Parallel Constraint Programming

Jean-Charles Régin and Arnaud Malapert

Abstract Constraint programming (CP) is an efficient technique for solving combinatorial optimization problems. In CP a problem is defined over variables that take values in domains and constraints which restrict the allowed combination of values. CP uses for each constraint an algorithm that removes values of variables that are inconsistent with the constraint. These algorithms are called while a domain is modified. Then, a search algorithm such as a backtracking or branch-and-bound algorithm is called to find solutions. Several methods have been proposed to combine CP with parallelism. In this chapter, we present some of them: parallelization of the propagator, parallel propagation, search splitting, also called work-stealing, problem decomposition, also called embarrassingly parallel search (EPS), and portfolio approaches. We detail the two giving the best performances in practice: the work-stealing approach and embarrassingly parallel search. We give some experiments supporting this claim on a single multi-core machine, on a data center and on the cloud.

1 Introduction

Constraint Programming (CP) is an efficient technique for solving combinatorial optimization problems. It is widely used for solving real-world applications such as rostering, scheduling, car sequencing, routing, etc. CP-based solvers are general and generic tools for modeling and solving problems [12, 14, 85, 101, 88, 102, 98]. The development of such solvers is an active topic of the CP community. In this chapter, we propose to consider different approaches for parallelizing a CP-based solver. Our

Jean-Charles Régin
Université Côte d'Azur, CNRS, I3S, France, e-mail: jcregin@gmail.com

Arnaud Malapert
Université Côte d'Azur, CNRS, I3S, France e-mail: arnaud.malapert@unice.fr

goal is to present methods that have been used to automatically parallelize CP-based solvers. This means that no particular action of the user is required.

CP is mainly based on the exploitation of the structure of the constraints and accepts constraints whose structure is different, unlike SAT or MIP which impose certain rules on allowable models of the problem: having only boolean variables and three clauses for SAT, or having only linear constraints for MIP.

This specificity of CP allows the use of any kind of algorithm for solving a problem. We could even say that we want to exploit as much as possible the capability to use different algorithms. Currently, when a problem is modeled in CP it is possible that a large variety of algorithms are used at the same time and communicate with each other. For instance, unlike with other techniques, it is really conceivable to have at the same time flow algorithms and dynamic programming.

In CP, a problem is defined using variables and constraints. Each variable is associated with a domain containing its possible values. A constraint expresses properties that have to be satisfied by a set of variables.

In CP, a problem can also be viewed as a conjunction of sub problems for which we have efficient resolution methods. These sub-problems can be very easy like $x < y$ or complex like the search for a feasible flow. These sub problems correspond to constraints. Then, CP uses for each sub problem the associated resolution method, often called a *propagator*. A propagator removes from the domains the values that cannot belong to any solution of the sub problem. This mechanism is called *filtering*. By repeating this process for each sub problem, so for each constraint, the domains of the variables are reduced.

After each modification of a variable domain, it is useful to reconsider all the constraints involving that variable, because that modification can lead to new deductions. In other words, the domain reduction of one variable may lead to deduce that some other values of some other variables cannot belong to a solution. So, CP calls all the propagators associated with a constraint involving a modified variable until no more modification occurs. This mechanism is called *propagation*.

Then, and in order to reach a solution, the search space will be traversed by assigning successively a value to each variable. The filtering and propagation mechanisms are, of course, triggered when a modification occurs. Sometimes, an assignment may lead to the removal of all the values of a domain: we say that a failure occurs, and the latest choice is reconsidered: there is a backtrack and a new assignment is tried. This mechanism is called *search*.

So, CP is based on three principles: filtering, propagation and search. We could represent it by reformulating Kowalski's famous definition of Algorithm (Algorithm = Logic + Control) [53] as:

$$CP = filtering + propagation + search \quad (1)$$

where filtering and propagation correspond to Logic and search to Control.

An objective can also be added in order to deal with optimization problems. In this case, a specific variable representing the objective is defined. When a better solution is found then this variable is updated, and this modification is permanent.

The relation between the objective variable and the other variables is usually via a constraint representing the objective function, which is often a sum constraint.

1.1 Filtering + Propagation

Since constraint programming is based on filtering algorithms [91], it is quite important to design efficient and powerful algorithms. Therefore, this topic caught the attention of many researchers, who discovered a large number of algorithms.

As we mentioned, a filtering algorithm directly depends on the constraint it is associated with. The advantage of using the structure of a constraint can be shown on the constraint $x \leq y$. Let $min(D)$ and $max(D)$ be respectively the minimum and the maximum value of a domain. It is straightforward to establish that all the values of x and y in the range $[min(D(x)), max(D(y)]$ are consistent with the constraint. This means that arc consistency can be efficiently and easily established by removing the values that are not in the above ranges. Moreover, the use of the structure is often the only way to avoid memory consumption problems when dealing with non-binary constraints. In fact, this approach prevents us from explicitly having to represent all the combinations of values allowed by the constraint.

One of the most famous examples is the ALLDIFFB constraint, which states that values taken by variables must be different, especially because the filtering algorithm associated with this constraint is able to establish arc consistency in a very efficient way by using matching techniques [90].

The propagation mechanism pushes the propagators associated with a variable when this variable is modified. There are usually two levels: a first level for the immediate propagation of the modification of a variable and a delayed level that aims at considering once and for all the modification of the variables involved in each propagator. The delayed level is called only when there are no more propagator to call in the first level. The delayed level is interrupted by the first level when the latter is no longer empty.

Of course, each propagator can be parallelized. However a synchronization between them is needed, so it is really difficult to obtain consistent speed up with such an approach. The propagation mechanism can also be parallelized, with the same issues.

Note that the mechanism that is used when solving a Sudoku puzzle corresponds to the application of rules, that is to the call of filtering algorithms (i.e. propagators) until we cannot make any deduction. Thus, this is a propagation mechanism.

1.2 Search

Solutions can be found by searching systematically through the possible assignments of values to variables. A *backtracking scheme* incrementally extends a partial

assignment that specifies consistent values for some of the variables toward a complete solution, by repeatedly choosing a value for another variable. The variables are assigned sequentially.

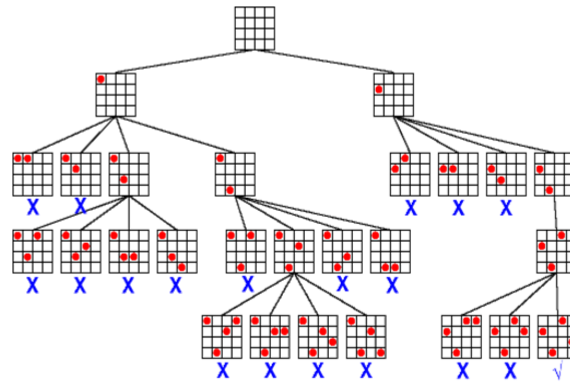

Fig. 1a: Search tree for the four queens problems without propagation.

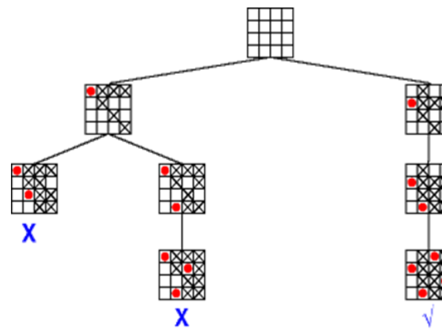

Fig. 1b: Search tree for the four queens problems with weak propagation.

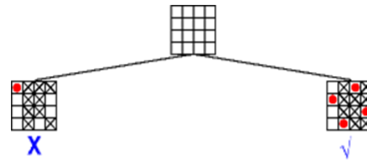

Fig. 1c: Search tree for the four queens problems with strong propagation.

At each node of the search tree, an uninstantiated variable is selected and the node is extended so that the resulting new branches out of the node represent alternative choices that may have to be examined in order to find a solution. The branching strategy, also called the variable-value strategy, determines the next variable to be instantiated, and the order in which the values from its domain are selected. Each

time a variable is assigned a value the propagation mechanism is triggered. If a partial assignment violates any of the constraints, that is if a domain becomes empty, then a backtrack is performed to the most recently assigned variable that still has alternative values available in its domain. Clearly, whenever a partial assignment violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of non-empty variable domains.

When a backtrack occurs, the refutation of the previous choice is usually added to the solver and the propagation mechanism is called. More precisely, if the assignment $x = a$ fails, then a backtrack is performed and the constraint $x \neq a$ is added.

The propagation mechanism allows the reduction of the variable domains and the pruning of the search tree whereas the branching strategy can improve the detection of solutions (or failures for unsatisfiable problems).

In the absence of specific knowledge, defining an efficient variable-value strategy for guiding the search for solutions of a given problem is not an easy task. Thus some generic variable-value strategies have been defined. They either try to apply generic principles such as the first fail principle (i.e., we should try to fail as quickly as possible) [39] or try to detect relations between variables and constraints. In the first case, we have strategies such as min-domain, which selects the variable having the minimum domain size, max-constrained, which prefers variables involved in a lot of constraints, or min-regret which selects the variable that may lead to the largest increase in the cost if it is not selected. The latter case is mainly formed by the impact-based strategy [89], the weighted degree strategy [7] and the activity-based strategy [71]. However, selecting a priori the best variable-value strategy is not an easy task, because no strategy is better than any other in general and because it is quite difficult to identify the types of problems for which a strategy is going to perform well. In addition, there is no robustness among the strategies. Any variable-value strategy can give good results for one problem and really bad results for some others. It is not rare to see the ratio of performance for a pair of strategies going to 1 to 20 (and even more sometimes) according to the problems which are solved.

1.2.1 Search Methods in Solvers

In generic solvers based on constraint programming, the search mechanism is an important part. It is a generic method for controlling the solver that is used to take decisions and to introduce refutation (i.e., the negation of decisions) for solving a whole problem. Decisions can correspond to the assignment of values to variables, but they can also be more complicated. Real-life applications are usually complex and the search mechanism is used for decomposing the problem, for adding some constraints and for solving some subparts. In other words it is used for performing different tasks. For instance, in scheduling applications it is common to deal with the relation between activities, that is which one starts before the other, instead of deciding at what precise moment an activity starts. In problems in which variables are continuous it is also common that a decision splits a variable into two equal parts.

The search is usually totally controlled by the user, who specifies functions that will be called to take a decision and to refute that decision. It is important to note that, in general, there is no other way to reach a state given by a sequence of decisions and refutations than replaying from the beginning that sequence.

1.3 Parallelism and Constraint Programming

In this chapter, we only discuss parallel constraint solving. Some surveys have been written about parallel logic programming [22, 35], and about parallel integer programming [21, 4, 28].

The main approaches to parallel constraint solving can roughly be divided into the following main categories: parallel propagators and propagation; search-space-splitting; portfolio algorithms; distributed CSPs; problem decomposition. Most approaches require communication and synchronization, but the most important issue is load balancing, which refers to the practice of distributing approximately equal amounts of work among tasks so that all processors are kept busy all the time.

After an introduction of the main principles of each approach we will detail the two most important ones: the search space splitting method (i.e., the work-stealing approach) and problem decomposition (i.e., embarrassingly parallel search).

1.3.1 Parallel Propagators and Propagation

The propagators of the constraints, that is the filtering algorithms associated with constraints, can be parallelized. However, this operation is not simple, and some synchronization issues of the domains of the variables arise. In addition, one of the most important drawback is that propagation requires a specific parallelization of each constraint. So, it is not popular because of the same synchronization problems, only a few studies can be found on the parallelization of the propagation mechanism [77, 37, 96]. Thus, parallelizing propagation is challenging [46] and the scalability is limited by Amdahl's law.

1.3.2 Search Space Splitting

Strategies exploring the parallelism provided by the search space are common approaches [87]: when a branching is done, different branches can be explored in parallel.

The work-stealing method dynamically splits the search space during the resolution. It was originally proposed by Burton and Sleep [10] and first implemented in Lisp parallel machines [36]. When a worker has finished exploring a subproblem, it asks other workers for another subproblem. If another worker agrees to the demand, then it dynamically splits its current subproblem into two disjoint subproblems and

sends one subproblem to the starving worker. The starving worker "steals" some work from the busy one. Note that some form of locking is necessary to avoid the case that several starving workers steal the same subproblem. The starving worker asks other workers in turn until it receives a new subproblem. Termination of the work-stealing method must be carefully designed to reduce the overhead when almost all workers are starving, but almost no work remains. Search space splitting is an active research area [110, 43, 69, 16]. This is also one of the methods giving the best results in practice.

