Parallel and Distributed Systems: Paradigms and Models

Riccardo Paoletti
Student ID: 532143

Year 2019/2020

Abstract

This report refers about several brute force parallel implementations of the well-known game *Sudoku*. Two implementations have been developed using only C++ threads, while one exploits the FastFlow library. All implementations rely on the exploration of the solution tree of the schema, making decisions and eventually running them backward if they turn out to be wrong.

1 Sequential resolution

The sequential implementation exploits recursion.

The Sudoku schema is represented as a 9x9 Matrix of cells. Every Cell is stored as a structure composed of an integer, representing the value assigned to the cell, or *UNASSIGNED*[1], and a vector of integers, representing the possible values that could be assigned to that cell.

Before actually begin to look for a solution, all singletons[2] are assigned with the only possible value they can have, then the exploration of the tree can begin.

The algorithm begins by detecting the unassigned cell with the fewer number of possible assignable values. Then for each of these values, the cell is assigned with that value, the possible values of the other cells are updated consequently and the algorithm is executed recursively on the new grid obtained.

If the decision turns out to lead to an empty solution space, then it is inverted, the cells returns to be *UNASSIGNED* and another value is chosen for that cell. If all the values turn out to be wrong then the schema is unsolvable and *false* is returned. If all cells in the schema turn out to be assigned with a value, i.e. a solution has been found, then *true* is returned.

[1] Special value that means that the cell is without a value.
[2] Cells with only one possible value.

2 C++ thread implementations

Here two parallel implementations of solutions to the brute force Sudoku resolution problem are presented. They exploit the C++ standard thread library. It's important to say that the schemes and the cells are represented in the same way described in the sequential section.

2.1 Single Queue Implementation

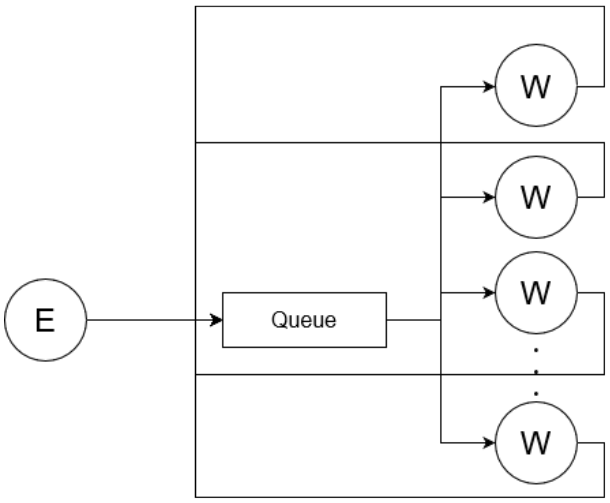
The implementation described here uses a single shared queue for all the threads.

The threads are stuck on the queue waiting for schemes to solve, while the main thread acts as emitter, splitting the original schema in new schemes.

There is an atomic boolean variable that is periodically checked by the threads indicating them whether a solution has been found or not, in such a way that once a thread finds a solution, it modifies this variables and all the others threads can quit the execution knowing that someone else has found a solution.

This communication method has been used also in the other implementation.

The schemes emitting procedures starts looking for the cell with the minimum number of possible assignable values. Once found it, for each of these values, a new schema with that value assigned is produced. The possible values of all cells are computed according to the new assignment. Once the schema is ready it is inserted in the vector of a thread, in a round-robin fashion. Once the emitting procedure is finished, all threads can start.



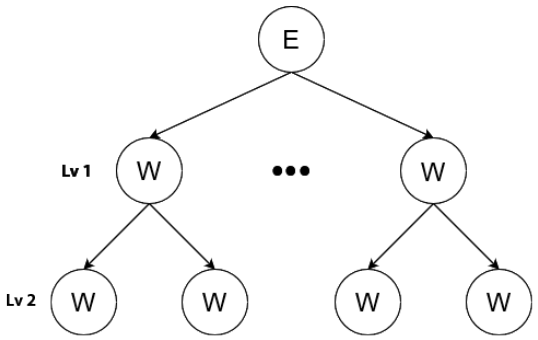
A thread popping out a schema from the queue splits the job in branches, tries to solve one of them and pushes the remaining ones in the queue to allow other threads to pop them out and try to solve them. In particular they select the cell with minimum number of possible assignable values and keep a value for themselves, while for the others they create new schemes and push them in the queue. Once finished splitting, they come back to the value kept and try to solve this branch recursively, eventually producing new schemes to be pushed in the queue.

