Parallel Satisfiability Solver Based on Hybrid Partitioning Method

***Abstract*—This paper presents a hybrid partitioning method used to improve the performance of solving a Satisfiability (SAT) problems. The principle of our approach consist firstly to apply a static partitioning to decompose the search tree in finite set of disjoint sub-trees, than assign each sub-tree to one computing core. However it is not easy to choose the relevant branching variables to partition the search tree. We propose in this context to partition the search tree according to the variables that occur more frequently then others. The advantage of this method is that it gives a good disjoint sub-trees. However, the drawback is the imbalance load between all computing cores of the system. To overcome this drawback, we propose as novelty to extend the static partitioning by combining with a new dynamic partitioning that assure a good load balancing between cores. Each time a new waiting core is detected, the dynamic partitioning selects automatically using an estimation function the computing core which has the most work to do in order to partition dynamically its sub-tree in two parts. It keeps one part and gives the second part to the waiting core. Preliminary result show that a good speedup is achieved using our hybrid method.**

***Keywords*-Parallelism; Satisfiability; Load balancing; Scheduling;**

I. Introduction

The past few years have seen enormous progress in SAT solving. Among others, this is due to evolution of hardware architectures as multi-core and Many Integrated Cores (MIC) machines. In this context, several parallel SAT solvers [4], [1], [9], [16] have been proposed. They are mainly based on two approaches:

- Search tree partitioning called Divide and conquer partitioning: the principle consists in decomposing the unique search tree generated by one search algorithm in a set of sub-trees, then schedule sub-trees between the different computing cores of the system. The partitioning of the search tree can be done using two methods: static or dynamic. We refers as example studies presented in [23], [5], [19].
- Portfolio parallelization: the principle consists in executing at the same time several search algorithm, then the first algorithm that finds a solution stops all the others. We refer as examples studies presented in [10], [20], [11], [3]

The contribution of this paper concerns the search tree partitioning approach. We aim at proposing a new hybrid method that combines static and dynamic partitioning to enjoy the benefits of each partitioning method. The static partitioning consist to partition the search tree in finite set of disjoints sub-trees, then affect each sub-tree to one computing core to be explored. This method has two major challenges: (1) *Choosing the partition variables*: for a given large SAT instance with hundreds of thousands of variables it is difficult to find the most relevant set of variables to divide the search space, and (2) *load balancing*: for some sub-problems it is easier to prove (un)satisfiability that others. Since the time needed to prove (un)satisfiability of sub-problems cannot be predicated, the work cannot be balanced prior to search. It is possible that some processors might be quickly idle while others take a long time to solve their sub-problems.

In our approach, we propose to choose for the static partitioning the variables that occur more frequently then others. The benefit of this method is that it selects variables which are a good candidates to partition the search tree in disjoint sub-trees. Then, to overcome the drawback of the load balancing problem, we propose as novelty to combine the static method with a new dynamic partitioning method. The advantage of our dynamic partitioning is to have a total control on the parallel search and the load balancing. The particularity is that each time a new waiting core is detected, the dynamic partitioning selects automatically using an estimation function the best working core which has a big amount of work to do in order to generate a new work (sub-tree). When the working core is selected, it partitions its sub-tree in two parts: it keeps one part and sharing the second part with the waiting core.

Our solution is proposed by doing an external parallelization of Glucose SAT solver [2]. That means the parallelization is not done inside the search algorithm. It is realised with the contribution of several sequential Glucose solvers. Glucose is an open source solver which has recently won the SAT competition 2015 [18].

The rest of this paper is structured as follows. Section II gives some preliminaries about solving SAT problems. Section III presents some research work related to our contribution. Section IV presents our hybrid partitioning method. Section V presents experimental results that shows the improvements of our partitioning method. Finally, section VI contains the concluding remarks and provides directions for future work.

II. Preliminaires

SAT problems are commonly represented in Conjunctive Normal Form (CNF). A CNF formula F [7] is represented by a set of boolean variables V ($V = v_1, v_2, v3, ..., v_n$) and a set of clauses C ($C = c_1, c_2, c_3, ..., c_k$). For each variable v_i it exists two literals: v_i and $\neg v_i$ called positive and negative literals. Each clause $c_i \in C$ is represented by a set of literals that is a disjunction ($v_1 \vee v_2 \vee v_3, ..., \vee v_i$). We say that the formula F is *satisfied*, if it exists an assignment π that satisfies all clauses C. π satisfies a clause c_i if it contains at last one positive literal. If π doesn't satisfy one clause c_i ($c_i \in C$) the formula F is said *unsatisfiable*. If each clause contains up to k literals, the problem is called k-SAT.

