Noname manuscript No.
(will be inserted by the editor)

Mixing Static and Dynamic Partitioning to Parallelize a Constraint Programming Solver

Tarek Menouer · Mohamed Rezgui ·
Bertrand Le Cun · Jean-Charles Régin

Received: date / Accepted: date

Abstract This paper presents an external parallelization of Constraint Programming (CP) search tree mixing both static and dynamic partitioning. The principle of the parallelization is to partition the CP search tree into a set of sub-trees, then assign each sub-tree to one computing core in order to perform a local search using a sequential CP solver. In this context, static partitioning consists of decomposing the CP variables domains in order to split the CP search tree into a set of disjoint sub-trees to assign them to the cores. This strategy performs well without adding an extra cost to the parallel search, but the problem is the load imbalance between computing cores. On the other hand, dynamic partitioning is based on preservation of the search state to generate, dynamically or on demand, the sub-trees that are assigned to the cores. This strategy offers good load balancing between the different computing cores, but computing overcosts appear due to the initialisation of the search when a sub-tree is migrated from one core to another. In this paper, we propose a new partitioning strategy that mixes the static and dynamic partitioning and enjoys the benefits of each strategy. This mixed partitioning is designed to run on shared and distributed memory architectures. The performances obtained are illustrated by solving the CP problems modelled using the FlatZinc format and solved using the Google OR-Tools solver on top of the parallel Bobpp framework.

Tarek Menouer and Bertrand Le Cun
Prism Laboratory
University of Versailles Saint-Quentin-en-Yvelines
45 avenue des Etats-Unis, Versailles, 78035, France
E-mail: Tarek.menouer@prism.uvsq.fr,Bertrand.lecun@prism.uvsq.fr

Mohamed Rezgui and Jean-Charles Régin
I3S laboratory
University of Nice Sophia Antipolis
28 Avenue Valombrose, Nice, 06100, France
E-mail: Rezgui@i3s.unice.fr, Jean-Charles.REGIN@unice.fr

Keywords Parallelism, Constraint Programming, Work Stealing, Dynamic Load Balancing.

1 Introduction

Parallel search trees have been widely studied in the different contexts of Divide and Conquer, Branch and Bound, and Constraint programming (CP)[1]. Most of these studies propose a parallelization of the entire tree search algorithm that must be used in place of the sequential one, as studies proposed in [40,4,39,2,10,30,34].

In this paper, we propose an external parallelization of CP search tree [2]. The main part of the search is performed by a sequential CP solver. In other words, we propose parallel algorithms that are used to dispatch or schedule the tasks which will be executed by each computing core using a sequential CP solver.

Parallel search tree algorithms are usually explained in terms of static or dynamic partitioning. The principle of static partitioning [36] is to partition the search tree into a finite set of distinct sub-trees according to the decomposition of the variables' domains within the CP problem, then assign each sub-tree to one computing core. The advantage of this strategy is that when a computing core begins searching a new sub-tree, it starts directly without adding an extra cost to the parallel search. However, the workload is not well-balanced. The only way to handle the balancing problem is to generate enough sub-trees to have a good load balance.

The second strategy is dynamic partitioning [20]. The principle of this strategy is based on a previously studied Work Stealing technique to partition the search tree into a set of sub-trees, and schedule them during the execution of the search algorithm in order to have good load balancing between the different computing cores. For technical reasons within the sequential CP solver (Google OR-Tools) used to perform the local search, when a computing core starts the search with a new sub-tree, some extra cost is added to the parallel search. This additional cost of each Work Stealing operation implies a specific scheduling and partitioning algorithm to obtain a good performance.

The contribution of this paper is to propose a new partitioning strategy to parallelize the CP search tree that mixes the static and dynamic strategies. For the needs of High Performance Computing (HPC), this new partitioning strategy will be run in shared and distributed memory architectures. This mixed partitioning starts by performing the static partitioning in order to generate a sufficient number of sub-trees for the different computing cores used in the resolution, and ensure that these starting sub-trees are the good sub-trees to be explored (without adding an extra cost to the search). In the second step, dynamic partitioning is used to ensure good load balancing between computing cores.

[1] Constraint Programming is sometimes called Branch and Infer

[2] This work is funded by PAJERO Bpifrance project

We illustrate the performance of the mixed partitioning by using the OR-Tools solver [41] on top of the parallel Bobpp framework [18].

OR-Tools is an open source sequential CP solver developed by a Google research team. Bobpp is a framework that provides an interface between solvers of combinatorial problems and parallel computers.

The next section presents related work. The static and dynamic partitioning strategies are described in section 3, mixed partitioning is presented in section 4. Section 5 presents some experiments using the Bobpp framework to solve CP problems. Finally, a conclusion and some perspectives are presented in section 6.

2 Related Work

A CP problem Π is a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, defined as follows:

- A set of n variables $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$
- A set of n finite domains $\mathcal{D} = \{D(x_1), D(x_2), \dots, D(x_n)\}$ with $D(x_i)$ the set of possible values for the variable x_i
- A set of constraints between the variables $\mathcal{C} = \{C_1, C_2, \dots, C_e\}$. A constraint C_i is defined on a subset of variables $X_{C_i} = \{x_{i_1}, x_{i_2}, \dots, x_{i_j}\}$ of $\mathcal{X}$ with a subset of the Cartesian product $D(x_{i1}) \times D(x_{i2}) \times \dots \times D(x_{ij})$, that states which combinations of values of variables $\{x_{i_1}, x_{i_2}, \dots, x_{i_j}\}$ are compatible

Each constraint C_i is associated with a filter algorithm that removes values from the domains of its variables when it is not possible to satisfy the constraint. The constraint propagation applies filtering algorithms of $\mathcal{C}$ to reduce the domains of variables in turn until no more reduction can be done. Thereby, it detects the combinations of locally inconsistent values that cannot be solutions.

Π is solved as follows:

First, all variables of Π are unassigned. For each step, a variable x_i is chosen and a value $a \in D(x_i)$ is assigned to x_i in turn. Each branch of a search tree computed by this search defines an assignment. Next, the constraint propagation mechanism checks the consistency of the partial assignment with the constraints $\mathcal{C}$. Each partial assignment creates a node in the search tree. Thus, we associate the consistency of a node with the consistency implied by an assignment.

