Brute Force Sudoku Solver

Parallel and Distributed Systems

Eleonora Di Gregorio
520655

July 2020

Contents

1 Introduction

Sudoku is a classic puzzle whose objective is to fill a 9 × 9 grid with digits so that each column, each row, and each of the nine 3 × 3 sub-grids that compose the grid contain all of the digits from 1 to 9. The initial configuration consists of having some cells that contain **fixed values**, the more they are, the easier finding a solution is.

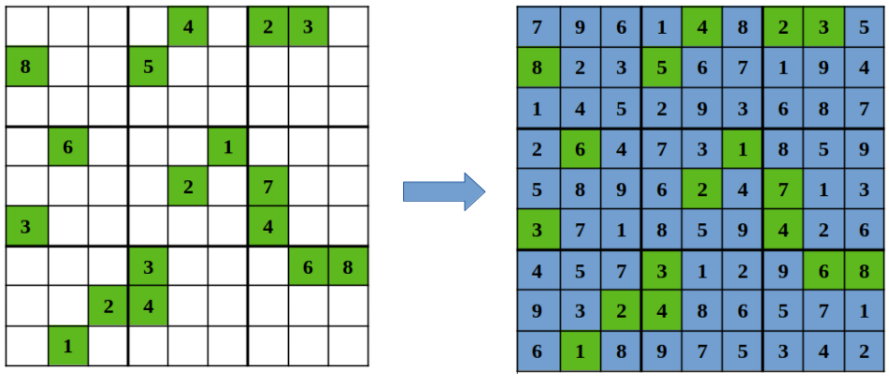


Figure 1: Sudoku resolution

A well-posed puzzle has a **single solution** and discovering it can be done quickly applying some smart considerations, but we are interested in a solution based on brute force. We want to visit the tree of configurations that are generated fixing the values of a cell to each one of the feasible values for that cell. The following image clarifies how the tree appears, the black cells are those with fixed values.

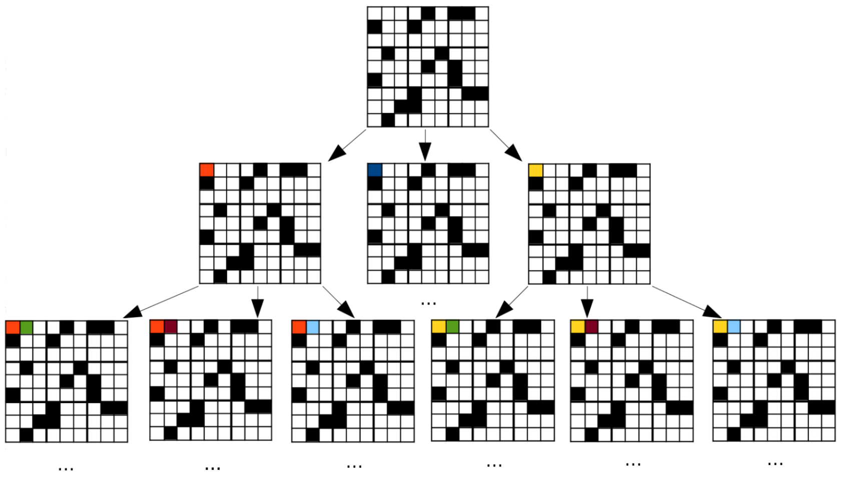


Figure 2: Partial configurations tree

The more interesting Sudoku to solve via brute force are those called minimum Sudoku, they have only 17 fixed cells which is the minimum number of fixed values to guarantee

a unique solution as shown in [2].
An example of a minimum Sudoku and its resolution is given in figure 1.

An instance of the problem is represented by the class Sudoku, which is made up of two configurations, respectively the starting one and the solution. A `configuration` is a `struct` as follows.

```
struct configuration{
    std::vector<std::vector<int16_t>> grid;
    std::vector<std::vector<int16_t>> rows_values;
    std::vector<std::vector<int16_t>> cols_values;
    std::vector<std::vector<int16_t>> grids_values;
};
```

Listing 1: Configuration struct

The `vector grid` contains the rows of the Sudoku, represented as vectors with their actual values, `vector rows_values` contains for each row the vector of values that are still no placed in that row, the same thing but for columns and sub-grids is done by the `vectors cols_values` and `grids_values` respectively. Each configuration uses 324 16-bit integers, plus the space for vector structures.