Some frameworks sharing the same search tree in memory have been proposed [83, 103]. In this case, a shared list of open nodes in the search tree is maintained (nodes that have at least one child that is still unvisited) and starved processors just pick up the most promising node in the list and expand it. Although this kind of mechanism intrinsically provides excellent load balancing, it is known not to scale beyond a certain number of processors; beyond that point, performance starts to decrease. Indeed, on a shared-memory system, threads must contend with each other to communicate with the memory and the problem is exacerbated by cache consistency transactions. Thus, other approaches that do not use shared memory are preferred.

However, even if the memory is not shared it is not easy to scale up to thousands of processors, because work-stealing consumes communication, synchronization and computation time. To address these issues, Xie and Davenport allocated specific processors to coordination tasks, allowing an increase in the number of processors (linear scaling up to 256 processors) that can be used on a parallel supercomputer before performance starts to decline [107].

Machado et al. proposed a hierarchical works-stealing scheme adapted to a cluster physical infrastructure, in order to reduce the communication overhead [61]. A worker first tries to steal from its local node, before considering remote nodes (starting with the closest remote node). This approach achieved good scalability up to 512 cores for the n-queens and quadratic assignment problems. For constraint optimization problems, maintaining the best solution for each worker would require a large communication and synchronization overhead. However, they observed that the scalability was lowered because of the lazy dissemination of the so-far best solution, i.e., because some workers use an obsolete best solution.

General-purpose programming languages designed for multi-threaded parallel computing such as Charm++ [45] and Cilk++ [56, 9] can ease the implementation of work-stealing approaches. Otherwise, a work-stealing framework such as Bobpp [26, 54] provides an interface between solvers and parallel computers. In Bobpp, the work is shared via a global priority queue and the search tree is decomposed and allocated to the different cores on demand during the search algorithm execution. Periodically, a worker tests whether starving workers exist. In this case, the worker stops the search and the path from the root node to the right highest open node is saved and inserted into the global priority queue. Then, the worker continues the search with the left open node. Otherwise, if no starving worker exists, the worker continues the search locally using the solver. Starving workers are notified of the insertions in the global priority queue, and each one picks up a node and starts

the search. Using or-tools as an underlying solver, Menouer and Le Cun observed good speedups for the Golomb Ruler problem with 13 marks (41.3 with 48 workers) and the 16-queens problem (8.63 with 12 workers) [66, 67] . Other experiments investigate the exploration overhead caused by their approach.

Bordeaux et al. proposed another promising approach based on a search-space-splitting mechanism not based on a work-stealing approach [5]. They use a hashing function implicitly allocating the leaves to the processors. Each processor applies the same search strategy in its allocated search space. Well-designed hashing constraints can address the load-balancing issue. This approach gives a linear speedup from 30 processors for the n-queens problem, but then the speedups stagnate at 30 until to 64 processors. However, it only got moderate results on 100 industrial SAT instances.

Sometimes, for complex applications where very good domain-specific strategies are known, the parallel algorithm should exploit the domain-specific strategy. Moisan et al. proposed a parallel implementation of the classic backtracking algorithm, Limited Discrepancy Search (LDS) [40], which is known to be efficient in a centralized context when a good variable-value selection heuristic is provided [74, 75]. Xie and Davenport proposed that each processor locally uses LDS to search in the trees allocated to them (by a tree-splitting, work-stealing algorithm) but the global system does not replicate the LDS strategy [107].

1.3.3 Portfolio Algorithms

Portfolio algorithms explore the parallelism provided by different viewpoints on the same problem, for instance by using different algorithms or parameter tuning. This idea has also been exploited in a non-parallel context [32].

No communication is required and an excellent level of load balancing is achieved (all workers visit the same search space). Even if this approach causes a high level of redundancy between processors, it shows really good performance. It was greatly improved by using randomized restarts [60] where each worker executes its own restart strategy. More recently, Cire et al. executed the Luby restart strategy, as a whole, in parallel [18]. They proved that it achieves asymptotic linear speedups and, in practice, often obtained linear speedups. Besides, some authors proposed to allow processors to share information learned during the search [38].

One challenge is to find a scalable source of diverse viewpoints that provide orthogonal performance and are therefore of complementary interest. We can distinguish between two aspects of parallel portfolios: if assumptions can be made on the number of available processors then it is possible to handpick a set of solvers and settings that complement each other optimally. If it is not possible to make such assumptions, then we need automated methods to generate a portfolio of any size on demand [5]. So, portfolio designers became interested in feature selection [30, 31, 33, 47]. Features characterize problem instances by number of variables, domain sizes, number of constraints, constraints arities. Many portfolios select the best candidate solvers from a pool based on static features or by learning

the dynamic behavior of solvers. The SAT portfolio `iSAC` [2] and the CP portfolio CP-Hydra [81] use feature selection to choose the solvers that yield the best performance. Additionally, CP-Hydra exploits the knowledge coming from the resolution of a training set of instances by each candidate solver. Then, given an instance, CP-Hydra determines the *k* most similar instances of the training set and determines a time limit for each candidate solver based on the constraint program maximizing the number of solved instances within a global time limit of 30 minutes. Briefly, CP-Hydra determines a switching policy between solvers (Choco, Abscon, Mistral).

In general, the main advantage of the portfolio algorithms approach is that many strategies will be automatically tried at the same time. This is very useful because defining good search strategies is a difficult task.

The best strategy can also be detected in parallel by using estimation techniques [82].

1.3.4 Distributed CSPs

Distributed CSPs is another idea that relates to parallelism, where the problem itself is split into pieces to be solved by different processors. The problem typically becomes more difficult to solve than in the centralized case because no processor has a complete view of the problem. So, reconciling the partial solutions of each subproblem becomes challenging. Problem splitting typically relates to distributed CSPs, a framework introduced by Yokoo et al. in which the problem is naturally split among agents, for example for privacy reasons [109]. Other distributed CSP frameworks have been proposed [41, 15, 24, 55, 104].

1.3.5 Problem Decomposition

The Embarrassingly Parallel Search (EPS) method based on search space splitting with loose communications was first proposed by Régin et al. [92, 93, 95, 63].

When we have *k* workers, instead of trying to split the problem into *k* equivalent subparts, EPS proposes to split the problem into a huge number of subproblems, for instance 30*k* subproblems, and to give these subproblems successively and dynamically to the workers when they need work. Instead of expecting to have equivalent subproblems, EPS expects that *for each worker the sum of the resolution time of its subproblems will be equivalent*. Thus, the idea is not to decompose a priori the initial problem into a set of equivalent subproblems, but to decompose the initial problem into a set of subproblems whose resolution time can be shared in an equivalent way by a set of workers. Note that the subproblems that will be solved by a worker is not known in advance, because this is dynamically determined. All the subproblems are put in a queue and a worker takes one when it needs some work.

The decomposition into subproblems must be carefully done. Subproblems that would have been eliminated by the propagation mechanism of the solver in a se-

quential search must be avoided. Thus, *only problems that are consistent with the propagation are considered*.

Fischetti et al. proposed another paradigm called SelfSplit in which each worker is able to autonomously determine, without any communication between workers, the job parts it has to process [25]. SelfSplit can be decomposed into three phases: the same enumeration tree is initially built by all workers (sampling); when enough open nodes have been generated, the sampling phase ends and each worker applies a deterministic rule to identify and solve the nodes that belong to it (solving); a single worker gathers the results from others (merging). SelfSplit exhibited linear speedups up to 16 processors and good speedups up to 64 processors on five benchmark instances. SelfSplit assumes that sampling is not a bottleneck in the overall computation whereas that can happen in practice [93].

This chapter is organized as follows. First we recall some preliminaries about parallelism and constraint programming. Then we detail the work-stealing method and the embarassingly parallel search. Next we give some results comparing the methods and showing their efficiency on different types of parallel machines. Finally, we conclude.

2 Background

2.1 Parallelism

For the sake of clarity, we will use the notion of *worker* instead of process, processor, core or thread. A worker is an entity which is able to perform some computations. It usually corresponds to a thread/core in a current computer.

2.1.1 Parallelization Measures and Amdahl's Law

Two important parallelization measures are speedup and efficiency. Let $t(c)$ be the wall-clock time of the parallel algorithm where c is the number of cores and let $t(1)$ be the wall-clock time of the sequential algorithm. The speedup $su(c) = t(1)/t(c)$ is a measure indicating how the parallel algorithm performs much faster due to parallelization. The efficiency $\mathit{eff}(c) = su(c)/c$ is a normalized version of speedup, which is the speedup value divided by the number of cores. The maximum possible speedup of a single program as a result of parallelization is known as Amdahl's law [3]. It states that a small portion of the program which cannot be parallelized will limit the overall speedup available from parallelization. Let $B \in [0,1]$ be the fraction of the algorithm that is strictly sequential. The time $t(c)$ that an algorithm takes to finish when being executed on c cores corresponds to $t(c) = t(1)\left(B + \frac{1}{c}(1-B)\right)$. Therefore, the theoretical speedup $su(c)$ is

$$su(c) = \frac{1}{B + \frac{1}{c}(1 - B)}$$

According to Amdahl's law, the speedup can never exceed the number of cores, i.e., a linear speedup. This, in terms of efficiency measure, means that efficiency will always be less than 1.

Note that the sequential and parallel branch-and-bound (B&B) algorithms do not always explore the same search space. Therefore, super-linear speedups in parallel B&B algorithms are not in contradiction with Amdahl's law because processors can access high-quality solutions in early iterations, which in turn bring a reduction in the search tree and problem size.

For the oldest approaches, scalability issues are still to be investigated because of the small number of processors, typically around 16 and up to 64 processors. One major issue is that all approaches may (and a few must) resort to communication. Communication between parallel agents is costly in general: in shared-memory models such as multi-core, this typically means an access to a shared data structure for which one cannot avoid some form of locking; the cost of message-passing cross-CPU is even significantly higher. Communication additionally makes it difficult to get insights on the solving process since the executions are highly inter dependent and understanding parallel executions is notoriously complex.

Most parallel B&B algorithms explore leaves of the search tree in a different order than they would on a single-processor system. This could be a pity in situations where we know a really good search strategy, which is not entirely exploited by the parallel algorithm.

For many approaches, experiments with parallel programming involve a great deal of non-determinism: running the same algorithm twice on the same instance, with identical number of threads and parameters, may result in different solutions, and sometimes in different runtimes.

2.2 Embarrassingly Parallel Computation

A computation that can be divided into completely independent parts, each of which can be executed on a separate worker, is called *embarrassingly parallel* [105].

An embarrassingly parallel computation requires none or very little communication. This means that workers can execute their task, without any interaction with other workers.

Some well-known applications are based on embarrassingly parallel computations, such as the Folding@home project, Low-level image processing, the Mandelbrot set (a.k.a. fractals) or Monte Carlo calculations [105].

Two steps must be defined: the definition of the tasks and the task assignment to the workers The first step depends on the application, whereas the second step is more general. We can either use a static task assignment or a dynamic one.

With a static task assignment, each worker does a fixed part of the problem which is known a priori.

With a dynamic task assignment, a work-pool is maintained and workers consult it to get more work. The work-pool holds a collection of tasks to be performed. Workers ask for new tasks as soon as they finish previously assigned task. In more complex work-pool problems, workers may even generate new tasks to be added to the work-pool.

2.3 Internal and External Parallelization

Techniques which aims at sharing the search tree, like work stealing can be implemented in two different ways in generic CP solvers. Either the solver integrates the capability of traversing the search by several workers at the same time, that is the parallelization is ad-hoc to the solver, or the solver provides some mechanisms like monitors to control the search from outside of the solver. In the former case, we say that the parallelization is made intra solver, whereas for the latter case we say that it is an extra solver parallelization.

Usually the former case is more powerful, but requires some modifications of the source code and so is less flexible and can only be done by the author of the solver [79]. The allocation of the part of the search tree is often specific and it is difficult to control, modify or change it.