2.2 Divide&Conquer

This implementation relies on the Divide&Conquer programming paradigm. The program starts with one main thread and takes as parameter the parallel degree that express how deeply the program should dig in splitting the problem into sub-problems before starting to

solve the schemes.

At the beginning it's checked if a solution has been found or not, if it has been found then the procedure ends, otherwise it is checked if the desired parallel degree has been reached, if it is so, then the schema is solved directly with the sequential procedure.



Otherwise the singletons are assigned and the cell with the minimum number of possible assignable values is selected. Then, for each of these values, a new schema is created copying the actual one and assigning that value, and a new thread is started, with this schema as parameter and this same function as body.

Eventually a thread will find a solution and will update the atomic boolean variable, otherwise all threads will finish their task and stop.

3 FastFlow implementation

This implementation uses the *ff_Farm* construct in a Master&Worker fashion, where the emitter node, that here is the master, once spread the schemes among the workers, remains active receiving feedback from them.

In particular, a worker that wants to solve a schema does not explore all the branches of the solution tree for that specified schema. Instead, it explores only one branch, and accumulate the other branches that it meets in a vector, that will be sent back to the master.

The master, receiving these vectors, will open them and transmit again all the schemes contained in these vectors to the farm of workers to try to solve them, eventually receiving from them other vectors resulting from the exploration of such schemes.

The program terminates when a solution is found or all possible schemes have been tested for a solution.

4 Performance modeling

Solution space of Sudoku resolution problem can be pictured as a tree where the leaves are at different depth and each internal node has a non-fixed number of children. The task of the parallel implementation is to explore this tree with different workers in such a way that each worker goes through its own path and either finds a solution, or arrives at the end of the path and goes to the next one.

In a situation like that, the performance of the sequential algorithm depends on where the solution is located. The more it is in the left of the tree, the more it is fast to find it and vice versa.

When the search is parallelized, each worker takes a path, even not started by it but only continuing someone else's job, expands the nodes to find every possible branch to assign to other workers and then continues in its own. If it finds a solution, the other workers are notified and all of them will stop e and quit.

In a situation like that, the scalability of a solution depends on the number of branches and path to surf before actually reach the solution. With a number of workers lesser than or equal to this number, the program will actually have a speedup in the execution, while going above this threshold introduces only overhead that is not justified by the work to do.

In addition to this, also when the job is quite parallelizable and a worker finds a solution in a small time, it's not sure that the execution time will be so much short, because another worker could be busy to navigate a particularly long path that doesn't lead to a solution but it will know it only when the path is finished.

Before actually quitting the execution, the master has to wait for this worker to end its path before actually finishing. This waste of time it's not actually so invalidating but it has to be kept in mind anyway.

5 Results

The experiments have been conducted on a data set of ten Sudoku schemes (index from 0 to 9), with different complexity. The programs have been tested using a varying number of workers (1-2-4-8-16-32-64-128-256). For the sake of simplicity, only results about three schemes (2-6-8) will be shown but they are quite different in such a way to have a quite complete glance of the situation.

5.1 Test structure

For the sequential implementation the execution syntax is:
`Sudoku-seq-BF <board_index>`
where `board_index` is the index of the board to solve.

