Università degli studi di Pisa
Scuola Superiore Sant'Anna

DEPARTMENT OF COMPUTER SCIENCE
DEPARTMENT OF INFORMATION ENGINEERING
AND SCUOLA SUPERIORE SANT'ANNA

Master in Computer Science and Networking

Academic Year 2015/2016

SPM final project report

Conway's Game of Life

Federico Conte
(451530)

1 Project Description

The application is described here [1] in details.
Game of Life (abbrev. **GOL**) is an example of a **cellular automaton** which is a way to abstract complex systems. GOL is a simulation of the behaviour of a community of biological entities, in particular, the alterations of a society of living organisms that can rise or fall under certain conditions. For example, it can abstract the behaviour of cells on a microscope slide. Taking the last example as guide, in the following, we will refer to a general biological entity as **cell**, and the place where it is collocated as **box**.

The simulation evolves through a sequence of discrete events, called **generations**, and is characterized by the following features:

- the space, where cells live, is modelled as a **2D toroidal grid** of boxes and a cell can occupy only one box;
- a cell behaviour is affected by its state and the states of its neighbours (see Figure 1; so the update is a **9-point stencil** operation) according to certain rules (Algorithm 2) ;
- all cells are affected simultaneously in a generation;
- initially each box contains a cell with probability of **50%**;

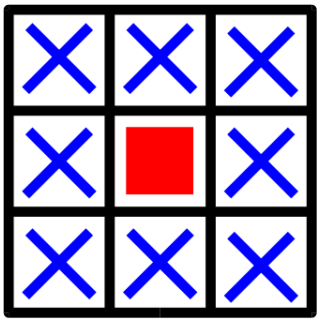


Figure 1: *The blue cross are the neighbours of the red square cell.*

The rules for cells evolution are descried by Algorithm 2. We can notice that these kind of rules can be reprocessed in order to be expressed by just one assignment statement, as it is shown in Algorithm 3, where the targeting box will contains or not a cell depending on the condition value. Algorithm 1 shows the pseudo-code of Game of Life.

Algorithm 1 Game of Life

INPUT: #Iterations, #Rows, #Columns

```
1: Matrix initilialization
2: for k = 1 to #Iterations do
3:     for i = 1 to #Rows do
4:         for j = 1 to #Columns do
5:             #Neighbours ← CalculateNeighbours( i, j )
6:              Box[ i, j ] ← Compute Rules ( See Algorithm 2 or 3 )
7:         end for
8:     end for
9: end for
10: Display the result
```

Algorithm 2 GOL Rules

```
1: if Cell is alive then
2:     if #Neighbours < 2 then    Cell dies due to under-population
3:     else if #Neighbours > 3 then    Cell dies for starvation
4:     else    Cell stays alive
5:     end if
6: else
7:     if #Neighbours == 3 then    New cell born by reproduction of one of the neighbours
8:     else    Box remains empty
9:     end if
10: end if
```

Algorithm 3 Reprocess GOL Rules

```
1: Box ← (( #Neighbours == 3 ) OR ( Cell is alive AND #Neighbours == 2 ))
```

The project have been developed in **C++** starting from a simple sequential solution. Then, we have tried to optimize it trying different paths, as it explained in next section, and choosing the one considered most appropriate for further works.
At this point, we have built two different parallel application: one using **low level threading mechanisms** and one using **FastFlow framework**; and we have compared the two solutions. Then, we have added the support for vectorization and we have tried to see if this optimization brings to performance improvements both in the sequential and parallel implementations.

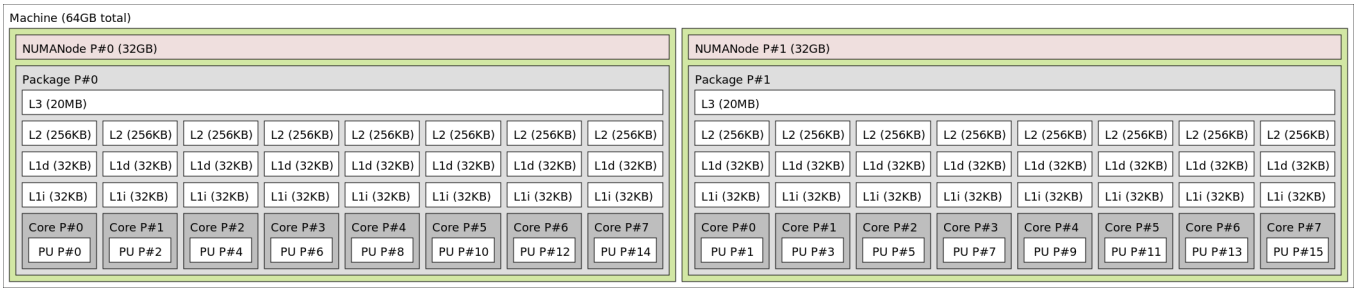


Figure 2: *Topology of our Xeon Host machine obtained with the command hwloc-ls.*

