture-8-stencil.pdf

[3] Mario Leyton; Jose Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5452456

[4] Patrizio Dazzi. Programming with Skeletons: Skandium. http://backus.di.unipi.it/~marcod/SPM1011/Skandium.pdf

[5] Java Documentation. nanoTime. http://docs.oracle.com/javase/6/docs/api/java/lang/System.html#nanoTime%28%29

[6] Marco Danelutto. Distributed systems: paradigms and models. Teaching notes.

6. Manual for Fish and Sharks application

How to copy the project to remote account:
scp -r path_to_the_FishShark your_acount@titanic.di.unipi.it:

How to compile
javac -cp /usr/local/Skandium/skandium-1.0b2.jar src/fish_and_sharks /*.java

How to run
java -cp /usr/local/Skandium/skandium-1.0b2.jar: fish_and_sharks.Run #threads #matrixSize #generations.
E.g: java -cp /usr/local/Skandium/skandium-1.0b2.jar: fish_and_sharks.Run 1 2000 50
The above example will run the application with 1 thread, matrix size of 2000 for 50 generations.

PS: In order to see the value of the original matrix in the **constructor** of the **class Matrix** you need to uncomment the method **printMatrix()** which is on line 46.

In order to see the value of the updated matrix after the given iteration, in the **class Run** you need to uncomment the method **matrix.printMatrix()** which is on line 57.