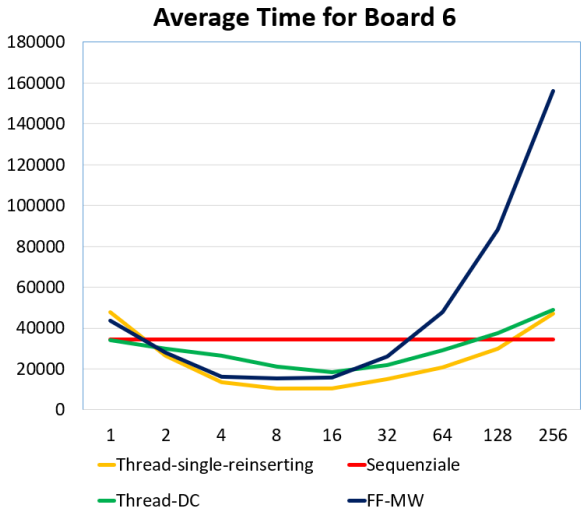
For the single-queue implementation the execution syntax is:
`Sudoku-single-queue <nw> <board_index>`
where `nw` is the number of workers and `board_index` is the index of the board to solve.

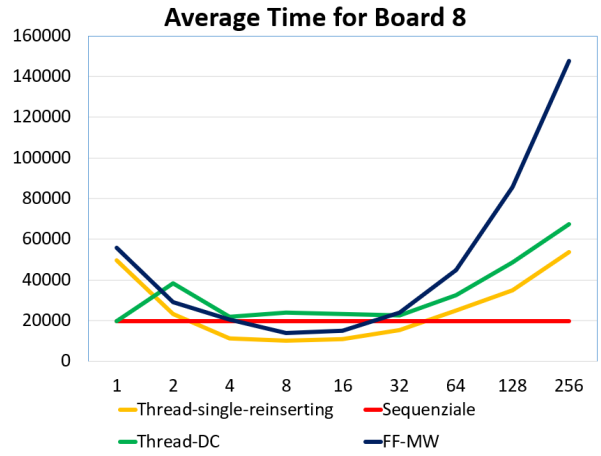
For the Divide&Conquer implementation the execution syntax is:
`Sudoku-DC <par_degree> <board_index>`
where `par_degree` is the maximum deepness of the solution tree to reach and `board_index` is the board to solve.

For the FastFlow implementation the execution syntax is:
`Sudoku-FF <nw> <board_index>`
where `nw` and `board_index` are the same as before.

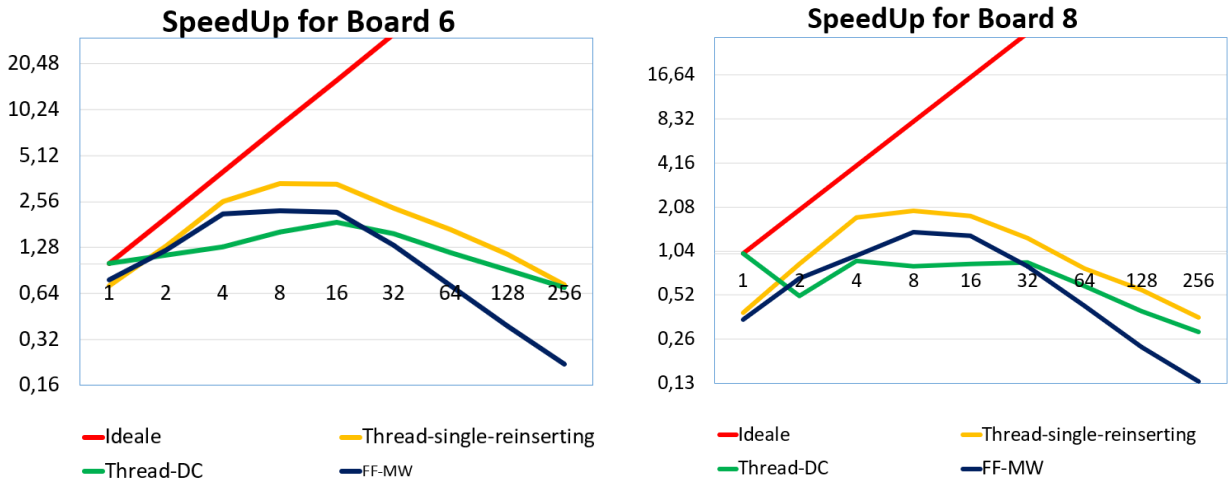
5.2 Average time and SpeedUp

In the figures below the average execution time for two schemes, number 6 and 8, is shown with respect to the number of workers used. Solution of schema 6 is more computationally costly than the one of schema 8, in particular, average sequential times for them are 34.504,9 microseconds for the first, and 19.675,6 microseconds for the second one.