The platform used to test and evaluate the project has the following characteristics:

Hardware: x86 64 architecture with Intel Xeon CPU E5-2650 @ 2.0GHz, 2 NUMA nodes with 8 cores each, 20MB L3 cache and 64GB of RAM equipped with Intel Xeon Phi™ coprocessor (see Figure 2)

Operating System: Red Hat Enterprise Linux Server release 6.5

Compiler: Intel C++ Compiler, Version 15.0.2

2 Sequential Version

In Algorithm 1, nothing is said about where to store the result of the new matrix configuration. Regarding our knowledge, there are two solution:

(a) use two different matrix, one for reading and one for writing and swap them after each iteration;

(b) exploit the technique described in [2], where the new matrix is stored upon the original one but shifted upwards and sideways of one box.

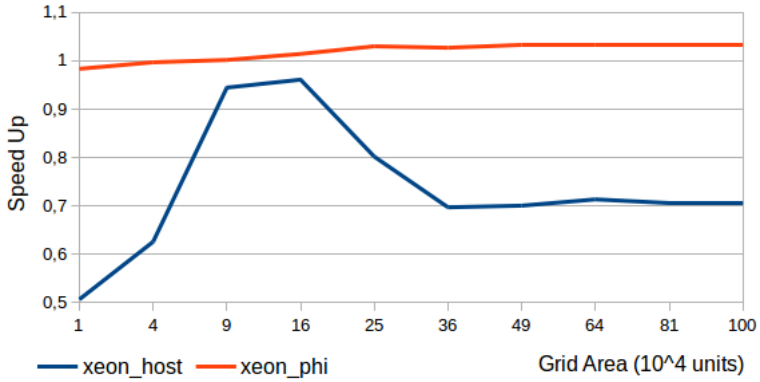


Figure 3: *Acceleration obtained using solution (b) instead of solution (a), on Xeon Host (blue) and Xeon Phi™ (red), where* $\#Iterations = 100$, $\#Rows = \#Columns = \sqrt{GridArea}$.

Figure 3 shows that we have a little gain on Xeon Phi™ using solution (b) but this behaves worse than solution (a) on Xeon Host. This is probably due to the fact that, in order to use just one matrix, the code contains more conditional and arithmetic statements.
Besides **solution (a)** presents a much simpler implementation, so we go on with this one.

Solution (a) is implemented **adding a border of 1-box thickness** to the grid (obtained increasing the number of rows and columns by two) whose values follow the logic of the 2D toroidal shape of the grid. In this way during the updating phase, we do not need to check if a box is on the original edges in order to perform the correct reference to the neighbours; in fact, **those neighbours have been copied properly on the added border**. So, during the update operations, the **neighbours offset is the same for all the boxes**.
Besides, we have experimentally proved (although we do not show the results here for brevity) that updating the left and right border of the grid, during the computation of a generation, is faster than trying to avoid these calculations through checks.

Then, we attempt to see which is the most efficient data structure for storing the grid. As you can see from Figure 4, it turns out that the **boolean array** is the simplest and most efficient one. The reason is probably due to the fact that:

- **vector<bool>** requires too much time to index its elements, since allocates them in raw format without any meta information;
- **bitset array** requires further checks to manage its data structure organized in blocks;

- **boost::dynamic_bitset** substructure should be similar to the bitset array, so probably suffers of the same problem.

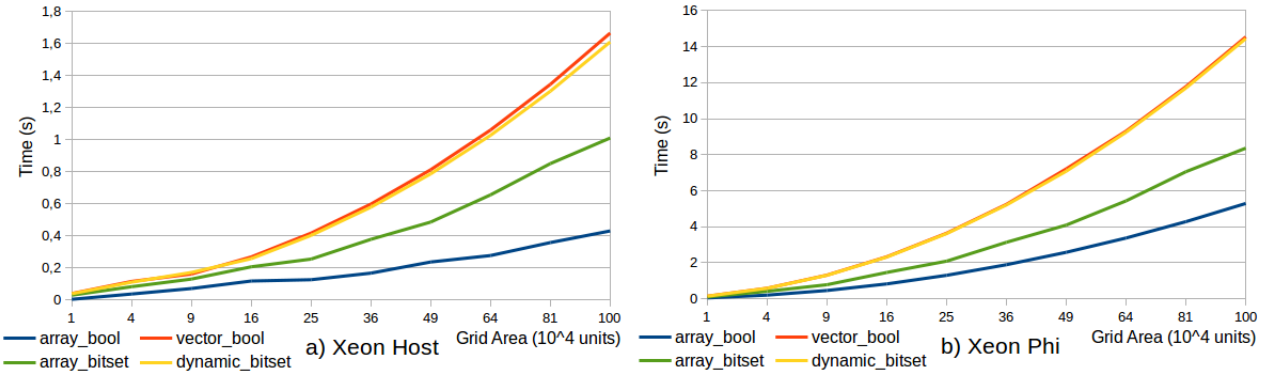


Figure 4: *Performance comparison between the four different data structure described above, respectively on Xeon Host (a) and Xeon Phi™ (b), where* $\#Iterations = 100$, $\#Rows = \#Columns = \sqrt{GridArea}$.