CP algorithms are used to solve combinatorial problems with great complexity, such as scheduling problems [3]. Several methods are proposed in order to parallelize the CP search-space, as in [33] [32] [35] [25] [42]. In the literature there are several parallel CP solvers such as Gecode [38], Parallel COMET [22], ILOG Parallel Solver [32], parallel Mozart solver [37], etc. Each solver uses a specific CP technique, but from a parallel point of view they all use the same principle based on Work Stealing [8, 5, 14, 12, 1].

Work Stealing is the most popular technique used to implement load balancing between computing cores. Each computing core does some tasks, then

when a core has nothing left to do, it steals tasks from another core to keep busy. Some studies which used Work Stealing to paralellize the exploration of the CP search space are presented in [43,19,16,6,45].

For example, the study presented by Xie et al. [43] proposes the masters/workers approach. Each master has its workers. The search space is divided between the different masters, then each master puts its attributed subtrees in a work pool to dispatch to the workers. When a node of the sub-tree is detected that is a root of large sub-tree, the workers generate a large number of sub-trees and put them in a work pool in order to have better load balancing. Fischetti et al. [19], propose a worker-pool without communication between workers. First, the workers decompose the initial problem during a limited sampling phase, during which each worker visits nodes randomly. Thus, they can visit redundant nodes. After the sampling phase, each worker is attributed its nodes by a deterministic function. During the resolution, if a node is detected to be difficult by an estimated function, it is put into a global queue. When a worker finishes the resolution of its node, it receives a hard node from the global queue and solves it. When the queue is empty and there is no work to do, the resolution is done. Jaffar et al. [16] propose the use of a master which centralizes all pieces of information (bounds, solutions and requests). The master evaluates which worker has the largest amount of work in order to give some work to a waiting worker.

There are other works which use new techniques, such as Portfolio parallelization. Bordeaux et al. [6], are the first authors to present a study on the scalability of constraint solving on more than 100 processors. They focus on the resolution of CP and boolean SATisfiability (SAT) problems. They use two approaches: portfolios and search space splitting. Without communication and using hashing constraints to split the search tree, their results show good speedups for up to 30 processors, but not beyond. Yun et al.[45] introduced the recursive splitting with iterative bisection partitioning method which decomposes the initial problem into a large number of sub-problems. This decomposition is based on previous unsuccessful resolutions.

In all previous works, the parallelization is performed inside the search algorithm. In this paper we propose an external parallelization of OR-Tools CP solver using a parallel Bobpp framework. The external parallelization is done without changing the OR-Tools source code.

3 The Partitioning Strategies

This section presents the different partitioning strategies proposed in order to perform an external parallelization of CP search tree by using the OR-Tools solver on top of the parallel Bobpp framework.

OR-Tools [41] is an open source library that implements a sequential CP solver. It is developed in C++ by a Google research team. The purpose of this library is to explore the search space in order to find one or all possible solutions using various constraint propagation methods.

Bobpp [11] is a parallel framework oriented towards solving Combinatorial Optimization Problems. It is developed in C++ and can be used as the runtime support. Bobpp provides several search algorithms that can be parallelized using different parallel programming methods. The goal is to propose a single framework for most classes of Combinatorial Optimization Problems, which can be solved in as many different parallel architectures as possible. Figure 1 shows how Bobpp interfaces with high-level applications (QAP, TSP, ...), CP solvers, and different parallel architectures using several parallel programming environments like Pthreads as well as MPI or more specialized libraries such as Athapascan/Kaapi.

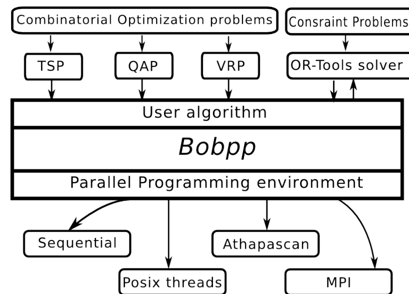


Fig. 1 Bobpp Framework

3.1 The Static Partitioning

The principle of static partitioning is to split the CP search tree into a set of disjoint sub-trees. The root node of each sub-tree is named a *BOB-static-node*, which has a partial assignment as a set of pairs (variable, value) checked by the propagation mechanism. This strategy is explain in detail in [36].

A simple algorithm could be to perform Breadth First Search (BFS) in the search tree until the n *BOB-static-node* is reached. Unfortunately, it is not easy to efficiently perform BFS, mainly because BFS is not an incremental algorithm like Depth First Search (DFS). Therefore, we have used Depth-bounded Depth First Search (DBDFS). Let n_k denote the number of *BOB-static-nodes* found with DBDFS at depth k. To reach the nth *BOB-static-node*, we repeat the DBDFS until we reach a depth k such that $n_{k-1} < n \leq n_k$.

This strategy can be compared with the Iterative Deepening Depth-First Search (IDDFS) [17], which is based on the same principle (iterations of DBDFS), but with some differences. IDDFS stops the search when it finds a solution, whereas static partitioning tries to generate n *BOB-static-nodes*.

As this strategy doesn't add extra cost to the parallel search, we use it to explore the first level of the search tree and to generate *BOB-static-nodes*, which are shared between computing cores to perform a parallel search. This

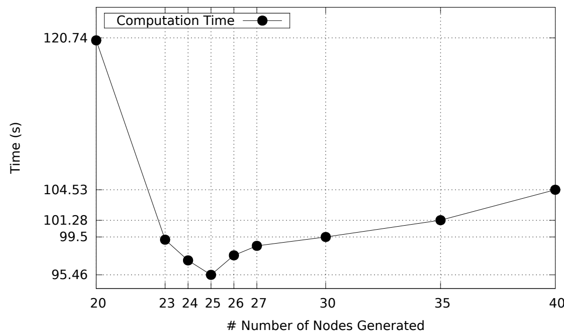


Fig. 2 Variation of computation time for solving the problem of Naval battle (Sb_sb_13_13_5_1) [27] on 12 cores according to the number of sub-trees generated

strategy performs well, but it has some problems. The first problem is the determination of the best number of *BOB-static-nodes* to generate.

