Journal of Artificial Intelligence Research 57 (2016) 421-464 Submitted 06/16; published 11/16

Embarrassingly Parallel Search in Constraint Programming

Arnaud Malapert ARNAUD.MALAPERT@UNICE.FR
Jean-Charles Régin JEAN-CHARLES.REGIN@UNICE.FR
Mohamed Rezgui REZGUI@I3S.UNICE.FR
Université Côte d'Azur, CNRS, I3S, France

Abstract

We introduce an Embarrassingly Parallel Search (EPS) method for solving constraint problems in parallel, and we show that this method matches or even outperforms state-of-the-art algorithms on a number of problems using various computing infrastructures. EPS is a simple method in which a master decomposes the problem into many disjoint subproblems which are then solved independently by workers. Our approach has three advantages: it is an efficient method; it involves almost no communication or synchronization between workers; and its implementation is made easy because the master and the workers rely on an underlying constraint solver, but does not require to modify it. This paper describes the method, and its applications to various constraint problems (satisfaction, enumeration, optimization). We show that our method can be adapted to different underlying solvers (`Gecode`, `Choco2`, `OR-tools`) on different computing infrastructures (multi-core, data centers, cloud computing). The experiments cover unsatisfiable, enumeration and optimization problems, but do not cover first solution search because it makes the results hard to analyze. The same variability can be observed for optimization problems, but at a lesser extent because the optimality proof is required. EPS offers good average performance, and matches or outperforms other available parallel implementations of `Gecode` as well as some solvers portfolios. Moreover, we perform an in-depth analysis of the various factors that make this approach efficient as well as the anomalies that can occur. Last, we show that the decomposition is a key component for efficiency and load balancing.

1. Introduction

In the second half of the 20th century, the frequency of processors doubled every 18 months or so. It has now been clear for a few years that this period of "free lunch", as put by Sutter and Larus (2005), is behind us. As outlined by Bordeaux, Hamadi, and Samulowitz (2009), the available computational power will keep increasing exponentially, but the increase will be in terms of number of available processors, not in terms of frequency per unit. **Multi-core processors** are now the norm which raises significant challenges for software development. **Data centers** for high-performance computing are readily accessible by many in academia and industry. **Cloud computing** (Amazon, Microsoft Azure, Google, . . .) offers massive infrastructures for rent on which computing and storage can be used on demand. With such facilities anyone can now gain access to super-computing facilities at a moderate cost. **Distributed Computing** offers possibilities to put computational resources in common and effectively obtains massive capabilities. Examples include `Seti@home` (Anderson, Cobb, Korpela, Lebofsky, & Werthimer, 2002), `Distributed.net` (Distributed Computing Technologies Inc, 20) and `Sharcnet` (Bauer, 2007). The main challenge is therefore to scale, *i.e.*, to cope with this growth.

Constraint programming (CP) is an appealing technology for a variety of combinatorial problems which has grown steadily in the last three decades. The strengths of CP are the use of constraint propagation combined with efficient search algorithms. Constraint propagation aims at removing combinations of values from variable domains which cannot appear in any solution. Over a number of years, possible gains offered by the parallel computing have attracted the attention.

Parallel computing is a form of computation in which many calculations are carried out simultaneously (Almasi & Gottlieb, 1989) operating on the principle that large problems can often be divided into smaller ones, which are then solved in parallel. Different forms of parallel computing exist: bit-level, instruction level, data and task parallelism. Task parallelism is a common approach for parallel branch-and-bound (B&B) algorithms (Mattson, Sanders, & Massingill, 2004) and is achieved when each processor executes a different thread (or process) on same or different data. Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. For example, when memory is shared, several tasks of an algorithm can modify the same data at the same time. This could render the program incorrect. Mutual exclusion allows a worker to lock certain resources to obtain exclusive access, but can create starvation because the other workers must wait until the worker frees the resources. Moreover, the indeterminism of the parallel programs makes the behaviour of the execution unpredictable, *i.e.* the results of different program runs may differ. So, communication and synchronization among different sub-tasks can address this issue, but are typically some of the greatest obstacles to good performance. Another central bottleneck is load balancing, *i.e.* keeping all processors busy as much as possible.

Wilkinson and Allen (2005) introduced the *Embarrassingly Parallel* paradigm which assumes that a computation can be divided into a number of completely independent parts and that each part can be executed by a separate processor. In this paper, we introduce an Embarrassingly Parallel Search (EPS) method for constraint problems and show that this method often outperforms state-of-the-art parallel B&B algorithms for a number of problems on various computing infrastructures. A master decomposes the problem into many disjoint subproblems which are then solved independently by workers. Since a constraint program is not trivially embarrassingly parallel, the decomposition procedure must be carefully designed. Our approach has three advantages: it is an efficient method; it involves almost no communication, synchronization, or mutual exclusion between workers; its implementation is simple because the master and the workers rely on an underlying constraint solver but does not require to modify it. Additionally, it is deterministic under certain restrictions.