We could have tried other optimizations, as suggested by [2], but we wanted to focus more on the parallelization phase.

In conclusion for this section, we have chosen to go on with the solution of **two matrixes** both stored on simple **boolean array**. Algorithm 4 shows the pseudocode revised taking into account what said until now.

Algorithm 4 Game of Life

INPUT: #Iterations, #Rows, #Columns

1: $rows \leftarrow$ #Rows + 2; $cols \leftarrow$ #Columns + 2; ▷*Adding a border of 1-box thickness*
2: $size \leftarrow rows \times cols$
3: $G \leftarrow$ Grid initilialization; ▷*Allocates boolean arrays G.W , G.R and fill G.R randomly*
4: G.CopyBorder() ▷*Copy neighbours on the added border*
5: **for** $k = 1$ to #Iterations **do**
6: **for** $pos = cols$ to $size - cols$ **do** ▷*Excludes the first and last row*
7: $num \leftarrow$ G.CalculateNeighbours(pos) ▷*Using the reading boolean array G.R*
8: $G.W[\ pos\ ] \leftarrow ((\ num == 3\ )$ OR $(\ G.R[\ pos\ ]$ AND $num == 2\ ))$
9: **end for**
10: G.Swap( ) ▷*Swapping the reading (G.R) and writing (G.W) boolean arrays*
11: G.CopyBorder( )
12: **end for**
13: G.Print( ) ▷*Display the result*

2.1 Adding Vectorization *("vect")*

In order to increase performance, we have tried to extends our code using the **array notation** whenever was possible. In particular, we have followed the suggestion reported in [3]: "use short-vector coding if you have data reuse between statements and N is big", where N is the number of cycle iterations. So, for the array notation we have used a vector coding of **length 16** on Xeon Host and **length 32** on Xeon Phi™, since the vector processing unit (VPU) in Xeon Host and Xeon Phi™ works respectively at **256 bits** and **512 bits** parallelism at a time.

We have used an integer array, of size **16** on Xeon Host and **32** on Xeon Phi™, to save the computation of the neighbours counting (line 7 of Algorithm 4) and express the updating assignment in array notation. Besides, we have used the **elemental function** notation to vectorize the neighbours counting function.

We have not succeeded to align data, despite we have declared all the data with **64 bit alignment**. Probably these is due to the nature of the computation: when accessing the neighbours of a box we cannot ensure alignment access. So in the code, we remove all the alignments.

Besides, we have used the **drand48** family of random number functions (as it is suggested in [4]), in order to speed up the initialization phase where the grid is filled randomly with zeros and ones.

3 Parallel Version

Clearly the innermost loop calculating a new generation, line 6 of Algorithm 4, is the most intensive part of the application. Furthermore, we cannot parallelize the outermost loop (line 5) since each generation depends from the previous one.

Since the ***G.CalculateNeighbours()*** function (line 7) has to read the neighbours values (see 1), the innermost loop can be parallelize using a **9-point stencil** pattern.
For simplicity, we will not try to parallelize the **initialization phase** (line 3) as well as ***G.CopyBorder()*** function (line 11). We are choosing the following parallel implementation. Let us view the reading and writing boolean arrays again as matrixes. The grid is row-wise divided into ***nw*** chunks, where nw is the number of Workers (actually we do not follow this rigid division, but the number of boxes are equally divided to Workers). A **barrier** is used in order to synchronize all Workers at the end of the computation of a generation. At this point, we can execute in sequential lines 10 and 11. Then, the Workers can restart computing a new generation. During the calculation of a generation, the first and last line of the reading matrix chunk, owned by a Worker, are **shared in read-only mode** between neighbour Workers; so we do not need looking mechanisms: synchronization of the worker while accessing the shared area is left to the system. Instead, the writing matrix chunk, owned by a Worker, is not shared, i.e. it is accessed by only that Worker and so we do not have consistency problems.

In the following, we will call **"CopyBorder"** to refer both instructions 10 and 11 ($T_{CopyBorder}$), $T_{Barrier}$ the time to synchronize the Workers, $T_{InitGrid}$ the time to initialize the Grid, $T_{InitThreads}$ the time to start the threads and $T_{CalcNeighbours}$ the time to compute the innermost loop.

