Parallel and Distributed Computing

Report 2nd Project

Parallel Sudoku using MPI

Professor Luís Guerra e Silva
2^{nd} Semester – 2017/2018
18/05/2018

Group 9:

81731 – Catarina Aleixo
81074 – Manuel Reis
81216 - Bernardo Amaral

I. Parallel implementation using MPI

- **Decomposition:**

In order to distribute the work properly and to avoid having just one processor dedicated to that, it was developed a ring structure in which each processor communicates with its neighbors (left and right). Initially the input file given has to be read by one processor only that compresses the board by sending the numbers that are fixed (its positions on the board and its values). Then it has to broadcast it to all the other processes to guarantee that every processor has a copy of the original board using the *MPI_Bcast()* function.

First, the program tries to settle, at least, one board for each processor since it will call the *checkPossibilities()* function, which calculates the number of possibilities in each index, until there are sufficient boards for every processor. When the number of possibilities is equal or greater than the number of processes, each processor will start working on a board with a possibility that is multiple of its id, which guarantees that each processor will work in different boards.

When a processor explored all its possibilities, it will ask for work, sending a message to its left neighbor. If its neighbor is currently solving its board game, it has to stop and give it the next valid board game and then it will either continue with the next valid guess on that board or backtracks if there's no more valid guesses for that index. Then when a processor receives a game board it has to decompress it and start working on it, calling the *decompress( )* and the *bruteforce()* functions.

- **Load balancing:**

In order to keep all processors occupied, when a certain processor is performing the algorithm, in its board, it will check if its right neighbor is idle. However, in order to decrease the overhead, it will only distribute a board if it's worth it, i.e, if the cost of sending the board is less than computing the rest of the possibilities left. This threshold is defined according to the size of the game board. It was tested the time that it takes to try every possibility, in each cell of a game board, until there are one million possibilities. The number of the index that accumulates that number of possibilities is the threshold defined.

Every time a process receives a board it has to invoke the *MPI_Iprobe()* and *MPI_Get_count()* functions to know the size of the string that will receive. Then it's ready to receive a game board, invoking the *MPI_Recv()* function.

- **Synchronization concerns:**

There are many synchronization problems that have to be thought and well structured. The termination of the program is one of them, since all processors have to be convinced that either there's solution and someone found it, or there's no solution and there are no processors currently working.

As it was noticed on the first implementation of the sudoku problem using OpenMP, there could be more than one thread claiming to have found a solution at the same time. In order to synchronize all processors, if someone finds a solution it will send to its left neighbor a message with its id minus p, that is the number of processors. Any processor that receives a negative number from its right neighbor knows that someone found a solution, and everyone will stop working. However, there's a problem if two or more processors discover a solution, at the same time, and only one can print it. This problem is overcomed using the following strategy: if there are two or more processors sending a message with its id minus p, the processor with that value more negative will be the one that will print the solution. This is guaranteed because all the processors that found a solution will only forward a message with a negative number if the number that they receive is more negative than the one they sent, meaning that only the processor with the lowest id minus p value will receive its own message. From that moment on, all the processors know that someone found a solution. Thereafter, the process that will print the solution will send a message with a special number that indicates to the other processors that they can terminate processing and then have to wait to receive the same message, before exiting the program. All the other processors only have to forward the message with this special number and then, they can exit the program.

In order to solve the synchronization problem when there's no solution, when some processor hasn't work to do, it has to ask its left neighbor for work sending a message with its id. If its neighbor doesn't have work too, then it will pass the message to its left neighbor, which means that if the process that asks for work receives its own message, there's no more work to do since, if some processor has work to give to its neighbor, it won't forward the messages with a positive id (that means the processor with that id needs work). If that happens, a message with the same special number that is used when there is a solution is sent and any processor that receives the special number knows that has to pass that message to its right number and then can exit the program. The processors that send the special number only have to wait for receiving it and then they can exit the program too. This solution works if two or more processors end their work and there's no more boards to explore, they only have to follow the procedure described above.

II. Experimental and Performance results

File	Nr of processors	Running time (m:s) (Cluster)	Running time (m:s) (Local PC)	Speed Up (Cluster)	Speed Up (Local PC)
9x9	1	0 : 0.85	0 : 0.67	0.15	0.20
	2	0 : 0.001	0 : 0.001	130	130
	4	0 : 0.001	0 : 0.001	130	130
	8	0 : 0.001	0 : 0.001	130	130
9x9-nosol	1	0 : 24.86	0 : 19.31	0.69	0.89
	2	0 : 12.92	0 : 10.10	1.33	1.70
	4	0 : 13.78	0 : 5.54	1.24	3.10
	8	0 : 28.57	0 : 17.39	0.60	0.99
16x16	1	0 : 33.61	0 : 25.54	0.42	0.62
	2	0 : 22.73	0 : 18.59	0.69	0.85
	4	0 : 16.71	0 : 17.10	0.94	0.92
	8	0 : 12.45	0 : 25.14	1.27	0.63
16x16-zeros	1	0 : 0.026	0 : 0.020	0.19	0.25
	2	0 : 0.026	0 : 0.021	0.19	0.24
	4	0 : 0.028	0 : 0.022	0.18	0.23
	8	0 : 0.1280	0 : 0.072	0.04	0.07
16x16-nosol	1	26 : 3.5154	20 : 14.89	0.75	0.96
	2	17 : 4.510	10 : 25.58	1.14	1.87
	4	13 : 18.89	6 : 25.44	1.46	3.03
	8	12 : 35.00	12 : 41.78	1.55	1.53

Table 1 – Parallel performance

File	Running time (m:s)
9x9	0 : 0.13
9x9-nosol	0 : 17.19
16x16	0 : 15.78
16x16-zeros	0 : 0.005
16x16-nosol	19 : 26.90

Table 2 – Serial performance

The results presented in both tables are medium values, since each file was tested at least 5 times. From table 1 it's possible to see that running on a local computer of 4 cores, when the number of processors is 8, the performance results are worst due to the fact there are 2 processes sharing the same core. However, there are results that are better in a local computer, since there is lower overhead in the communications. Some of the tests that were done using the cluster were influenced by the overloading of the other users.