Figure 2 shows an example concerning the variation of computation time according to the number of sub-trees generated by static partitioning to solve the Naval battle problem (Sb_sb_13_13_5_1) [27], using 12 cores on an Intel machine (12 cores and 48 GB of RAM). Thus, each time we increase the number of sub-trees generated we measure the computation time until an optimal number of sub-trees is found (in figure 2 this is 25 sub-trees). After the optimal value found in this example, as you can see, the computation time slows down again.

The second problem concerns the load imbalance between computing cores. Indeed, the load imbalance implies that sometimes one computing core performs almost all of the search while the other computing cores wait until the first core finishes.

3.2 The Dynamic Partitioning

The principle of the dynamic partitioning is that different computing cores share the work via the Bobpp Global Priority Queue, called *GPQ*. The search tree is decomposed and allocated to the different cores on demand and during the execution of the search algorithm. Periodically, a working core tests if waiting core(s) exist(s). If this is the case, the working core stops the search in the left OR-Tools node. Next, the path from the root node to the highest right OR-Tools node is saved in what we called the *BOB-dynamic-node*. This *BOB-dynamic-node* is inserted into the GPQ, then the working core continues the search with the left OR-Tools node. Otherwise, if no waiting cores exist, the working core performs the search locally using the OR-Tools solver. The waiting cores are notified by the insertion of a new *BOB-dynamic-node* in the GPQ. If a new *BOB-dynamic-node* is inserted in the GPQ, the waiting core picks up the node and starts the search. This partitioning strategy is presented in detail in [20].

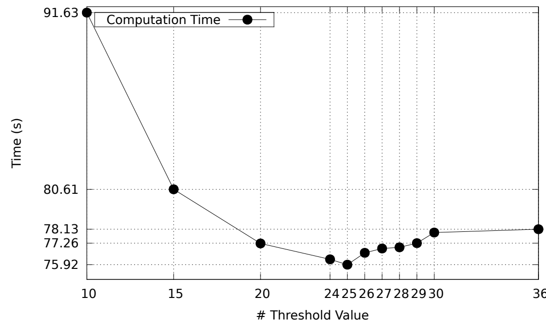


Fig. 3 Variation of the computation time for solving the problem of Naval battle (Sb_sb_13_13_5_1) [27] on 12 cores according to the value of the threshold

For technical reasons in the OR-Tools solver, each time a computing core starts the search with a new *BOB-dynamic-node*, redundant nodes must be explored. To limit the exploration of redundant nodes, it is proposed to add a threshold, which limits the depth of each *BOB-dynamic-node*. The choice of the value of the threshold is a difficult problem. Choosing a very small threshold makes the algorithm similar to static partitioning, with a limited number of sub-trees explored by the different computing cores. Conversely, choosing a high threshold makes the algorithm similar to dynamic partitioning without a threshold, which makes load balancing easier between the cores but increases the exploration of the redundant OR-Tools nodes.

Figure 3 shows the variation in computation time according to the value of the threshold to solve the Naval battle problem (Sb_sb_13_13_5_1) [27] using 12 cores on an Intel machine (12 cores and 48 GB of RAM). As a result, the computation time decreases while increasing the threshold value until an optimal threshold (value of 25) is reached. After this optimal value the computation time increases again.

4 Mixing Static and Dynamic Partitioning

In short, each of the two previous strategies has the following advantages and disadvantages:

- Static Partitioning generates good sub-trees to be explored by the computing cores without adding an extra cost to the parallel search, but the disadvantages are:
  - The determination of the best number of sub-trees to generate
  - The workload can be imbalanced
  - It is developed in [36], to run only for a shard memory architecture
- Dynamic Partitioning gives a well-balanced work, but the problems are:
  - Each time a sub-tree migrates from one computing core to another, redundant nodes are explored and an extra cost is added to the parallel search

Algorithm 1 Mixed Partitioning

Require: $\mathcal{N}$, the threshold of static partitioning
Require: $\mathcal{S}$, the threshold of dynamic partitioning
  Let K, represent the current number of sub-trees generated by static partitioning
  Let P, represent the depth of the current OR-Tools node
  if $K < N$ **then**
    Generate a new sub-tree using static partitioning
  else
    if $\exists$ at least one waiting core AND $P < S$ **then**
      Stop the search on the left branch
      Create a *BOB-dynamic-node*
      Insert the *BOB-dynamic-node* in the global queue
      Restart the search
    else
      Continue the sequential exploration of the search-space
    end if
  end if

– It is also developed in [20], to run only for a shard memory architecture

The following section presents a new partitioning strategy that performs in shared and distributed memory architectures. The aim of this new strategy is to mix static and dynamic partitioning in order to generate at the beginning good initial sub-trees which are explored without adding an extra cost to the parallel search. Then, in the second step, dynamic partitioning is used to have good load balancing, as presented in algorithm 1.

Figure 4 shows the variation in computation time according to the number of sub-trees generated during static partitioning and the value of the threshold used by dynamic partitioning to solve the Naval battle problem (Sb_sb_13_13_5_1). This experiment was realised using 12 cores on an Intel machine (12 cores and 48 GB of RAM).

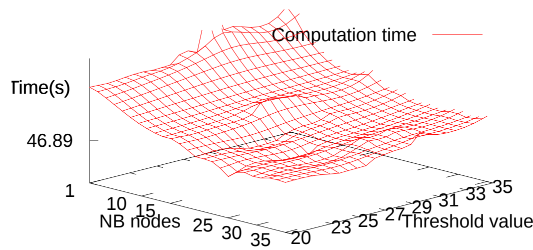
Figure 4:a presents the variation in computing time in a 3D format. A first remark is that the optimal number of sub-trees generated with static partitioning is 25.

In figure 4:b, the number of sub-trees generated by the static partitioning is fixed at 25 and we present the computation time obtained by varying the threshold value used by the dynamic partitioning method. Coincidentally, the optimal threshold for the dynamic partitioning is also 25.

