Game of Life

SPM project report

Federico Soave

July 20, 2016

In this paper I will describe the steps I have taken to study and write a fast implementation of Conway's Game of Life, a well-known cellular automaton. Given my past experience and personal taste, I picked `C++` as my programming language of choice.

Apart from the sequential implementation, two parallel versions of the program will be compared side by side: the first one, called *threads*, has been realized using plain `C++11` threads and synchronization mechanisms, offered by the Standard Template Library. The second one, called *fastflow*, has been realized with the aid of the parallel programming framework of the same name. A graphical version has also been implemented to check for correctness of the algorithms, and for fun.

1 Features

All implementations share the same business logic code which mainly consists in the `Matrix::updateRows` function. At execution, a matrix of the desired size is allocated and randomly initialized. Then, `updateRows` and `swap` are called in a loop for a given number of iterations.

Instead of the random initialization, the user can request that some well-known patterns (such as Gliders, Gosper's Guns or Schick's Engines) be drawn on the grid, and witness their evolution in time if using the graphical version. For a given height and width pair (h, w), the matrix will always be randomly initialized in the same way for all implementations.

As a means of verifying the consistency of the different implementations, I devised a simple hashing mechanism allowing the user to check that for a given dimension and number of iterations, the matrix will always end up at the same final state. The hashing is done in a single thread and as such it is a costly operation and should not be used when measuring performance.

2 Algorithms and data structures

The *Game* takes place on a 2-dimensional toroidal space, where each cell can be in either one of two possible states, and evolves at discrete time steps. To emulate this I used a matrix of items of one byte each, taking into account the toroidal property of the world. A more compact data structure (e.g. a bitmap) could have been used at the cost of performance, but given the abundant amount of main and cache memory present on the test system, I chose the former.

In each iteration, the state of each cell is recomputed by looking at the state of its eight neighbour cells at the previous iteration. For this reason a couple of twin matrices is used, where each matrix is read or written at each iteration in an alternative fashion. This is realized by simply swapping two pointers.

Despite being dynamically allocated as a contiguous area of memory (for sake of locality), the matrices are accessed via double pointers (i.e. an array of pointers to the rows). I took advantage of this to emulate the toroid on the y coordinate: the array of row pointers is allocated in such a way that the item at position -1 points to the last (h^{th}) row of the matrix, and the item at position h points to the first row of the matrix, allowing to seamlessly jump from one side of it to the other without the need of checking the index at runtime.

3 Parallelization

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. At the next iteration, the cells lying on the border of a partition must be propagated to the workers in charge of the adjacent partitions. This is easily recognized as a *stencil* parallel pattern.

For the sake of simplicity and data locality I chose to partition the matrix row-wise. Each worker thread is assigned a chunk of rows and must compute them. Also, before beginning the next iteration, it must wait that all the other workers have computed their partitions too. To this end, a cyclic thread barrier was employed, that all threads must wait upon at every loop.

3.1 The barrier

At first, I wrote my own barrier implementation using synchronization data types of the STL, such as `mutex` and `condition_variable`. In particular, `condition_variable::wait` is a system primitive which ultimately suspends the current thread until a `notify` or `notify_all` is called on that same `condition_variable`. Acquiring the `mutex` can also suspend the thread for a significant amount of time, and has quite a overhead. This became apparent when executing the *threads* version on the MIC with a large number of threads.

For this reason, I switched to a "spin" barrier, realized using only `C++11`'s `atomic` types. This solution has the advantage of working in userspace only and threads are not suspended, but blocked in a short busy-wait loop instead.

While the execution speed improved w.r.t the previous solution, it behaved quite badly with some specific number of workers, especially in the high end range.

3.2 Dynamic scheduling

In an attempt to improve speed at high degrees of parallelism, I introduced a new lightweight synchronization data structure aiming to balance the load among workers. It is a thread-safe queue loaded with fine-grained chunks of computation that all threads must fetch and compute. When the queue becomes empty, the barrier is called and the queue is reset for the next iteration.

