Project Report: Game of Life

"Distributed system: paradigms and models"
(prof. M. Danelutto)

Student: Andrea Tesei (464561)

Master degree in Computer Science and Networking
University of Pisa and SSSUP Sant'Anna

-a.y. 2015/2016-

Table of Contents

1. Introduction

The project consists in the implementation of three different versions of the Game of Life: one parallel version implemented with threads; another parallel version implemented with an existing parallel programming framework named FastFlow (studied during the course); and a sequential version, used for performance evaluation.
In order to describe the logic behind this game I would quote the definition from the GOL's Wikipedia page: "*The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970*".
For other informations and to read the rules of the game, the reader can visit https://en.wikipedia.org/wiki/Conway's_Game_of_Life.
The rest of this report is organized in this way: in the second chapter, all the design choices will be described, including the algorithms choosen and a model for the completion time for both parallel versions; in the third chapter the detail of the implementation will be described even with some considerations about optimization done in the code; in the fourth chapter the way to prove the correctness of the programs will be discussed; in the fifth chapter some details about the various tests and performance evaluations done; and in the last chapter there will be the user manual.
All tests have been perfomed on a workstation with two Intel Xeon CPU E5-2650 with 8 cores each, and a Xeon Phi coprocessor with 60 cores.

2. Design choices

In this chapter the abstract design of the parallel and sequential algorithms are presented, providing a model for the completion time for both parallel versions (Thread and FastFlow).
The cellular automaton is modeled by a 2D toroidal matrix in which each cell can assume only two states: *1* if it's alive, and *0* if not. At each iteration of the algorithm, the next state of a cell is computed considering its current state and the ones of the eigth neighbours surrounding it, according to the rules of the Game of Life.

2.1 Sequential version

The following pseudo-code snippet describes the behaviour of the sequential algorithm implemented in the project:

```
1  #GLOBAL num_rows, num_columns, matrixRead, matrixWrite
1  #INPUT num_rows num_columns num_iterations seed
2  Matrix matrixRead = initialize_matrix(seed); //Initialize matrix with random values
3  Matrix matrixWrite = matrixRead;
4  for n in 1..num_iterations do
5      for i in 0..rows-1 do
6          for j in 0..num_columns-1 do
7              neighbours = # of neighbours alive;
8                  matrixWrite[i*cols+j] = computeValue(matrixRead[i*cols+j], neighbours);
9          done
10     done
11     matrixWrite.swap(matrixRead);
12 done
```

- Algorithm 1: Game of Life sequential -

First, the matrix is allocated and initialized with random values (line 2-3). Then, the first loop (line 4-12) controls the number of iterations given as input by the user: at this point, for each row i and column j, the number of the alive neighbours is calculated accessing the previous and the next rows (line 7), and the new value for the cell (i,j) is computed (line 8) according to the GOL's rules (wrapped into the *computeValue* function). At the end of each iteration, the matrix used for writing becomes the new *matrixRead* and all is ready for a new generation (line 11).
In the next chapter we will analyze the detail of the implementation.

2.2 Parallel versions

The following pseudo-code snippet express the behaviour of a generic worker in the thread implementation of the parallel algorithm, implemented with standard C++11 threads:

```
1  #GLOBAL barrier, matrixRead, matrixWrite
2  #INPUT start_row, end_row, num_rows, num_columns, num_iterations
3  for i in 1..num_iterations do
4      for i in start_row..end_row do
5          for j in 0..num_columns-1 do
6              neighbours = # of neighbours alive;
7              matrixWrite[i*cols+j] = computeValue(matrixRead[i*cols+j], neighbours);
8          done
9      done
10     barrier.spinWait(function swapMatrix);
10 done
```

- Algorithm 2: Parallel GOL with threads -

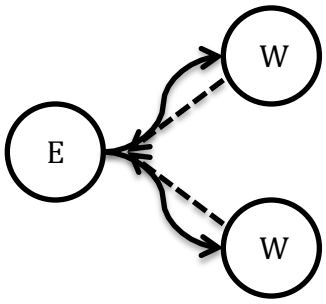
As we can see, this piece of code is quite similar to the *algorithm 1,* except that that in this implementation only a small partition of the rows is assigned to a generic worker, and it is the only one responsible for the update of the assigned partition (static partitioning). Actually the distribution of the rows is done statically by the *main* function, which computes the division's result x of the number of rows by the number of workers and this will be the size of each partition: if the result of the division has any remainder, the main function assign one more row to each thread until the remainder become zero. In this implementation, a busy wait is executed by each thread while waiting to start the next iteration (line 10): when the last thread enter in the barrier, it has to swap the two matrices in order to let start the next iteration.
Given M the number of rows, N the number of columns, K the number of iterations, *nw* the number of threads, assuming an even distribution of the rows among the *nw* workers, we can model the completion time of this computation with the following formula:

$$T_C \approx T_{init-matrix}(N * M) + K * \left(\frac{M}{nw} * T_{update-row}(N) + T_{barrier}(nw) \right)$$

Where:

- $T_{init\text{-}matrix}(N * M)$ = the time to initialize the whole matrix
- $T_{update\text{-}row}(N)$ = the time to update a single row
- $T_{barrier}(nw)$ = the average time spent in the barrier with nw workers

The other FastFlow implementation use a well-known pattern of parallel computation which is called *master-worker*, implemented with a farm with feedback and without any collector.