The extra solver parallelization adds an algorithm which aims at supervising and controlling the search for solutions. This algorithm also manages the work made by each worker. It interacts with the sequential solver which provides it with some parts of the search tree. The advantage of this approach is that the sequential search is not really modified. The parallel algorithm is defined on the top of the sequential mechanism and use the sequential search in parallel for each worker. Some function are usually given in order to be able to define different kinds of task allocations and to have a better control of the parallelization. Unfortunately, this also has some costs: there are more communications, the protocol must be general and some functions must be provided by the solver like the capability to give a part of the search tree and to restart the search from a given node of the search tree. Thus, this method is often dedicated to some methods of parallelization like the work stealing. For instance, the or-tools solver gives monitors which directly interacts with the internal search.

Some generic framework have been developed in order to deal with external parallelization. Such frameworks provide the user with some features for controlling the search and that are independent of the solver. The role of the framework is to implement the interface with the solver.

Bobpp [27] is a parallel framework oriented towards solving Combinatorial Optimization Problems. It provides an interface between solvers of combinatorial problems and parallel computers. 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 frame-

work for most classes of Combinatorial Optimization Problems, so that they may be solved in as many different parallel architectures as possible. Figure 2 shows how Bobpp interfaces with high-level applications (QAP, TSP, etc.), CP solvers, and different parallel architectures using several parallel programming environments such as Pthreads as well as MPI or more specialized libraries such as Athapascan/Kaapi.

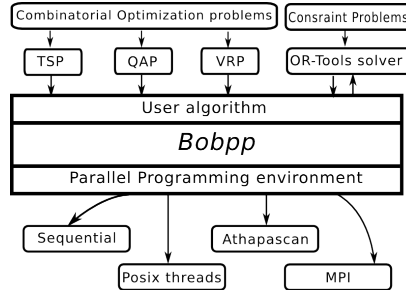

Fig. 2: Bobpp Framework

2.4 Constraint Programming

A constraint network $\mathscr{CN} = (\mathscr{X}, \mathscr{D}, \mathscr{C})$ is defined by

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

Each constraint C_i is associated with a filtering algorithm, often called a propagator, which removes values from the domains of its variables that are not consistent with it. The propagation mechanism applies the filtering algorithms of $\mathscr{C}$ to reduce the domains of variables in turn until no reduction can be done. One of the most interesting properties of a filtering algorithm is arc consistency. We say that a filtering algorithm associated with a constraint establishes arc consistency if it removes all the values of the variables involved in the constraint that are not consistent with the constraint. For instance, consider the constraint $x+3=y$ with the domain of x equal to $D(x) = \{1,3,4,5\}$ and the domain of y equal to $D(y) = \{4,5,8\}$. Then establishing arc consistency will lead to $D(x) = \{1,5\}$ and $D(y) = \{4,8\}$.

For convenience, we will use the word ”problem” to designate a constraint network when it is used to represent the constraint network and not the search for a

solution. We say that a problem P is consistent with the propagation if and only if running the propagation mechanism on P does not trigger a failure.

Now, we can detail the general methods giving the best results in practice.

3 Parallel Search Tree

One of the most popular techniques for combining constraint programming and parallelism is to define a parallel search tree. In other words, we try to traverse the search space in parallel. Usually, this result is achieved by splitting the search tree. This can be done either before the beginning of the search or dynamically during the search. The former case is named static partitioning while the latter is named dynamic partitioning

3.1 Static Partitioning

When we want to use k workers for solving a problem, we can split the initial search tree into k disjoint parts and give one subproblem to each worker. Then, we gather the different intermediate results in order to produce the results corresponding to the whole problem. The advantage of this method is its simplicity. Unfortunately, it suffers from several drawbacks that arise frequently in practice: the times spent to solve each part are rarely well balanced and the communication of the objective value is not good when solving an optimization problem (the workers are independent). The main issue is that the balancing of the workload of the workers is equivalent to the balancing of the parts. Some works has been done on decomposing search trees based on their size in such a way as to equilibrate the parts to be solved [51, 19, 48]. However, the tree size is only approximated and is not strictly correlated with the solving time. In addition, we do not know how to have equivalent subtrees because the propagation mechanism will modify the tree during the search for solutions. Thus, as mentioned by Bordeaux et al. [6], it is quite difficult to ensure that each worker will receive the same amount of work. Hence, this method suffers from some issues of scalability, since the resolution time is the maximum of the resolution time of all workers. In order to remedy these issues, dynamic partitioning of the search tree is preferred.

3.2 Dynamic Partitioning

This strategy, called the work-stealing method, aims to partition the search tree into a set of subtrees, and schedule them during the execution of the search algorithm in order to have good load balancing between the different workers. Thus, workers

each solve part of the problems and when a worker is waiting, it ”steals” some work from another worker. This general mechanism can be described as follow: when a worker W no longer has any work, it asks another worker V whether it has some work to give it. If the answer is positive, then the worker V splits the search tree it is currently solving into two subtrees and gives one of them to the waiting worker W. If the answer is negative then W asks another worker U, until it gets some work to do or all the workers have been considered. The work-stealing approach partly resolves the balancing issue of the simple static decomposition method, mainly because the decomposition is dynamic. Therefore, it does not need to be able to split a search tree into well-balanced parts at the beginning.

This method has been implemented in a lot of solvers (Comet [70] or ILOG Solver [84] for instance), and in several ways [99, 44, 111, 17] depending on whether the work to be done is centralized or not, on the way the search tree is split (into one or several parts) or on the communication method between workers.

For example, the study presented by Xie and Davenport [106] 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 subtree is detected that is a root of a large subtree, the workers generate a large number of its subtrees and put them in a work-pool in order to have better load balancing. Fischetti et al. [25], propose a work-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 estimation 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. [44] propose the use of a master that 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.

In the Bobpp framework, the work is shared thanks to a Global Priority Queue (GPQ). The search tree is decomposed and allocated to the different workers on demand and during the execution of the search algorithm. A unit of work corresponds to the solving of a subtree of the search tree. This subtree is the subtree of the search tree rooted at a given node, called the local root. This subtree is called the local search tree.

Periodically, a working worker tests whether waiting worker(s) exist(s). If this is the case, the working worker stops the search in the current node and gives a part of its local search tree, that is the subtree rooted at a node. In other words, it puts a node of the local search tree in the GPQ and continues to solve the remaining part of its local search tree. The waiting workers are notified by the insertion of a new node in the GPQ, and a waiting worker picks up the node and starts the solving of the subtree rooted at this node. This partitioning strategy has been presented in detail by Menouer and Le Cun [65].

There are two main questions that have to be answered to efficiently implement this mechanism: How do we start the search for a solution is a given subtree? And which subtree is given? In the next sections we will see that there is no perfect answer to these questions.

3.2.1 Local Subtree Solving

Conceptually there is no difficulty to start a search from a given node of the search tree. However, in practice, this is quite different. The main question is to be able to set the solver in the correct state corresponding to the root node. A state of a solver is defined by the domains of the variables and the internal data structure required by the propagators and some other data that may have been defined by the user. This means that some actions have been performed to reach a state. Thus, the question is how can we restore a state or how can we move from one state to another state? This is the continuation problem in computer science [94]. The restoration of a state of the search tree depends on the solver. Some solvers, such as or-tools [86] or Choco [13] or Oscar [98], use a trail mechanism. This means that they save some data when the search is going down in order to be able to restore them when the search is going up (i.e., backtracking). Some others, such as Gecode, use different mechanisms to avoid restoring the memory. The internal mechanism may lead to different strategies to move from one state to another one. Some solvers directly implement continuations [102].

There are usually three possibles methods: the state is explicitly saved, the state is recomputed from the current state or the state is recomputed from scratch, that is from the root of the search tree.

The possible implementations of these methods depend on whether the solver uses a generic search procedure or an internal one.

Internal search procedure.

Such a procedure means that the solver has total control over what can be done during the search for a solution. Some interactions with the search are possible but these are limited and the user cannot define its own data structures in a way which is not controlled by the solver. Usually internal search corresponds to a search method in which the only decisions are assignments or refutations. In this case, this means that nodes can be seen as subproblems. Thus, computing a state is equivalent to restarting the search from a given subproblem of the initial problem. This can be easily done by simply imposing the specific definition of the subproblem and running the propagation of the solver once. Therefore the cost is not really expensive. In addition saving a sub problem is not costly in memory so it is a good alternative to the explicit saving of the state.

The two other methods can also be used, that is we can easily backtrack to the lowest common ancestor (lca) of the current and the target node and then replay

the search from the lca to the target node. We can measure whether this is more efficient than direct instantiation with the subproblem of the target node or not. The replay from scratch is usually less interesting than restarting the problem with the constraints defining the subproblem of the target node.

Generic search procedure.

As we mentioned in the Introduction, if a generic search is used then there is no way to deduce the state of a node of the search tree, mainly because we cannot know what are the data structures that are used. We have no information about the data that are defined by the user. This means that the state needs to be recomputed from the path going from the root to the target node. In this case, we say that the search is replayed. Unfortunately this has a cost.

Since we cannot define precisely the structure of a state, the memorization of a state can only be done by copying the whole memory, which is possible only when there are only a few variables and constraints. So in general the first method is not possible. The second method is only possible for the current worker. Therefore when some work is given to another worker, the third method is usually used. Replaying the search from the root node has a cost that depends on the length of the path. Therefore, it is common to study the consequences of some choices. This is the purpose of the next section.

3.2.2 Subtree Definition

When a worker needs some work it asks the other workers to give it a part of their current work. We discuss here how a worker can answer this request. All jobs, that is all given subtrees are not equivalent for several reasons: it can be expensive to replay the corresponding state and the solving times of the subtrees may strongly differ.

The first problem can be solved by the worker which gives some work by defining a strategy for selecting the given node of its local search tree. The simplest strategy consists of giving the current node and triggering a local backtrack in order to continue solving the local search tree (See Figure 3). It is also possible to give the next available open node.

However this method does not take into account the time to replay the state of a node. Thus, some other methods [64] have been developed. Notably, the node that is the closest to the root can be transferred. Some experiments have shown that this decreases the number of decision replays by a factor of 2. Figure 4 illustrates this approach.

The second question about the amount of work that is given is more important for the global solving time. In fact, dynamic partitioning has a termination issue. When the whole search for solutions is almost done, there are more and more workers without work and so there are more and more workers asking for some work. At the

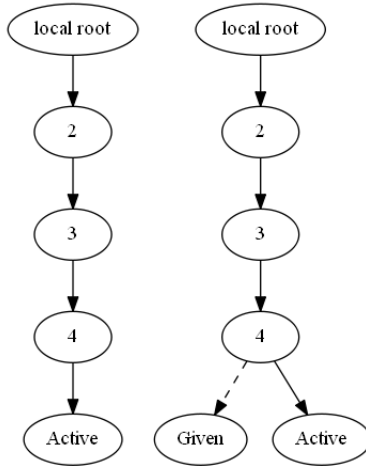

Fig. 3: Simple work separation

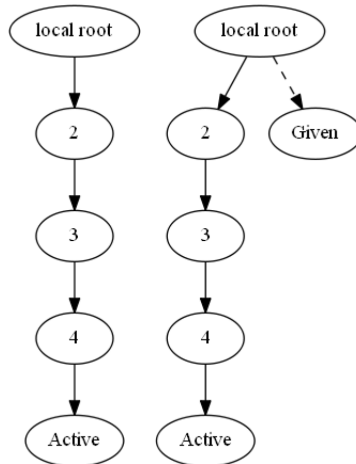

Fig. 4: Transmission of the subtree rooted at the node closest to the root.

same time, there are fewer and fewer workers that can give a part of their work. In addition the quantity of work that can be given is less and less important. Thus, we have more requests, fewer possible responders and less work to give. This is why we often observe a decrease in performance when the search is almost ended. Thus, we generally observe that the method scales well for a small number of workers whereas it is difficult to maintain a linear gain when the number of workers becomes larger, even though some methods have been developed to try to remedy this issue [108, 70]. Note that it is possible to have an immediate failure, that is the propagation of the new node may fail.

