Instituto Superior Técnico

Parallel and Distributed Computing

Game of Life 3D

Gonçalo Gaspar 58803   Yuliya Plotka 76467   Helena Cruz 78190

1 Serial version

The language chosen for the implementation of this project was C++, because of its data structures and their performance. The approach for the implementation of the serial version was very simple. The idea was to basically store all the initial cells in a data structure corresponding to the initial generation and then start evolving, using only two data structures - current and next generation. It was also used another data structure for optimizing the algorithm. Cells were represented as a class with three integer attributes: `x`, `y` and `z`.

1.1 Data structures

The data structures used in this project were:

- `unordered_set`
- `unordered_map`

Cells are unique in the generation, meaning that it's only possible to have one cell with a set of coordinates `(x, y, z)`. For this reason, the best option to store cells was a `set`. The option of choosing the `unordered_set` instead of the `ordered_set` was easy: the need for ordering cells was only in the end of all the computation, when printing the results.

Even though `ordered_set` has better performance when compared to the worst case of the `unordered_set`, the amount of data and a good hashing function proved it to be worse. The same happened with the `unordered_map`, which purpose is going to be explained in the next section.

1.2 Serial algorithm

The computation of the next generation is made by iterating over the each cell of the current generation. For each cell, we calculate its number of neighbors. The number of neighbors is calculated by generating the 6 neighbor cells and then seeing if they're present in the set. If so, then it's a neighbor. If not, it's a dead cell. If the number of neighbors is between 2 and 4, inclusive, than it will survive into the next generation - we just add it to the set corresponding to the next generation. If the number of neighbors is less than 2 or bigger than 4, than the cell dies and we do nothing.

To optimize the algorithm when it comes to a dead cell becoming a live one, instead of brute forcing (for each possible dead cell calculate the number of neighbors), another strategy was adopted.

It's trivial to realize than any dead cell that is going to become a live one in the next generation has neighbors. Taking this into account, when calculating the number of neighbors of a live cell, if the cell is not a neighbor than it's dead. That cell is added to the `unordered_map` where `(key, value)` are represented by `(cell, number of neighbors)`.

Then just by iterating this map of dead cells it's very easy to decide its future: if its number of live neighbors is 2 or 3, it is added to the set corresponding to the next generation.

2 Parallel version

Even though the strategy used in the serial version worked perfectly, it was not the ideal one for parallelizing the algorithm.

2.1 Synchronization concerns

The problems with parallel computing are generally related to operations that need to be executed inside critical sections - delaying the whole program - and to barries in the synchronization - waiting for all threads to finish.

The straightforward solution to parallelize our algorithm would be to parallelize the analysis of each cell (the computation of its neighbors and deciding if its going to live or die). However, there are a few issues with this:

- The insertion of cells to the next generation set must be done inside a critical zone, by one thread at a time.
- The insertion of dead cells to the map of dead cells must also be done inside a critical zone.

This would introduce a lot of overhead in the computation. For this reason, a few adaptations were made to the algorithm and to the data structures.

2.2 Parallel algorithm

To parallelize the serial algorithm, it was crucial to determine the most computationally intensive parts of the algorithm and divide its work load through the various used threads. To accomplish this the following was made:

- Declaration of a fixed number of the previously described data structures and their insertion into a vector to be iterated by threads, in which cells are inserted through an index calculation function that depends on cells coordinates.
- The threads iterate through the sets, each at a time. Using OpenMP dynamic schedule, if a thread finishes processing its sets, then it will start on another batch of sets - batches of 4.

3 Results

A summary table is presented below with the analysis of the performance of the parallel algorithm versus the serial implementation.
T(s) - Time in seconds, **S** - Speedup $= \frac{t_{serial}}{t_{parallel}}$

			Serial	**2 threads**		**4 threads**		**8 threads**	
File name	**Size**	**N. gen**	**T(s)**	**T(s)**	**S**	**T(s)**	**S**	**T(s)**	**S**
s5e50.in	5	10	0.02	0.02	1	0.03	0.67	0.04	0.5
s20e400.in	20	500	8.79	3.61	2.43	2.22	3.96	1.6	5.49
s50e5k.in	50	300	116.19	28.94	4.01	16.92	6.87	12.36	9.4
s50e5k2.in	50	600	270.67	64.43	4.2	35.83	7.55	26.68	10.15
s150e10k.in	150	1000	3.88	3.68	1.05	2.47	1.57	2.45	1.58
s200e50k.in	200	1000	15.61	10.17	1.53	5.91	2.64	4.57	3.42
s500e300k.in	500	2000	91.24	43.73	2.09	26.65	3.42	19.02	4.78
s500e300k.in	500	5000	228.76	105.84	2.16	63.65	3.59	46.27	4.94

Table 1: Results - Time and speedup comparison