Class Sudoku provides a method *print* to print the configuration passed as a parameter and a method *fix_value* to fix a value in a configuration and consequently update the vectors of available values.

The next sections describe the problems faced and the different implementation proposed.

2 Sequential versions

Finding a solution to the problem means finding the best way to **generate and visit the tree** of possible configurations. The position of the solution varies depending on the structure of the starting configuration of Sudoku but it always appears as a leaf in that tree. Indeed all the leaves of the tree are invalid configurations, except for one which is the solution.

Given the starting configuration, the first thing to do is fixing all the cells with only one feasible value, these choices are mandatory and help in **limiting the growth of the tree**. The function which performs this operation is *fix_valid_values* and returns the number of elements that have been fixed. This operation is inserted in a loop because each time a new value is fixed, a new constrain is added and this could lead to the previous scenario. This operation is repeated not only at the beginning but also when a new configuration from the tree is evaluated.

When no more cells can be assigned we can be in three different situations:

- the solution is reached, very simple Sudoku can be solved just repetitively applying this function.
- no solution is possible, the configuration set so far is not valid.

- the configuration is a valid one but we have to choose a new element and try to solve the problem using all the possible values that fit in the cell.

The third scenario is the most interesting for the brute force resolution of the puzzle. In figure 1 is shown as an example that the first element is chosen to be fixed but this is not the best choice, indeed we want to limit the number of configurations generated and proceed down in the tree as fast as possible. The best choice is the cell with the minimum number of feasible values, such that the **branching factor is minimized**. The cell is discovered by the function *brute_fix*. At this point a strategy to visit the tree is needed, three implementations are proposed and studied applying it on minimum Sudoku.

- *BFS_seq* : the tree of configurations is generated and visited through a breadth-first search, the implementation is done using a `deque`, the new configurations are pushed at the end of the queue and the new ones to test are popped from the front.
- *DFS_seq* : the tree of configurations is generated and visited through a depth-first search, the implementation is done using a `stack`, in this way the last configurations generated are the first to be explored.
- *mix_seq* : the two previous approaches are mixed, when we are evaluating a new configuration we go on fixing the elements, but then if the solution is not found in that branch the new configuration to evaluate is picked up as we were running a BFS.

As expected, considering that the solution is a leaf in the tree, a pure BFS is not the best choice. The best approach is strictly related to the structure of the puzzle, for this reason, I have tested the different approaches on the set of last 1000 Sudoku in [1].

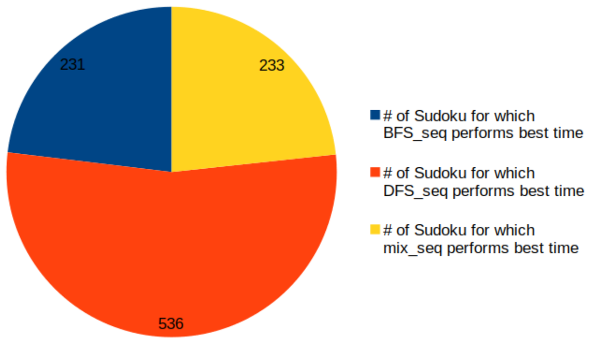
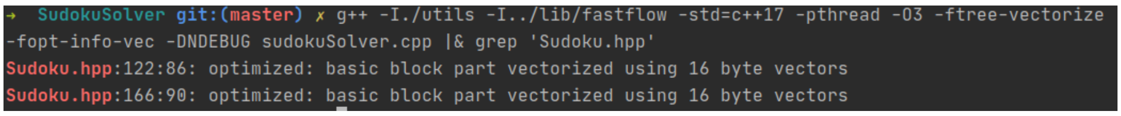


Figure 3: Partial configurations tree

Using the *DFS_seq* function we achieve the best performances on most of the cases, also when it is not the best case the time spent by this approach is not so far from the best result obtained and averaging timings spent to find the solution *DFS_seq* is still the best alternative. Even excluding the simplest Sudoku that can be solved just applying *fix_valid_values* the proportions above are respected.