Equation 1 shows the **cost model to calculate the completion time of GOL**, where N is the number of boxes (previously called *"GridArea"*).

$$T_c \sim T_{InitGrid} + T_{InitThreads} + \#Iterations \cdot (\frac{N}{nw} \cdot T_{CalcNeighbours} + T_{Barrier} + T_{CopyBorder}) \quad (1)$$

a) Xeon Host	Normal version		Vectorized version	
	25M	**100M**	**25M**	**100M**
Serial Time (ms)	314	1490	20	126
Sequential Time (ms)	17056	63484	5322	21782
Serial Fraction	1.84%	2.35%	0.38%	0.58%
Maximum Speedup	**54**	**42**	**266**	**172**

b) Xeon Phi™	Normal version		Vectorized version	
	25M	**100M**	**25M**	**100M**
Serial Time (ms)	2468	9457	294	764
Sequential Time (ms)	132059	532323	17754	71953
Serial Fraction	1.87%	1.78%	1.66%	1.06%
Maximum Speedup	**53**	**56**	**60**	**94**

Table 1: *Showing the maximum speedup that we can achieve, calculated using the* ***Amdahl's law****, with different input data on Xeon Host (a) and Xeon Phi™ (b).*

In the following experiments we are going to use **two** different values for N : **25M** and **100M**, where M stands for millions. Table 1 shows the maximum speedup that we can achieve, where:

Sequential Time = $T_{InitGrid} + \#Iterations \cdot N \cdot (T_{CalcNeighbours} + T_{CopyBorder})$

Serial Fraction = $\frac{(T_{InitGrid} + \#Iter \cdot T_{CopyBorder})}{SequentialTime}$ % **Maximum Speedup** = $\frac{1}{SerialFraction}$

We can notice that the **maximum speedup is not so high**, about **50** but some exceptions. This will limit us, especially on Xeon Phi™. This is due to the fact that the initialization phase is not a negligible time, so in a future work we should try to parallelize it as well.

3.1 Version 1.0

3.1.1 Low Level Threading Mechanisms *("thread")*

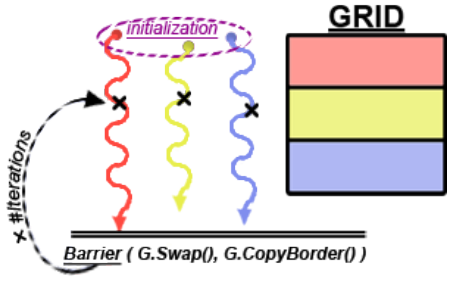


Figure 5: *Shows the scheme of the parallel application behavior.*

The schema is shown in Figure 5. When the **thread pool** is initialized, each thread receives the grid portion where has to work with. When it completes a generation on its chunk, if it is the last one it executes the *"Serial Phase"* and **notify all** the waiting threads the barrier ending, otherwise it **waits** on a **condition variable** until the barrier has not been reached by all threads. We repeat this schema for the number of iterations.

3.1.2 FastFlow framework *("ff")*

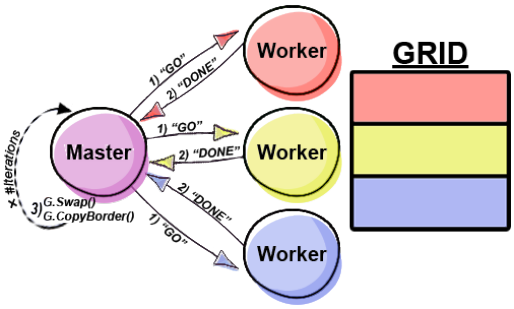
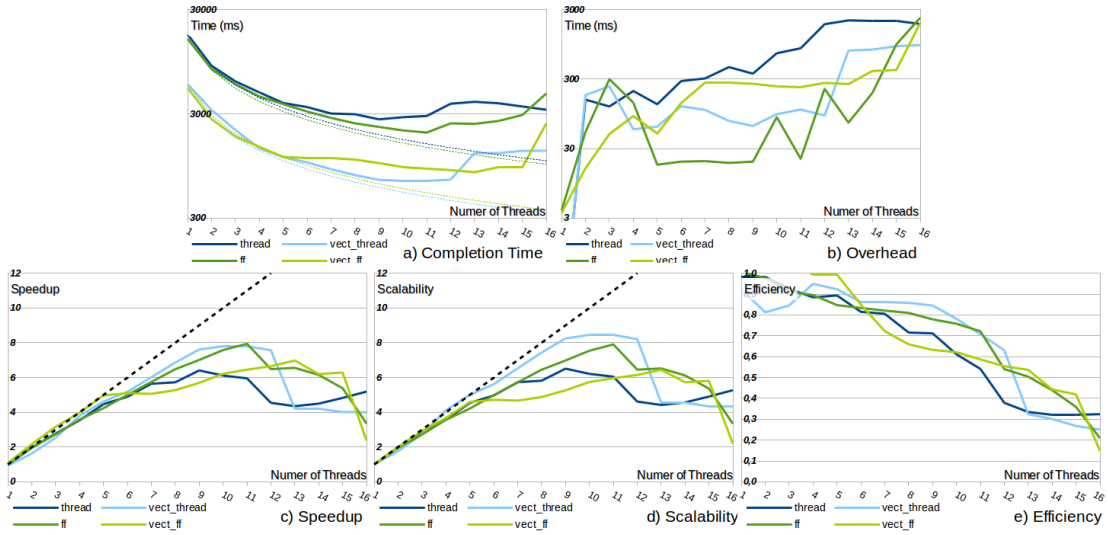