In order to speed up the termination of the algorithm, we should avoid giving a search tree that will be too small to be solved. Unfortunately, it is difficult to estimate the time that will be required to traverse a search tree, otherwise we would be able to have nice decompositions. One possible solution is to consider the depth of a node and to relate it to the solving time of the subtree rooted at that node. Thus, if a node is at a depth that is greater than a given threshold then the node cannot be given to another worker. This means that some workers will not be able to give any node. This idea usually improves the global behavior of the parallelization. However, it can be further improved by using a dynamic threshold that mainly depends on the depth. 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 subtrees explored by the different workers. Conversely, choosing a high threshold makes the algorithm similar to dynamic partitioning without a threshold, which makes load balancing easier between the workers but increases the exploration of redundant nodes. For instance, Menouer [64] uses a threshold equal to $2\log(\#workers)$, where $\#workers$ is the number of available workers. In addition, the threshold is increased each time a worker no longer has work. The maximum value is defined by $7\log(\#workers)$.

Even if the depth is a poor estimation of the quantity of work needed to solve a subtree, a threshold based on depth improves the work-stealing approach in practice. Figure 5 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) [73] using 12 cores on an Intel machine (12 cores and 48 GB of RAM). As a result, the computation time decreases with increasing threshold value until an optimal threshold (value of 25) is reached. After this optimal value the computation time increases again.

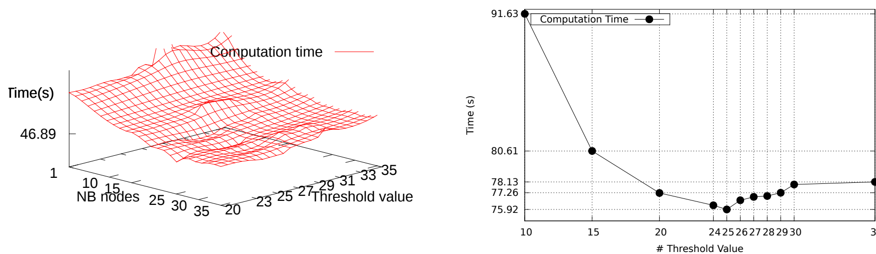

Fig. 5: Variation of the computation time for solving the Naval Battle problem on 12 cores according to the threshold value [68]

4 Problem Decomposition

The idea of Embarrassingly Parallel Search (EPS) is to statically decompose the initial problem into a huge number of subproblems that are consistent with propagation (i.e., running the propagation mechanism on them does not detect any inconsistency). These subproblems are added to a queue, which is managed by a master. Then, each waiting worker takes a subproblem from the queue and solves it. The process is repeated until all the subproblems have been solved. The assignment of the subproblems to workers is dynamic and there is no communication between the workers. EPS is based on the idea that if there is a large number of subproblems to solve then the resolution times of the workers will be balanced even if the resolution times of the subproblems are not. In other words, load balancing is automatically obtained in a statistical sense.

We will detail this method in this section.

4.1 Principles

This approach relies on the assumption that the resolution time of disjoint subproblems is equivalent to the resolution time of the union of these subproblems. If this condition is not met, then the parallelization of the search of a solver (not necessarily a CP Solver) based on any decomposition method, such as simple static decomposition, work stealing or embarrassingly parallel method may be unfavorably impacted.

This assumption does not seem too strong because experiments do not show such a poor behavior with a CP Solver. However, it has been observed in some cases with a MIP Solver.

We have seen that decomposing the initial problem into the same number of subproblems as workers may cause unbalanced resolution times for different workers. Thus, the idea of EPS is to strongly increase the number of considered subproblems, in order to define an embarrassingly parallel computation leading to good performance.

Before going into further details of the implementation, a property can be established. While solving a problem, we will use the following terminology:

- *active time of a worker:* the sum of the resolution times of a worker (the decomposition time is excluded).
- *inactive time of a worker:* the difference between the elapsed time for solving all the subproblems (the decomposition time is excluded) and the active time of the worker.

The EPS approach is mainly based on the following remark.

Remark 1. The active time of all the workers may be well balanced even if the resolution time of each subproblem is not well balanced.

Since a worker may solve several subproblems, their resolution times can be different while their sum remains equal to a given value.

The main challenge of a decomposition is not to define equivalent problems, it is to avoid having some workers without work whereas some others are running. We do not need to know in advance the resolution time of each subproblem. We just expect that the workers will have equivalent activity time. In order to reach that goal, EPS decomposes the initial problem into a lot of subproblems. This increases our chance to obtain well-balanced activity times for the workers, because we increase our chance to be able to obtain a combination of resolution times leading to the same activity time for each worker.

For instance, when the search space tends to be not equilibrated, there are subproblems that will take a longer time to be solved. By having a lot of subproblems we increase our chance to split these subproblems into several parts having comparable resolution time and so to obtain a well-balanced load for the workers at the end. It also reduces the relative importance of each subproblem with respect to the resolution of the whole problem.

Here is an example of the advantage of using a lot of subproblems. Consider a problem which requires 140s to be solved sequentially and for which we have four workers. If we split the problem into four subproblems then we have the following resolution times: $20, 80, 20, 20$. We will need 80s to solve these subproblems in parallel. Thus, we gain a factor of $140/80 = 1.75$. Now if we split again each subproblem into four subproblems we might obtain the following subproblems represented by their resolution time: $((5,5,5,5),(20,10,10,40),(2,5,10,3),(2,2,8,8))$. In this case, we might use the following assignment: worker1 : $5+20+2+8 = 35$; worker2 : $5+10+2+10 = 27$; worker3 : $5+10+5+3+2+8 = 33$ and worker4 : $5+40 = 45$. The elapsed time is now 45s and we gain a factor of $140/45 = 3.1$. By splitting the subproblems again, we will reduce the average resolution time of the subproblems and expect to break the 40s subproblem. Note that decomposing a subproblem further does not run away the risk of increasing the elapsed time.

Property 1. Let P be an optimization problem, or a satisfaction problem in which we search for all solutions. If P is split into subproblems whose maximum resolution time is *tmax*, then

(i) the minimum resolution time of the whole problem is *tmax*;

(ii) the maximum inactivity time of a worker is less than or equal to *tmax*.

Proof. Suppose that a worker W has an inactivity time which is greater than *tmax*. Consider the moment where W started to wait after its activity time. At this time, there are no more available subproblems to solve, otherwise W would be active. All active workers are then finishing their last task, whose resolution is bounded by *tmax*. Thus, the remaining resolution time of each of these other workers is less than or equal to *tmax*. Hence a contradiction.

The next section shows that the decomposition should be carefully done.

4.1.1 Subproblems Generation: a Top-Down Method

We assume that we want to decompose a problem into q subproblems.

Unlike the work-stealing approach, EPS does not aim to decompose the search tree into subtrees instead, it aims to decompose the whole problem into a set of subproblems. These two decompositions are really different even if at first glance they look similar. Notably the relation tp the sequential approach is different. There exists one rule when we try to parallelize a sequential process that should not be forgotten: *We should avoid doing something in parallel that we would not have done sequentially*.

The simplest method that can be considered does not satisfy this remark. It is a simple decomposition that is done as follows:

1. We consider any ordering of the variables $x_1,...,x_n$.
2. We define A_k to be the Cartesian product $D(x_1) \times ... \times D(x_k)$.
3. We compute the value k such that $|A_{k-1}| < q \leq |A_k|$.

Each assignment of A_k defines a subproblem and so A_k is the sought decomposition.

This method works well for some problems such as the nqueens or the Golomb ruler, but it is really bad for some other problems, because a lot of assignments of A may be trivially not consistent. Consider for instance that x_1, x_2 and x_3 have the three values $\{a,b,c\}$ in their domains and that there is an alldiff constraint involving these three variables. The Cartesian product of the domains of these variables contains 27 tuples. Among them only six ((a,b,c), (a,c,b), (b,a,c),(b,c,a),(c,a,b), (c,b,a)) are not inconsistent with the alldiff constraint. That is, only $6/27 = 2/9$ of the generated subproblems are not trivially inconsistent. It is important to note that most of these inconsistent problems would never be considered by a sequential search, and so we violate the previous rule. For some problems we have observed more than 99% of the generated problems were detected inconsistent by running the propagation (Figure 6). Thus, another method is needed to avoid this issue.

EPS solves this issue by *generating only subproblems that are consistent with the propagation*, that is such that if we run the propagation mechanism on them then there is no failure. This means that they are not known to be inconsistent. Such subproblems will also be considered by a sequential process, so they no longer violate the parallel-sequential rule we mentioned.

The generation of q such subproblems becomes more complex because the number of subproblems consistent with the propagation may not be related to the Cartesian product of some domains. A simple algorithm could be to perform a Breadth First Search (BFS) in the search tree, until the desired number of subproblems consistent with the propagation is reached. Unfortunately, it is not easy to perform a BFS efficiently mainly because BFS is not an incremental algorithm like Depth-First Search (DFS). Therefore, we can use a process similar to an iterative deepening depth-first search [52]: we repeat a Depth-Bounded Depth First Search (DBDFS), in other words a DFS that never visits nodes located at a depth greater than a given value, increasing the bound until we have generated the right number of subproblems. However, even if the depth of a search tree can be precisely defined, it is not

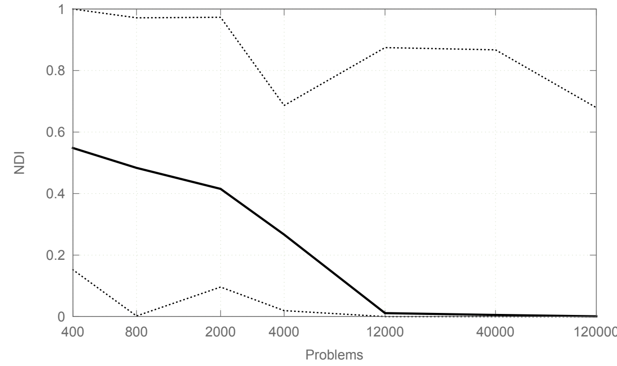

Fig. 6: Percentage of subproblems consistent with the propagation (NDI) generated by the simple decomposition method for all problems. The geometric mean is in bold, dashed lines represent minimum and maximum values

easy to relate this notion to the number of variable already assigned. In fact, some variables may be assigned by propagation, and this is the case for the latest values of the domain of the variables. Thus, it is better to replace the depth limit by another simple limit.
We consider a set $Y \subseteq X$ of variables: we only assign the variables of Y and we stop the search when they are all assigned. In other words, we never try to assign a variable that is not in Y. This process is repeated until all assignments of Y consistent with the propagation have been found. Each branch of a search tree computed by this search defines an assignment. We will denote by A_Y the set of assignments computed with $Y \subseteq X$. To generate q subproblems, we repeat the DBDFS by adding variables to Y if necessary until we have $|A_Y| \geq q$.

For convenience and simplicity, a static ordering of the variables is used.

This method can be improved in two ways:

1. We try to estimate some good set of variables Y in order to avoid repeating too many DBDFS: For instance, if for a given Y we produce only $q/1000$ subproblems and if the size of the domains of the next three non-assigned variables is 10, then we can deduce that we need to add at least three variables to Y.
2. In order to avoid repeating the same DFS for the first variables while repeating DBDFS, we store in a table constraint the previously computed assignments. More precisely, if we have computed A_Y then we use a table constraint containing all these assignments when we look for $A_{Y'}$ with $Y \subseteq Y'$.

Large Domains

This method can be adapted to large domains. A new step must be introduced in the algorithm in the latest iteration. If the domain of the latest considered variable,

denoted by lx, is large then each of its values cannot be considered individually. In this case, its domain is split into a fixed number of parts and we use each part as a value. Then, either the desired number of subproblems is generated or we have not been able to reach that number. In the latter case, the domain of lx is split again, for instance by splitting each part into two new parts (this multiplies by at most 2 the number of generated subproblems) and we check whether the generated number of subproblems is fine or not. This process is repeated until the right number of subproblems is generated or the domain of lx is totally decomposed, that is each part corresponds to a value. In the latter case, we continue the algorithm by selecting a new variable.

Parallelization of the decomposition

When there are a lot of workers, for instance 500, the decomposition into subproblems may represent an important part of the resolution if it is done sequentially. Two reasons can explain this behavior: the ratio between a sequential method and a parallel one is large because we have 500 workers and not 6, 12 or 40. Since there are a lot of workers, there is also much more work to do because the initial problem needs to be decomposed into a larger number of subproblems. Thus, between a sequential solution with w workers and another one with $W > w$ workers, the potential loss in term of computation power is W/w whereas we have at least W/w more work to do. So, it is can be necessary to parallelize the decomposition into subproblems.