2.1 Vectorization

brute_fix and *fix_valid_values* are two fundamental functions that are called by all the versions of sudoku solver. The choice of `type uint_16_t` for the type of the integers in configuration structure is because in the two functions is repeated an operation over the elements of a configuration and so vectorization is possible.



```
→  SudokuSolver git:(master) ✗ g++ -I./utils -I../lib/fastflow -std=c++17 -pthread -O3 -ftree-vectorize
-fopt-info-vec -DNDEBUG sudokuSolver.cpp |& grep 'Sudoku.hpp'
Sudoku.hpp:122:86: optimized: basic block part vectorized using 16 byte vectors
Sudoku.hpp:166:90: optimized: basic block part vectorized using 16 byte vectors
```

Figure 4: vectorization

3 Parallel version

Studying the problem, we can observe that is a form of **stream parallelism**, indeed we cannot generate all the configurations and then go through them. Threads generate configurations that represent the task for themselves and the other threads. A thing to observe is that the complexity of tasks varies and so the time spent for its completion is not fixed.

In the next subsections, two versions of implementation using P-thread and one using Fast Flow are presented.

3.1 P-thread

Considering that the best performances are obtained through a DFS visit, the natural step is trying to parallelize that approach. But what happened in sequential version can't be automatically applied in the parallel scheme, indeed in this case using a BFS up to a certain point could differentiate the range in which the solution is searched and allow to reach the solution faster.

3.1.1 DFS parallel version

This version is implemented by the function *parallel_pt_dfs* in `SudokuPTStack.hpp` file. The first thing that's done is setting up a synchronized stack which is implemented by the class `systack` in `stack.cpp` file. The attribute of the class are:

- a `stack`, which contains all the configurations produced by the different threads and it's initialized with the starting configuration of the puzzle.
- a `mutex` and a `condition variable` needed to guarantee mutual exclusion access to the stack.
- an `atomic boolean` which notifies when the stack has to be dismissed and threads don't have to push or pop elements anymore.

Each thread performs a cycle, in this loop, it pops a configuration from the stack and executes the function *fix_valid_values* on it, as in sequential all cells with only one feasible value are filled. At this point, if the configuration is invalid it is deleted and the cycle restarts, if the solution is found, the thread calls the stack's method *dismiss_stack* in order to notify to all the elements to stop their jobs on the stack and an atomic variable shared among threads is set such that represent that a solution is found.

In the third scenario, the thread identifies the cell on which the brute force filling must be applied, it's done in the same manner of sequential version, using *brute_fix*. At this point, all the configurations generated are pushed into the stack except for one, which is kept by the thread itself, in this way the synchronized operation to pop an element from the stack is avoided.

The synchronization and communication among threads are implemented through the use of a shared data structure, we are in the context of **auto-scheduling** indeed each thread menages itself and whenever it fishes its works checks if there is some new task to compute.

3.1.2 Mixed parallel approach

In this case, we have the same approach but the structure shared among thread is a queue, it is implemented by the class `sysqueue` in `queue.cpp` file. The attribute of the class are:

- a `queue`, which contains all the configurations produced by the different threads and it's initialized with the starting configuration of the puzzle.
- a `mutex` and a `condition variable` needed to guarantee mutual exclusion access to the queue.
- an `atomic boolean` which notifies when the queue has to be dismissed and threads don't have to push or pop elements anymore.

The algorithm that is performed by the thread is the same of DFS parallel approach, the different thing is the order in which the configurations are evaluated. Indeed when a value is fixed via brute force the configurations generated are pushed back into the queue except for one which is kept by the thread that continues to be evaluated. Furthermore, each time a thread generates a new configuration the length of the queue is checked, if the number of tasks becomes too high, the tread starts solving sequentially and recursively the configuration. The limit value is fixed in the define `LIMIT_PT` to the value 320, this choice has been chosen experimentally, excluding a proportional value to the number of threads, indeed, considering that the mean branching factor is often 3, fixing that values means that about 5 values have been fixed, which means that lots of constraints are introduced and the sequential computation consequently simplified.

Limiting the number of sequences generated is not considered in DSF version because threads pick up from the stack values that have usually the biggest number of cells already fixed and so they go straight to the solution or to declaring not valid that configuration.

