# CPD Report
# MAXSAT Solver

Group 17
Helder Duarte N 84982
Samuel Gomes N 76415
Rui Santos N 76384

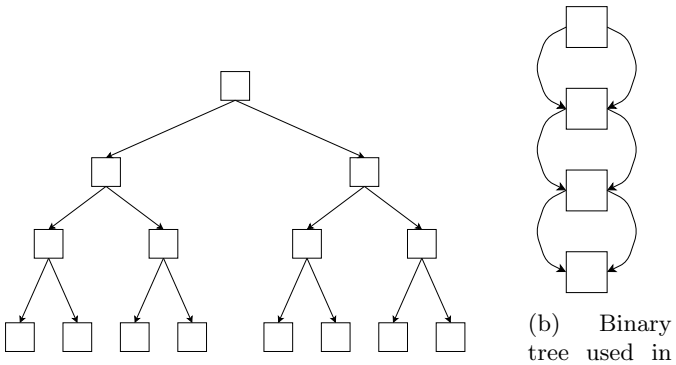
August 28, 2020

# Chapter 1

# Serial implementation

## 1.1 Data structures

After parsing the input, a tree (struct *tree_t*) is created to help search through the various possible attributions, in which each level corresponds to a variable. Each item of the tree (struct *tree_item_t*) corresponds to the attribution of a given variable, using a pointer to represent a negative attribution (*struct tree_item* falsePath* pointer) and another (*struct tree_item* truePath* pointer) for a positive attribution.
Then a matrix is created (*int** *mainHash*) to store one clause per row (the values in the columns are equivalent to the attributions given in a certain clause).
In order to reduce memory footprint, since all nodes on the same level will be the same, instead of an usual tree structure, as in Fig 1.1a, only one node per level exists, effectively changing the tree to what can be seen on Fig 1.1b. With larger examples, this changes from being a way to more efficiently store the tree to an absolute requirement, for example, on a problem with 400 variables, the tree would need to have $2^{400}$ nodes, which is impractical.



(a) Regular binary tree.

(b) Binary tree used in the project.

Figure 1.1: Binary tree data structure.

## 1.2 Solution description

To traverse the tree a recursive DFS was implemented. The conditions in each node are tested for the current variable.
The function *void treeDFS* is used to initialize the search and call a recursive auxiliary function *void DFSAux*. This function has two main states:

- While the current node is not a leaf, the attribution's *sat* value is updated (using the *int* testConditions* function), the auxiliary array is updated and it tests if the current state is eligible for prunning (in that case it returns immediately to the previous node);
- If the current node is a leaf the maxSat value is updated (if the current sat is bigger than the *maxSat*, the *maxSat* is overwritten) and the number of times the value occurs is also updated (for the case the current *sat* value is equal to

the *maxSat*).

An auxiliary array (*partialSatisfiedConditions*)was created to divide the clauses in three groups:

- **0**: clause not yet satisfied
- **1**: clause satisfied (at least one of the variable satisfy the condition)
- **2**: clause not satisfied (none of the variables satisfy the condition)

This array is updated in the function *testConditions()* and it is called for each node.

## 1.3 Pruning

In order to make the search faster and more efficient, a pruning operation is made. It consists of checking whether the number of unsatisfied clauses up to that point is greater than the number of clauses yet to be tested. If this happens, then there is no solution in the current path. Therefore, the search can be stopped, cutting a lot of unnecessary searching.

# Chapter 2

# Shared memory implementation with OpenMP

The biggest problem we faced when trying to parallelize the sequential implementation was the load balancing issue. We wanted to dynamically separate the tree search depending on the number of threads specified. For that to happen, the partial tree would have to be calculated dynamically based on the number of the current thread. Our solution consists in three main concepts:

- First, we add a new attribute to each node called carry, which starts at half the number of threads and is the result of the division of the previous value of it by 2. The objective of this number is to help in further calculations.
- Then, when the tree is being traversed, the modulus of the carry by the thread number is compared with half of the carry. This divides the threads in two groups (one for the falsePath - the calculation is smaller than $carry/2$; and one for the truePath - the calculation is greater than $carry/2$).
- Finally, when the carry is equal to 1, we know that there is only one more possible division left. In that case, we compare the thread number with $\frac{numberofthreads}{2}$ and associate one path to one of them and the other path with the other.