- Figure 1: the master-worker farm -

The custom emitter has to initialize the matrix, send the various partitions to workers and swapping the matrices at each iteration in order to compute the next generation.
The next pseudo-code snippet describe the computation of the custom Emitter, subclass of the ff_node_t<T> of the FastFlow library (the worker's code is the same as in *Algorithm 2*, except the *spinWait)*:

```
1 #GLOBAL matrixRead, matrixWrite
2 #INPUT loadbalancer, num_rows, num_columns, num_iterations
3 matrixRead = initialize_matrix(seed);    // Initialize matrix with random values
4 matrixWrite = matrixRead;
5 received_iterations_results = 0;
6 svc(GOLTask t){
7     if(t != null) then
8           received_iterations_results++;
9           if(received_iterations_results == loadbalancer.getNWorkers())
10                matrixWrite.swap(matrixRead);
11                num_iterations--;
12    -- If other iterations have to be performed, new tasks will be sent        --
13    -- If not, an EOS is sent to each worker in order to stop computation      --
14}
```

- Algorithm 3: Game of Life FastFlow implementation -

When the matrix is initialized (line 3), the *emitter* starts to send each partition (GOLTask) to any ready worker (the farm load balancer use the on_demand policy). When it has finished to send all the tasks, it will wait reply from each worker. Whenever it has received all results (line 9), it perfoms the matrix swapping and send another set of task to the available workers (line 10-13). The same partitioning as the thread implementation is done by this custom *emitter.*
Given M the number of rows, N the number of columns, K the number of iterations, *nw* the number of workers, assuming an even distribution of the rows among them, we can model the completion time of this computation with the following formula:

$$T_C \approx T_{init-matrix}(N * M) + K * \left(\frac{M}{nw} * T_{update-row}(N) + T_{emitter}(nw)\right)$$

Where:

- $T_{init\text{-}matrix}(N * M)$ = the time to initialize the whole matrix
- $T_{update\text{-}row}(N)$ = the time to update a single row
- $T_{emitter}(nw)$ = the time for the emitter's operation (swapping matrices and sending new tasks)

In both cases the initialization phase can be parallelized as well in order to minimize the completion time of this operation.

3. Implementation details

In this chapter the details of the implementation will be discussed, with some considerations about the code optimizations done. First of all, the matrix is stored as a single pointer to an array of *unsigned_char*: working with single bit is considered not so efficient; so small integers are better choice in this kind of computation. The implementation as a single pointer to an array leads some good performance gain, since it can reduce the overhead of the memory when accessing the previous and the next row, in order to compute the next generation of the cellular automaton; on the other hand, i have experience that, since the GOL's space is modeled by a 2D toroidal matrix, this implementation produce some perfomance degradation when the index of the row is computed dynamically with the modulo operation (especially when vectorization is applied).

3.1 GOLMatrix class & Vectorization

GOLMatrix is a custom class which is actually a wrapper of two matrices, one used for reading the previous generation's values and the other used for writing the next one. This class is included in all three version of GOL, and actually contains all the *functional code* needed by this program to performs the computation. The only two worth of noting functions are the *GOLMatrix::computeNextGenerationRow(i),* which actually performs the computation of next generation for the given row, and the *GOLMatrix::initializeMatrix()* used in the initlization phase. The code of this two functions is described below:

```
1 void GOLMatrix::computeNextGenerationRow(const int i){
2       const int cols = this->columns;
3       const int rowslen = this->rows;
4       unsigned char *previous = matrixRead + (((i-1+cols) % rowslen) * cols);
5       unsigned char *thisline = matrixRead + (i * cols);
6       unsigned char *nextline = matrixRead + (((i+1) % rowslen) * cols);
7       const int AlivesForZero =(int) previous[cols-1] + (int) previous[0] + (int) previous[1] +
8                                (int) thisline[cols-1] +                     (int) thisline[1] +
9                                (int) nextline[cols-1] + (int) nextline[0] + (int) nextline[1];
10      matrixWrite[(i*cols)] = computeValue(thisline[0], AlivesForZero);
11      const int AlivesForLast = (int) previous[0] + (int) previous[cols-1] + (int) previous[cols-2] +
12                                (int) thisline[cols-2] +                     (int) thisline[0] +
13                                (int) nextline[0] + (int) nextline[cols-1] + (int) nextline[cols-2];
14      matrixWrite[(i*cols)+cols-1] = computeValue(thisline[cols-1], AlivesForLast);
15      #pragma ivdep
16      for(auto j = 1; j < cols-1; j++){
17              const int cell = thisline[j];
18              const int neighboursAlive = (int) previous[j-1] + (int) previous[j] + (int) previous[j+1] +
19                                          (int) thisline[j-1] +                     (int) thisline[j+1] +
20                                          (int) nextline[j-1] + (int) nextline[j] + (int) nextline[j+1];
21              matrixWrite[(i * cols) + j] = (neighboursAlive == 2)*(cell == 1) + (neighboursAlive == 3);
22      }
23}
```