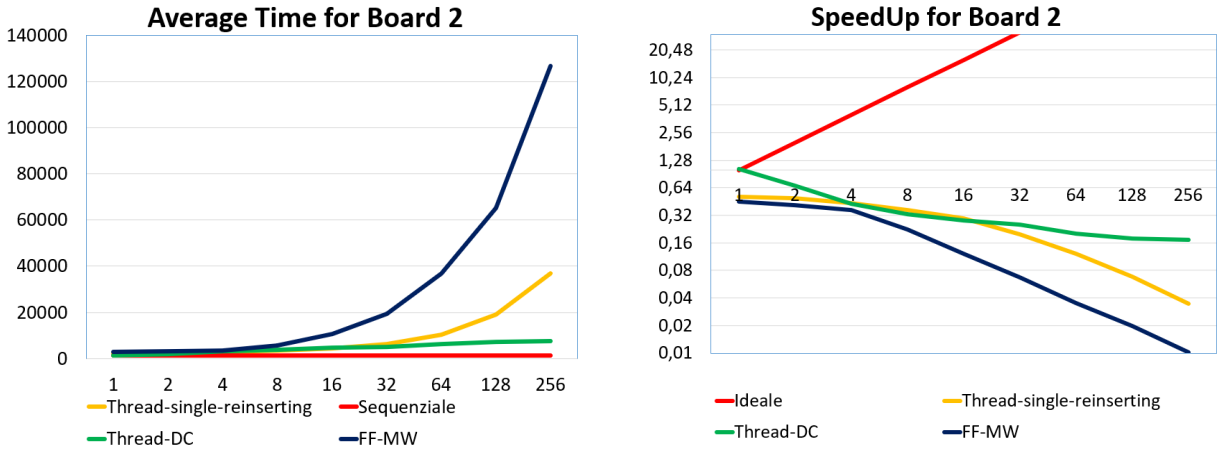


As shown in the figure above, all implementations have better performance on more computationally costly schemes, in particular the single-queue one has the best.



This is confirmed also by the SpeedUp shown in the figure above. The single-queue implementation perform better than the other two with any number of workers. The FastFlow Master&Worker one is better than the Divide&Conquer one up to 32 workers, when the behaviour is inverted and the M&W performance deteriorate quickly.

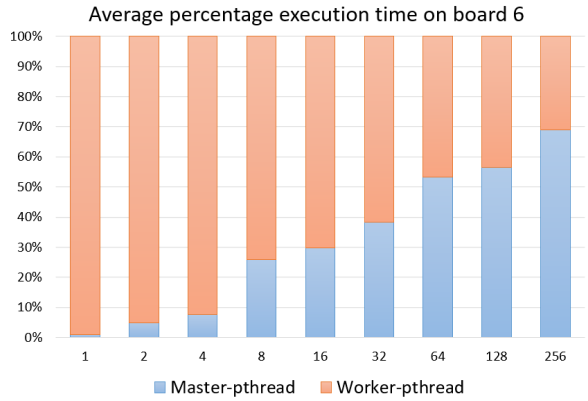
The situation changes for very computationally cheap schemes, like the schema 2. The third scheme has an average sequential time of 1288,06 milliseconds.