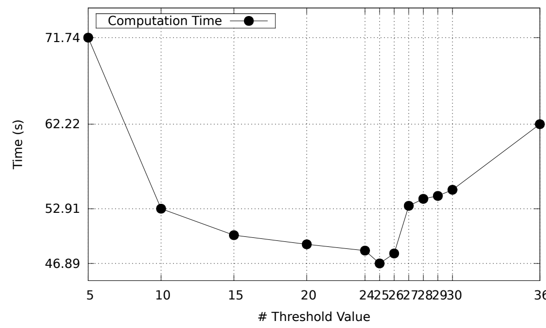
Figure 4:c shows a table describing computation time values obtained by varying the number of sub-trees generated during static partitioning and the value of the threshold used by the dynamic partitioning. Figure 4:c is presented in order to explain figure 4:b in detail.

We note in figures 2, 3 and 4:b) that:

– The computation time obtained in figure 2 by using an optimal number of sub-trees generated by the static partitioning is *95.46* seconds
– The computation time obtained in figure 3 by using the optimal dynamic partitioning threshold is *75.92* seconds



(a)



(b)

Number of Static sub-trees	Dynamic Threshold Value	Computation Time (seconds)
1	24	86,845
15	24	63,375
20	24	49,59
25	24	48,31
30	24	49,08
35	24	49,5
1	25	85,955
15	25	63,285
20	25	48,335
25	***25***	***46,89***
30	25	48,24
35	25	48.7
1	26	86,255
15	26	65
20	26	48,555
25	26	47,99
30	26	48,91
35	26	49.5

(c)

Fig. 4 Variation of computation time for solving the Naval battle problem (Sb_sb_13_13_5_1) [27] on 12 cores according to the number of sub-trees generated by the static partitioning and threshold value of the dynamic partitioning

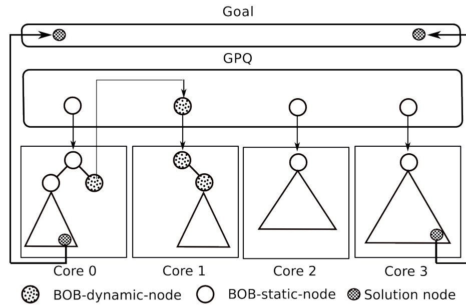


Fig. 5 Mixed partitioning in a shared memory architecture

– The computation time obtained in figure 4:a by using the optimal number of sub-tree generated by the static partitioning and the optimal dynamic partitioning threshold is *46.89* seconds

So, using a mixed partitioning strategy allows one to obtain a good computation time.

4.1 Mixed Partitioning in a Shared Memory Architecture

To explore the search tree on parallel computers, the primary issue is load balancing. The idea is to use the GPQ of the parallel Bobpp framework as a global pool of nodes (tasks) in which each computing core will pick up a node and perform a local search using a sequential CP solver.

As presented in figure 5, the mixed partitioning starts by inserting sub-trees generated using a static partitioning into the Bobpp GPQ. These sub-trees are represented as *BOB-static-nodes*. As the search tree generated by the OR-Tools solver is binary, we propose to fix the number of the sub-trees generated by the static partitioning to 2×(Number of computing cores used in the search). The mixed partitioning algorithm does not require re-compilation for each new target machine or new execution context. For every new execution, the software checks the number of computing cores used in the search and generates the *BOB-static-nodes*. During the second step, in order to keep a good load balance between the different computing cores, we use dynamic partitioning with an automatic threshold. In order to accomplish this, the starting value of the threshold is fixed, then increased each time a load imbalance is detected. This automatic threshold performs well, it is presented in [20].

If a solution is found, it will be inserted into the Bobpp Goal, which stores all of the solutions.

4.2 Mixed Partitioning in a Distributed Memory Architecture

Parallel search on a distributed memory architecture relies on the same principle as a parallel search on a shared memory architecture. On a distributed

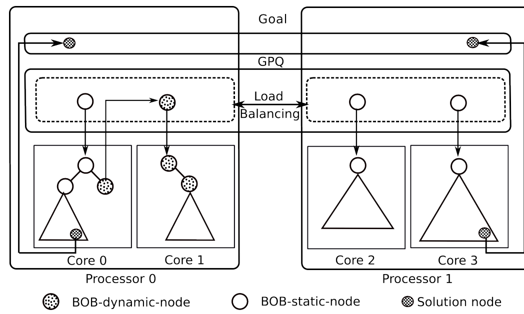


Fig. 6 Mixed partitioning in a distributed memory architecture

memory machine, the Bobpp GPQ has to be split among the number of processors used and we assign to each processor one Sub Priority Queue (SPQ). The main difference between these two types of memory is the priority queue management.

On a shared memory, there is one priority queue shared between all cores. In a distributed memory, we use many SPQs and each SPQ will be shared between the different cores used by each processor. In order to obtain a good load balancing between the different *BOB-static/dynamic-nodes* used by each SPQ, we use the Work Stealing approach. This means that *BOB-static/dynamic-nodes* are moved once a starvation risk is detected.

As soon as the distributed memory of the priority queue is used, Bobpp is able to execute a shared memory strategy on each processor using a hybrid parallel programming environment (Pthreads+MPI).

Figure 6 shows how a hybrid environment is used to perform the mixed partitioning.

5 Experimentation

In order to validate and compare the different approaches used in this study, experiments were performed using two Linux machines, M1 and M2. The two machines have the same configuration: each machine is a bi-processor Intel Xeon X5650 (2.67 GHz) computer with 12 cores and 48 GB of RAM. The network uses Ethernet technology with IPV6 protocol to obtain a speed of 188552 Kbit/s. The goal of this experimentation is to show the performances of our parallelization approach. We use for the implementation the OR-Tools solver (version: 2727). It is evident that all results depends on the OR-Tools solver because it is used as the main CP solver. All CP problems solved in this paper were proposed in the MiniZinc Challenge 2012 [27] and modelled using the FlatZinc format [28]. The goal of the MiniZinc Challenge is to compare the performances of solvers and various constraint solving technologies on a set of problems. FlatZinc is a low-level solver input language designed to specify problems at the level of an interface to CP solvers. The following computation