- GOLMatrix::computeNextGenetationRow() function in GOLMatrix.cpp -

```
1 void GOLMatrix::initializeMatrix(const int seedc, const int startRow,
                                   const int endRow){
2       unsigned short seed[3] = {seedc,0,seedc};
3       const int rowslen = this->rows;
4       const int cols = this->columns;
5       for(auto i = startRow; i <= endRow; i++){
6               #pragma ivdep
7               for(auto j = 0; j < cols; j++){
8                       double rand = nrand48(&seed[0]);
9                       matrixRead[(i * cols) + j] = ((int)ceil(rand)) % 2;
10              }
11      }
12}
```

- GOLMatrix::initializeMatrix function in GOLMatrix.cpp -

As we can see from the code snippets, the vectorization is applied to the two inner-most loop of the two functions: *computeNextGenerationRow(int i)* is actually called inside an external loop executed in a generic worker (see Algorithm 2), so we can conclude that this is the inner-most loop that can be parallelized; the second is clearly the inner-most loop of the initialization function and so we can apply here this optimization as well.

It is also worth noting that the **nrand48** function included in the <stdio.h> header is considered to be vectorizable by the Intel Developer community (interested reader can read this: https://software.intel.com/en-us/articles/random-number-function-vectorization), but the vectorization report, given by the compiler with the "-vec-report" flag enabled, extimate a low potential speedup in any case. On the other hand, the optimization of the inner-most loop of the "next generation computation" lead to important performance gain. It is woth noting that in a first version, I use to not compute the first and the last element outside the loop, while using the modulo operation to compute the right index of the array, in order to match the toroidal behaviour: I've experienced that any kind of mathematic operations (heavy or lightweight) may lead to performance degradation and limit the powerful of vectorization. I've decided to store the three rows needed to compute the next generation for the considered row in three different auxiliary pointers and then compute the first and the last element outside the vectorized loop. This optimization reach a good estimated-speedup in the vectorization report.

3.2 Thread version implementation & Barrier class (Thread)

Actually the thread version is implemented using the standard C++11 threads with a custom class named *Barrier*, which implements a spin barrier with only one method called *busyWait(std::function<void>)*. This method is implemented using the *std::atomic<T>* of C++11 and performing a busy waiting on an atomic variable. The *main* function of this version is only responsible to initialize the matrix, allocate and initialize the threads and wait till the termination of the set of workers. The swapping phase is demanded to the last thread which enters in the barrier, which will execute the functional given as input to the *busyWait* method.
The *pthread_setaffinity_np* is used to "stick" the thread to the cores, in order to avoid heavy memory operations due to OS's re-scheduling of the considered thread: in this way, each thread can resume this execution in the same core as before being scheduled, without any additional memory operations.

3.3 Custom Emitter (FastFlow)

In the section 2.2, the behaviour of the custom Emitter is described with pseudo-code snippet. The full source code of this custom struct is given below:

```
1 struct Emitter: ff_node_t<GOLTask> {
2       ff_loadbalancer *const lb;
3       int rows, columns, numIterations, grain, receivedResults, rest, division, nworkers, seed;
4       bool send;
5
6       Emitter(ff_loadbalancer *const lb, int rows, int columns, int iterations, int nworkers, int seed)
                        :lb(lb),rows(rows),columns(columns),numIterations(iterations), grain(grain),
                        nworkers(nworkers),seed(seed),receivedResults(0), send(true){
7               division = this->rows / this->nworkers;
8               matrix = new GameOfLife::GOLMatrix(this->rows, this->columns);
9               matrix->initializeMatrix(this->seed, 0, this->rows-1);
10      }
11      GOLTask *svc(GOLTask *partialResult) {
12              const int nw = lb->getNWorkers();
13              if(partialResult != nullptr){
14                      delete partialResult;
15                      receivedResults++;
16                      if(receivedResults == this->nworkers){
17                              matrix->swapMatrices();
18                              numIterations--;
19                              receivedResults = 0;
20                              this->send = true;
21                      }
22              }
23              if (send) {
24                      if(numIterations == 0)
25                              lb->broadcast_task(EOS);
26                      else {
27                              // Produce iteration
28                              int rest = this->rows % nw;
29                              for(int x = 0; x < nw; x++){
30                                      if(rest > 0){
31                                              GOLTask *n = new GOLTask(division*x, division*x + division);
32                                              rest--;
33                                              ff_send_out( (void*)n);
34                                      } else {
35
36                                              GOLTask *n = new GOLTask(division*x, division*x + division - 1);
37                                              ff_send_out( (void*)n);
38                                      }
39                              }
40                              this->send = false;
41                              return GO_ON;
41                      }
43              }
44              return GO_ON;
45      }
46};
```