The queue is simply realized with an `atomic` pointer to the next chunk to be computed, is created once and spans the entire time of execution. This new solution allowed me to achieve the performance demonstrated below.

3.3 Thread affinity

As suggested during the lab lessons of the course, I tried to bind the worker threads to the hardware threads of the CPU at the start of the computation. This is called setting the thread "affinity". For the MIC architecture in particular, one should make sure that different threads bind to different physical cores, if possible.

However, setting the thread affinity properly did not seem to improve performance in any case. On both architectures, the operating system seemed to do a very good job at assigning the threads itself, so I did not enable this feature.

3.4 Expected speedup

For brevity, I will only take into consideration the *threads* version executed on the MIC processor. In a typical execution, the main thread allocates the main data structures and then spawns the workers. In turn, they start by initializing the matrix at random and then proceed to compute the Game of Life on a partition of it. At every iteration of the Game, they synchronize each other by using the barrier and the queue of chunks for dynamic scheduling. One could model the total time of execution as follows:

$$T = T_{\text{seq}}(p) + \frac{h}{p}\Big[T_{\text{initRow}}(w) + s\, T_{\text{computeRow}}(w)\Big] + s\, T_{\text{synch}}(p) \qquad (1)$$

where the time of the sequential part T_{seq} may depend on the number of threads p to spawn, the parallel part is executed on partitions of $\frac{h}{p}$ rows each, and the main loop is executed s times. The time for synchronization T_{synch} may depend on the number of workers, too.

Rearranging the expression, we get

$$T = T_{\text{seq}}(p) + \frac{hw}{p}(T_{\text{initCell}} + s\, T_{\text{computeCell}}) + s\, T_{\text{synch}}(p) \tag{2}$$

from which it can be seen that increasing the parallelism degree p can cut down the compute time, but there may also be some effects on T_{seq} and T_{synch}. However, experimental measurements revealed that while T_{synch} does not significantly increase with p, both T_{seq} and, surprisingly, T_{initCell}/p increase approximately linearly for $p \geqslant 40$.

If for the sequential part a linear dependency on p is quite clear, because compute chunks must be allocated and threads must be started, the same cannot be said for T_{initCell}/p. One possible explanation is that, since in the initialization phase the matrix is accessed for the first time, the cost to fetch it from main memory must be paid right away. This holds true especially because, even when avoiding to randomly initialize, the total execution time does not change significantly.

3.5 FastFlow implementation

Given that the business logic code was already there and I had already realized the *threads* parallel version, plugging in the FastFlow framework was very easy. The `ParallelFor` class provides exactly the functionality I need. It is sufficient to call the `ParallelFor::parallel_for_idx` function in the iteration loop. The framework takes care of spawning and synchronizing the threads, and assigning them the tasks.

By default, synchronization in fastflow is realized via blocking system primitives. As an option left to the programmer, `ParallelFor` allows to enable a "spin" version, that synchronizes via shared atomic variables. As in the *threads* version, on the MIC architecture this modality performs better that the blocking one, whereas there is no improvement on the Xeon one. If anything, in some cases it causes the execution to terminate in unusually long times when run in the high end range of worker threads.

There is a difference worth noting between my *threads* implementation and the *fastflow* one: whereas in the former the main thread spawns the worker threads and then gets suspended until the computation completes, in the latter the main thread is in charge of scheduling the tasks on the workers at every iteration and takes up one hardware thread on its own (or triggers several context switches, degrading performance even more).

4 Optimizations

Every bit of the program has been written with speed in mind. In order to get a good sequential code speed, I paid great attention to the following aspects:

- not strictly necessary memory accesses and arithmetic operations (e.g. computing a remainder to wrap-around on the matrix) should be avoided;
- memory access patterns should be as "regular" as possible, preserving spatial/temporal locality;
- avoid unnecessary branches;
- unroll short loops with statically known loop count;
- virtual function calls should be avoided and inlined when possible;

The peculiar nature of the `Matrix` data structure allowed me to write the main code of the computation with nearly no conditional branches. Almost all functions are automatically inlined by the compiler.