Also in this case the synchronization and communication among threads are implemented through the use of a shared data structure, a `queue`. This approach has been tested using a non-blocking multi produced and multi consumer non-blocking queue, which was developed for the assignment image processing, however, in order to work well in this scenario some modifications have been introduced and the effect has been that the non blocking aspect has been lost. This experiment let me understand how difficult is working with non blocking synchronization, and let me appreciate the simple way to use this mechanism for free in fast flow communication among threads.

3.2 Fast Flow

The parallel version implemented using Fast Flow is based on the farm `ff::farm`, it is implemented by the *parallel_ff_solve* in `SudokuFF.hpp` file . The skeleton used is that of master-worker,which is illustrated in figure 5, so given a parallelism degree n the farm will have $n - 1$ workers.

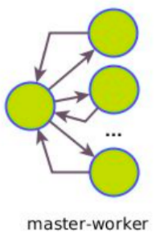


Figure 5: Farm skeleton

The is no Collector since the Emitter acts like a it. In particular it acts like a Coordinator and the parallel computation works in the following way: first of all the `main` thread create the Emitter and the Workers. The Emitter has type: `struct Emitter : ff_monode_t<Task>` , the Workers have type `struct Worker : ff_node_t<Task>` . The `Task struct` is implemented as follow.

```
struct Task{

    configuration *conf;
    bool solution = false;
};
```

Listing 2: Task struct

The default scheduling of the tasks in a Fast Flow farm is Round Robin but in this case that is not a good solution because each task can differ a lot in terms of time spend to compute it, and so there could happen unbalanced distribution of tasks. Also, the *set_scheduling_ondemand(n)* possibility is not so good, indeed when the queue has free space it means the Worker is "ready" and new job is assigned to him even if itself or some other worker found the solution. So this solution takes a little more than a real on demand scheduling which has been implemented.

When the Emitter is initialized it counts the number of output channels, and sets up a boolean vector which contains whether a worker is ready or not. The first time the Emitter receives a task, it is from main and so select a ready worker through the apposite method *selectReadyWorker()*, and send the task to it. When it receives a new task from one of the feedback channel from the emitter, it checks the field solution of the task, if it is true this is interpreted as an `EOS`, indeed at this point the Emitter has to stop sending Task to the workers. If the field solution is false means that a configuration was not valid and so the emitter performs a cycle to send new tasks to ready workers, always preferring the deeper configuration in the available set. In case the number of configuration generated starts to begin to much there is the risk that the overhead of the communication becomes consistent and so a limit `LIMIT_FF` is fixed, after the generation of 320 configurations the Emitter sends a task to a Worker with the solution field set to true, so the worker understands that has to solve the task sequentially.

When a worker receives a task he starts the usual cycle calling the function *fix_valid_values*. After this the new configurations generated via brute force are sent to the Emitter, except for one which is kept by the worker itself. This worker is considered busy by the Emitter until the worker itself send the task that notifies that a solution is found or that is not possible with that configuration.

4 Experiments

To run the code it is needed to create a new directory called `lib` and clone in it the FastFlow code (https://github.com/fastflow) and then run the script: `/ff/mapping_string.sh`. The command to compile is:

```
g++ -I./utils -I../lib/fastflow -std=c++17 -pthread -O3 -DNDEBUG sudokuSolver.cpp
```

Once the executable has been obtained the parameters needed are the following:

- $filename$: name of the file that contains starting configuration of Sudoku (mandatory). A starting configuration is a string of 81 digits,it contains the digits of the sudoku from the first row to the last, from left to right. For the empty cells the digit used is 0. An example of a valid configuration is:

  ```
  400007000000000020000001000003050060000800900000000000401200000700008000016002000000
  ```

- n : Parallelism degree (mandatory)
- mod : a string between $all, seq, pt\_dfs, pt\_mix, ff$ (default all).
- *repetition* : Number of times the experiment must be repeated (default 1).

- *print* : Boolean for printing the starting configuration and the solution after each execution(default false).

The script used for experiments is in the file `experiments.sh`. The Sudoku used to test the different implementations are in the file *tests.txt*. The most difficult aspect of the Sudoku problem is that they differ a lot one each other, a bad implementation to find the solution of one of it could obtain the best performance on the others.