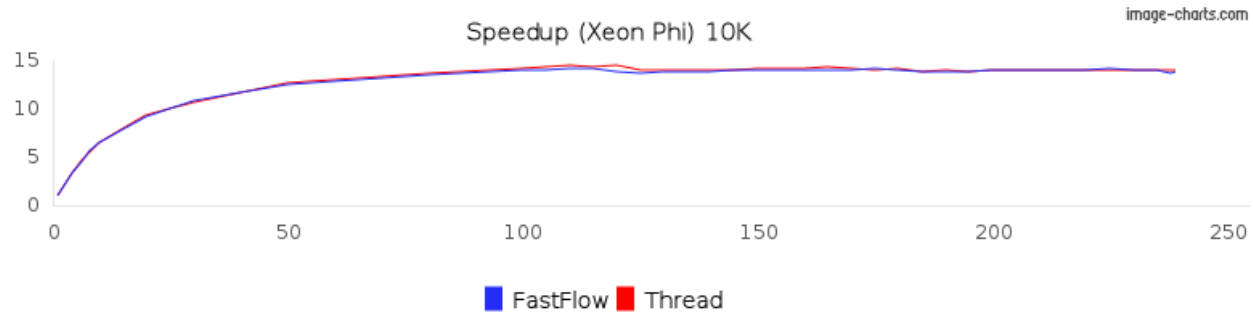
As described in section 2.2, the emitter sends new tasks first time it spawn (the variable *send* is initialized to true when the farm is executed for the first time) and whenever all the tasks comes back from the workers. At this point, the swapping phase is performed and the next iterations will be started. The initialization phase is done in the constructor of the Emitter: also in this case it is worth noting that it can be parallelized as well sending a "fake" task of initialization to the workers.

4. Correctness

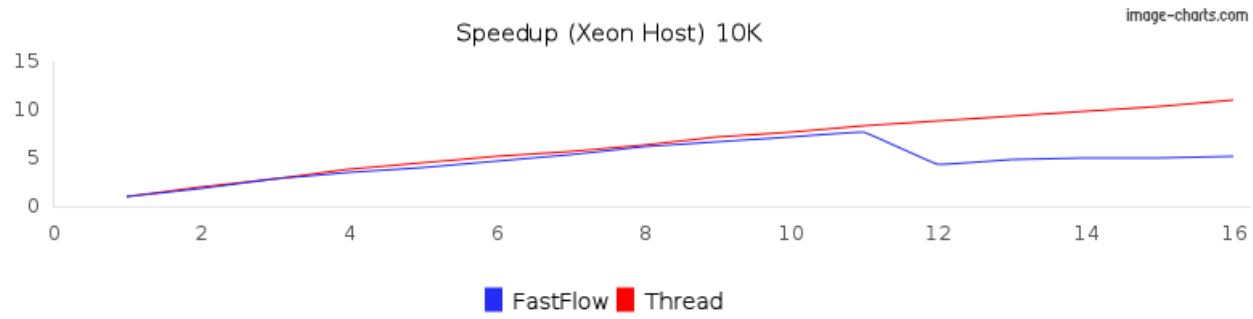
Correctness can be tested using the given flag, available in the sequential version only for easy-to-show reason (see last chapter for the user manual), which actually causes the program to execute some known patterns of GOL configuration hand-coded. Three different tests are implemented: the "Blinker" with period 2 (a simple line of contiguous lives cell with each two iterations comes back to the initial condition); the "Glider" spaceships which traslate across the board indefinitely; and an example of "unbounded growth" which is really valid in a theoretical infinite table, but let us to notice some interesting behaviour of this cellular automaton.
The consistency between the various output of the sequential and parallel version, are checked by computing the MD5 hash function of the final matrix and comparing it with the others in order to discover inconsistency. This will be checked also when the set of results tests are parsed.