Experiments give some information:

1. The difference in the total work (i.e., activity time) done by the workers decreases when the number of subproblems increases. This is not a linear relation. There is a huge difference between the activity times of the workers when there are fewer than five subproblems per worker. These differences decrease when there are more than five subproblems per worker.
2. A simple decomposition into subproblems that may be inconsistent quickly causes some issues because inconsistencies are detected very quickly.
3. Splitting an initial problem into a small set of subproblems is fast compared to the overall decomposition time and compared to the overall resolution time.

These observations show that a compromise has to be found and an iterative process decomposing the initial problem in three phases has to be defined. In the first phase, the whole problem is decomposed into only a few subproblems because the relative cost is small even with an unbalanced workload. However, we should be careful with the first phase (i.e., starting with probably inconsistent subproblems) because it can have an impact on the performance. Finally, the most important thing seems to be to generate five subproblems because we could restart from these subproblems to decompose further and such a decomposition should be reasonably well balanced.

Thus, a method in three main phases has been designed:

- An initial phase where one subproblem per worker is generated as quickly as possible. This phase does not consume time and may remain sequential.
- A main phase which aims to generate five subproblems per worker. Each subproblem is consistent with the propagation. This phase can be divided into several steps to reach that goal while balancing the work among the workers.
- A final phase which consists of generating $K \geq 30$ subproblems per worker from the set of subproblems computed by the main phase.

4.1.2 Subproblems Generation: a Bottom-Pp Method

Another method for finding the requested subproblems has been proposed by Malapert et al [63]. It is a bottom-up decomposition that tries to find in a depth-first manner the depth d at which we can generate the q subproblems (Figure 7).

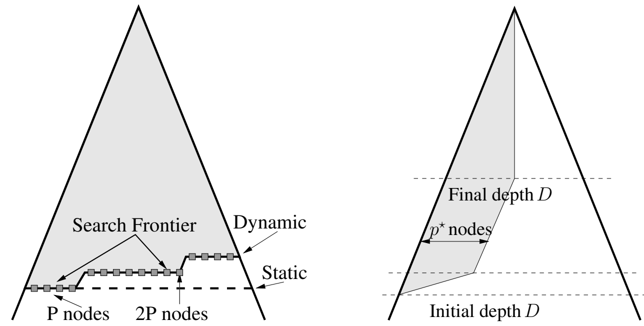

Fig. 7: Bottom-up decomposition and estimation

The algorithm aims to identify the topmost search frontier with approximately $p^* = q$ open nodes by sampling and estimation of the sought depth. The procedure can be divided into three phases:

1. a partial tree is built by sampling the top of the real search tree;
2. we estimate the level widths of the real tree;
3. we determine the decomposition depth d with a greedy heuristic.

Since we need to explore the top of the search tree, an upper bound D on the decomposition depth is fixed. The maximum decomposition depth D must be chosen according to the number of workers and the expected number of subproblems per worker. If D is too small, the decomposition could generate too few subproblems. If D is too large, the sampling time increases while the decomposition quality could decrease.

The sampling phase builds a partial tree with at most p^* assignments on a level using a depth-first search. The number of assignments (i.e., open nodes in the search tree) at each level is counted by a callback. The maximum depth D is reduced each

time there are p^* assignments at a given level. If the sampling ends within its limits, then the top of the tree has been entirely visited and no estimation is needed. Otherwise, we need to estimate the widths of the topmost levels of the tree depending on the partial tree. This estimation is a straightforward adaptation of the one proposed by Cornuejols et al. [20] to deal with n-ary search trees. In practice, the main issue is that the higher the arity is, the lower the precision of the estimation. Therefore, the heuristics that is used minimizes the absolute deviation between the estimated number of nodes and the expected number p^*. If several levels have an identical absolute deviation, then the lowest level with an estimated number of subproblems greater than or equal to p^* is selected.

4.1.3 Implementation

EPS involves three tasks: the definition of the subproblem (TaskDefinition) the task assignment of subproblems to the workers (TaskAssignment) and a task that aims at gathering solutions and/or objective values: TaskResultGathering. In this step, the answers to all the subproblems are collected and combined in some way to form the output (i.e., the answer to the initial problem).

For convenience, we create a master (i.e., a coordinator process) which is in charge of these operations. So, it creates the subproblems (TaskDefinition) holds the work-pool and assigns tasks to workers (TaskAssignment) and fetches the computations made by the workers (TaskResultGathering).

We detail these operations for the satisfaction and optimization problems.

Satisfaction Problems

- The TaskDefinition operation consists of computing a partition of the initial problem P into a set S of subproblems.
- The TaskAssignment operation is implemented by using a FIFO data structure (i.e., a queue). Each time a subproblem is defined it is added to the back of the queue. When a worker needs some work it takes a subproblem from the queue.
- The TaskResultGathering operation is quite simple: when searching for a solution it stops the search when one is found; when searching for all solutions, it just gathers the solutions returned by the workers.

Optimization Problems

In case of optimization problems we have to manage the best value of the objective function computed so far. Thus, the operations are slightly modified.

- The TaskDefinition operation consists of computing a partition of the initial problem P into a set S of subproblems.
- The TaskAssignment operation is implemented by using a queue. Each time a subproblem is defined it is added to the back of the queue. The queue is also associated with the best objective value computed so far. When a worker needs some work, the master gives it a subproblem from the queue. It also gives it the best objective value computed so far.
- The TaskResultGathering operation manages the optimal value found by the worker and the associated solution.

Note that there is no other communication, that is when a worker finds a better solution, the other workers that are running cannot use it for improving their current resolution. So, if the absence of communication may increase our performance, this aspect may also lead to a decrease in performance. Fortunately, we do not observe this bad behavior in practice. We can see here another argument for having a lot of subproblems in case of optimization problems: the resolution of a subproblem should be short in order to improve the transmission of a better objective value and to avoid performing work that could have been ignored with a better objective value.

4.1.4 Size of the partition

One important question is: how many subproblems should be generated?

This is mainly an experimental question. However, in order to have good scalability, this number should be defined in relation to the number of workers that are involved. More precisely, it is more consistent to have q subproblems per worker than a total of q subproblems.

It appears that this number does not depend on the type of problem that is considered. Some experiments show that a minimum of 30 subproblems per worker is required.

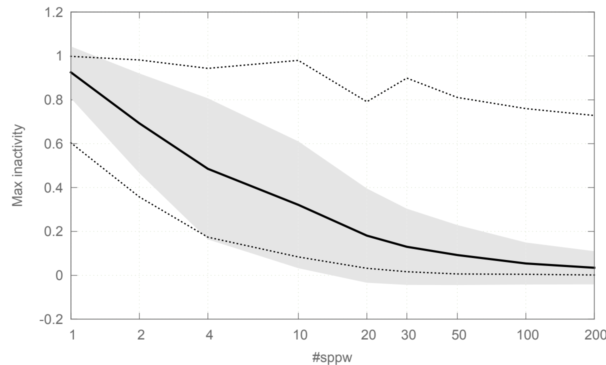

Fig. 8: Percentage of maximum inactivity time of the workers (geometric mean)

Figure 8 shows that the percentage of the maximum inactivity time of the workers decreases when the number of subproblems per worker is increased. From 20 subproblems per worker, we observe that on average the maximum inactivity time represents less than 20% of the resolution time.

4.2 Determinism

EPS can be modified to return the same solution as the sequential algorithm, which can be useful in several scenarios such as debugging or performance evaluation.

We assume that the whole problem is decomposed into the subproblems $P_1, \ldots, P_p$ in that order and that they are selected by respecting that order.

The first solution found by the sequential algorithm belongs to the satisfiable subproblem P_i with the smallest index, that is the leftmost solution. Consider that the parallel algorithm finds the first solution for the subproblem P_j such that $j > i$. Then, it is not necessary to solve problems P_k such that $k > j$ and one must only wait for each problem P_k such that $k < j$ and then determine the leftmost solution, the satisfiable subproblem with the smallest index.

This can easily be extended for optimization problems by slightly modifying the cutting constraints. Usually, when a new solution is found a cutting constraint is stated that only allows strictly better solutions. On the contrary to other constraints, the cutting constraint is always propagated while backtracking. Here, if a solution is found when solving the subproblem P_j, then the cutting constraint only allows strictly improving solution for subproblems $k \geq j$, but also allows equivalent solutions for subproblems $k < j$.

So, the parallel algorithm returns the same solution as the sequential one if the subproblems are visited in the same order. Moreover, the solution returned by the parallel algorithm does not depend on the number of workers, but only on the decomposition.

5 Comparison Between the Work-Stealing Approach and EPS

The EPS method has several advantages compared to the work-stealing approach. We can cite the most important ones:

• there is almost no communication between workers and the communication between the master and the workers is really weak.

• the method is independent of the solver. There is no need to know the solver in detail or to have access to internal data structures. It can be used with a generic search without any problem.

• there is no termination issue.

• very easy problems can be considered by a worker without causing any issue. There is no need for any threshold.

• there is no issue of replaying a part of the search with a generic search, because the problem is decomposed and not split.
• the method is quite simple.
• the method can be easily adapted to use distributed machines.
• by saving the order in which the subproblems have been executed, we can simply replay a resolution in parallel. This costs almost nothing and helps a lot with the debugging of applications. Determinism is easy to achieve.

The work-stealing approach has several advantages:
• we have fine and dynamic control over the way the search is explored and split.
• the method manages the repartition of the work and if only one worker is working then its work will be shared, whereas this may not be the case with EPS.
• there is no setup time because there is no a priori decomposition of the problem

6 Experiments

These experiments come from Malapert et al. [63]. More information and more details can be found in [63].

6.1 Benchmark Instances

We consider instances of satisfaction and optimization problems. We ignore the problem of finding a first feasible solution because the parallel speedup can be completely uncorrelated to the number of workers, making the results hard to analyze. We consider optimization problems for which the same variability can be observed, but to a lesser extent because an optimality proof is required.

We perform a huge number of tests and we select the most representative ones.

The first set, called `fzn`, is a selection of 18 instances selected from more than 5000 instances either from the repository maintained by [49] or directly from the Minizinc 1.6 distribution written in the FlatZinc language [78]. Each instance is solved in more than 500 seconds and less than 1 hour with Gecode. The selection is composed of one unsatisfiable, six enumerations, and 11 optimization instances.

The set `xcsp` is composed of instances from the categories ACAD and REAL of XCSP 2.1 [97]. It consists of difficult instances that can be solved within 24 hours by `Choco2` [62]. A first subset, called `xcsp1`, is composed of five unsatisfiable and five enumeration instances whereas the second subset, called `xcsp2`, is composed of 11 unsatisfiable and three enumeration instances. The set `xcsp1` is composed of instances easier to solve than those of `xcsp2`.

6.1.1 Implementation Details

Three CP solvers are used: Choco2 2.1.5 written in Java, Gecode 4.2.1 and OR-tools rev. 3163 written in C++. Threads [76, 50] and MPI [57, 34] technologies are used. The typical difference between them is that threads (of the same process) run in a shared memory space, while MPI is a standardized and portable message-passing system to exchange information between processes running in separate memory spaces. Therefore, Thread technology does not handle multiple nodes of a cluster whereas MPI does.

In C++, Threads are implemented by using pthreads, a POSIX library [76, 50] used by Unix systems. In Java, the standard Java Thread technology [42] is used.

OR-tools uses a sequential top-down decomposition and C++ Threads. Gecode uses a parallel top-down decomposition and C++ Threads or MPI technologies. In fact, Gecode will use C++ pthread on the multi-core computer, OpenMPI on the data center, and MS-MPI on the cloud platform. Gecode and OR-tools both use the lex variable selection heuristic because the top-down decomposition requires a fixed variable ordering. Choco2 uses a bottom-up decomposition and Java Threads. In every case, the jobs are scheduled in FIFO to mimic as much as possible the sequential algorithm so that speedups are relevant. We always take the value selection heuristic that selects the smallest value, whatever heuristic that may be.

6.1.2 Execution Environments

We use three execution environments that are representative of computing platforms available nowadays: multi-core, data center and cloud computing.