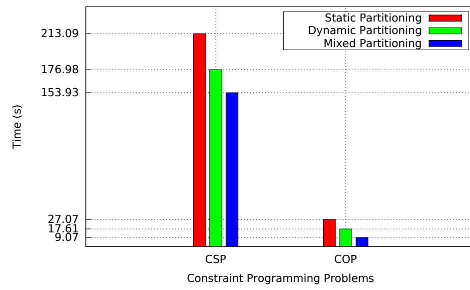


Fig. 7 Computation time for solving the Constraint Programming problems depending on the partitioning strategies

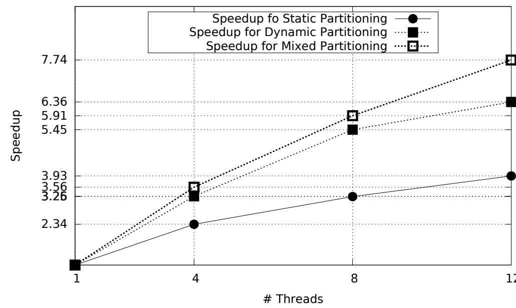


Fig. 8 Average speedup obtained using static, dynamic and mixed partitioning for solving 6 Constraint Satisfaction Problems

time presented in this section are given in seconds and are averages of several runs.

5.1 Solving Constraint Programming Problems using a Shared Memory Architecture

Figure 7 compares the three partitioning strategies - static, dynamic and mixed - for solving the two CP problems listed below. 12 cores were used in testing each strategy:

- The Quasi Group problem (Quasigroup7_10) [27], which is a Constraint Satisfaction Problem: labelled CSP in figure 7
- The Level Packing problem (2DLevelPacking_Class8_20_9) [27], which is a Constraint Optimization Problem: labelled COP in figure 7

The performance of static partitioning is limited, as shown by its low speedup compared to other partitioning strategies. The third partitioning (mixing static and dynamic) is clearly the best partitioning method for solving Constraint Satisfaction and Optimization Problems.

Figures 8 and 9 show the average speedup obtained using each of the three partitioning strategies for solving 10 CP problems, 6 are Constraint

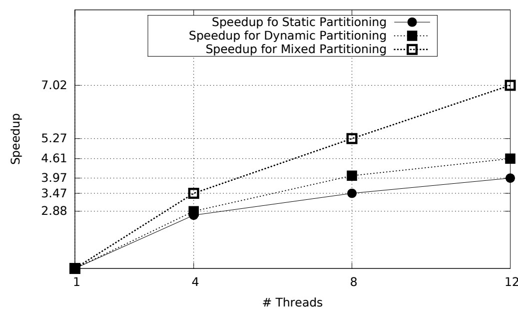


Fig. 9 Average speedup obtained using static, dynamic and mixed partitioning for solving 4 Constraint Optimization Problems

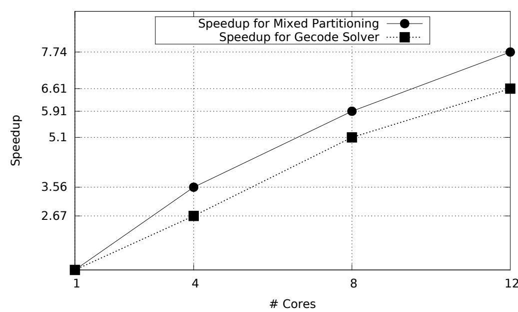


Fig. 10 Comparison between an average speedup obtained using mixed partitioning and Gecode CP solver for solving 6 Constraint Satisfaction Problems

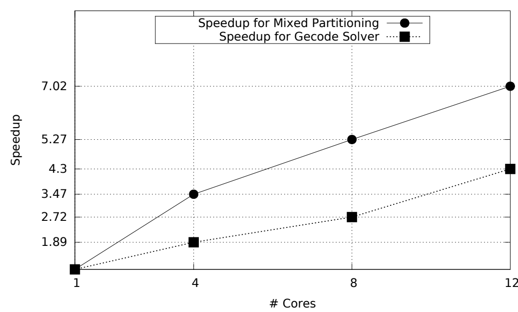


Fig. 11 Comparison between an average speedup obtained using mixed partitioning and Gecode CP solver for solving 4 Constraint Optimization Problems

Satisfaction Problems and 4 are Constraint Optimization Problems. Using mixed partitioning, an average speedup of 7.74 is reached in the resolution of 6 Constraints Satisfaction Problems using 12 cores and an average speedup of 7.02 is obtained for solving 4 Constraint Optimization Problems using 12 cores.

Problems	Mixed Partitioning run times (seconds)			
	1 core	4 cores	8 cores	12 cores
quasigroup7_10.fzn	1377.25	432.43	227.55	162.71
sb_sb_13_13_5_1.fzn	653.58	157.42	93.83	56.65
sb_sb_14_14_6_0.fzn	152.25	51.23	29.63	23.81
sb_sb_15_15_7_4.fzn	284.73	91.86	57.47	49.13
sb_sb_12_12_5_4.fzn	64.51	14.34	8.51	7.46
sb_sb_15_15_7_3.fzn	248.95	71.52	52.12	44.27

Table 1 Execution times obtained using mixed partitioning for solving 6 Constraint Satisfaction Problems

Problems	Gecode solver run times (seconds)			
	1 core	4 cores	8 cores	12 cores
quasigroup7_10.fzn	239.69	73.66	37.46	24.74
sb_sb_13_13_5_1.fzn	301.44	107.80	60.75	50.85
sb_sb_14_14_6_0.fzn	151.69	57.2	28.23	22.15
sb_sb_15_15_7_4.fzn	411.24	139.56	78.84	66.26
sb_sb_12_12_5_4.fzn	42.33	16.8	9.24	6.90
sb_sb_15_15_7_3.fzn	335.73	175.97	81.38	68.48

Table 2 Execution times obtained using Gecode CP solver for solving 6 Constraint Satisfaction Problems