5. Performance evaluation

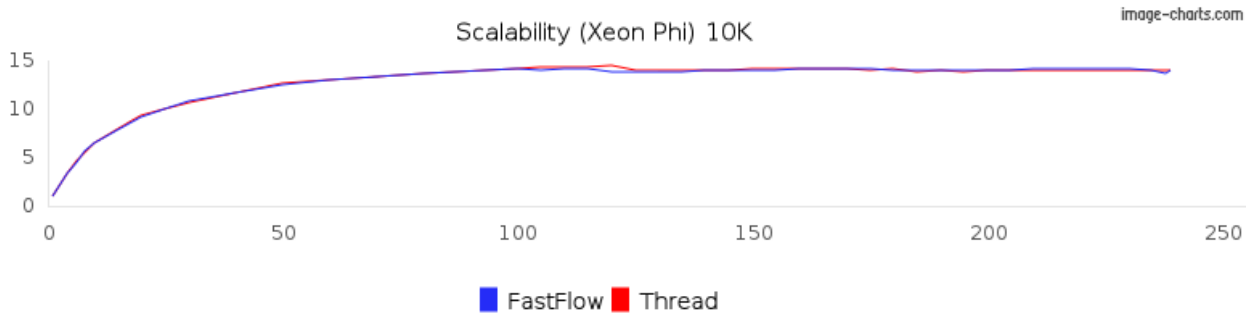
In this chapter will be presented the tests executed on the two machines indicated in chapter 1 for what concerns Speedup, Scalability and Efficiency with two different input: the first test use a matrix of 5000 rows and 5000 columns with number of iterations equal to 1000; the second test use a matrix of 10000 rows and 10000 columns with number of iterations equal to 250. The charts of the *completion time* are missing here for space reasons: they can be found in the "*charts"* directory of the project.



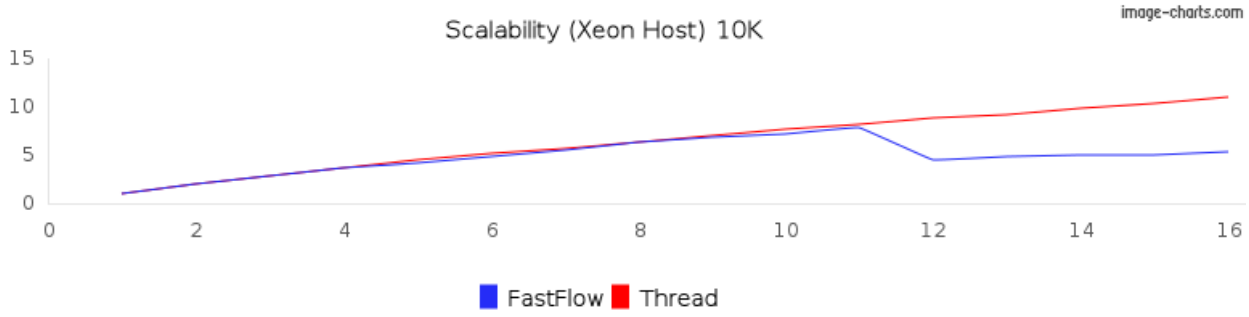
- Chart 1: Speedup plot with M=10000 N=10000 K=250 on Xeon Phi-



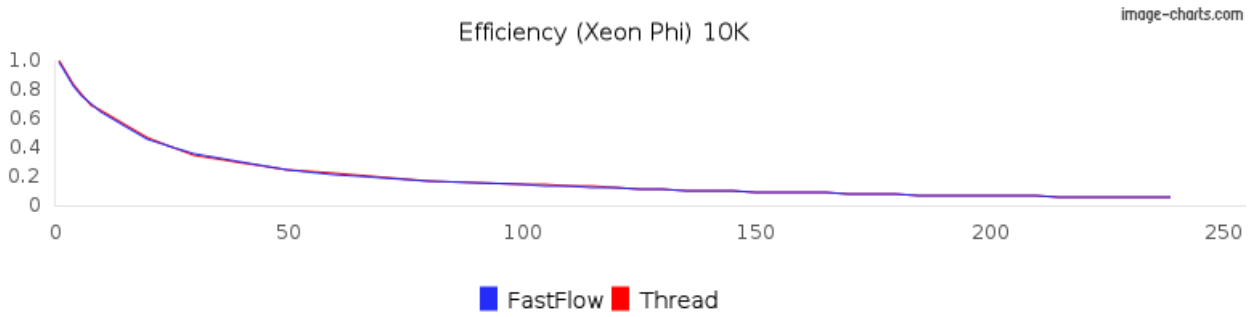
- Chart 2: Speedup plot with M=10000 N=10000 K=250 on Xeon Host-