Figure 6: *Shows the scheme of the parallel application behavior.*

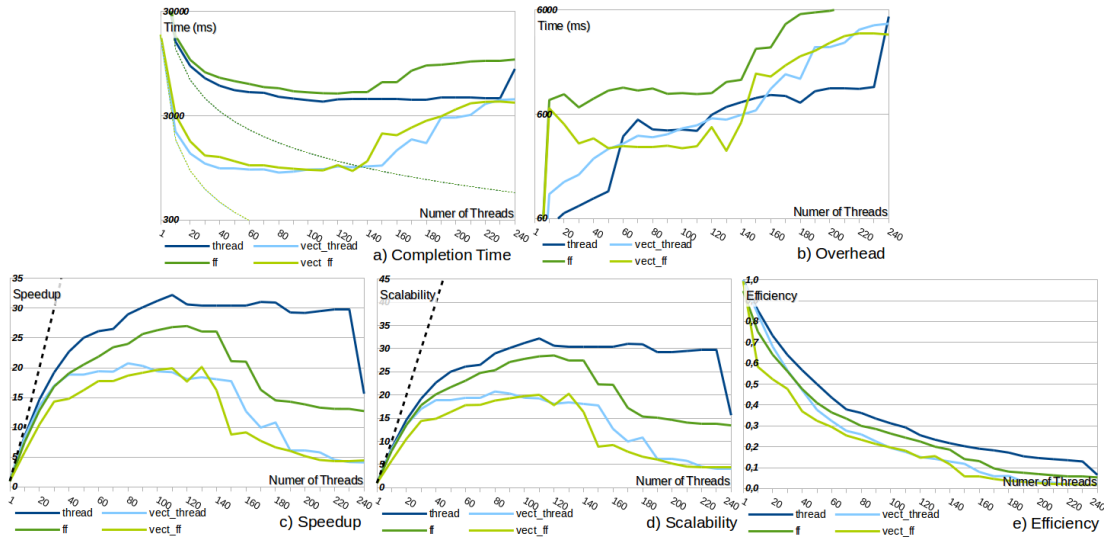
The schema is shown in Figure 6. When each Worker is initialized, it receives the grid portion where has to work with. The Master sends a *"GO"* message to all the Workers. When it completes a generation on its chunk, it sends a *"DONE"* message to the Master. When the Master receives *"DONE"* from all the Workers, it executes lines 10 and 11. We repeat this schema for the number of iterations.