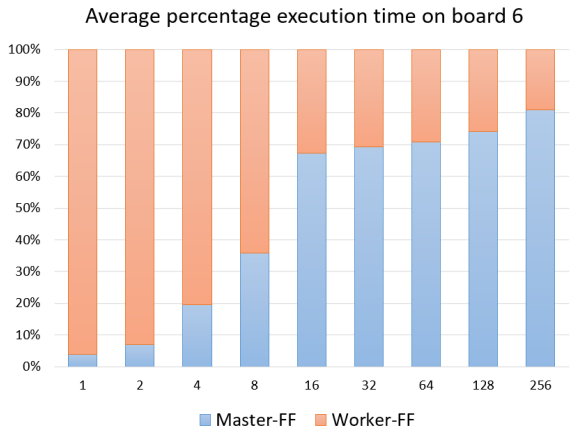


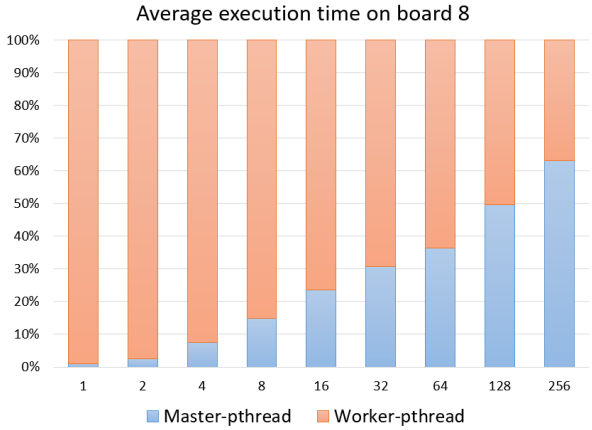
As it can be seen in the figure above, all implementation performs quite bad because of the overhead introduced by parallelism that is not rewarded by the small workload of the base cases in simple schemes.

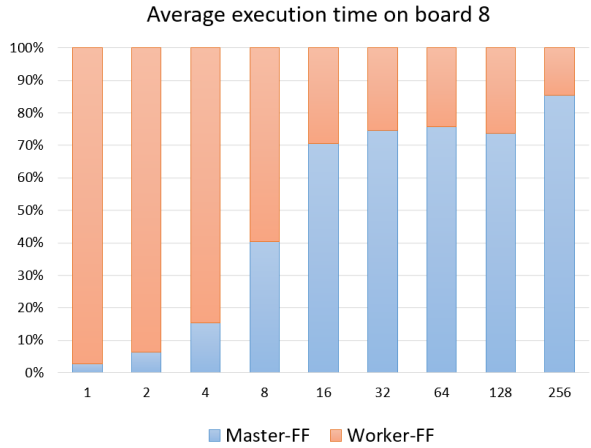
5.3 Master and Workers performance

In the following figures it's shown the percentage of work, with respect to the total execution time, that is taken by the master, i.e. the main thread, in the C++ pthread implementation and in the FastFlow one on boards 6 and 8.