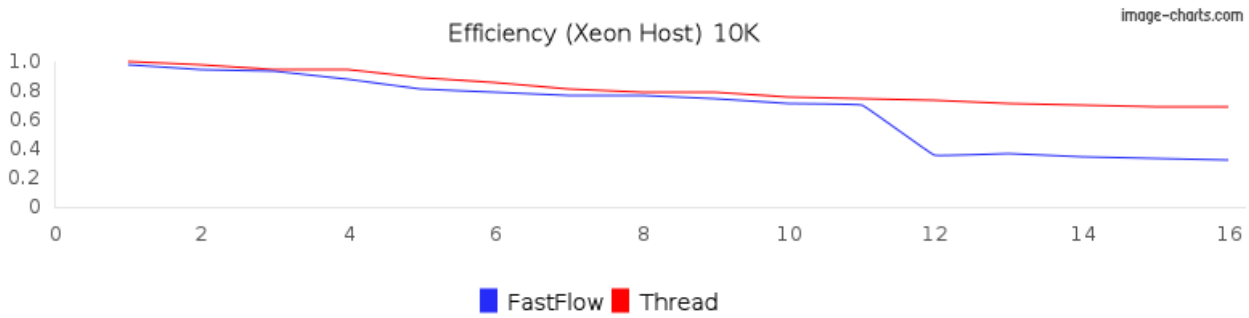
- Chart 3: Scalability plot with M=10000 N=10000 K=250 on Xeon Phi-



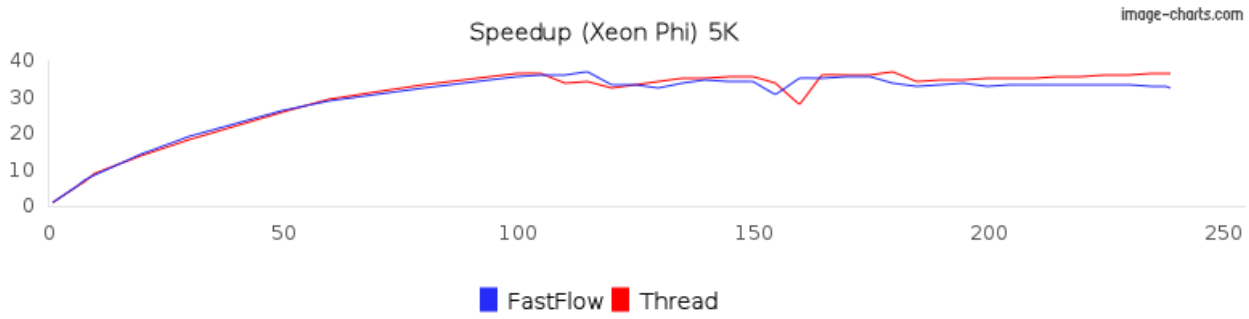
- Chart 4: Scalability plot with M=10000 N=10000 K=250 on Xeon Host-



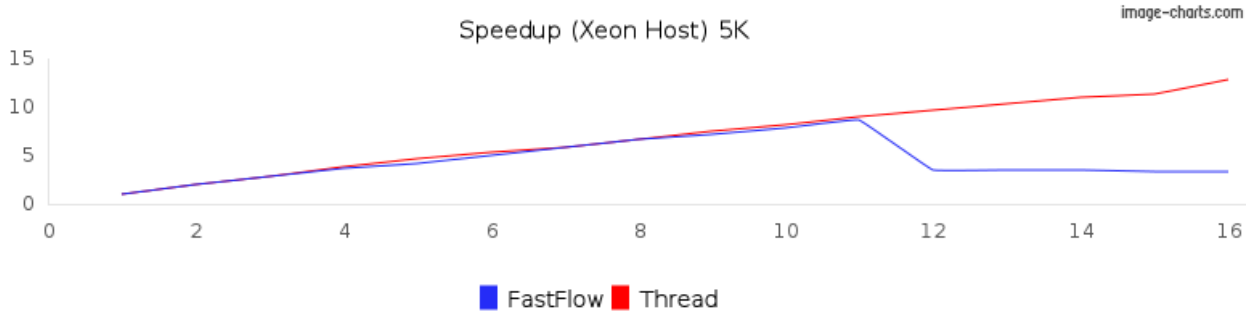
- Chart 5: Efficiency plot with M=10000 N=10000 K=250 on Xeon Phi-



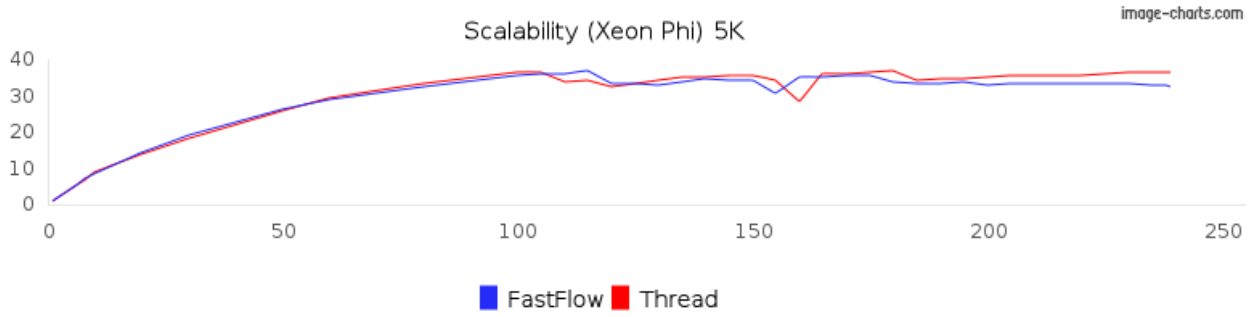
- Chart 6: Efficiency plot with M=10000 N=10000 K=250 on Xeon Host-