In the literature, many academic and industrial problems [6] [13] are encoded as SAT problems then solved by a SAT-solver. Generally encoding SAT problem is easy then writing a particular algorithm for a specific problem.

For solving any SAT problem, first, all variables are unassigned. For each step, a variable v_i is chosen and it will be assigned with boolean value in turn. Each branch of a search tree computed by this search defines an assignment. Next, the propagation mechanism checks the consistency of the partial assignment of variables with a set of clauses in order to generate a new node in the search tree. If all clauses are satisfied the node represents a solution, otherwise the node represent a failure and a back track is applied.

For example, for solving the problem γ which is modelled using two variables (v_1 and v_2) and two clauses ($v_1 \vee \neg v_2$ and $\neg v_2$), several algorithms or heuristic exists. The main difference between them is the choice of the branching variables and the choice of the first value assigned to each branching variable. Figure 1 shows an example of tree search generated to solve the γ problem by using an heuristic that selects the first unassigned variable and assigns it in first by true value. This search tree is generated as follows:

- The first selected variable is v_1 and it is assigned with the true value. In this step, only the first clause is satisfied ($v_1 \vee \neg v_2$)
- In the second step, the variable v_2 is selected and assigned with true value. In this case the node represents a failure because the second clause ($\neg v_2$) is not satisfied,
- In the third step a back track is applied and the variable v_2 is assigned with false value. In this step a solution is found that satisfies all clauses.

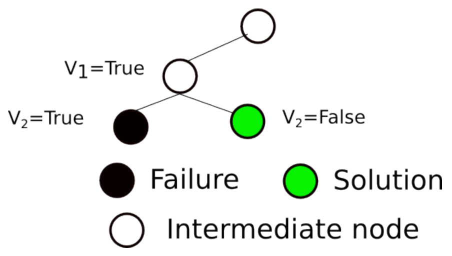

Figure 1. Search tree generated to solve a SAT problem

III. Related Work

During the last few decades, there has been considerable evolution in the innovations of hardware architectures as multi-core parallel machines and parallel techniques used to solve combinatorial problems using SAT solvers. In this context, different studies are presented, including but not limited to works presented in [22], [17], [5], [12], [10], [21], [16], [14]. The majority of these studies can be classified according to the parallelization approaches: search tree partitioning, Portfolio or combining search tree partitioning and Portfolio parallelization.

As example of search tree partitioning approaches we mention the following studies:

Plaza et al. [17] propose an approach based on a static partitioning. The principle consists in partitioning the search-tree in finite set of sub-trees. Then explore each sub-tree using one core. To choose the branching variables, the authors propose to use the VSIDS [15] heuristic. Additionally, it implements a parallel processing based on recursive learning.

Böhm et al. [5] proposes to partition the search tree using a dynamic partitioning to decompose the input formula into disjoint sub-formulas. Then, each sub-formula is solved by a sequential SAT solver. To modify the boolean formulas, the authors propose to use an optimized data structures. It's proposed also to use an efficient workload algorithm to assure a good load balancing between cores.

Jurkowiak et al. [12] proposes to apply the work stealing technique to assure a good load balancing between all computing core. The approach proposed in this study is based on the master/slaves communication model. The master is responsible for distributing the sub-trees among the slaves. Then, each slave executes one sub-tree.

As example of Portfolio approaches we mention the following studies:

Xu et al. [21] proposes the *Satzilla* solver which is a Portfolio-based algorithm selection. Satzilla is a portfolio-based solvers which have been proposed in the sequential context. It consists in running several solvers, then the first one which finds a solution stops others.

Hamadi et al. [10] proposes the *ManySAT* solver which is a Portfolio of complementary sequential algorithm. The novelty of this solver in the context of Portfolio parallelization is the sharing of clauses between the solvers to improve the global performance of the system.

As example of studies which combines search partitioning and Portfolio we mention the following works:

Ohmura et al. [16] proposes to use for the search partitioning approach a master-slave architecture. The master handles clauses sharing, deletion of redundant clauses and the dynamic partitioning of the search trees, while the slaves perform search in each sub-tree using different search algorithms heuristics.