This paper integrates results from a series of publications (Régin, Rezgui, & Malapert, 2013, 2014; Rezgui, Régin, & Malapert, 2014). However, this paper includes novel contributions, implementations, and results. A new implementation of EPS on the top of the Java library `Choco2` (Choco, 2010) uses a new decomposition procedure. New results are given for the implementations on the top of the C++ library `Gecode` (Schulte, 2006) and `OR-tools` (Perron, Nikolaj, & Vincent, 2012), More problem's types and instances are tested. EPS is compared with other parallelizations of `Gecode` and with several static solvers portfolios, We perform in-depth analysis of various components, especially the decomposition procedures, as well as the anomalies that can occur.

The paper is organized as follows. Section 2 presents the constraint programming background, Amdahl's law, and related work about parallel constraint solving. Section 3 gives a detailed description of our embarrassingly parallel search method. Section 4 gives extensive experimental results for the various implementations (`Gecode`, `Choco2`, `OR-tools`) on different computing infrastructures (multi-core, data center, cloud computing) as well as comparisons to other state-of-the-art parallel implementations and static solver portfolios.

2. Related Work

Here, we present the constraint programming background, two important parallelization measures related to Amdahl's law, and related work about parallel constraint solving.

2.1 Constraint Programming Background

Constraint programming (CP) has attracted high attention among experts from many areas because of its potential for solving hard real-life problems. For an extensive review on constraint programming, we refer the reader to the handbook by Rossi, Van Beek, and Walsh (2006). A *constraint satisfaction problem* (CSP) consists of a set $\mathcal{X}$ of variables defined by a corresponding set of possible values (the domains $\mathcal{D}$) and a set $\mathcal{C}$ of constraints. A *constraint* is a relation between a subset of variables that restricts the possible values that variables can take simultaneously. The important feature of constraints is their declarative manner, *i.e.* they only specify which relationship must hold. The current domain $\mathcal{D}(x)$ of each variable $x \in \mathcal{X}$ is always a (non-strict) subset of its initial domain. A *partial assignment* represents the case where the domains of some variables have been reduced to a singleton (namely a variable has been assigned a value). A solution of a CSP is an assignment of a value to each variable such that all constraints are simultaneously satisfied.

Solutions can be found by searching systematically through the possible assignments of values to variables. A *backtracking scheme* incrementally extends a partial assignment $\mathcal{A}$ that specifies consistent values for some of the variables, toward a complete solution, by repeatedly choosing a value for another variable. The variables are labeled (given a value) sequentially. 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 determines the next variable to be instantiated, and the order in which the values from its domain are selected. If a partial assignment violates any of the constraints, backtracking 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 variable domains.

A filtering algorithm is associated with each constraint which removes inconsistent values from the domains of the variables, *i.e.* assignments which cannot belong to a solution of the constraint. Constraints are handled through a *constraint propagation mechanism* which allows the reduction of the domains of variables until a global fixpoint is reached (no more domain reductions are possible). In fact, a constraint specifies which relationship must hold and its filtering algorithm is the computational procedure that enforces the relationship.

Generally, consistency techniques are not complete, *i.e.* they do not remove all inconsistent values from the domains of the variables.

Both backtracking scheme and consistency techniques can be used alone to completely solve a CSP, but their combination allows the search space to be explored in a complete and more efficient way. 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).

Here, we consider a complete standard backtracking scheme with depth-first traversal of the search tree combined to the following variable selection strategies. Note that different variable selection strategies can be used although only one at a time. `lex` selects a variable according to lexicographic ordering. `dom` selects the variable with the smallest remaining domain (Haralick & Elliott, 1980). `ddeg` selects a variable with largest dynamic degree (Beck, Prosser, & Wallace, 2005), that is, the variable that is constrained with the largest number of unassigned variables. Boussemart, Hemery, Lecoutre, and Sais (2004) proposed conflict-directed variable ordering heuristics in which every time a constraint causes a failure during search, its weight is incremented by one. Each variable has a weighted degree, which is the sum of the weights over all constraints in which this variable occurs. `wdeg` selects the variable with the largest weighted degree. The current domain of the variable can be incorporated to give `dom/ddeg` or `dom/wdeg` which selects the variable with minimum ratio between current domain size and its dynamic or weighted degree (Boussemart et al., 2004; Beck et al., 2005). `dom/bwdeg` is a variant which follows a binary labeling scheme. `impact` selects the variable/value pair which has the strongest impact, *i.e.* leads in the strongest search space reduction (Refalo, 2004).