Multi-core is a Dell computer with 256 GB of RAM and four Intel E7-4870 2.40 GHz processors running on Scientific Linux 6.0 (each processor has 10 cores).

Data Center is the “Centre de Calcul Interactif” hosted by the Université Nice Sophia Antipolis, which provides a cluster composed of 72 nodes (1152 cores) running on CentOS 6.3, each node with 64 GB of RAM and two Intel E5-2670 2.60 GHz processors (eight cores). The cluster is managed by OAR [11], i.e., a versatile resource and task manager. As Thread technology is limited to a single node of a cluster, Choco2 can use up to 16 physical cores whereas Gecode can use any number of nodes thanks to MPI.

Cloud Computing is a cloud platform managed by the Microsoft company (Microsoft Azure) that enables applications to be deployed on Windows Server technology [58]. Each node has 56 GB of RAM and Intel Xeon E5-2690E 2.6 GHz processors (eight physical cores) We were allowed to simultaneously use three nodes (24 cores) managed by the Microsoft HPC Cluster 2012 [72].

Some computing infrastructures provide hyper-threading technologies, which improves parallelization of computations (doing multiple tasks at once). For each core that is physically present, the operating system addresses two logical cores, and shares the workload among them when possible. The multi-core computer pro-

vides hyper-threading, whereas it is deactivated on the cluster, and not available on the cloud.

The time limit for solving each instance is set to 12 hours whatever be the solver. Usually, we use two workers per physical core ($w = 2c$) because hyper-threading is efficient in our experiments. The target number p of subproblems depends linearly on the number w of workers ($p = 30 \times w$), which allows statistical balance of the workload without increasing too much the total overhead [92].

Let t be the solving time (in seconds) of an algorithm and let su be the speedup of a parallel algorithm. In the tables, a row gives the results obtained by different algorithms for a given instance. For each row, the best solving times and speedups are indicated in bold. Dashes indicate that the instance is not solved by the algorithm. Question marks indicate that the speedup cannot be computed because the sequential solver does not solve the instance within the time limit. Arithmetic means, abbreviated AM, are computed for solving times, whereas geometric means, abbreviated GM, are computed for speedups and efficiency. Missing values, i.e., dashes and question marks, are ignored when computing statistics.

6.2 Multi-core

In this section, we use parallel solvers based on Thread technologies to solve the instances of `xcsp1` or the nqueens problem using a multi-core computer. Let us recall that there are two workers per physical core because hyper-threading is activated ($w = 2c = 80$). We show that EPS frequently gives linear speedups, and outperforms the work-stealing approach proposed by [100] and [80].

Table 1 gives the solving times and speedups of the parallel solvers using 80 workers for the `xcsp1` instances. `Choco2`, `Gecode` and `OR-tools` use `lex`. They are also compared to a work stealing approach denoted `Gecode`-WS [100, 80]. First, implementations of EPS are faster and more efficient than the work-stealing. EPS often reaches linear speedups in the number of cores whereas it never happens for the work stealing. Even worse, three instances are not solved within the 12-hour time limit using work-stealing whereas they are using the sequential solver.

Decomposition is the key to the bad performance on the instances `knights-80-5` and `lemma-100-9-mod`. The decomposition of `knights-80-5` takes more than 1,100 seconds and generates too many subproblems, which precludes any speedup. The issue is lessened using the sequential decomposition of `OR-tools` and is resolved by the parallel top-down decomposition of `Gecode`. Note also that the sequential solving times of `OR-tools` and `Gecode` respectively are 20 and 40 times higher. Similarly, the long decomposition time of `Choco2` for `lemma-100-9-mod` leads to a low speedup. However, the moderate efficiency of `Choco2` and `Gecode` for `squares-9-9` is not caused by the decomposition.

`Gecode` and `OR-tools` are often more efficient and faster than `Choco2`. The solvers show different behaviors even when using the same variable selection heuris-

Instances	Choco2		Gecode		OR-tools		Gecode-WS	
	t	*su*	*t*	*su*	*t*	*su*	*t*	*su*
costasArray-14	240.0	**38.8**	62.3	19.1	**50.9**	33.4	594.0	2.0
knights-80-5	1133.1	1.5	**548.7**	**37.6**	2173.9	18.5	–	–
latinSquare-dg-8_all	328.1	39.2	251.7	**42.0**	**166.6**	35.2	4488.5	2.4
lemma-100-9-mod	123.4	4.1	6.7	10.1	**1.8**	**22.9**	3.0	22.3
ortholatin-5	249.9	36.0	421.7	13.5	**167.7**	**38.1**	2044.6	2.8
pigeons-14	899.1	15.5	**211.8**	**39.1**	730.3	18.5	–	–
quasigroup5-10	123.5	**32.5**	18.6	26.4	**17.0**	**36.9**	22.8	21.5
queenAttacking-6	**622.5**	**28.5**	15899.1	?	–	–	–	–
series-14	39.3	32.9	**11.3**	**34.2**	16.2	28.7	552.3	0.7
squares-9-9	1213.0	16.1	**17.9**	18.4	81.4	**35.0**	427.8	0.8
AM (*t*) or GM (*su*)	497.2	17.4	1745.0	24.0	378.4	28.7	1161.9	3.3

Table 1: Solving times and speedups (40-cores machine). Gecode and OR-tools use the lex heuristic

tic because their propagation mechanisms and decompositions differ. Furthermore, the parallel top-down decomposition of Gecode does not preserve the ordering of the subproblems with regard to the sequential algorithm.

6.3 Data Center

In this section, we study the influence of the search strategy on the solving times and speedups, the scalability up to 512 workers, and compare EPS to a work-stealing approach.

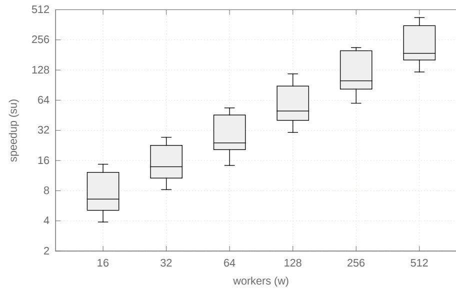

Fig. 9: Scalability up to 512 workers (Gecode, lex, data center).

Table 2 compares the Gecode implementations of EPS and work-stealing (WS) for solving xcsp instances using 16 or 512 workers. EPS is faster and more efficient than work-stealing. With 512 workers, EPS is on average almost 10 times

Instances	$w = 16$				$w = 512$			
	EPS		WS		EPS		WS	
	t	su	t	su	t	su	t	su
cc-15-15-2	–	–	–	–	–	–	–	–
costasArray-14	**64.4**	**13.6**	69.3	12.7	**3.6**	**243.8**	17.7	49.8
crossword-m1c[1]	**240.6**	**13.1**	482.1	6.6	**18.7**	**168.6**	83.1	38.0
crossword-m1[2]	**171.7**	**14.5**	178.5	13.9	**13.3**	**187.3**	57.8	43.0
knights-20-9	**5190.7**	?	38347.4	?	**153.4**	?	3312.4	?
knights-25-9	**7462.3**	?	–	–	**214.9**	?	–	–
knights-80-5	**1413.7**	**11.5**	8329.2	2.0	**49.3**	**329.8**	282.6	57.5
langford-3-17	24351.5	?	**21252.3**	?	**713.5**	?	7443.5	?
langford-4-18	**3203.2**	?	25721.2	?	**94.6**	?	5643.1	?
langford-4-19	**26871.2**	?	–	–	**782.5**	?	–	–
latinSquare-dg-8_all	**613.5**	**13.1**	621.2	13.0	**23.6**	**341.7**	124.4	64.7
lemma-100-9-mod	**3.4**	**14.7**	5.8	8.6	**1.0**	**51.4**	2.5	19.7
ortholatin-5	**309.5**	**14.1**	335.8	13.0	**10.4**	**422.0**	71.7	61.0
pigeons-14	**383.3**	**14.5**	6128.9	0.9	**15.3**	**363.1**	2320.2	2.4
quasigroup5-10	**27.1**	**13.5**	33.7	10.8	**1.7**	**211.7**	9.8	37.3
queenAttacking-6	42514.8	?	**37446.1**	?	**1283.9**	?	9151.5	?
ruler-70-12-a3	**96.6**	**15.1**	105.5	13.8	**3.7**	**389.3**	67.7	21.5
ruler-70-12-a4	**178.9**	**14.4**	185.2	13.9	**6.0**	**429.5**	34.1	75.5
series-14	**22.5**	**13.4**	56.9	5.3	**1.1**	**264.0**	8.2	36.9
squares-9-9	**22.8**	**11.1**	44.3	5.7	**1.3**	**191.7**	7.6	33.7
AM (t) or GM (su)	5954.8	13.5	8196.7	7.4	178.53	**246.2**	1684.6	33.5

[1]crossword-m1-words-05-06 [2]crossword-m1c-words-vg7-7_ext

Table 2: Speedups and solving times for xcsp (Gecode, lex, data center, $w = 16$ or 512)

faster than work-stealing. It is also more efficient because they both parallelize the same sequential solver. On the multi-core machine, Gecode is faster than Choco2 on most instances of xcsp1. Five instances that are not solved within the time limit by Gecode are not reported in Table 2. Six instances are not solved with 16 workers whereas twelve instances are not solved with the sequential solver. By way of comparison, only five instances are not solved by Choco2 using the lex heuristics whereas all instances are solved in sequential or parallel when using dom/wdeg or dom/bwdeg. Once again, this highlights the importance of the search strategy.

Figure 9 is a boxplot of the speedups with different numbers of workers for solving fzn instances. The median of speedups are around $\frac{w}{2}$ on average and their dispersion remains low.

Instance	EPS		WS	
	t	su	t	su
market_split_s5-02	**467.1**	**24.3**	658.6	17.3
market_split_s5-06	**452.7**	**24.4**	650.7	17.0
market_split_u5-09	**468.1**	**24.4**	609.2	18.7
pop_stress_0600	**874.8**	**10.8**	2195.7	4.3
nmseq_400	**342.4**	**8.5**	943.2	3.1
pop_stress_0500	**433.2**	**10.1**	811.0	5.4
fillomino_18	**160.2**	**13.9**	184.6	12.1
steiner-triples_09	**108.8**	**17.2**	242.4	7.7
nmseq_300	**114.5**	**6.6**	313.1	2.4
golombruler_13	**154.0**	**20.6**	210.4	15.1
cc_base_mzn_rnd_test.11	**1143.6**	**7.3**	2261.3	3.7
ghoulomb_3-7-20	**618.2**	**6.8**	3366.0	1.2
still_life_free_8x8	**931.2**	**9.6**	1199.4	7.5
bacp-6	**400.8**	**16.4**	831.0	7.9
depot_placement_st70_6	**433.9**	**18.3**	1172.5	6.8
open_stacks_01_wbp_20_20_1	**302.7**	**17.6**	374.1	14.3
bacp-27	**260.2**	**16.4**	548.4	7.8
still_life_still_life_9	**189.0**	**16.9**	196.8	16.2
talent_scheduling_alt_film117	**22.7**	**74.0**	110.5	15.2
AM (t) or GM (su)	**414.7**	**15.1**	888.4	7.7

Table 3: Solving times and speedups for fzn (Gecode, lex, cloud, $w = 24$)

6.4 Cloud Computing

The systems are deployed on the Microsoft Azure cloud platform. The available computing infrastructure is organized as follows: cluster nodes compute the application; one head node manages the cluster nodes; and proxy nodes load-balance communication between cluster nodes. Unlike a data center, cluster nodes may be far from each other and communication time may take longer. Proxy nodes requires two cores and are managed by the service provider. Here, three nodes of height cores with 56 GB of RAM memory provide 24 workers (cluster nodes) managed by MPI.

Table 3 compares the Gecode implementations of EPS and work stealing for solving the fzn instances with 24 workers. Briefly, EPS is always faster than work-stealing, and therefore more efficient because they both parallelize the same sequential solver. Work–stealing suffers from a higher communication overhead in the cloud than in a data center. Furthermore, the architecture of the computing infrastructure and the location of cluster nodes are mostly unknown, which precludes improvements to work-stealing such as those proposed by Machado et al. [61] or Xie and Davenport [107].