Martins et al. [14] proposes to use at first the static partitioning to decompose the search tree in a set of sub-tree. This method is based on the VSIDS heuristic to select the branching variables. Then, switch to a portfolio approach when load balancing becomes an issue or when a cutoff is reached.

All of the previous research studies proposed in the context of search tree partitioning use only one method (static or dynamic) to decompose the initial search tree and schedule sub-trees between the different cores. In contrast to these related studies, our work propose a hybrid algorithm that combines static and dynamic partitioning. This hybrid method is presented in the next section.

IV. Hybrid Partitioning Method

The contribution of this paper is based on proposing a new hybrid partitioning method which performs a parallel search on multi-core machine to improve the performance of solving SAT problems. The goal of this method consists in overcoming the problem of the load balancing obtained by using a static partitioning. The principle consists to start the search by using static partitioning in order to partition the search tree in a good set of disjoint sub-trees. Then in a second step, apply a dynamic partitioning to obtain a good load balancing between different cores of the computing system. In what follows we present in detail the principle of the static and the dynamic partitioning.

A. Static Partitioning

This method aims at decomposing the initial search tree in finite set of sub-trees, than explore each sub-tree using one computing core. All sub-trees are generated such as they are disjoints and each sub-tree contains a specific guiding path which is different form other. The guiding path is generated by assigning all the branching variables with a particular boolean value. However, the problem is how we choose the branching variables to construct the initial guiding path. Plaza et al. [17] proposes to sequentially run a master thread for a short time to allow the VSIDS [15] heuristic to generate a new information which can be used to select variables. In this paper we use an other method which is also used by Gil et al. [8]. It consists in choosing the variables that occur more frequently then others.

In our context, the number of partitioning variables is fixed according to the number of computing cores used in the parallel search. For example using p computing cores, we select n partitioning variables which occur more often than other variables and which are unassigned ($n = floor(\log_2(p))$). Then, this selected variables are used to generate n^2 sub-problems with a different guiding path. For each sub-problem a new static node is generated that represents a structure which contains the original problem with all variables, all clauses and the new guiding path. Then, this new static nodes are inserted in a Global Priority

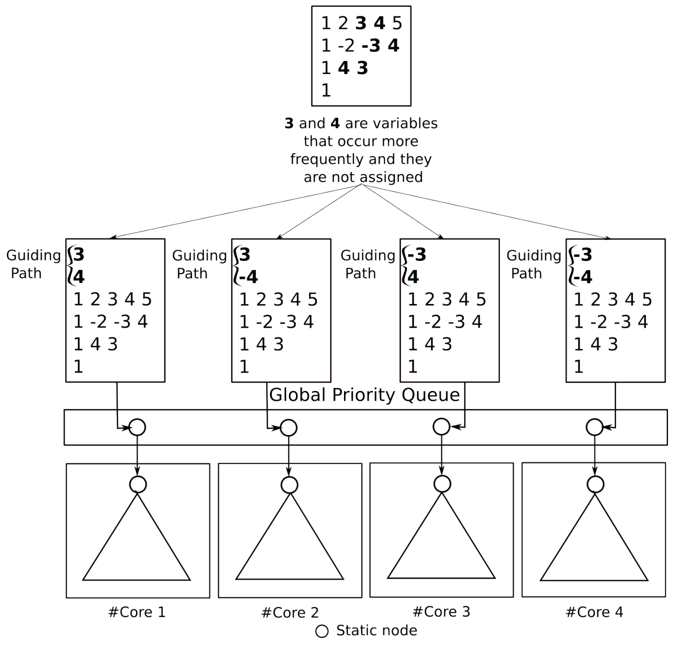

Figure 2. Static Partitioning

Queue (*GPQ*) which is a global pool of nodes shared between all computing cores. Then, each core takes one node and starts the search.

Figure 2 shows an example of using static partitioning with 4 computing cores. We start by selecting 2 ($floor(\log_2(4))$) partitioning variables, in our case the variables which occur more often than other are variables 3 and 4. In reality, variable 1 is the most occur variable, but it is not selected as a branching variable because in the initial problem the variable 1 is assigned with true value. After fixing the branching variables, we generate new sub-problems with a unique guiding path for each one. Each node is encapsulated in a static node and inserted in the *GPQ*. Finally, each core start the exploration of the search with one static node.

