PDC Project (MPI version)

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

May 19, 2017

1 Introduction

In the second part of this project, an implementation of the Game of Life in MPI has been developed, in order to support a Distributed Memory architecture. In this type of architecture parallelization represents a considerable challenge, since it's not possible to rely on shared memory regions. As a consequence, the algorithm used was changed considerably with respect to the one used in the serial implementation. However the same data structure to keep track of alive cells has been used, namely a series of hash tables.

2 Foster's methodology

In order to find a good parallelization approach, Foster's methodology has been applied to the problem:

1. **Primitive task:** the primitive task is to determine if a cell is alive or dead in the next generation;

2. **Communication:** in order to accomplish the primitive task, each cell needs to know the state of the six surrounding cells;

3. **Agglomeration:** in order to minimize the communication, multiple primitive tasks have been grouped together. In particular we opted for a row wise decomposition of the space along the x dimension. With this approach each process needs only to send the cells that are on the boundaries of its assigned region (ghost rows) to the processes managing adjacent regions. First, we implemented a version where each process sends one ghost row to the process directly above and one to the process directly below. However, in an attempt to reduce the amount of communications, we changed it so that the processes would actually send two *ghost rows* instead of one. This allows for processes to compute the next state of the first ghost row outside their boundaries and use it in the next iteration of the algorithm. This approach should make the algorithm faster since communications between processes only happen once every two iterations. This decomposition, although valid, doesn't represent the best possible decomposition: a checkerboard decomposition requires less communication and scales optimally. Unfortunately, in order to meet the deadline, it was not possible to re-factor our code in order to take advantage of this decomposition.

Summarizing:

1. Process 0 computes the boundaries of the region assigned to each process. See section 3 for more informations;

2. Each process receives their respective boundaries from process 0, along with the number of iterations to perform and the size of the cube;

3. For the first generation, each process reads and stores in a table the living cells that belong to the region assigned to it. Other two tables are used to store the cells of the upper and lower processes. Another

approach has been tried: the master process (rank 0) was responsible of sending the packets of cells to each process. This approach is particularly useful if the file is big and the nodes in the network have memory constraints. However, this approach has been discarded, since it considerably slowed down the algorithm, especially in big instances.

4. Processes compute the future generation of their rows and the rows immediately outside their boundaries. In this last computation we take advantage of the additional ghost row to avoid performing a communication;

5. Processes compute once again the state of the next generation using the rows calculated in 4;

6. Every two iterations, each process serialize the cells to send to the adjacent processes in an array of integers and send the cells up and down. At the same time, the ghost rows of the adjacent processes are received and the cells are inserted in their respective tables. In order to avoid deadlocks, non-blocking receive functions have been used.

7. Process repeat steps 4-6 until the maximum number of iterations is reached.

8. Processes order their cells and send them to process zero for printing. This last communication takes a considerable amount of time if the number of cells is big. An alternative approach would be to store the output of each process in a text file and join the files afterwards.

3 Load Balancing

Load balancing is an important issue that needs to be addressed in order to get good performances in parallel programs. In our case we opted for a partially dynamic load balancing. At the beginning, the master process compute the boundaries of the regions assigned to each process, trying to split the initial alive cells evenly among processes. The number of alive cells is computed and, given this information, it's possible to determine approximately the number of cells each process should receive. An even distribution is not possible, and on almost all the situations, one of the processes receives more cells than the others. However, the difference, especially for a big number of processes, is negligible. A completely dynamic approach has been considered, but this would have required resizing of the regions at each iteration (or every n iterations) thus increasing the complexity of the algorithm and the amount of communication.

4 Results

The following results have been obtained by running the instances on the RNL cluster with the specified number of processors. Given our decomposition approach, it was not always possible to use 8/16/32/64 processes on instances with cubes with a very small size. Moreover, the bigger instance (s600e20M) was in large part not computed, due to errors in the cluster caused probably by an extremely big run time.
S = Speedup

Table 1

	Generations	Serial	2 Thread	4 Threads	8 Threads
s5e50.in	10	0.0476s	0.032s (S=1.48)	0.015s (S=3.17)	NA
s20e400.in	500	3.563s	2.733s (S=1.30)	2.651s (S=1.34)	1.301s (S=2.74)
s50e5k.in	300	38.579s	24.480s (S=1.58)	12.150 (S=2.48)	8.555s (S=4.51)
s150e10k.in	1000	2.919s	3.142s (S=0.93)	2.698s (S=1.08)	0.842(S=3.46)
s200e50k.in	1000	8.442s	5.498s (S=1.54)	4.723s (S=1.79)	2.641s (S=3.20)
s500e300k.in	2000	38.442s	28.312s (S=1.38)	13.302s (S=2.89)	10.359s (S=3.71)
s500e5M.in	10	979s	1279s (S=0.76)	1823s (S=0.54)	528s (S=1.85)

Table 2

	Generations	Serial	16 Thread	32 Threads	64 Threads
s5e50.in	10	0.0476s	NA	NA	NA
s20e400.in	500	3.563s	1.189s (S=3.00)	NA	NA
s50e5k.in	300	38.579s	5.91s (S=6.53)	4.385 (S=8.80)	NA
s150e10k.in	1000	2.919s	0.571s (S=5.11)	0.9095s (S=3.21)	0.477s (S=6.12)
s200e50k.in	1000	8.442s	1.089s (S=7.73)	0.723s (S=11.67)	0.542s (S=15.58)
s500e300k.in	2000	38.442s	5.456s (S=7.04)	4.977s (S=7.72)	2.208s (S=17.40)
s500e5M.in	5	979s	229s (S=4.28)	77s (S=12.71)	22.53s (S=43.45)
s600e20M.in	5	$\geq 21600s$	NA	NA	9064s ($S \geq 2.38$)

The results, reported in Table 1 and Table 2 are approximately as expected. The algorithm scales very poorly and this is due to the fact that a row-wise decomposition has been used. Moreover, the data structure we used proved to be inefficient, since the dimension of the tables influence a lot the times, probably due to frequent cache misses when tables are big. Unfortunately, it's not possible to reduce the tables dimensions considerably, because otherwise, for large instances, the search of cells becomes very slow. As a consequence, the obtained times and speedups are very variable and multiple iterations of the same algorithm may give very different results (especially in smaller instances). In general, it's possible to achieve an efficiency that lies between 25% and 50% depending on the instance. The best results are obtained with the *s500e5M* instance, where we get a speedup of 43.45, with an efficiency of almost 68%. In some configurations, unexpected results have been obtained (low speedups even with high number of processes, speedups below zero, etc.). It was not possible to identify the causes with precision, but it's possible that the dimensions of the tables and the communication overhead sometimes influence the times in an unpredictable way. Due to the large execution times and errors in the clusters, it was not possible to measure reliable times for the bigger instance (*s600e20M.in*).