Problems	Mixed Partitioning run times (seconds)			
	1 core	4 cores	8 cores	12 cores
2DLevelPacking_Class8_20_9.fzn	102.65	34.01	21.37	14.23
fastfood_ff58.fzn	37.28	9.33	5.54	4.53
open_stacks_01_problem_15_15.fzn	46.36	13.27	8.26	7.18
open_stacks_01_wbp_30_15_1.fzn	87.27	25.70	17.58	14.07

Table 3 Execution times obtained using mixed partitioning for solving 4 Constraint Optimization Problems

Problems	Gecode solver run times (seconds)			
	1 core	4 cores	8 cores	12 cores
2DLevelPacking_Class8_20_9.fzn	19.17	9.96	12.01	7.97
fastfood_ff58.fzn	21.04	9.34	7.32	6.99
open_stacks_01_problem_15_15.fzn	96.82	40.55	19.75	13.97
open_stacks_01_wbp_30_15_1.fzn	197.71	196.88	127.85	40.57

Table 4 Execution times obtained using Gecode CP solver for solving 4 Constraint Optimization Problems

Currently there are many available computing resources, such as data centers and the cloud computing, so the majority of parallel CP solvers proposed in the literature, such as Choco [7], CPHYDRA [9] and Numberjack solver [29] use a Portfolio parallelization [15]. The idea of the Portfolio parallelization is to run different search strategies in parallel and the first strategy to find a solution or meet the needs of the user stops all other strategies. In our approach we propose an external parallelization of one search strategy instead of

a Portfolio parallelization. Few CP solvers exist that offer a parallelization of one search strategy and solve problems modelled using FlatZinc format, such as the Gecode solver [38]. Gecode [38] is an open source parallel CP solver developed in C++. It proposes an internal parallelization of a search algorithm using the Work Stealing technique. The idea of this parallelization is described as follows: when a computing core has nothing to do, it steals work from other cores which have tasks to perform.

Figures 10 and 11 show a comparison between the average speedup obtained using mixed partitioning, which is an external parallelization of the OR-Tools solver that is among the best CP solvers selected in the MiniZinc Challenge 2013 [26], and the Gecode solver (version 4.2.1), which was selected as the best parallel CP solver from 2008 until 2012 in the MiniZinc Challenge [26] for solving 6 Constraint Satisfaction Problems and 4 Constraint Optimization Problems.

Tables 1, 2, 3 and 4 show the absolute run time (giving in seconds) obtained using the mixed partitioning and Gecode CP solver for solving 6 Constraint Satisfaction Problems and 4 Constraint Optimization Problems.

The average speedup obtained using the mixed partitioning is better than the average speedup obtained using the Gecode solver. This result allows validation of the performance of the mixed partitioning.

Our approach has some benefits, such as:

- It proposes an external parallelization of the OR-Tools solver without changing the OR-Tools code, it is used directly with the next OR-Tools release
- As it is an external parallelization, it is possible to combine the mixed partitioning and the portfolio parallelization [21]
- It is designed for both shared and distributed memory architectures, as opposed to the other parallel CP solvers which are designed for only one memory architecture.

The problems with our approach are:

- The resolution of problems depend on the threshold of static and dynamic partitioning
- The parallel search algorithm is not deterministic
- Only solves problems modelled using FlatZinc format

Figure 12 shows the load balance for solving the Quasi Group problem (Quasigroup7_10) [27], which is a Constraint Satisfaction Problem using 12 cores according to the partitioning strategies.

Figure 12:a is obtained using static partitioning. It is clear that this strategy gives a load imbalance between the computing cores. We observe that the computing time is *213.09* seconds.

Figure 12:b is obtained using dynamic partitioning. With this strategy, the computation time is *176.98* seconds.

Figure 12 :c is obtained using mixed partitioning. With this strategy, the load is good balanced and the computation time is *153.93* seconds.

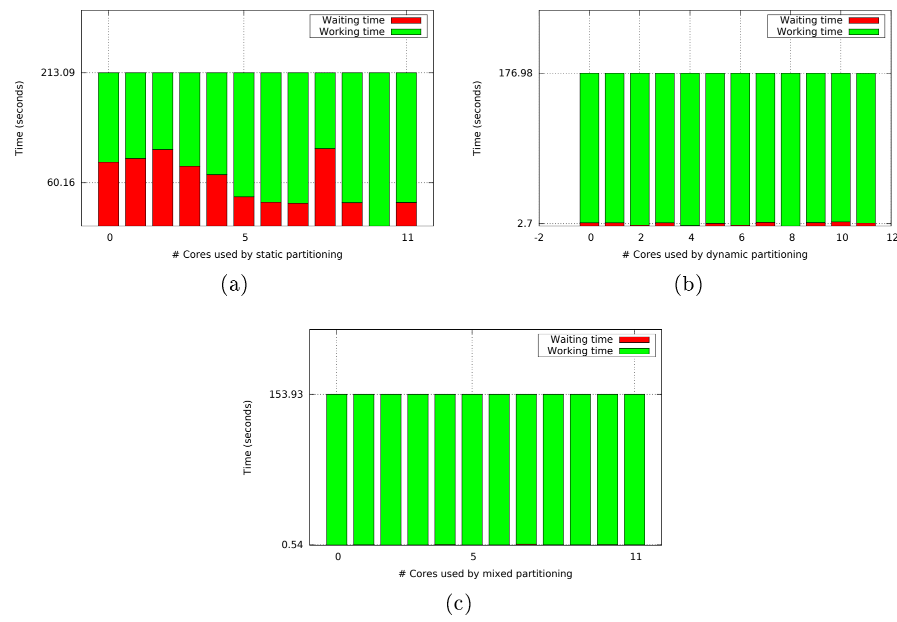


Fig. 12 Load balancing for solving the Quasi Group problem (Quasigroup7_10) [27], which is a Constraint Satisfaction Problem using 12 cores according to the partitioning strategies