The advantage of static partitioning is that partitions the search tree in a good disjoint sub-trees. But the drawback is the load balancing between computing cores. For some problems it is easier to prove unsatisfiability than others. When a core finish its work and there is no new sub-tree to explore, it waits until a solution is found or all working cores finish their work. This waiting time has a negative effect on the global performance of the system. To solve this problem we propose to extend the static partitioning with a dynamic partitioning in order to generate, dynamically, new works (sub-trees) each time a waiting core is detected.

B. Dynamic Partitioning

The goal of the dynamic partitioning is to give a good load balancing between all computing cores. This principle is presented in algorithm 1. Each time a new waiting core is

Algorithm 1 Dynamic Partitioning

```
$GPQ$, a Global Priority Queue shared between all cores
$C_x$, working core which have the most work to do compared to the others cores
if $\exists$ at last one waiting core and the current working core is $C_x$ then
    Stop the search
    Create a new dynamic node
    Insert the dynamic node in the $GPQ$
    Restart the search
else
    Continue the exploration of the sub-tree
end if
```

detected, the algorithm selects using an estimation function the working core which has the most work to do compared to the other working cores to generate a new sub-tree. We propose to select the core which has the most work to do because it is the best candidate to share his work. Then, the selected working core fixes his next branching variable to true value and generate a new dynamic node. The dynamic node contains all the variables assigned with the working core and the next branching variable assigned to false value. The new dynamic node is inserted in the *GPQ* to be shared with the waiting core that takes effects by the insertion of a new dynamic node in the *GPQ*. However, if no waiting core exists, all the working cores perform a sequential exploration of the sub-tree search.

To have a good performance and improve the cooperation between all workers, each time a computing core finishes it works it saves all the learned clauses in a data base, then when a new core restarts the search with a new dynamic node, it picks up the clauses saved in the data base and starts the search. The exchange of learned clauses between computing cores is very efficient to reduce the search space and to improve the global performance of the system.

The estimation function used in this dynamic partitioning is based on the number of assigned variables after the propagation mechanism. Each time we need to select a core to generate a new work, we select the core which has the smallest estimation. That means we select the core that has the big number of unsigned variables and which probably has more work to do in the future than others cores.

1) Example of Progress of Dynamic Partitioning: Figures 3 and 4 show the progress of the dynamic partitioning. In figure 3: cores 1, 3 and 4 explore a static sub-tees where core 2 is idle. To give a new work to core 2, figure 4 shows the mechanism. As core 1 is the working core which has the smallest estimation of work to do compared to the others cores (estimation=20), it assigns the next branching variable v selected by the local search algorithm to true value (v=true). Then, core 1 creates and inserts in the *GPQ* a new dynamic node that contains all necessary informations to

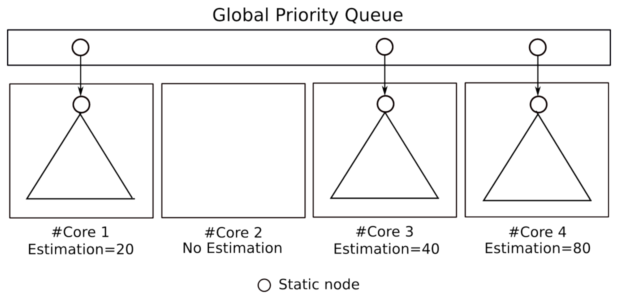

Figure 3. Dynamic partitioning: first step

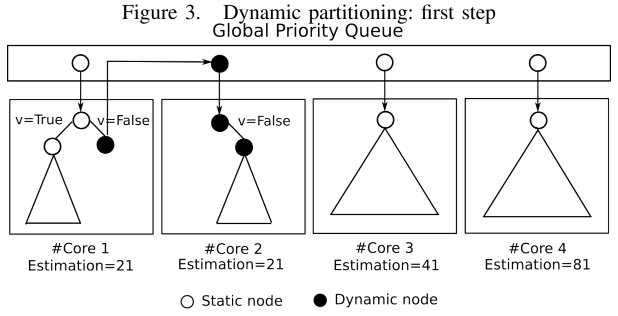

Figure 4. Dynamic partitioning: second step

restart the search by an other core. The core 2 takes this new dynamic node and starts the search. By following this mechanism, we can say that the dynamic partitioning gives a good load balancing between all cores by creating new work each time a waiting core is detected.