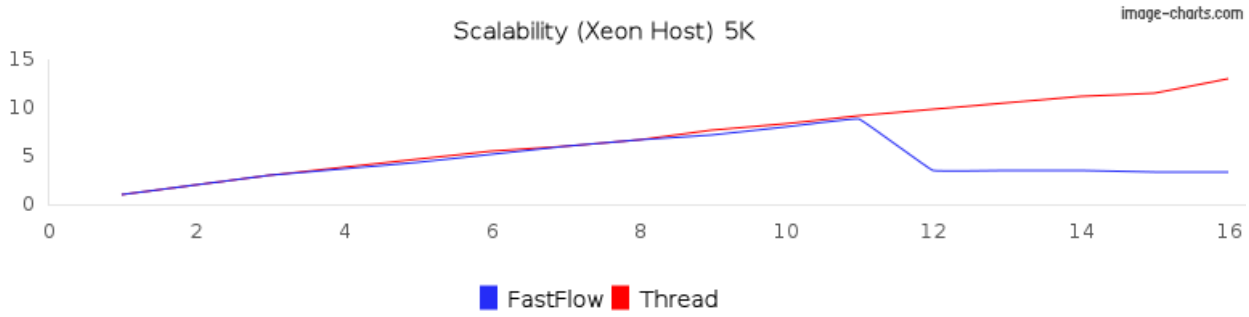
- Chart 7: Speedup plot with M=5000 N=5000 K=1000 on Xeon Phi-



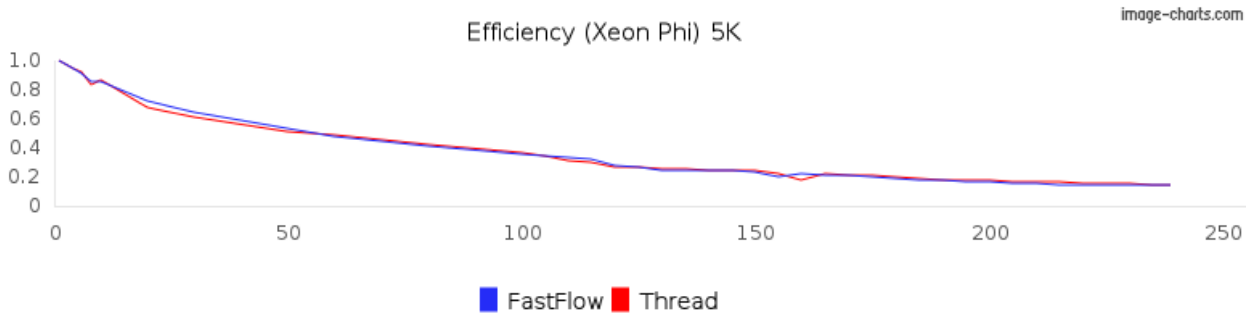
-Chart 8: Speedup plot with M=5000 N=5000 K=1000 on Xeon Host-



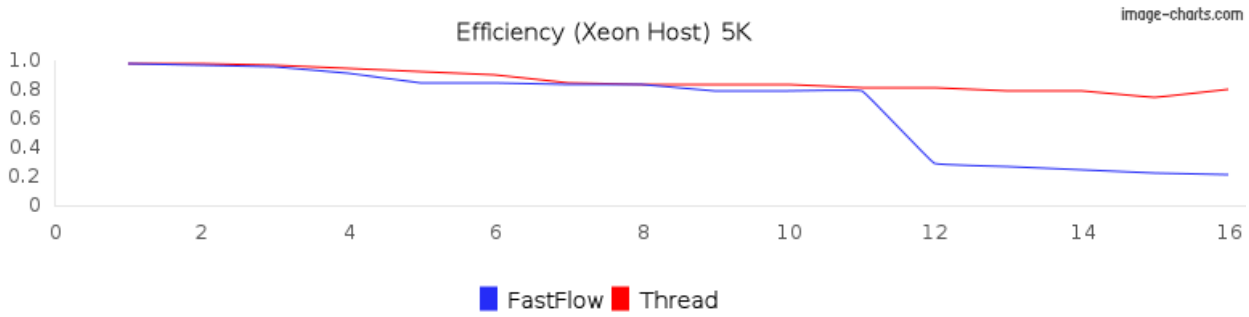
- Chart 9: Scalability plot with M=5000 N=5000 K=1000 on Xeon Phi-