For this reason, to show the result an average on 10 Sudoku is done, these Sudoku have been chosen randomly among that were solved from 1 to 5 seconds with the sequential algorithm on the Xeon phi, indeed these are more interesting to observe those that can be solved in hundreds of milliseconds. The following graph shows the speedup that we can obtain with the parallel mix approach (*pt_mix*), parallel DFS (*pt_dfs*), and with Fast Flow implementation.

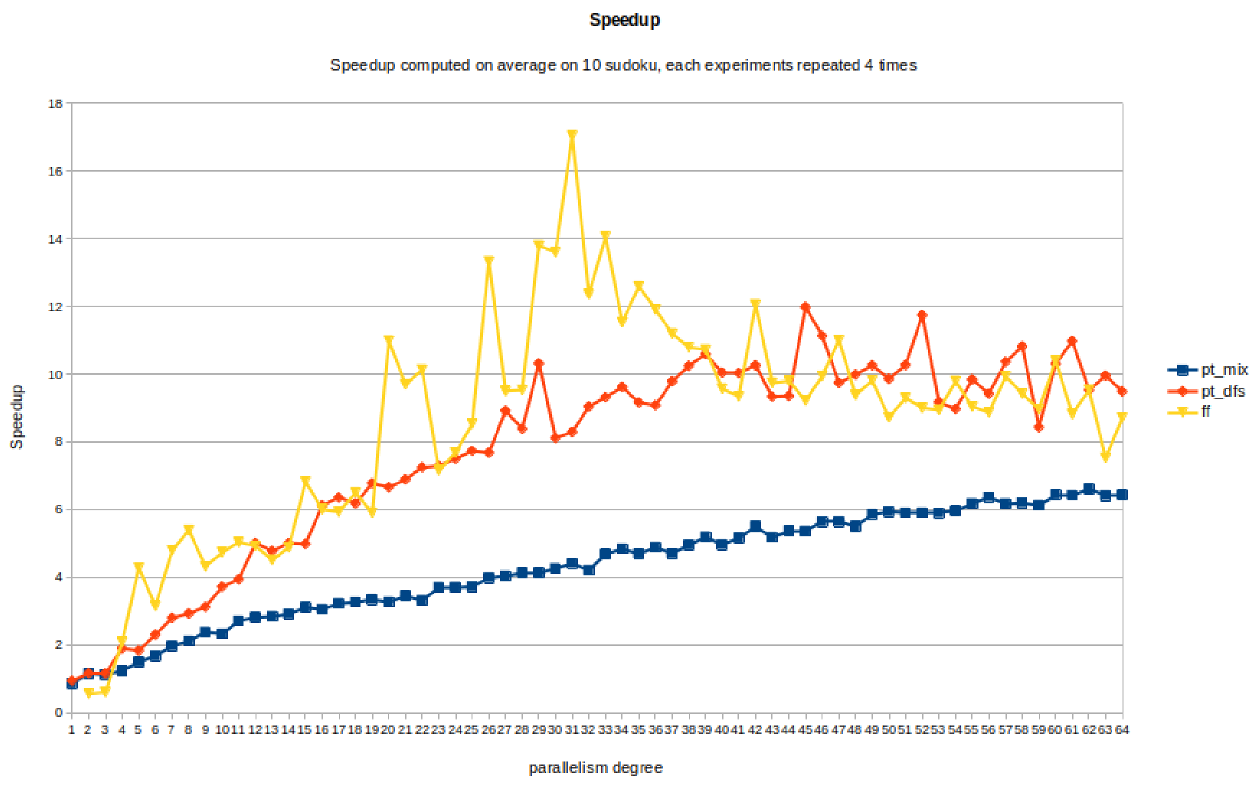


Figure 6: Speedup

As we can see, the best performances are obtained by Fast Flow implementation up to 50 threads and then by *pt_dfs* implementation but always comparable with Fast Flow performances. The great gap is among the two different approaches and we can say that visiting the tree of configuration through a deep search is the winning one.

References

[1] `https://staffhome.ecm.uwa.edu.au/~00013890/sudoku17`.

[2] Gary McGuire, Bastian Tugemann, and Gilles Civario. “There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem”. In: *CoRR* abs/1201.0749 (2012). arXiv: `1201.0749`. URL: `http://arxiv.org/abs/1201.0749`.