3.1.3 Performance of Version 1.0

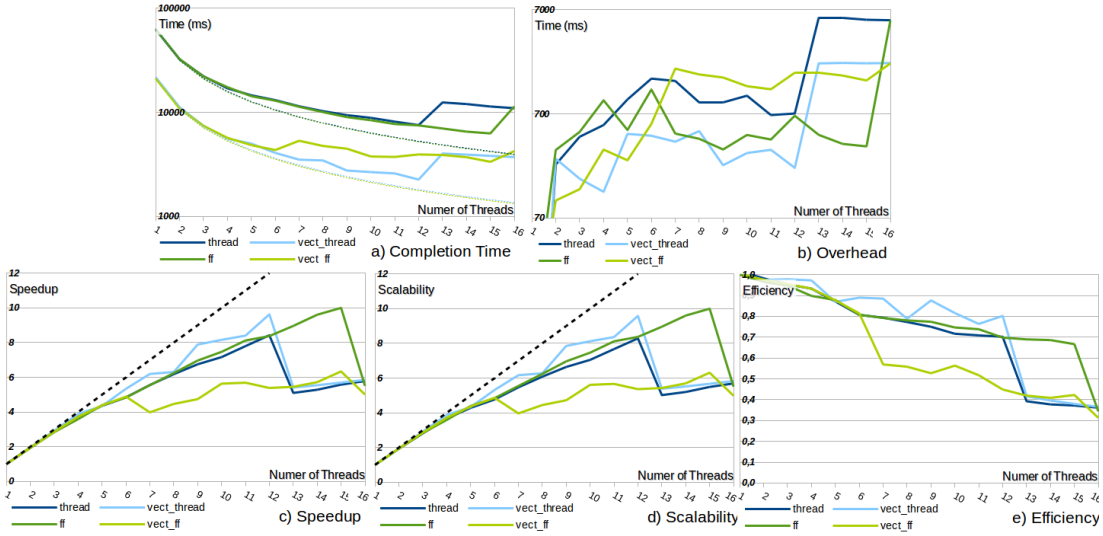
N = 25M on Xeon Host



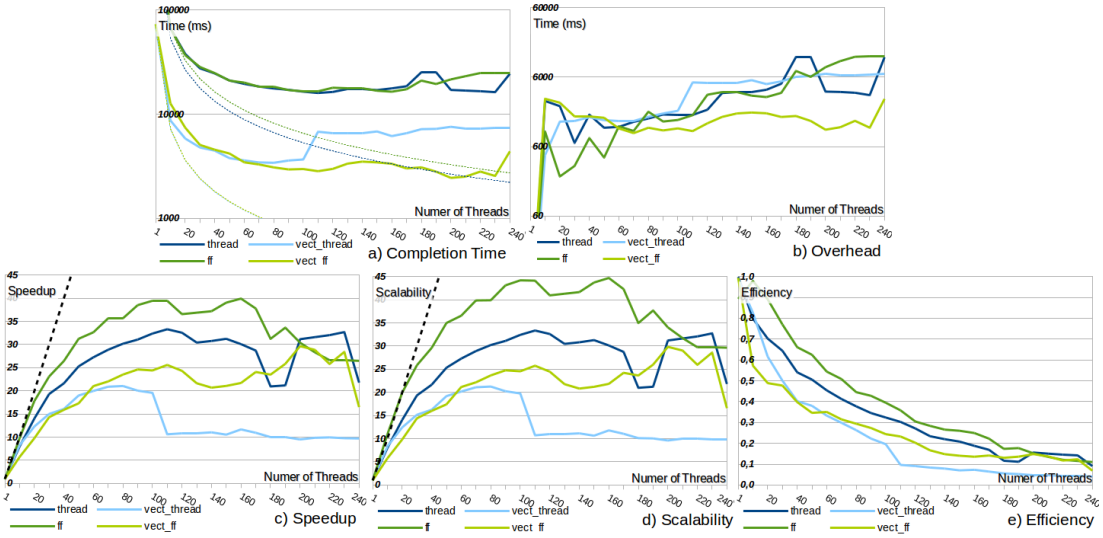
N = 25M on Xeon Phi™



N = 100M on Xeon Host



N = 100M on Xeon Phi™



We can see that two development methodology: *"thread"* and *"ff"*, **are comparable for both the normal and vectorized versions**. We have a maximum speedup between **8** and **10** on Xeon Host and between **30** and **40** on Xeon Phi™.

3.2 Version 2.0

When we saw the first results, we noticed that the **"Barrier Phase"**, i.e. the overall time needed to synchronize all the Workers, was taking almost **55%** on Xeon Host and **75%** on Xeon Phi™ of the completion time, and **10%** more adding the vectorized version. First we have tried to improve the barrier mechanism, of the "thread" methodology, using **atomic** variables and implementing the waiting through a **busy-looping**.
Since the situation did not improve much, we realized that was a problem of **load unbalancing**: there was a difference of some milliseconds (depending on the methodology) between the first and last Worker that were reaching the barrier, and multiplying this value for the number

of iteration generates a non acceptable overhead.