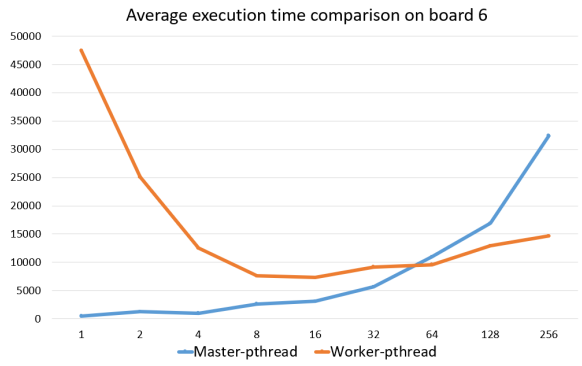


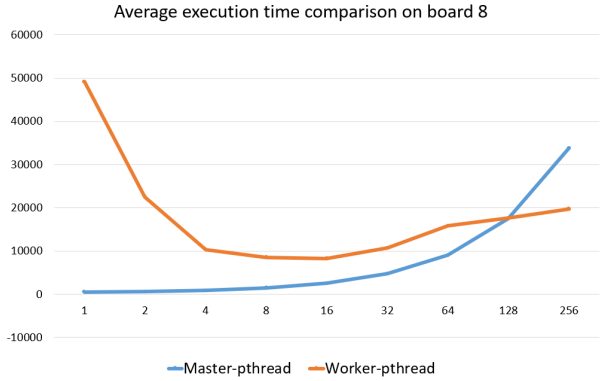


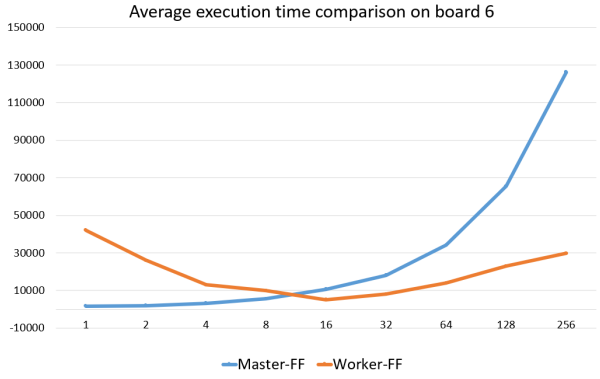


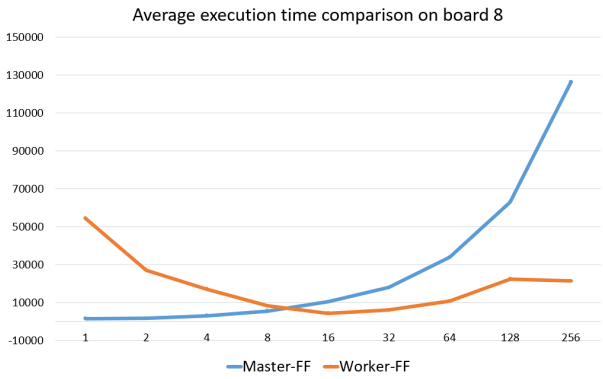


As it can be seen, with a small number of workers the master has a low computational load, and almost all the execution time depends on the workers. Increasing the number of workers makes the workload of each one of them to reduce while the one of the master to increase, due to all the communication overhead introduced by parallelism. In fact, not only the percentage raises up but also the average execution time as it can be seen in the figures below.