6.5 Comparison with Portfolios

Portfolio approaches exploit the variability of performance that is observed between several solvers, or several parameter settings for the same solver. We use four portfolios. The portfolio `CPHydra` [81] uses feature selection on top of the solvers `Mistral`, `Gecode` and `Choco2`. `CPHydra` uses case-based reasoning to determine how to solve an unseen problem instance by exploiting a case base of problem solving experience. It aims to find a feasible solution within 30 minutes. It does not handle optimization or all-solution problems and the time limit is hard coded. The other static and fixed-size portfolios (`Choco2`, CAG, `OR-tools`) use different variable selection heuristics as well as randomization and restarts. Details about `Choco2` and CAG can be found in [62]. The CAG portfolio extends the `Choco2` portfolio by also using the solvers `AbsCon` and `Gecode`. So, CAG always produces better results than `Choco2`. The `OR-tools` portfolio was the gold medalist of the Minizinc challenge 2013 and 2014. It can seem unfair to compare parallel solvers and portfolios using different numbers of workers, but designing scalable portfolios (up to 512 workers) is a difficult task and almost no implementation is publicly available.

Table 4 gives the solving times of EPS and portfolios for the `xcsp` instances on the data center. First, `CPHydra` with 16 workers only solves two among 16 unsatisfiable instances (`cc-15-15-2` and `pigeons-14`), but in less than 2 seconds whereas these are difficult for all other approaches. `OR-tools` is the second-least efficient approach because it solves fewer problems and often takes longer as confirmed by its low Borda score. The parallel `Choco2` using `dom/wdeg` is better on average than the `Choco2` portfolio even if the portfolio solves a few instances much faster such as `scen11-f5` or `queensKnights-20-5-mul`. In this case, the diversification provided by the portfolio outperforms the speedups offered by the parallel B&B algorithm. This is emphasized for the CAG portfolio, which solves all instances and obtains several of the best solving times. The parallel `Gecode` with 16 workers is often slower and less robust than the portfolios `Choco2` and CAG. However, increasing the number of workers to 512 clearly makes it the fastest solver, but still less robust because five instances are not solved within the time limit.

To conclude, the `Choco2` and CAG portfolios are more robust thanks to their inherent diversification, but their solving times vary more from one instance to another. With 16 workers, implementations of EPS outperform the `CPHydra` and `OR-tools` portfolio, are competitive with the `Choco2` portfolio, and are slightly dominated by the CAG portfolio. In fact, the good scaling of EPS is a key to beat the portfolios.

7 Conclusion

We have presented different methods for combining constraint programming techniques and parallelism, such as parallelization of the propagator or parallel propa-

Instances	EPS			Portfolio		
	Choco2	Gecode		Choco2	CAG	OR-tools
	$w = 16$	$w = 16$	$w = 512$	$w = 14$	$w = 23$	$w = 16$
cc-15-15-2	2192.1	–	–	1102.6	**3.5**	1070.0
costasArray-14	649.9	64.4	**3.6**	6180.8	879.4	1368.8
crossword-m1-words-05-06	204.6	240.6	**18.7**	512.3	512.3	22678.1
crossword-m1c-words-vg7-7_ext	1611.9	171.7	**13.3**	721.2	721.2	13157.2
fapp07-0600-7	2295.7	–	–	37.9	**3.2**	–
knights-20-9	491.3	5190.7	153.4	3553.9	**0.8**	–
knights-25-9	1645.2	7462.3	214.9	9324.8	**1.1**	–
knights-80-5	1395.6	1413.7	**49.3**	1451.5	301.6	32602.6
langford-3-17	3062.2	24351.5	**713.5**	8884.7	8884.7	–
langford-4-18	538.3	3203.2	**94.6**	2126.0	2126.0	–
langford-4-19	2735.3	26871.2	**782.5**	12640.2	12640.2	–
latinSquare-dg-8_all	294.8	613.5	**23.6**	65.1	36.4	4599.8
lemma-100-9-mod	145.3	3.4	**1.0**	435.3	50.1	38.2
ortholatin-5	362.4	309.5	**10.4**	4881.2	4371.0	4438.7
pigeons-14	2993.3	383.3	**15.3**	12336.9	5564.5	12279.6
quasigroup5-10	451.5	27.1	**1.7**	3545.8	364.3	546.0
queenAttacking-6	**706.4**	42514.8	1283.9	2644.5	2644.5	–
queensKnights-20-5-mul	5209.5	–	–	235.3	**1.0**	–
ruler-70-12-a3	42.8	96.6	**3.7**	123.5	123.5	8763.1
ruler-70-12-a4	1331.3	178.9	**6.0**	1250.2	1250.2	–
scen11-f5	–	–	–	45.3	**8.5**	–
series-14	338.9	22.5	**1.1**	1108.3	302.1	416.2
squares-9-9	115.9	22.8	**1.3**	1223.7	254.3	138.3
squaresUnsat-19-19	**3039.8**	–	–	4621.1	4621.1	–
Arithmetic mean	1385.0	5954.8	**178.5**	3293.8	1902.7	7853.6

Table 4: Solving times of EPS and portfolio (data center)

gation. We have detailed the most popular methods: work-stealing methods based on search tree splitting, and EPS, the embarrassingly parallel search method, which is based on problem decomposition. These methods give good results in practice. They have been tested on a single multi-core machine, on a data center and on the cloud. However, it seems that the scaling performance of EPS is better. In addition EPS is simple and easy to implement. The future will tell us whether it can replace the work stealing approach in CP solvers. In any case, the obtained results show that we can efficiently combine parallelism and CP and often expect results that are almost linear.

References

1. 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA. IEEE Computer Society (2004). URL http:

//ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=9460
2. Amadini, R., Gabbrielli, M., Mauro, J.: An Empirical Evaluation of Portfolios Approaches for Solving CSPs. In: C. Gomes, M. Sellmann (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, *Lecture Notes in Computer Science*, vol. 7874, pp. 316–324. Springer Berlin Heidelberg (2013). DOI 10.1007/978-3-642-38171-3_21. URL http://dx.doi.org/10.1007/978-3-642-38171-3_21
3. Amdahl, G.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67, pp. 483–485. ACM, New York, NY, USA (1967)
4. Bader, D., Hart, W., Phillips, C.: Parallel Algorithm Design for Branch and Bound. In: H. G (ed.) Tutorials on Emerging Methodologies and Applications in Operations Research, *International Series in Operations Research & Management Science*, vol. 76, pp. 5–1–5–44. Springer New York (2005). DOI 10.1007/0-387-22827-6_5
5. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with Massively Parallel Constraint Solving. In: Boutilier [8], pp. 443–448
6. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Boutilier [8], pp. 443–448
7. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI, vol. 16, p. 146 (2004)
8. Boutilier, C. (ed.): IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009 (2009)
9. Budiu, M., Delling, D., Werneck, R.: DryadOpt: Branch-and-bound on distributed data-parallel execution engines. In: Parallel and Distributed Processing Symposium (IPDPS), 2011 IEEE International, pp. 1278–1289. IEEE (2011)
10. Burton, F.W., Sleep, M.R.: Executing Functional Programs on a Virtual Tree of Processors. In: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, FPCA '81, pp. 187–194. ACM, New York, NY, USA (1981)
11. Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounie, G., Neyron, P., Richard, O.: A Batch Scheduler with High Level Components. In: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02, CCGRID '05, pp. 776–783. IEEE Computer Society, Washington, DC, USA (2005). URL http://dl.acm.org/citation.cfm?id=1169223.1169583
12. Choco, T.: Choco: an open source java constraint programming library. Ecole des Mines de Nantes, Research report **1**, 10–02 (2010)
13. Choco, T.: Choco: an open source java constraint programming library. Ecole des Mines de Nantes, Research report **1**, 10–02 (2010)
14. Choco solver
http://www.emn.fr/z-info/choco-solver/ (2013). Accessed: 14-04-2014
15. Chong, Y.L., Hamadi, Y.: Distributed Log-Based Reconciliation. In: Proceedings of the 2006 Conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva Del Garda, Italy, pp. 108–112. IOS Press, Amsterdam, The Netherlands, The Netherlands (2006). URL http://dl.acm.org/citation.cfm?id=1567016.1567045
16. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-Based Work Stealing in Parallel Constraint Programming. In: Gent [29], pp. 226–241
17. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Gent [29], pp. 226–241
18. Cire, A.A., Kadioglu, S., Sellmann, M.: Parallel Restarted Search. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, AAAI'14, pp. 842–848. AAAI Press (2014). URL http://dl.acm.org/citation.cfm?id=2893873.2894004
19. Cornuéjols, G., Karamanov, M., Li, Y.: Early estimates of the size of branch-and-bound trees. INFORMS Journal on Computing **18**(1), 86–96 (2006)

20. Cornuéjols, G., Karamanov, M., Li, Y.: Early Estimates of the Size of Branch-and-Bound Trees. INFORMS Journal on Computing **18**, 86–96 (2006)
21. Crainic, T.G., Le Cun, B., Roucairol, C.: Parallel branch-and-bound algorithms. Parallel combinatorial optimization **1**, 1–28 (2006)
22. De Kergommeaux, J.C., Codognet, P.: Parallel logic programming systems. ACM Computing Surveys (CSUR) **26**(3), 295–336 (1994)
23. De Nicola, R., Ferrari, G.L., Meredith, G. (eds.): Coordination Models and Languages, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 24-27, 2004, Proceedings, *Lecture Notes in Computer Science*, vol. 2949. Springer (2004)
24. Ezzahir, R., Bessière, C., Belaissaoui, M., Bouyakhf, E.H.: DisChoco: A platform for distributed constraint programming. In: DCR'07: Eighth International Workshop on Distributed Constraint Reasoning - In conjunction with IJCAI'07, pp. 16–21. Hyderabad, India (2007). URL https://hal-lirmm.ccsd.cnrs.fr/lirmm-00189778
25. Fischetti, M., Monaci, M., Salvagnin, D.: Self-splitting of workload in parallel computation. In: H. Simonis (ed.) Integration of AI and OR Techniques in Constraint Programming: 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings, pp. 394–404. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-07046-9_28. URL http://dx.doi.org/10.1007/978-3-319-07046-9_28
26. Galea Fran c., Le Cun, B.: 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), pp. 779–785 (2007)
27. Galea, F., Le Cun, B.: 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), pp. 779–785 (2007)
28. Gendron, B., Crainic, T.G.: Parallel branch-and-bound algorithms: Survey and synthesis. Operations research **42**(6), 1042–1066 (1994)
29. Gent, I.P. (ed.): Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings, *Lecture Notes in Computer Science*, vol. 5732. Springer (2009)
30. Gomes, C., Selman, B.: Algorithm Portfolio Design: Theory vs. Practice. In: Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence, pp. 190–197 (1997)
31. Gomes, C., Selman, B.: Search strategies for hybrid search spaces. In: Tools with Artificial Intelligence, 1999. Proceedings. 11th IEEE International Conference, pp. 359–364. IEEE (1999)
32. Gomes, C., Selman, B.: Hybrid Search Strategies For Heterogeneous Search Spaces. International Journal on Artificial Intelligence Tools **09**, 45–57 (2000)
33. Gomes, C., Selman, B.: Algorithm Portfolios. Artificial Intelligence **126**, 43–62 (2001)
34. Gropp, W., Lusk, E.: The MPI communication library: its design and a portable implementation. In: Scalable Parallel Libraries Conference, 1993., Proceedings of the, pp. 160–165. IEEE (1993)
35. Gupta, G., Pontelli, E., Ali, K.A., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. ACM Transactions on Programming Languages and Systems (TOPLAS) **23**(4), 472–602 (2001)
36. Halstead, R.: Implementation of Multilisp: Lisp on a Multiprocessor. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, LFP '84, pp. 9–17. ACM, New York, NY, USA (1984)
37. Hamadi, Y.: Optimal Distributed Arc-Consistency. Constraints **7**, 367–385 (2002)
38. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a Parallel SAT Solver. Journal on Satisfiability, Boolean Modeling and Computation **6**(4), 245–262 (2008)
39. Haralick, R., Elliot, G.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14**, 263–313 (1980)