This method breaks the search statically, i.e., the breaking is predetermined, and adds a particular subtree search to each particular thread, as seen in Fig 2.1, for an example of 4 threads.
A consequence of having several threads accessing the partial result variables at the same time was the occurrence of race conditions. To avoid this problem, global variables were replaced by thread private variables to store the partial results (which were passed to the functions and updated by reference).
To combine the results calculated by each thread, to the threads that give the greatest maximum SAT value, it is added to the SAT value the number of times that result occurred in that thread.
*Note: The calculation of the assignment is done at the end of the search rather than before.*

Table 2.1 lists the computation time of each exercise, of both parallel and serial versions. The parallel version manages, at most, a speedup of 2.4 times when compared to the serial version, using 4 threads.
Although these results show big improvements, they shy away from the expected results. This is due to a lack of dynamic load balancing. The threads that finish sooner due to prunning, do not help the others. This spawned several efforts to introduce dynamic load balancing, ranging from completely altering the tree traversal algorithm to an iterative BFS with the use of a queue, which due to the number of nodes in the tree proved impossible (the memory usage exploded), to implementing a complete tree rather then the single node per level tree, in order to include node specific data, which, similar to the BFS, provoked huge memory utilization.

Fig 2.2a shows a logarithmic plot of all the timings. To better point out the differences, Fig 2.2b and 2.2c show the scalar plots of examples 1 and 2, and 3, 4 and 5, respectively. After analyzing the plots, the overhead of the parallel version is clear in the first two examples, these being small examples where the partitioning of the work has a negative influence on the efficiency of the program. With larger examples, the gain of parallelization becomes noticeable.

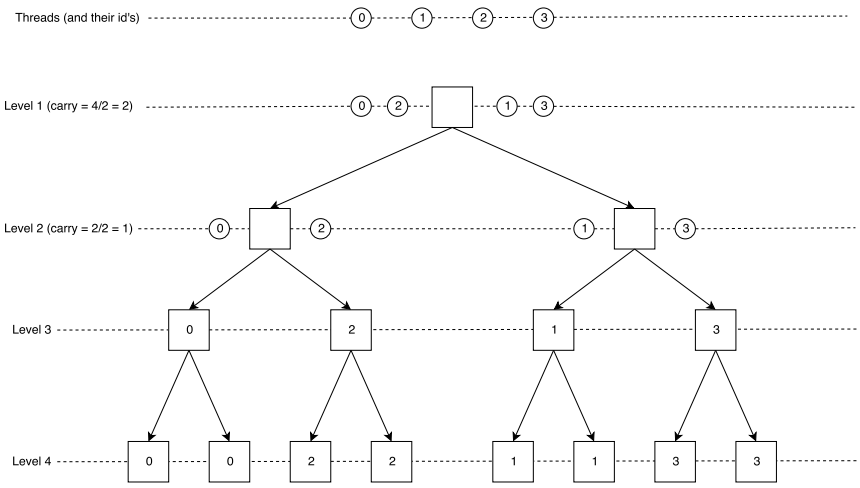
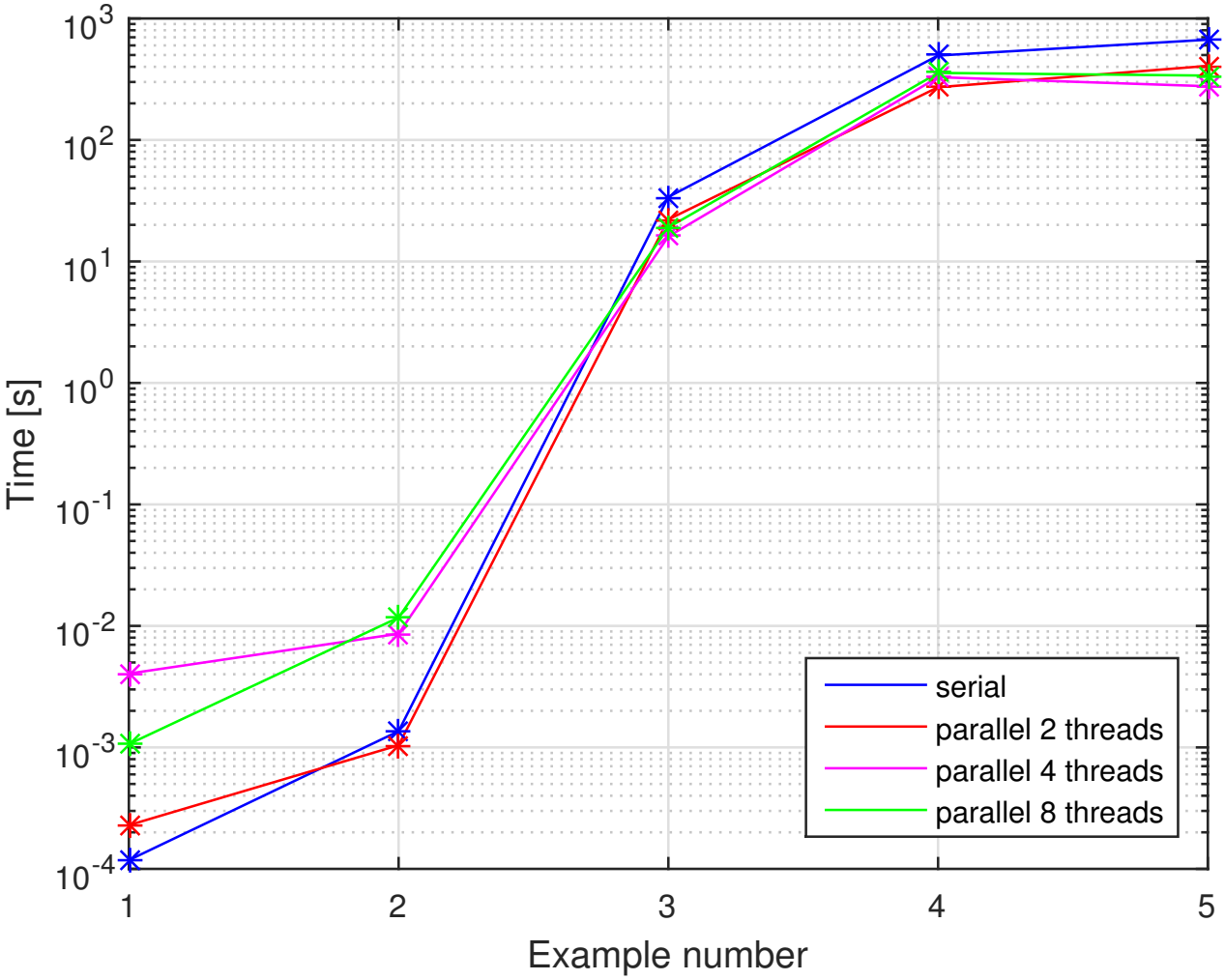