For optimization problems, we consider a standard top-down algorithm which maintains a lower bound, lb, and an upper bound, ub, on the objective value. When $ub \leq lb$, the subtree can be pruned because it cannot contain a better solution.

2.2 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 many times the parallel algorithm performs 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 (Amdahl, 1967). 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 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 brought a reduction in the search tree and problem size.

2.3 Parallel Constraint Solving

Designing and developing parallel programs has been a manual process where the programmer was responsible for both identifying and implementing parallelism (Barney & Livermore, 2016). In this section, we only discuss parallel constraint solving. About parallel logic programming, we refer the reader to the surveys of De Kergommeaux and Codognet (1994), and Gupta, Pontelli, Ali, Carlsson, and Hermenegildo (2001). About parallel integer programming, we refer the reader to the surveys of Crainic, Le Cun, and Roucairol (2006), Bader, Hart, and Phillips (2005), and Gendron and Crainic (1994).

The main approaches to parallel constraint solving can roughly be divided into the following main categories: search space shared in memory; search space splitting; portfolio algorithms; problem splitting. 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.

2.3.1 Search Space in Shared Memory

These methods are implemented by having many cores sharing a list of open nodes in the search tree (nodes for which there is at least one of the children that is still unvisited). Starved processors just pick up the most promising node in the list and expand it. By defining different node evaluation functions, one can implement different strategies (DFS, BFS and others). Perron (1999) proposed a comprehensive framework tested with at most 4 processors. Vidal, Bordeaux, and Hamadi (2010) reported good performance for a parallel best-first search up to 64 processors. 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 for communicating with memory and the problem is exacerbated by cache consistency transactions.

2.3.2 Search Space Splitting

Search Space Splitting strategies exploring the parallelism provided by the search space are common approaches: when a branching is done, different branches can be explored in parallel (Pruul, Nemhauser, & Rushmeier, 1988). One challenge is load balancing: the branches of a search tree are typically extremely imbalanced and require a non-negligible overhead of communication for work stealing (Lai & Sahni, 1984).

The work stealing method was originally proposed by Burton and Sleep (1981) and first implemented in Lisp parallel machines (Halstead, 1984). The search space is dynamically split during the resolution. When a worker finished to explore a subproblem, it asks other workers for another subproblem. If another worker agrees to the demand, then it splits dynamically its current subproblem into two disjoint subproblems and sends one subproblem to the starving worker. The starving worker "steals" some work to the busy one. Note that some form of locking is necessary to avoid that several starving workers steal the same subproblems. The starving worker asks other workers in turn until it receives a new subproblem. Termination of work stealing method must be carefully designed to reduce the overhead when almost all workers are starving, but almost no work remains. Recent works based on this approach are those by Zoeteweij and Arbab (2004), Jaffar, Santosa, Yap, and Zhu (2004), Michel, See, and Hentenryck (2009), and Chu, Schulte, and Stuckey (2009).

Because work stealing uses both communication, synchronization and computation time, this cannot easily be scaled up to thousands of processors. To address these issues, Xie and Davenport (2010) 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.

Machado, Pedro, and Abreu (2013) proposed a hierarchical work stealing scheme correlated to the cluster physical infrastructure, in order to reduce the communication overhead. 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. But, Machado et al. observed that the scalability was lowered because the lazy dissemination of the so-far best solution, *i.e.* because some workers use obsolete best solution.

General-purpose programming languages designed for multi-threaded parallel computing like Charm++ (Kale & Krishnan, 1993) and Cilk++ (Leiserson, 2010; Budiu, Delling, & Werneck, 2011) can ease the implementation of work stealing approaches. Otherwise, a work stealing framework like Bobpp (Galea & Le Cun, 2007; Le Cun, Menouer, & Vander-Swalmen, 2007) 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 if starving workers exist. In this case, the worker stops the search and the path from the root node to the highest right 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. The 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 (2013), and Menouer and Le Cun (2014) 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). Other experiments investigate the exploration overhead caused by their approach.

Bordeaux et al. (2009) proposed another promising approach based on a search space splitting mechanism not based on a work stealing approach. They use a hashing function allocating implicitly 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 for up to 30 processors for the n-queens problem, but then the speedups stagnate at 30 until to 64 processors. However, it only got moderate results 100 industrial SAT instances.