- Chart 10: Scalability plot with M=5000 N=5000 K=1000 on Xeon Host-



- Chart 11: Efficiency plot with M=5000 N=5000 K=1000 on Xeon Phi-



- Chart 12: Efficiency plot with M=5000 N=5000 K=1000 on Xeon Host-

As we can view from the charts, in both of the tests #1 and #2 we have the very same performance's values on the Xeon Phi MIC, while in the Host machine the FastFlow's implementation mantains comparable performance with respect to Thread version till *nw*=11. With parallel degree greater than this value, the Thread version on the Xeon Host seems to be better than FF's implementation in terms of speedup and scalability (Chart 2/4-8/10).
In both of tests for what concerns the executions on the Xeon Phi, we can observe that even if the performance are slightly equal, the FastFlow's implementation uses a lower number of allocated resources when it match the top performance's values (Table 2/4 next section).
In test #2 for sure the initialization phase (measured mean initialization time is about 6 secs) introduce some degrade on performance values in both versions since it is not parallelized: the use of vectorization with the **nrand48** function seems to be an useless optimization if we consider the effective performance gain (expecially on the Xeon Phi where it is not vectorized at all). This naturally leads the speedup to remain at low values (no more than 14 on Xeon Phi): with the parallelization of the initialization phase we can obtain for sure better performances.
Also the presence of the barrier in the Thread's version leads to performance degradation: I've measured that this impact is naturally dominated by the initialization phase in test #2 (where the input is quite big), while it participate with a great value in the performance degradation of the first version, expecially when the number of active threads grows.
Another thing that is worth to point out, is that the vectorization of the update-row-phase leads to an important improvement in both versions expecially on the Xeon Phi, which exploit the vectorization in a more powerful way compared to the host machine (as we can see in the vectorization report, compiling the program with the command *make perf*).

5.1 Top values

Here are the top values collected from the two test cases, listed for better readability (values are in seconds):

Version	$T_C(1)$	n_{wopt}	$sp(n_{wopt})$	$scalab(n_{wopt})$	$T_C(n_{wopt})$	$\varepsilon(n_{wopt})$
Seq	56,06	-	-	-	-	-
Thread	56,71	16	12,35	12,49	4,54	0,8
FastFlow	57,33	11	8,67	8,85	6,48	0,79

- Table 1: top values for Test #1 on Xeon Host -

Version	$T_C(1)$	n_{wopt}	$sp(n_{wopt})$	$scalab(n_{wopt})$	$T_C(n_{wopt})$	$\varepsilon(n_{wopt})$
Seq	167,54	-	-	-	-	-
Thread	167,63	180	36,75	36,81	4,56	0,2
FastFlow	167,60	115	36,83	36,84	4,55	0,34

- Table 2: top values for Test #1 on Xeon Phi -

Version	$T_C(1)$	n_{wopt}	$sp(n_{wopt})$	$scalab(n_{wopt})$	$T_C(n_{wopt})$	$\varepsilon(n_{wopt})$
Seq	55,76	-	-	-	-	-
Thread	56,64	16	10,60	10,77	5,26	0,7
FastFlow	56,48	11	7,75	7,88	7,30	0,7

- Table 3: top values for Test #2 on Xeon Host -

Version	$T_C(1)$	n_{wopt}	$sp(n_{wopt})$	$scalab(n_{wopt})$	$T_C(n_{wopt})$	
Seq	166,54	-	-	-	-	
Thread	166,20	120	14,43	14,43	11,54	0,12
FastFlow	167,52	115	14,15	14,23	11,77	0,12

- Table 4: top values for Test #2 on Xeon Phi -

6. User manual

6.1 Compilation

The user can find a **Makefile** in the root directory, which contains all istructions to compile the project:

- **make golhost**: compilation for Xeon Host machine;
- **make golmic**: compilation for Xeon Phi machine;
- **make debug**: compilation without any optimization creating binary useful for debug ops;
- **make perf:** compilation with –pg flag (GNU gprof) and –vec-report5 (vectorization report);

6.2 Usage

For the sequential version the Makefile produce the "**SequentialGOL[.mic].out**" binary file which have the following possible parameters:

- **"-r"**: indicating the number of rows;
- **"-c"**: indicating the number of columns;
- **"-i"**: indicating the number of iterations;
- **"-s"**: indicating the seed for random generation of values in the initialization phase;
- **"-t"**: indicating the test code (possible values: 1, 2, 3);
- **"-p"**: to print the matrices at each iteration (not so readable with big tables).

The other two executable for the parallel versions have the same parameters of the sequential one, except the possibility to execute test and print matrix and with the additional parameter "-n" indicating the number of workers to be instantiated.

6.3 Folders

In the root directory the user can find the **Makefile** with 3 other folders:

- **"src"** directory contains all the source code for the three GOL versions, including the GOLMatrix and Barrier class described in chapter 3;
- **"tests"** directory contains some script used to automatize the test phase in the two target machine;
- **"charts"** directory which contains all the output of the tests done with the various charts generated with the computed average values.