# PDC Project

**Marco Stellin, 85795**
**Carlos Costa, 85063**
**Tomás Mendes, 84926**

April 7, 2017

## 1 Serial implementation

At first, we attempted to solve the problem by storing all the living cells in a simple linked list, and iterating over every cell in the cube, living or dead, to check whether it was alive or dead in the next generation. This approach, however, proved to be extremely inefficient: checking every cell and its neighbors wasn't practical, as it led to a complexity of $O(n^3)$, and the linked list proved to be quite slow in terms of inserting and searching elements. We then opted to only analyze the cells that were alive, and then check the dead neighbors around those cells to see if they should be alive in the next iteration. This yielded much better results especially for the larger examples. Instead of a Linked List we implemented an hash table with buckets (implemented as Linked Lists) to handle collisions. This data structure provides insertions in O(1) and, on average, also searches are performed in O(1). The hash function is generated as follows:

$$hash(x, y, z) = (x * 667 + y) * 971 + z$$

We use prime numbers in this equation to lower the probability of collision. Operations on the bits are also performed in order to make the hash as uniform and "random" as possible. The position on the table is found by performing a modulo operation on the hash with the table dimension. The table dimension is a large prime number as well, in order to avoid as many collisions as possible.

## 2 Parallel implementation using OpenMP

*Note: from now on we will address the hash table that stores the current generation of alive cells as Current Generation Table (CGT) and the one that stores next generation of alive cells as Next Generation Table (NGT).*

The algorithm performs basically two key operations:

1. Search in the cube domain for a particular cell to determine if the cell is alive or dead in generation $n$;
2. Insert in the cube that represents generation $n + 1$ the new alive cells;

When a search is performed in the CGT no problem can arise due to parallelization since we are performing reading operations on a table that is never changed in that generation. Problems arise when searching in the NGT, since, in this case, the table is modified multiple times in order to insert new alive cells. If no precaution is taken, race conditions are very likely to happen and this can lead to reading outdated data or even to the breaking of the program (due to invalid pointers in linked lists). Insertions are only performed in the NGT. This is a critical operation since we are modifying the state of the table, so no insertions or searches (at least in the cell of the NGT interested by the insertion) are allowed. These are the *critical* operations that we must pay attention to during the development of a parallel version.

In order to speedup the aforementioned operations and lower as much as possible the impact of critical sections, we used two OpenMP concepts: loop iterations distribution using the directive **#pragma omp**

**for** and locks. The first directive is used whenever there's an iteration over the cells of the CGT and the NGT. In this way, each thread performs the necessary operations just on the subset of alive cells contained in the assigned hash table cell. Locks are important in order to lower the impact of critical sections: a lock is created for each cell of the CGT and the NGT. Whenever a critical operation is executed, the thread that is performing that operation gets the lock of that position, performs the operation, and release the lock. All the other threads have to wait for the lock to be released to perform operations on that cell of the hash table. This approach is better than using critical sections, because only individual cells are locked and not the entire hash table.

Some other things were taken into account:

- ✓ **Data locality:** both in the CGT and in the NGT, the head of the linked list in each cell is stored as a structure instead of a pointer. On average we expect that each cell of the hash table will contain just one element. If this is the case, since data in an array is stored in consecutive positions in memory, there's a higher proability of finding the data we need in the cache. Storing pointers would have been inefficient, since jumps in other memory positions could invalidate the entire block of memory we are using. A test performed using Valgrind showed an almost perfect cache hit rate in L1 and L3 cache levels.

- ✓ **Load balancing:** if the hash algorithm distributes the cells uniformly in the buckets of the hash table, load balancing is not an issue, because every thread will receive approximately the same number of living cells to manage. Some tests performed on the hash table showed good distribution of elements and few empty buckets. The usage of OMP scheduling techniques didn't bring to noticeable speedups.

## 3 Results

Configuration: Ubuntu VM, CPU Intel Core I7-6700HQ with 4 physical cores, 1 thread/core, L1 Cache: 32K, L2 Cache: 256K, L3 Cache: 6144K.
S = Speedup

| | Generations | Serial | 1 Thread | 2 Threads | 4 Threads | 8 Threads |
|---|---|---|---|---|---|---|
| s5e50.in | 10 | 0.0476s | 0.0427s | 0.0276s (S=1.72) | 0.0677s (S=0.70) | 0.0523s (S=0,91) |
| s20e400.in | 500 | 3.563 | 3.818s | 2.023s (S=1.76) | 1.141s (S=3.12) | 1.606s (S=2.21) |
| s50e5k.in | 300 | 37.579s | 39.929s | 21.348s (S=1.76) | 11.376s (S=3.30) | 13.114s (S=2.86) |
| s150e10k.in | 1000 | 2.919s | 3.187s | 1.774s (S=1.64) | 1.090s (S=2.67) | 1.366s (S=2.137) |
| s200e50k.in | 1000 | 8.442s | 9.466s | 4.892s (S=1.72) | 2.748s (S=3.07) | 3.140s (S=2.68) |
| s500e300k.in | 2000 | 38.442s | 42.234s | 22.094s (S=1.74) | 12.364s (S=3.11) | 13.110s (S=2.93) |

The results are approximately as expected. The first instance is not indicative because it's too small and the overhead introduced by OpenMP decrease the overall performance. The most significative instances are *s50e5k* and *s500e300k*, because the algorithm deals with very large populations. In these cases we get good results with 2 threads and acceptable results with 4 threads. The algorithm doesn't scale well with the number of threads. This can be a result of too much overhead due to the presence of locks or, more trivially, could be related to the fact that we are running the tests on a VM, thus not exploiting fully the underlying CPUs. The poor results of running the algorithm with 8 threads were expected: the machine used for the tests has only 4 physical cores, so some threads are actually sharing the same core, thus decreasing the performance.