# Parallel Sudoku @ MPI

**Group 10:**
Artur Esteves - 91603
Ricardo Morais - 91064
Luís Fonseca - 79770

## 1 Parallel solution

The parallel solution is based on the serial solution used in the OpenMP delivery. The MPI sudoku application uses the master-slave model where each slave correponds to a machine where work will be done. The master manages the communitacion between both entities.

### 1.1 Decomposition

Given that our solving algorithm traverses a tree to reach a solution, each processor will be given a sub-tree where each node corresponds to a candidate for an empty cell. Initially, we generate the first candidates for a tree and push them to a LIFO queue. The slaves request work when possible and try to solve their sub-tree.

### 1.2 Load Balancing

Our solution uses a centralized dynamic load balancing scheme. Whenever a node is free, which means it didn't found a solution, it requests new work from the master's work pool. As long as the work pool is not empty, every processor has always work to do. One issue with this approach is if the work pool is empty and the computation finishes earlier than the other slaves resulting in work unbalance.

### 1.3 Synchronization and Communication

Our solution uses a synchronous communication scheme through the use of $MPI\_Send$, $MPI\_Recv$ and $MPI\_Probe$ functions. The master process listens to the slaves messages inside a loop. Firstly, we use the probe blocking call to determine the type of message and size of the buffer. Then, we received the buffer of the message. To detect termination in the the master process, whenever a solution found message is received or the number of available process reaches zero, we exit the loop. To assure the master doesn't terminates before the slaves, we check what slaves are still active and send a send a termination message.

## 2 Results

These results were obtained in the cluster of 9 machines dedicated to the course. Each node runs on an Intel® Xeon® CPU E3-1230 v2 (Ivy Bridge) @ 3.30GHz with 4GB of RAM.

The results dipslay the average time of 10 runs of each puzzle. The elapsed time excludes the MPI initialization and the time after finding a solution. The number of processors correspond to n slaves + 1 master.

| Puzzle | Elapsed time (s) |
|---|---|
| 9x9 | 0.204590 |
| 9x9-nosol | 19.479164 |
| 16x16 | 26.965905 |
| 16x16-nosol | 1485.200749 |
| 16x16-zeros | 0.008854 |

Table 1: Execution results of the series implementation

| Puzzle | Processors | Elapsed time | Speedup |
|---|---|---|---|
| 9x9 | 2 | 0.723814 | 0.283 |
| 9x9 | 3 | 0.390272 | 0.524 |
| 9x9 | 4 | 0.004589 | 31.476 |
| 9x9 | 6 | 0.005906 | 34.641 |
| 9x9-nosol | 2 | 39.025948 | 0.500 |
| 9x9-nosol | 3 | 39.236490 | 0.497 |
| 9x9-nosol | 4 | 27.031722 | 0.721 |
| 9x9-nosol | 6 | 40.46801 | 0.480 |
| 16x16 | 2 | 88.430755 | 0.305 |
| 16x16 | 3 | 42.484877 | 0.635 |
| 16x16 | 4 | 29.892401 | 0.92 |
| 16x16 | 6 | 27.009316 | 0.998 |
| 16x16-zeros | 2 | 0.091952 | 0.101 |
| 16x16-zeros | 4 | 0.036409 | 0.220 |
| 16x16-zeros | 6 | 0.068061 | 0.118 |

Table 2: Execution results of the parallel implementation

Puzzles with minimal backtracking or none register negative speedups on account of the communication overhead. Due to synchronization problems, some test display outcome but the program stays locked.

## 3 Conclusions

Our obtained results were underwhelming due mainly to our data decomposition. Each availabe processor solves a big tree instead of distributing the work from lower nodes in the tree to other available processors causing a poor load balance. The fact that our communication is synchrounous causes deadlocks sometimes when solving puzzles leading to unpredictable behaviour. As further work, we should have implemented non-blocking communication to avoid this types of situations.