40. Harvey, W.D., Ginsberg, M.L.: Limited Discrepancy Search. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes, pp. 607–615 (1995)
41. Hirayama, K., Yokoo, M.: Distributed Partial Constraint Satisfaction Problem. In: Principles and Practice of Constraint Programming-CP97, pp. 222–236. Springer (1997)
42. Hyde, P.: Java thread programming, vol. 1. Sams (1999)
43. Jaffar, J., Santosa, A.E., Yap, R.H.C., Zhu, K.Q.: Scalable Distributed Depth-First Search with Greedy Work Stealing. In: 16th IEEE International Conference on Tools with Artificial Intelligence [1], pp. 98–103. URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=9460
44. Jaffar, J., Santosa, A.E., Yap, R.H.C., Zhu, K.Q.: Scalable distributed depth-first search with greedy work stealing. In: ICTAI [1], pp. 98–103. URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=9460
45. Kale, L., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++, vol. 28. ACM (1993)
46. Kasif, S.: On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction networks. Artificial Intelligence **45**, 275–286 (1990)
47. Kautz, H., Horvitz, E., Ruan, Y., Gomes, C., Selman, B.: Dynamic Restart Policies. 18th National Conference on Artificial Intelligence AAAI/IAAI **97**, 674–681 (2002)
48. Kilby, P., Slaney, J.K., Thiébaux, S., Walsh, T.: Estimating search tree size. In: AAAI, pp. 1014–1019 (2006)
49. Kjellerstrand, H.: Håkan Kjellerstrand's Blog. http://www.hakank.org/ (2014)
50. Kleiman, S., Shah, D., Smaalders, B.: Programming with threads. Sun Soft Press (1996)
51. Knuth, D.E.: Estimating the efficiency of backtrack programs. Mathematics of Computation **29**, 121–136 (1975)
52. Korf, R.: Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence **27**, 97109 (1985)
53. Kowalski, R.: Algorithm = logic + control. Commun. ACM **22**(7), 424–436 (1979)
54. Le Cun, B., Menouer, T., Vander-Swalmen, P.: Bobpp. http://forge.prism.uvsq.fr/projects/bobpp (2007)
55. Léauté, T., Ottens, B., Szymanek, R.: FRODO 2.0: An open-source framework for distributed constraint optimization. In: Boutilier [8], pp. 160–164
56. Leiserson, C.E.: The Cilk++ concurrency platform. The Journal of Supercomputing **51**(3), 244–257 (2010)
57. Lester, B.: The art of parallel programming. Prentice Hall Englewood Cliffs, NJ (1993)
58. Li, H.: Introducing Windows Azure. Apress, Berkely, CA, USA (2009)
59. Lodi, A., Milano, M., Toth, P. (eds.): Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 6140. Springer (2010)
60. Luby, M., Sinclair, A., Zuckerman, D.: Optimal Speedup of Las Vegas Algorithms. Inf. Process. Lett. **47**, 173–180 (1993)
61. Machado, R., Pedro, V., Abreu, S.: On the Scalability of Constraint Programming on Hierarchical Multiprocessor Systems. In: ICPP, pp. 530–535. IEEE (2013)
62. Malapert, A., Lecoutre, C.: À propos de la bibliothèque de modèles XCSP. In: 10èmes Journées Francophones de Programmation par Contraintes(JFPC'15). Angers, France (2014)
63. Malapert, A., Régin, J., Rezgui, M.: Embarrassingly parallel search in constraint programming. J. Artif. Intell. Res. (JAIR) **57**, 421–464 (2016). DOI 10.1613/jair.5247. URL http://dx.doi.org/10.1613/jair.5247
64. Menouer, T.: Paralllisations de Mthodes de Programmation Par Contraintes. Ph.D. thesis, Universit de Versailles Saint-Quentin-en-Yvelines (2015)
65. Menouer, T., Cun, B.L.: 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, pp. 1771–1777 (2013). DOI 10.1109/IPDPSW.2013.210. URL http://doi.ieeecomputersociety.org/10.1109/IPDPSW.2013.210

66. Menouer, T., Le Cun, B.: 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, pp. 1771–1777 (2013)
67. Menouer, T., Le Cun, B.: 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)
68. Menouer, T., Rezgui, M., Cun, B.L., Régin, J.: Mixing static and dynamic partitioning to parallelize a constraint programming solver. International Journal of Parallel Programming **44**(3), 486–505 (2016). DOI 10.1007/s10766-015-0356-7. URL http://dx.doi.org/10.1007/s10766-015-0356-7
69. Michel, L., See, A., Hentenryck, P.V.: Transparent Parallelization of Constraint Programming. INFORMS Journal on Computing **21**, 363–382 (2009)
70. Michel, L., See, A., Hentenryck, P.V.: Transparent parallelization of constraint programming. INFORMS Journal on Computing **21**(3), 363–382 (2009)
71. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems, pp. 228–243. Springer (2012)
72. Microsoft Corporation: Microsoft HPC Pack 2012 R2 and HPC Pack 2012. http://technet.microsoft.com/en-us/library/jj899572.aspx (2015)
73. Minizinc challenge http://www.minizinc.org/challenge2012/challenge.html (2012). Accessed: 14-04-2014
74. Moisan, T., Gaudreault, J., Quimper, C.G.: Parallel Discrepancy-Based Search. In: Principles and Practice of Constraint Programming, *Lecture Notes in Computer Science*, vol. 8124, pp. 30–46. Springer Berlin Heidelberg (2013)
75. Moisan, T., Quimper, C.G., Gaudreault, J.: Parallel Depth-bounded Discrepancy Search. In: H. Simonis (ed.) Integration of AI and OR Techniques in Constraint Programming: 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings, pp. 377–393. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-07046-9_27. URL http://dx.doi.org/10.1007/978-3-319-07046-9_27
76. Mueller, F., et al.: A Library Implementation of POSIX Threads under UNIX. In: USENIX Winter, pp. 29–42 (1993)
77. Nguyen, T., Deville, Y.: A distributed arc-consistency algorithm. Science of Computer Programming **30**(12), 227 – 250 (1998). DOI http://dx.doi.org/10.1016/S0167-6423(97)00012-9. URL http://www.sciencedirect.com/science/article/pii/S0167642397000129. Concurrent Constraint Programming
78. NICTA Optimisation Research Group: MiniZinc and FlatZinc. http://www.g12.csse.unimelb.edu.au/minizinc/ (2012)
79. Nielsen, M.: Parallel Search in Gecode. Master's thesis, KTH Royal Institute of Technology (2006)
80. Nielsen, M.: Parallel Search in Gecode. Master's thesis, KTH Royal Institute of Technology (2006)
81. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Irish Conference on Artificial Intelligence and Cognitive Science, pp. 210–216 (2008)
82. Palmieri, A., Régin, J., Schaus, P.: Parallel strategies selection. In: M. Rueher (ed.) Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings, *Lecture Notes in Computer Science*, vol. 9892, pp. 388–404. Springer (2016). DOI 10.1007/978-3-319-44953-1_25. URL http://dx.doi.org/10.1007/978-3-319-44953-1_25
83. Perron, L.: Search Procedures and Parallelism in Constraint Programming. In: Principles and Practice of Constraint Programming – CP'99: 5th International Conference, CP'99, Alexandria, VA, USA, October 11-14, 1999. Proceedings, pp. 346–360. Springer Berlin Heidelberg, Berlin, Heidelberg (1999). DOI 10.1007/978-3-540-48085-3_25. URL http://dx.doi.org/10.1007/978-3-540-48085-3_25

84. Perron, L.: Search procedures and parallelism in constraint programming. In: CP, *Lecture Notes in Computer Science*, vol. 1713, pp. 346–360 (1999)
85. Perron, L., Nikolaj, V.O., Vincent, F.: Or-Tools. Tech. rep., Google (2012)
86. Perron, L., Nikolaj, V.O., Vincent, F.: Or-Tools. Tech. rep., Google (2012)
87. Pruul, E., Nemhauser, G., Rushmeier, R.: Branch-and-bound and Parallel Computation: A historical note. Operations Research Letters **7**, 65–69 (1988)
88. cois Puget, J.F.: ILOG CPLEX CP Optimizer : A C++ implementation of CLP. http://www.ilog.com/ (1994)
89. Refalo, P.: Impact-based search strategies for constraint programming. In: M. Wallace (ed.) CP, *Lecture Notes in Computer Science*, vol. 3258, pp. 557–571. Springer (2004)
90. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings AAAI-94, pp. 362–367. Seattle, Washington (1994)
91. Régin, J.C.: Global Constraints: a Survey, chap. Global Constraints: a survey. Springer (2011)
92. Régin, J.C., Rezgui, M., Malapert, A.: Embarrassingly Parallel Search. In: Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings, pp. 596–610. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-40627-0_45. URL http://dx.doi.org/10.1007/978-3-642-40627-0_45
93. Régin, J.C., Rezgui, M., Malapert, A.: Improvement of the Embarrassingly Parallel Search for Data Centers. In: B. O'Sullivan (ed.) Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8656, pp. 622–635. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-10428-7_45. URL http://dx.doi.org/10.1007/978-3-319-10428-7_45
94. Reynolds, J.C.: The discoveries of continuations. Lisp and Symbolic Computation. **6**(3/4), 33248. (1993)
95. Rezgui, M., Régin, J.C., Malapert, A.: Using Cloud Computing for Solving Constraint Programming Problems. In: First Workshop on Cloud Computing and Optimization, a conference workshop of CP 2014. Lyon, France (2014)
96. Rolf, C.C., Kuchcinski, K.: Parallel Consistency in Constraint Programming. PDPTA '09: The 2009 International Conference on Parallel and Distributed Processing Techniques and Applications **2**, 638–644 (2009)
97. Roussel, O., Lecoutre, C.: Xml representation of constraint networks format. http://www.cril.univ-artois.fr/CPAI08/XCSP2_1Competition.pdf (2008)
98. Schaus, P.: Oscar, operational research in scala. URL https://bitbucket.org/oscarlib/oscar/wiki/Home
99. Schulte, C.: Parallel search made simple. In: N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Mller, L. Perron, C. Schulte (eds.) Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000. Singapore (2000). URL http://www.ict.kth.se/~cschulte/paper.php?id=Schulte:TRICS:2000
100. Schulte, C.: Parallel Search Made Simple. In: "Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000, pp. 41–57. Singapore (2000)
101. Schulte, C.: Gecode: Generic Constraint Development Environment. http://www.gecode.org/ (2006)
102. Van Hentenryck, P., Michel, L.: The objective-cp optimization system. In: C. Schulte (ed.) Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 8124, pp. 8–29. Springer (2013). DOI 10.1007/978-3-642-40627-0_5. URL http://dx.doi.org/10.1007/978-3-642-40627-0_5
103. Vidal, V., Bordeaux, L., Hamadi, Y.: Adaptive K-Parallel Best-First Search: A Simple but Efficient Algorithm for Multi-Core Domain-Independent Planning. In: Proceedings of the Third International Symposium on Combinatorial Search. AAAI Press (2010)

104. Wahbi, M., Ezzahir, R., Bessiere, C., Bouyakhf, E.H.: DisChoco 2: A Platform for Distributed Constraint Reasoning. In: Proceedings of the IJCAI'11 workshop on Distributed Constraint Reasoning, DCR'11, pp. 112–121. Barcelona, Catalonia, Spain (2011). URL http://dischoco.sourceforge.net/
105. Wilkinson, B., Allen, M.: Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers, 2nd edition edn. Prentice-Hall Inc. (2005)
106. Xie, F., Davenport, A.: 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, pp. 53–58 (2009)
107. Xie, F., Davenport, A.: Massively Parallel Constraint Programming for Supercomputers: Challenges and Initial Results. In: Lodi et al. [59], pp. 334–338. DOI 10.1007/978-3-642-13520-0_36. URL http://dx.doi.org/10.1007/978-3-642-13520-0_36
108. Xie, F., Davenport, A.J.: Massively parallel constraint programming for supercomputers: Challenges and initial results. In: Lodi et al. [59], pp. 334–338
109. Yokoo, M., Ishida, T., Kuwabara, K.: Distributed Constraint Satisfaction for DAI Problems. In: Proceedings of the 1990 Distributed AI Workshop. Bandara, TX (1990)
110. Zoeteweij, P., Arbab, F.: A Component-Based Parallel Constraint Solver. In: De Nicola et al. [23], pp. 307–322
111. Zoeteweij, P., Arbab, F.: A component-based parallel constraint solver. In: De Nicola et al. [23], pp. 307–322