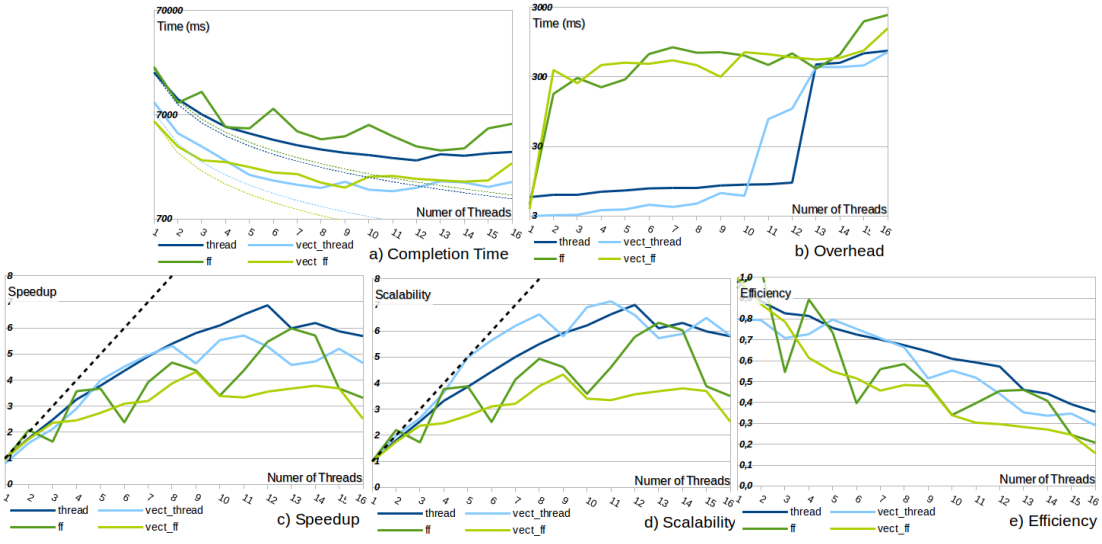
For the **second version** of the application, we decide to introduce the possibility of choosing the **grain size** of the chunk assigned to each Worker with the goal of reducing the load unbalancing problem. Unfortunately, we did not have time to properly explore this solution. The first results obtained are not good as expected. We were able to **reduce significantly the overhead only in "thread" and "vect + thread" methodologies on Xeon Host**, but in any case we obtain worse results than version 1.0 (maybe due to an overloaded of the machine). In fact, after some attempts, we noticed that Xeon Phi™ does not benefit to a dynamic scheduling (grain size set to **zero** , so we will not reports the graphs since they are similar to version 1.0) and on Xeon Host a good grain seems **9000**.
Below we describe the idea of the dynamic scheduling implementation.

"thread": All threads loops until the "Master" (represented by the thread executing the *main()*) do not terminate them, though an **atomic shared variable**. All threads are waiting for jobs. When the Master assigns a task to the first available thread, this thread executes it and than signals that is free again, though another **atomic variable** shared only with the Master.

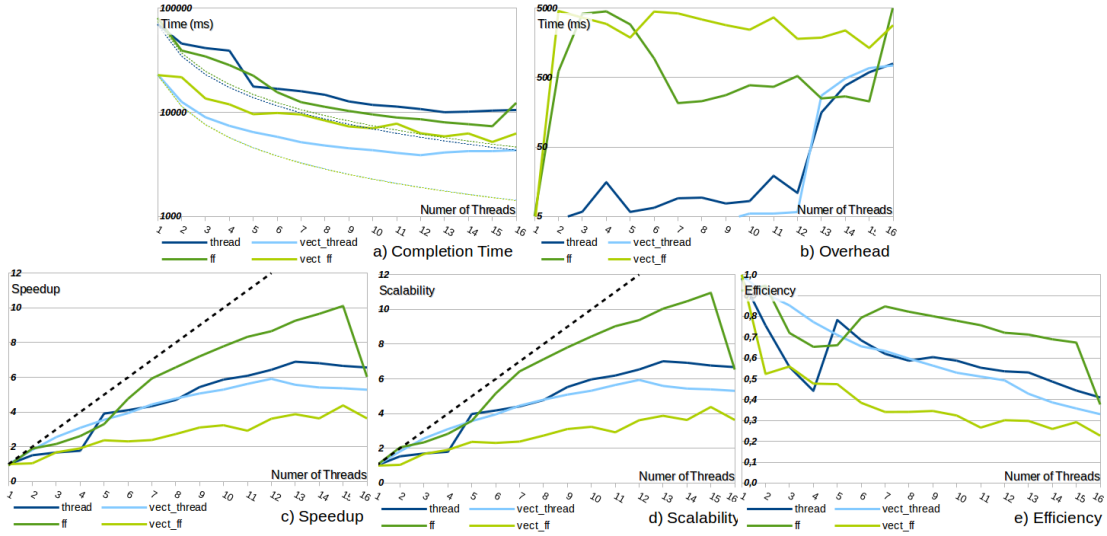
"ff": Now, instead of a "GO" message, we are passing to a Worker the starting and ending indexes of the grid working area in which the Worker has to compute the generation.