Since it is called once per cell update, I put some effort in implementing the `Matrix::lifeLogic` function, determining the new state of the cell from the number of its live neighbours. The fastest alternative turned out to be based on (just 7) bitwise logic operations, that the CPU can quickly work out without any memory access, or code jump.

4.1 Random initialization

In its default behaviour, the program initializes the matrix before starting the computation using a pseudo-random number generator. Each cell has 0.5 probability to be initialized *live*, and 0.5 *dead*. Even if the calls to the random generator are kept at a minimum, for a rather big matrix of 8000 by 8000 this can translate in 2×10^6 calls.

In order to speed up the process, I parallelized the initialization phase and delegated it to the same workers executing the main computation code. This way, the initialization cost scales up well with the number of threads available. The `lrand48_r` random-generating function is used.

4.2 Vectorization

Processor vectorization is nowadays an essential tool to improve performance on data-intensive computations. Each of the 60 cores of the MIC processor comes with a 512-bit wide Vector Processing Unit which is exploited by using a new set of instructions, specific to this architecture.

The Xeon processor, on the other hand, supports the 256-bit wide AVX instruction set. In order to exploit this extension, the programmer needs to explicitly tell the compiler via a compilation flag, otherwise it will generate vectorized code targeting a more generic, less performing extension (e.g. SSE).

To check that vectorization of the kernel of the computation was successful, compiler reports were generated. It then sufficed to rearrange code around and issue a "`#pragma ivdep`" directive on the innermost loop. The compiler did the rest of the job.

For comparison, executables of the *threads* version compiled with different vectorization flags were put at a performance test: no vectorization, default compiler vectorization, and AVX extension vectorization (Xeon CPU only) were enabled alternatively. Figure 1 shows the different computation bandwidths obtained by running the different executables on square matrices of 4000 cells on each side, 1000 iterations, minimum (1) and maximum (16 and 240) parallelism degrees.

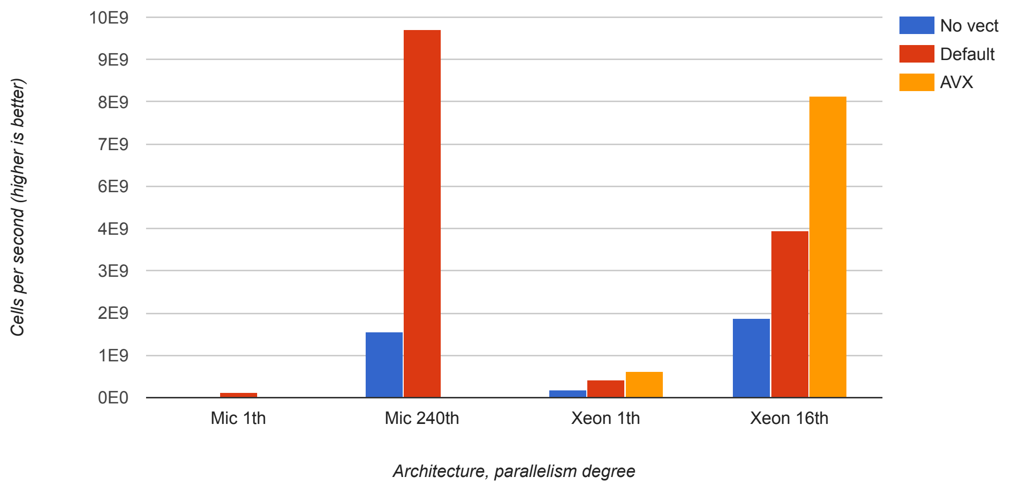

Figure 1: Vectorization speedup

5 Performance

To measure performances of the program, the three implementations *sequential*, *threads* and *fastflow* were run on the machine made available for the project, on both the main Intel Xeon processor, and the Intel Xeon Phi coprocessor (in native mode). Table 1 summarizes the main features of the two CPUs.