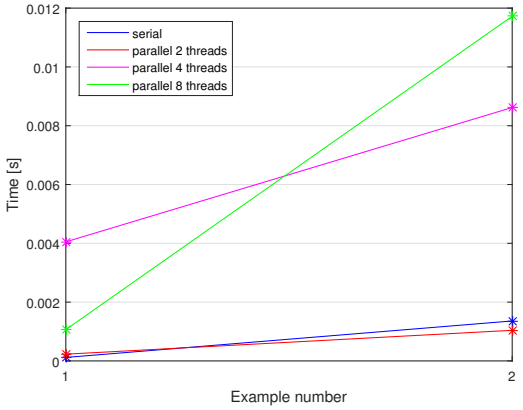

Figure 2.1: Diagram representing search thread division.

Table 2.1: Timings (the 8 threaded results were obtained on a 4 core processor).

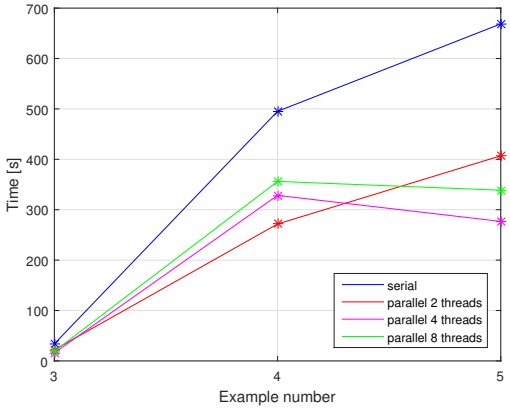
| | Serial | Parallel | | |
|---|---|---|---|---|
| | | 2 threads | 4 threads | 8 threads |
| ex1 | 118 us | 229 us | 4.05 ms | 1.06 ms |
| ex2 | 1.36 ms | 1.04 ms | 8.62 us | 11.7 ms |
| ex3 | 33.9 s | 22.2 s | 16.0 s | 19.0 s |
| ex4 | 495 s | 271 s | 328 s | 356 s |
| ex5 | 668 s | 406 s | 277 s | 338 s |




(a) Logarithmic plot of the timings.




(b) Scalar plot of example 1 and 2.




(c) Scalar plot of example 3, 4 and 5.

Figure 2.2: Plots of timings.