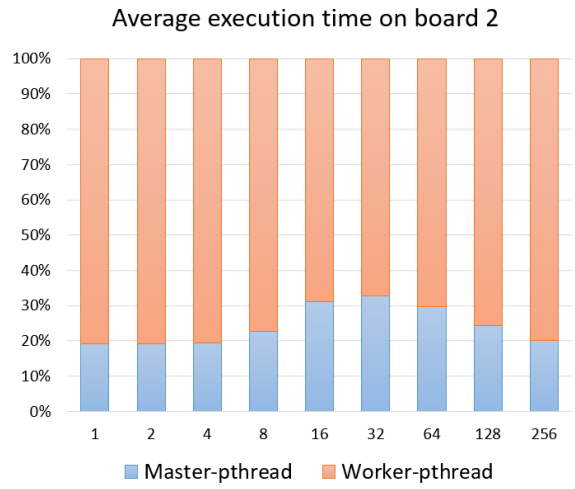


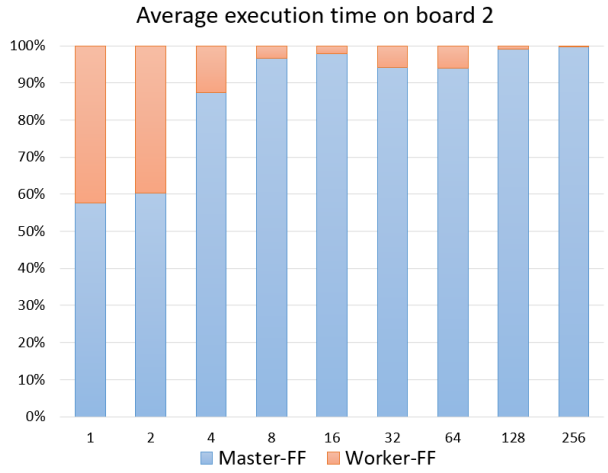


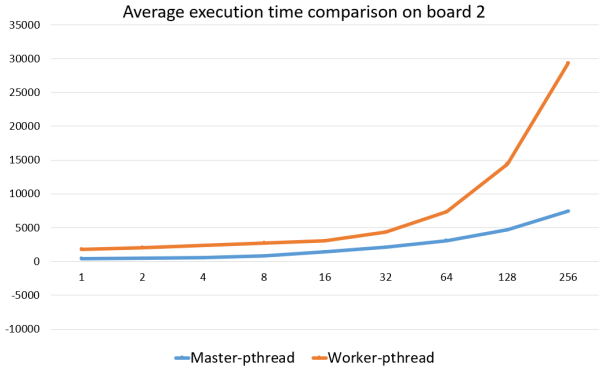
It can be observed how, in the FastFlow implementation the master workload and the workers one are complementary. With few workers the master workload is high and the workers' one is low, while vice versa when the number of workers increases the master workload tends to decrease and the workers' one goes up.

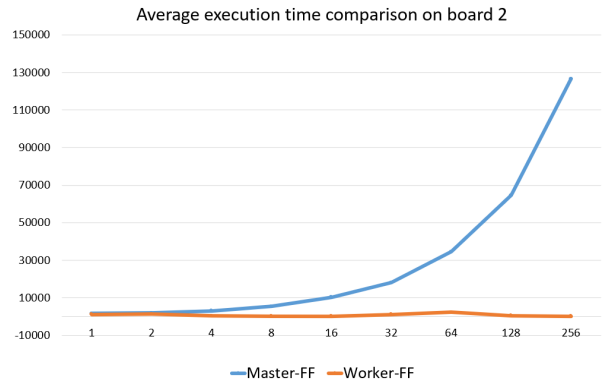
The pthread implementation behaves quite similarly, but less regularly, with the cross point in the graph that is moved from the center to the right to a bigger number of workers.

For very simple schemes, like scheme 2 that is shown in the figures below, the situation is quite different.









For the FastFlow implementation, because of the small amount of work needed to solve the problem, the workers' workload is quite low and never raises, while the master's one is low for few workers and tends to increase. This is due to the overhead introduced by the management of the parallelism.

For the pthread implementation, the situation is opposite. The master's workload is quite low for any number of workers, while the worker's workload tends to increase as the number of workers increasing. In particular it's peculiar how the master's workload percentage on the total execution time remains almost unvaried, even with 256 threads to manage. This is due to the fact that in this implementation there is a single shared queue where all the threads are awaiting, the job of the master is only to start them and send the first schemes, but this job is done independently of the number of workers are active. In particular, before of the explosion of the execution time due to the fact that there are too many workers solving branches while the solution has already been discovered, coming around 64 workers, the master's workload tends to increase a bit with the number of workers but this raise is minimal due to the fact that the solution is discovered very early and some threads not even have time to receive the first schema to solve because they have to quit.

6 Conclusions

This report evaluates several parallel implementations of a Sudoku solver based on C++ threads and FastFlow library.

Traversing the solution space tree during a search in parallel is the best way to reach a solution faster. Different solutions have been implemented. The main difference among them consists in how a thread handles the assignment and resolution of a schema: they use respectively a single blocking queue, Divide&Conquer approach and Master&Worker programming paradigm.

For quite difficult puzzles, it has been observed an immediate speedup with all implementations by increasing the number of threads up to eight. It has been achieved a maximum of up to 3,4 times speedup with our best parallel algorithm which uses one queue shared among all threads and eight threads. Overhead caused by increasing the number of threads beyond eight did not allow better time performance.

In particular, since the left-first visit has been used on the solution tree, the more a solution lies on the right of the tree, the more the computation is parallelizable, because it is more computationally costly. Conversely the more a solution is to the left of the tree, the more the computation becomes sequential, getting closer to a chain rather than to a tree, and making it quite hard to parallelize.

The pthread implementation has less overhead on the master, and in general performs better than the FastFlow one. The Divide&Conquer one is the worst in terms of average execution time and speedup but it's also the implementation that suffers less overhead due to the thread management, because of it's tree structure.