	Xeon E5-2650	Xeon Phi 5100
Num. of cores/threads	8 / 16	60 / 240
Frequency	2.00 Ghz	1.00 Ghz
L2 cache	8 × 256 KB	60 × 256 KB
L3 cache	20 MB shared	—
TDP	95 W	245 W

Table 1:

As a rule of thumb, input sizes were chosen as to obtain short enough and low-variance running times. For this, all tests were performed on square matrices of size 4000×4000 and 8000×8000, and 1000 iterations. From the running times, scalability, speedup and efficiency were calculated. Results are presented in the following charts. Since the execution times of *threads* and *fastflow* for $\mathrm{p} = 1$ are equal to those of the sequential version, the speedup chart is exactly like the scalability one and thus can be omitted.

At the maximum available parallelism degree, I was able to achieve a top speedup of 105.7 on the MIC, and 11.1 on the Xeon, with efficiency 44.1% and 72.8%, respectively.

6 Conclusions

The most interesting results were achieved on the MIC processor, given that on the Xeon all implementations behaved very similarly.

On this highly parallel architecture, it is essential to employ userspace, busy-wait synchronization mechanisms in order to unlock its true potential. A careful exploitation of vector units is also of paramount importance, as it can be seen in Figure 1 that it allowed to outperform the Xeon processor, in the end.

After a quick look at the documentation, FastFlow's `ParallelFor` turned out to be extremely easy to use and allowed to achieve very good performance with minimum effort and very little code. However, on the MIC processor the *threads* version outperformed it in every test, almost doubling its efficiency. On the other hand, it cost me a much bigger effort and some occasional headaches.