3.2.1 Performance of Version 2.0

N = 25M on Xeon Host



N = 100M on Xeon Host



a) Version 1.0	25M				100M			
	Xeon Host		Xeon Phi™		Xeon Host		Xeon Phi™	
Methodology	Time	NW	Time	NW	Time	NW	Time	NW
"thread"	2662	9	4230	100	7539	12	15985	110
"thread" + "vect"	681	10	856	80	2263	12	3419	80
"ff"	1987	11	4902	120	6272	15	16430	160
"ff" + "vect"	826	13	887	130	3351	15	2449	200

b) Version 2.0	25M				100M			
	Xeon Host		Xeon Phi™		Xeon Host		Xeon Phi™	
Methodology	Time	NW	Time	NW	Time	NW	Time	NW
"thread"	2554	12	6475	110	10028	13	24855	230
"thread" + "vect"	1296	11	3069	40	3878	12	10700	110
"ff"	3183	13	1263	110	7349	15	23599	230
"ff" + "vect"	1402	9	2768	60	5197	15	10366	170

Table 2: *Showing the best completion time (in milliseconds) obtained for the different parallel methodologies for Version 1.0 (a) and Version 2.0 (b) on Xeon Host and Xeon Phi™.*

4 Workspace

The application can be found here [5].
In the following we will refer to **this workspace**, that is organized as following:

"attempts" folder: contains the source files of the attempts made in order to choose the starting point of our sequential implementation.

"include" and "src" folders: contains respectively the header and source files of our implementation.

"performance_comparison" folder:
containing the following files:

avg_time.sh: given a sequence of values, removes the smallest and biggest and returns the average of remaining values.

compile_app.sh: compile the application both on Xeon Host and Xeon Phi™.

test_app.sh: script testing a specific development methodology used in the application(normal, vectorized, using low level threading mechanisms, using FastFlow framework), grid size and parallelism degree on both Xeon Host and Xeon Phi™.

test_app_multiple.sh: use *compile_app.sh* to test the applications for all the different development methodologies.

test_attempts.sh: script that compares the attempts among each other.

test_best_grain.sh & test_best_grain_multiple.sh: two scripts that helped us to individuate a good grain for version 2.0 of the application.

results_grain: folder containing the results for the searching of a good grain performed with the scripts described above.

A text file and the respective spreadsheet for each experiment made, showing the result in table and graph format. Where the results for **version 1.0** and **2.0** are respectively in folders **"results_version1"** and **"results_version2"**.

5 Test Methodology

For performance verification, we have used the **chrono** library with which we calculated the following time values:

Initialization phase: time to allocate and initialize the grid, $T_{InitGrid}$

Copy border: overall time to execute lines 10 and 11, $\#Iterations \cdot T_{CopyBorder}$

Creating threads: time to create and start the threads, $T_{InitThreads}$

Barrier phase: overall time to synchronize the Workers, $\#Iterations \cdot T_{Barrier}$

Complete Game of Life = $T_c - T_{InitGrid}$ (see Equation 1)

For each experiment, we have repeated the measure **5** times and we have given these values to **avg_time.sh** in order to obtain a good temporal candidate for the experiment.

6 User Manual

We delivered two compressed folder: respectively for **version 1.0** and **2.0** of the application. These folders have the same structure, as explained in Section 4.

6.1 Compilation

The folder contains a **Makefile** that allow us, typing the command **make**, to compile both the parallel methodologies: **"GOL_thread"** and **"GOL_ff"**. Besides, this **Makefile** can be configured setting the following control variables:

MIC: if set to true, it compiles the application for Xeon Phi™, else for Xeon Host (default false).

DEBUG: if set to true, activate the **"debug mode"** of the application, showing the evolution of the Grid (only if is a small one) and checking the correctness of the result comparing it with an easier, sequential and more trustful GOL implementation (default false).

MACHINE_TIME: if set to true, shows the time values in microseconds, otherwise it shows them in a more understandable format (default true).

For example, you can compile as following:

```
mkdir build
make MIC=true MACHINE_TIME=true
```

6.2 Usage

The two executables can be found in the **build** folder; in order to execute them, you can just invoking them on Xeon Host, while on Xeon Phi™ they must first be copied to **mic**.
Table 3 shows all possible options for the application.

For example, you can execute the following:

```
scp build/GOL_thread build/GOL_ff mic1:
ssh mic1 ./GOL_thread --width 5000 --height 5000 --thread 240
ssh mic1 ./GOL_ff --width 5000 --height 5000 --thread 240
```

Option	Description
--vect	activate the vectorization version
--grain NUM	chunk size assigned to threads (zero for static scheduling)
--width NUM	grid width
--height NUM	grid height
--seed NUM	seed used to initialize the grid (zero for timestamp seed)
--iterations NUM	number of iterations to perform
--thread NUM	number of threads (zero for the sequential version)
--help	shows all the options that can be set in the application

Table 3: *Showing all possible options of the application.*

References

[1] Wikipedia. Conway's game of life. *https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life*.

[2] Tim Tyler. Cellular automata optimisation. *http://cell-auto.com/optimisation*. Retrieved Gennuary 4, 2016.

[3] C.J. Lin Hideki Saito. Extracting vector performance with intel compilers. May 2012.

[4] Ronald W. Green. Random number function vectorization. September 2012. Last edited by AmandaS on September 3, 2014.

[5] Federico Conte. Game of life. *https://github.com/Draxent/GameOfLife*.