V. Experementations

To validate the approach used in this study, we realised a set of experiments on a linux system. The used parallel machine was a bi-processor Intel Xeon X5650 (2.67 GHz) with Hyper-Threading technology (6 physical cores for 12 *threads*) and 48 GB of RAM. The Glucose version used in this experimentation is Glucose 4.0 [2]. All solved problems were proposed in the SAT Competition 2015 [18] and modelled using "DIMACS CNF" format, which is a simple text format [7].

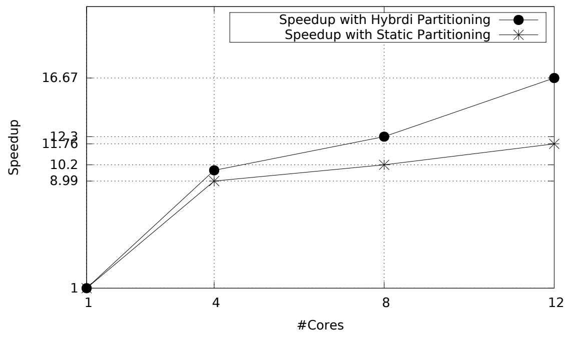

Figure 5. Average speed-up for solving 27 SAT problems using hybrid partitioning method

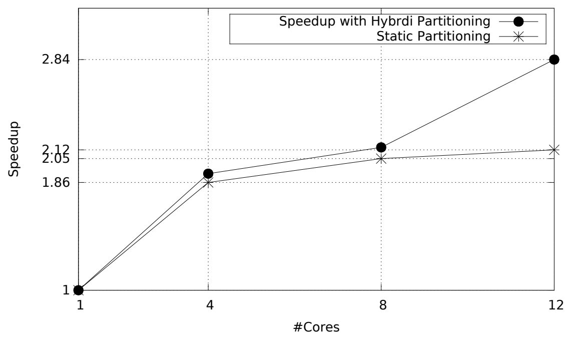

Figure 6. Average speed-up for solving 17 UNSAT problems using hybrid partitioning method

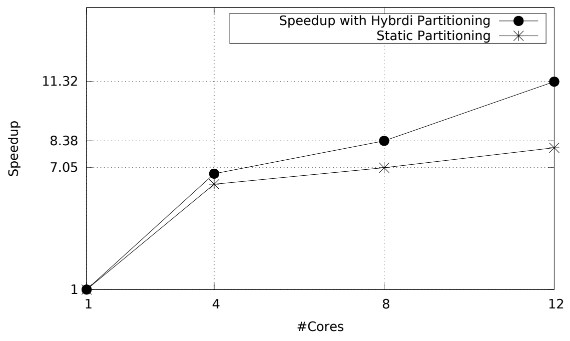

Figure 7. Average speed-up for solving 27 SAT + 17 UNSAT problems using hybrid partitioning method

Figures 5 (resp. figures 6) shows a comparison between the average speedup obtained using a static and hybrid partitioning methods for solving 27 SAT (resp. 17 UNSAT) problems. All solved instances in this experimentation refers to the instances which are solved in less then 10800 seconds (3 hours) with different configurations (1,4, 8 and 12 cores). It is clear that our hybrid partitioning method gives a good performance compared to the static partitioning, a speedup of 18.96 is reached for solving 27 SAT problems with 12 cores and a speedup of 2.73 is reached for solving 17 UNSAT problems with 12 cores. Figure 7 shows an average speedup for solving a set of 27 SAT and 17 UNSAT problems.

A. Comparison between the hybrid partitioning and other externals solvers

To validate our approach, figures 8 (resp. 9) shows a comparison between speedup obtained using Plingling solver, Treengling solver, Glucose solver and our hybrid partitioning method for solving 11 SAT (resp. 6 UNSAT) problems. All solved instances in this experimentation refers to the instances which are solved at the same time in less then 10800 seconds (3 hours) with different configurations (1,4, 8 and 12 cores) using Plingling solver, Treengling solver, Glucose solver and our hybrid partitioning method. According to the result of SAT competition 2015 [18] for the parallel track, Glucose solver was ranked as the best solver, Treengling solver was ranked in the second position and Plingeling was ranked in the third position.