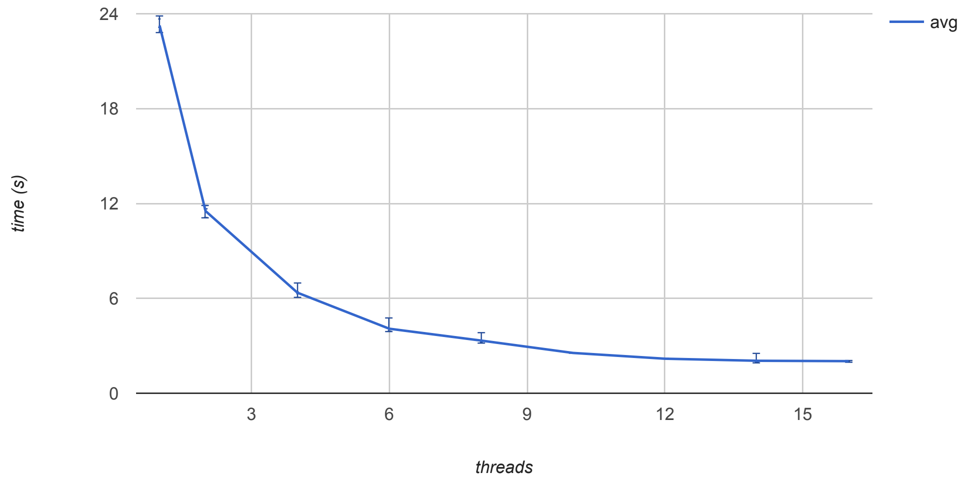

Figure 2: Running times, Xeon, 4000 × 4000, *fastflow*

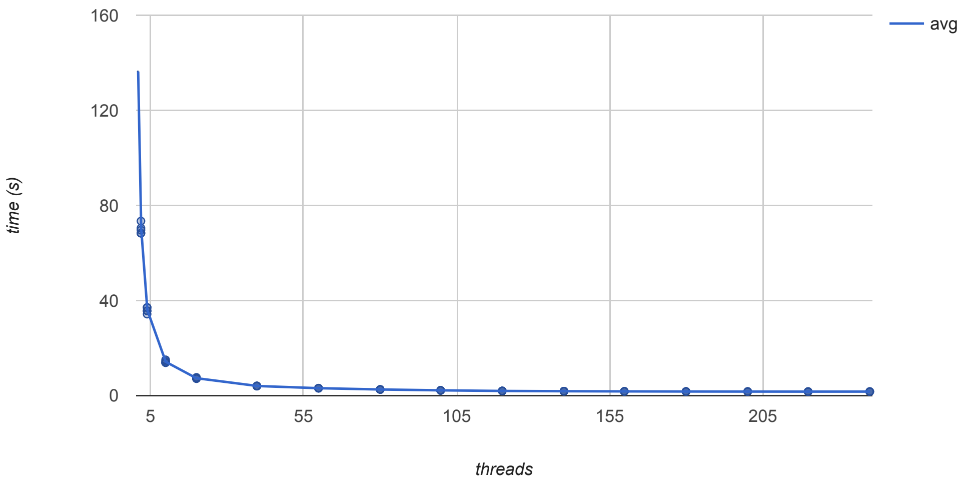

Figure 3: Running times, MIC, 4000 × 4000, *threads*

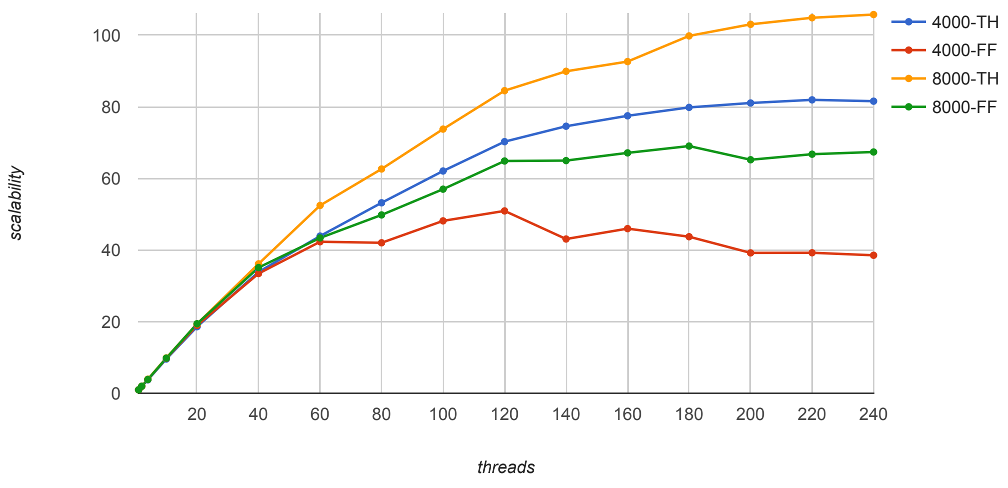