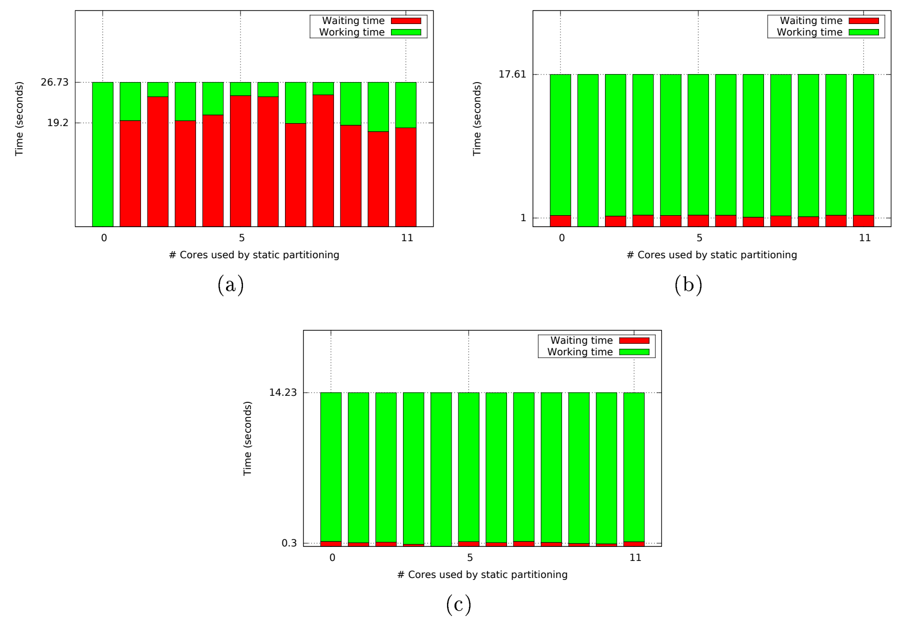


Fig. 13 Load balancing for solving the Level Packing problem (2DLevelPacking_Class8_20_9), which is a Constraint Optimization Problem using 12 cores according to the partitioning strategies

Figure 13 shows the load balance for solving the Level Packing problem (2DLevelPacking_Class8_20_9) [27], which is a Constraint Optimization Problem using 12 cores according to the partitioning strategies (static, dynamic and mixed).

As a result, for both problems (Constraint Satisfaction and Optimization Problems), the mixed partitioning allows all cores to work and wait for an equivalent time.

5.2 Solving Constraint Programming Problems using a Distributed Memory Architecture

Computer(s)	Number of processors	Number of cores/ processor	Total number of cores	speedup
M1	2	4	8	5.04
M1	3	4	12	5.54
M1&M2	4	4	16	5.93
M1&M2	4	5	20	7.02
M1&M2	2	12	24	10.04

Table 5 Average speedup for solving 6 Constraint Satisfaction Problems using a hybrid environment

Computer(s)	Number of processors	Number of cores/ processor	Total number of cores	speedup
M1	2	4	8	3.3
M1	3	4	12	3.43
M1&M2	4	4	16	3.6
M1&M2	4	5	20	4.89
M1&M2	2	12	24	7.78

Table 6 Average speedup for solving 4 Constraint Optimization Problems using a hybrid environment

The static and dynamic partitioning strategies are developed only for shared memory architectures. In the following section, we present some experiments on a distributed memory architecture with only the mixed partitioning strategy.

Tables 5 and 6 show the average speedups for solving Constraint Satisfaction and Optimization Problems using a hybrid (Pthreads+MPI) parallel programming environment.

There are some studies which present distributed algorithms for solving CP problems, as in [31,44,37,24,23]. However, to our knowledge, there is no open source CP solver which offers a parallel search algorithm adapted for

distributed memory architectures and solves our instances, which are modelled using FlatZinc format. This is why we can not compare our approach with other distributed CP solvers.

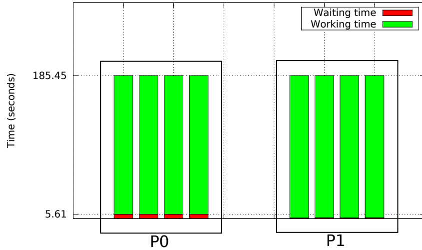


Fig. 14 Load balancing for solving the Quasi group problem (Quasigroup7_10), which is a Constraint Satisfaction Problem using a hybrid parallel programming environment with 2 processors and 4 cores/processor

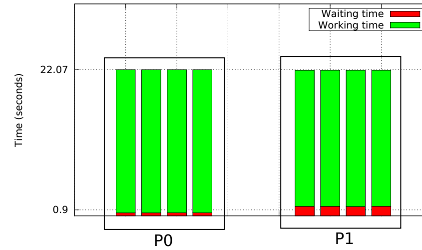


Fig. 15 Load balancing for solving the Level Packing problem (2DLevelPacking_Class8_20_9), which is a Constraint Optimization Problem using a hybrid parallel programming environment with 2 processors and 4 cores/processor

Figures 14 and 15 show the load balance when solving Constraint Satisfaction and Optimization Problems. The same result as in the shared memory architecture was obtained. All cores work and wait for an equivalent time.

6 Conclusion

This paper presents an external parallelization of a Constraint Programming solver, called OR-Tools, using the parallel Bobpp framework.

This solution presents a new strategy that mixes static and dynamic partitioning. Static partitioning is used to generate the best sub-trees and start the search without adding an extra cost. Dynamic partitioning is used to perform dynamic partitioning of different sub-trees during the execution of the search algorithm with a good load balance between computing cores.

It would be good to always give the user the same solution for a specific problem. In a sequential resolution, this is easy because it is the first solution obtained. However, in parallel, it is not necessarily the first solution. In the future, this mixed partitioning algorithm used to parallelize the OR-Tools solver will be adapted to always give the same solution whether using a parallel or a sequential run if it is asked by the user.

Finally, to enrich and extend the set of problems solved by the Bobpp framework, it would be interesting to port on top of the Bobpp framework a boolean SATisfiability (SAT) solver such as *Glucose* [13] solver.