Figure 10 notes that in generally our hybrid method give a good performance for solving SAT and UNSAT problems.

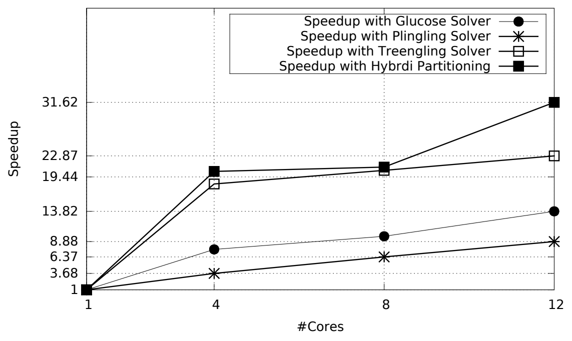

Figure 8. Comparison of average speed-up obtained for solving 11 SAT problems with Plingling solver, Treengling solver, Glucose solver and hybrid partitioning method

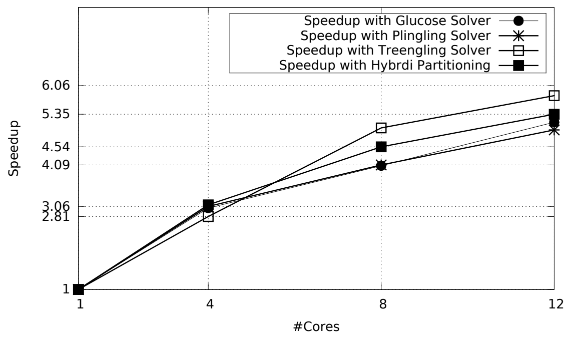

Figure 9. Comparison of average speed-up obtained for solving 6 UNSAT problems with Plingling solver, Treengling solver, Glucose solver and hybrid partitioning method

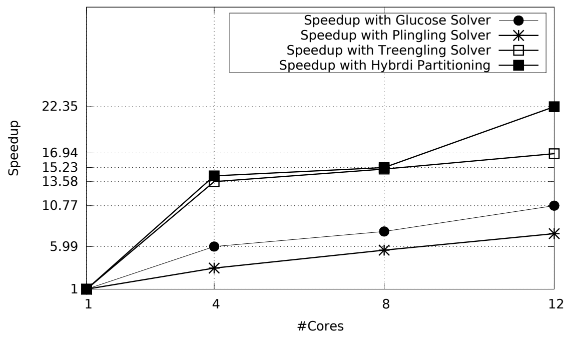

Figure 10. Comparison of average speed-up obtained for solving 11 SAT + 6 UNSAT problems with Plingling solver, Treengling solver, Glucose solver and hybrid partitioning method

B. Load Balancing

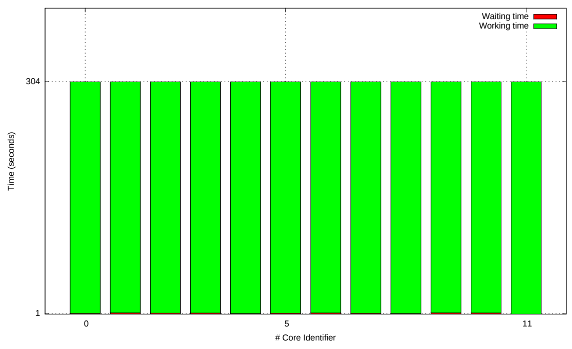

Figure 11. Load balancing for solving SAT problem (50bits_10.dimacs.cnf) using 12 computing cores

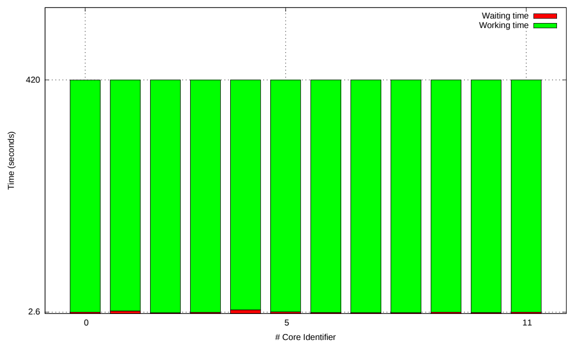

Figure 12. Load balancing for solving UNSAT problem (manthey_single-ordered-initialized_w48_b9.cnf) using 12 computing cores

Figures 11 and 12 shows the behaviour of the computing cores when we solve SAT and UNSAT problems using our hybrid partitioning method. As result, all cores worked and waited for an equivalent amount of time.