Figure 4: Scalability, MIC

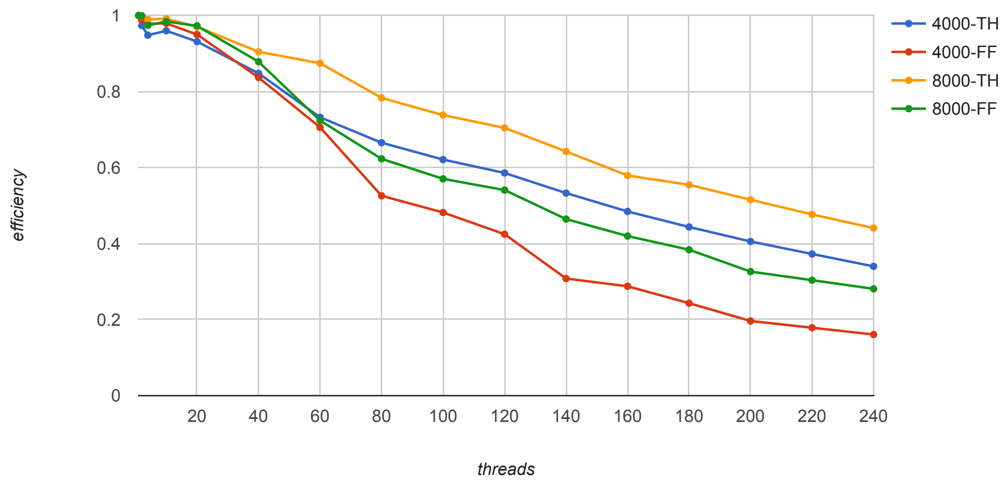

Figure 5: Efficiency, MIC

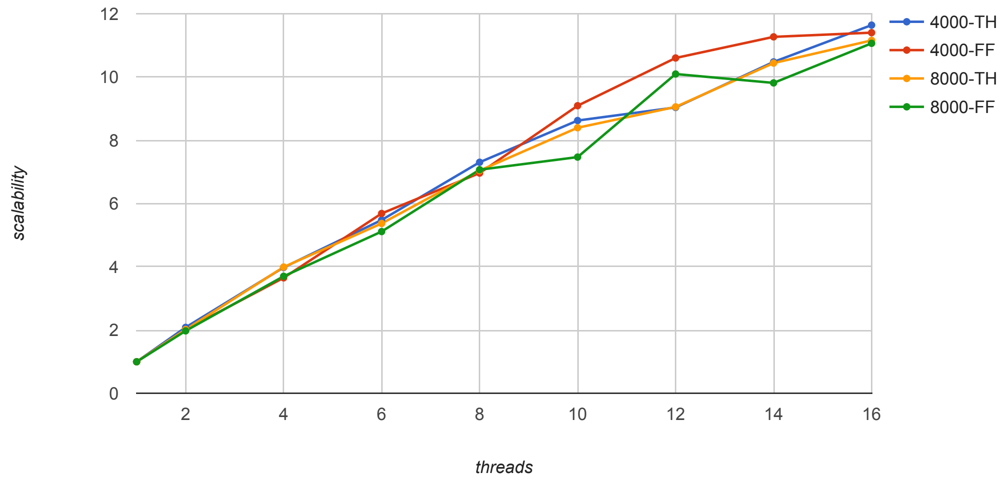

Figure 6: Scalability, Xeon

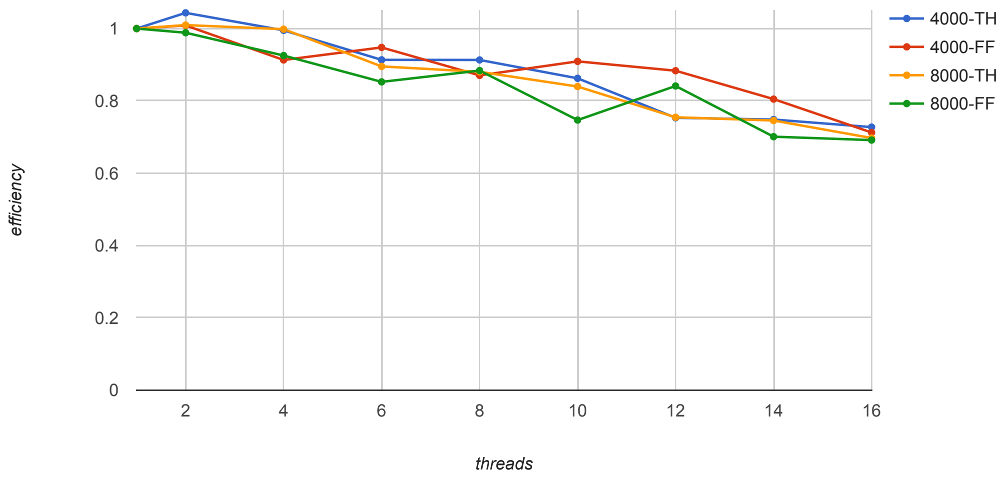

Figure 7: Efficiency, Xeon