References

1. U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM.
2. P. M. P. Alfonso Ferreira. *Solving Combinatorial Optimization Problems in Parallel Methods and Techniques.* Springer, lecture notes in computer science, vol. 1054 edition, 1996.
3. P. Baptiste, C. L. Pape, and W. Nuijten. *Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems.* International Series in Operations Research and Management Science, Paris, volume 39- springer edition, 2001.
4. B.Gendron and T.G.Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operational Research*, 42(06):1042–1066, 1994.
5. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.
6. L. Bordeaux, Y. Hamadi, and H. Samulowitz. Experiments with massively parallel constraint solving. In *IJCAI*, pages 443–448, 2009.
7. Choco solver
`http://www.emn.fr/z-info/choco-solver/`, 2013. Accessed: 14-04-2014.
8. G. Chu, C. Schulte, and P. J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *Principles and Practice of Constraint Programming-CP 2009*, pages 226–241. Springer, 2009.
9. e. o'mahony, E. Hebrard, A. Holland, and C. Nugent. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.
10. A. Ferreira and M. Morvan. Models for Parallel Algorithm Design: An Introduction. In A. Migdalas, P. Pardalos, and S. Storoy, editors, *Parallel Computing in Optimization*, pages 1–26. Kluwer Academic Publisher, Boston (USA), 1997.
11. F. Galea and B. Le Cun. Bob++ : a framework for exact combinatorial optimization methods on parallel machines. In *International Conference High Performance Computing & Simulation 2007 (HPCS'07) and in conjunction with The 21st European Conference on Modeling and Simulation (ECMS 2007)*, pages 779–785, June 2007.
12. T. Gautier, J. Roch, and G. Villard. Regular versus irregular problems and algorithms. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 1995.
13. Glucose sat solver
https://www.lri.fr/ simon/?page=glucose. Accessed: 14-04-2014.
14. Y. Guo, J. Zhao, V. Cave, and V. Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.
15. B. Hurley, L. Kotthoff, Y. Malitsky, and B. O'Sullivan. Proteus: A hierarchical portfolio of solvers and transformations. *arXiv preprint arXiv:1306.5606*, 2013.
16. J. Jaffar, A. E. Santosa, R. H. C. Yap, and K. Q. Zhu. Scalable distributed depth-first search with greedy work stealing. In *ICTAI*, pages 98–103, 2004.
17. R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, Sept. 1985.
18. B. Le Cun, T. Menouer, and P. Vander-Swalmen. Bobpp. `http://forge.prism.uvsq.fr/projects/bobpp`. Accessed: 14-04-2014.
19. M. M. Matteo Fischetti and D. Salvagnin. Self-splitting of workload in parallel computation. In *CPAIOR'14*, 2014.
20. T. Menouer and B. L. Cun. Anticipated dynamic load balancing strategy to parallelize constraint programming search. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1771–1777, 2013.
21. T. Menouer and B. L. Cun. Adaptive n to p portfolio for solving constraint programming problems on top of the parallel bobpp framework. In *2014 IEEE 28th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2014.

22. L. Michel, A. See, and P. Hentenryck. Parallelizing constraint programs transparently. In C. Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 514–528. Springer Berlin Heidelberg, 2007.
23. L. Michel, A. See, and P. V. Hentenryck. Transparent parallelization of constraint programs on computer clusters, 2008.
24. L. Michel, A. See, and P. Van Hentenryck. Distributed constraint-based local search. In F. Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, volume 4204 of *Lecture Notes in Computer Science*, pages 344–358. Springer Berlin Heidelberg, 2006.
25. L. Michel, A. See, and P. Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS JOURNAL ON COMPUTING*, 21(3):363–382, 2009.
26. Minizinc challenge
`http://www.minizinc.org/challenge.html`. Accessed: 14-04-2014.
27. Minizinc challenge
`http://www.minizinc.org/challenge2012/challenge.html`, 2012. Accessed: 14-04-2014.
28. NICTA. Specification of zinc and minizinc. Technical report, Victoria Research Lab, Melbourne, Australia, 2011.
29. Numberjack solver
`http://numberjack.ucc.ie/`, 2013. Accessed: 14-04-2014.
30. P. M. Pardalos. *Parallel Processing of Discrete Problems.* Springer-Verlag, volume 106 of the ima volumes in mathematics and its applications edition, 1999.
31. V. Pedro and S. Abreu. Distributed work stealing for constraint solving. *CoRR*, abs/1009.3800, 2010.
32. L. Perron. Search procedures and parallelism in constraint programming. *Principles and Practices of Constraint Programming*, 1999.
33. I. P.Gent, C. Jefferson, I. Miguel, N. C. Moore, P. Nightingale, P. Prosser, and C. Unsworth. A preliminary review of literature on parallel constraint solving. *Proceedings PMCS'11 Workshop on Parallel Methods for Constraint Solving*, 2011.
34. M. R. P.M. Pardalos and K. Ramakrishnan. *Parallel Processing of Discrete Optimization Problems.* American Mathematical Society, dimacs series vol. 22 edition, 1995.
35. C. C. Rolf. *Parallelism in Constraint Programming.* PhD thesis, Department of Computer Science, Lund University, Oct 2011.
36. J.-C. Régin, M. Rezgui, and A. Malapert. Embarrassingly parallel search. In *19th International Conference CP 2013 Uppsala Sweden*, 2013.
37. C. Schulte. Parallel search made simple. In *University of Singapore*, pages 41–57, 2000.
38. C. Schulte, G. Tack, and M. Z. Lagerkvist. *Modeling and Programming with Gecode.*
39. O. V. Shylo, T. Middelkoop, and P. M. Pardalos. Restart strategies in optimization: parallel and serial cases. *Parallel Computing*, 37(1):60 – 68, 2011.
40. C. R. Theodor Crainic, Bertrand Le Cun. *Parallel Branch and Bound Algorithms*, chapter 1, pages 1–28. John Wiley and Sons, USA, 2006.
41. N. van Omme, L. Perron, and V. Furnon. Or-tools. Technical report, Google, 2012.
42. P. Vander-Swalmen, G. Dequen, and M. Krajecki. Designing a parallel collaborative sat solver. In *17th International Conference on Parallel and Distributed Processing Techniques and Applications*, USA, 2011. CSREA Press.
43. F. Xie and A. Davenport. Solving scheduling problems using parallel message-passing based constraint programming. In *Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems COPLAS*, pages 53–58, 2009.
44. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.
45. X. Yun and S. L. Epstein. A hybrid paradigm for adaptive parallel search. In *CP*, pages 720–734, 2012.