VI. Conclusion

This paper presents a parallel SAT solver based on hybrid partitioning method that combine a static and dynamic partitioning. Static partitioning is applied to generate a good disjoint sub-trees, then dynamic partitioning is apply to assure a good load balancing between computing cores. This hybrid method is tested with shared memory architecture to solve 44 problems, and a good speed-up is achieved. As a first perspective, we propose to adapt the development of our hybrid method for distributed memory architecture and cloud computing to performs a test with industrial problems using high computing power.

In this paper, our contribution is proposed only in the parallelization of one search tree generate by one search algorithm. However in the Portfolio parallelization, several search algorithms are executed, then the first algorithm that found a solution stop others. As a second perspective we propose to apply our hybrid method for each search algorithm in the Portfolio parallelization.

As a third perspective, it is interesting to mix the parallel SAT and Constraint Programming (CP) solvers in the same framework to extend the set of combinatorial problems that can be solved by the same framework.

References

[1] G. Audemard, J.-M. Lagniez, and L. Simon. Improving glucose for incremental sat solving with assumptions: Application to mus extraction. In *16th International Conference on Theory and Applications of Satisfiability Testing(SAT'13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 309–317. Springer, 2013.

[2] G. Audemard and L. Simon. Glucose sat solver. http://www.labri.fr/perso/lsimon/glucose. Accessed: 08-10-2016.

[3] T. Balyo, P. Sanders, and C. Sinz. *HordeSat: A Massively Parallel Portfolio SAT Solver*, pages 156–172. Springer International Publishing, Cham, 2015.

[4] A. Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT Competition*, pages 51–52, 2013.

[5] M. Böhm and E. Speckenmeyer. A fast parallel sat-solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(2):381–400, 1996.

[6] R. Béjar and F. Manyà. Solving the round robin problem using propositional logic. In *AAAI/IAAI*, pages 262–266. AAAI Press / The MIT Press, 2000.

[7] DIMACS challenge. Satisfiability. Suggested format., 1993.

[8] L. Gil, P. Flores, and L. M. Silveira. Pmsat: a parallel version of minisat. *ournal on Satisfiability, Boolean Modeling and Computation, vol. 6*, 2008.

[9] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2008.

[10] Y. Hamadi and L. Sais. Manysat: a parallel sat solver. *JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT)*, 6, 2009.

[11] H. Hoos, K. Leyton-Brown, T. Schaub, and M. Schneider. Algorithm configuration for portfolio-based parallel sat-solving. In *Workshop on Combining Constraint solving with Mining and Learning (CoCoMiLe)*, 2012.

[12] B. Jurkowiak, C. M. Li, and G. Utard. Parallelizing satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9:174–189, 2001.

[13] H. A. Kautz, D. A. McAllester, and B. Selman. Encoding plans in propositional logic. In *KR*, pages 374–384. Morgan Kaufmann, 1996.

[14] R. Martins, V. Manquinho, and I. Lynce. Improving search space splitting for parallel sat solving. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 336–343, Oct 2010.

[15] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[16] K. Ohmura and K. Ueda. c-sat: A parallel sat solver for clusters. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 524–537. Springer Berlin Heidelberg, 2009.

[17] S. Plaza, I. Kountanis, Z. Andraus, V. Bertacco, and T. Mudge. Advances and Insights into Parallel SAT Solving. In *IWLS 2006: 15th International Workshop on Logic and Synthesis*, June 2006.

[18] Sat competition 2015
http://baldur.iti.kit.edu/sat-race-2015/index.php?cat=results.
Accessed: 08-10-2016.

[19] P. Vander-Swalmen, G. Dequen, and M. Krajecki. A collaborative approach for multi-threaded sat solving. *International Journal of Parallel Programming*, 37(3):324–342, 2009.

[20] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32(1):565–606, June 2008.

[21] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32(1):565–606, June 2008.

[22] H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: A distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4-6):543–560, June 1996.

[23] H. Zhang, M. P. Bonacina, M. Paola, Bonacina, and J. Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.