We have presented earlier works on the Embarrassingly Parallel Search method based on search space splitting with loose communications (Régin et al., 2013, 2014; Rezgui et al., 2014).

Fischetti, Monaci, and Salvagnin (2014) 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. SelfSplit can be decomposed in 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 (Régin et al., 2014).

Sometimes, for complex applications where very good domain specific strategies are known, the parallel algorithm should exploit the domain-specific strategy. Moisan, Gaudreault, and Quimper (2013), and Moisan, Quimper, and Gaudreault (2014) proposed a parallel implementation of the classic backtracking algorithm, Limited Discrepancy Search (LDS), that is known to be efficient in centralized context when a good variable/value selection heuristic is provided (Harvey & Ginsberg, 1995). Xie and Davenport (2010) proposed that each processor locally uses LDS to search in the trees allocated to them (by a tree splitting or work stealing algorithm) but the global system does not replicate the LDS strategy.

Cube-and-Conquer (Heule, Kullmann, Wieringa, & Biere, 2012) is an approach for parallelizing SAT solvers. A cube is a conjunction of literals and a DNF formula a disjunction of cubes. The SAT problem is split into several disjoint subproblems that are DNF formulas which are then solved independently by workers. Cube-and-Conquer using the Conflict-Driven Clause Learning (CDCL) solver Lingeling outperforms other parallel SAT solvers on some instances of the SAT 2009 benchmarks, but is also outperformed on many other instances. Thus, Concurrent Cube-and-Conquer (Van Der Tak, Heule, & Biere, 2012) tries to predict on which instances it works well and abort the parallel search after a few seconds in favor of a sequential CDCL solver if not.

2.3.3 Las Vegas Algorithms / Portfolios

They 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 (Gomes & Selman, 2000). 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 (Luby, Sinclair, & Zuckerman, 1993) where each worker executes its own restart strategy. More recently, Cire, Kadioglu, and Sellmann (2014) executed the Luby restart strategy, as a whole, in parallel. 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 (Hamadi, Jabbour, & Sais, 2008).

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 we want to face an arbitrarily high number of processors, then we need automated methods to generate a portfolio of any size on demand (Bordeaux et al., 2009). So, portfolio designers became interested in feature selection (Gomes & Selman, 1997, 1999, 2001; Kautz, Horvitz, Ruan, Gomes, & Selman, 2002). Features characterize problem instances like 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 behaviour of solvers. The SAT portfolio `iSAC` (Amadini, Gabbrielli, & Mauro, 2013) and the CP portfolio `CPHydra` (O'Mahony, Hebrard, Holland, Nugent, & O'Sullivan, 2008) use feature selection to choose the solvers that yield the best performance. Additionally, `CPHydra` exploits the knowledge coming from the resolution of a training set of instances by each candidate solver. Then, given an instance, `CPHydra` determines the k most similar instances of the training set and determines a time limit for each candidate solver based on constraint program maximizing the number of solved instances within a global time limit of 30 minutes. Briefly, `CPHydra` determines a switching policy between solvers (`Choco2`, `AbsCon`, `Mistral`).

Many recent SAT solvers are based on a portfolio such as ManySAT (Hamadi et al., 2008), SATzilla (Xu, Hutter, Hoos, & Leyton-Brown, 2008), SArTagnan (Stephan & Michael, 2011), Hydra (Xu, Hoos, & Leyton-Brown, 2010), Pminisat (Chu, Stuckey, & Harwood, 2008) based on Minisat (Een & Sörensson, 2005). Most of them combine portfolio-based algorithm selection to automatic algorithm configuration using different underlying solvers. For example, SATzilla (Xu et al., 2008) exploits the per-instance variation among solvers using learned runtime models.

In general, the main advantage of the algorithms portfolio 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.

2.3.4 Problem Splitting

Problem Splitting is another idea that relates to parallelism, where the problem itself is split into pieces to be solved by each processor. The problem typically becomes more difficult to solve than in the centralized case because no processor has a complete view on the problem. So, reconciling the partial solutions of each subproblem becomes challenging. Problem splitting typically relates to distributed CSPs, a framework introduced by Yokoo, Ishida, and Kuwabara (1990) in which the problem is naturally split among agents, as for privacy reasons. Other distributed CSP frameworks have been proposed such as those by Hirayama and Yokoo (1997), Chong and Hamadi (2006), Ezzahir, Bessière, Belaissaoui, and Bouyakhf (2007), Léauté, Ottens, and Szymanek (2009), and Wahbi, Ezzahir, Bessiere, and Bouyakhf (2011).

2.3.5 Parallel Constraint Propagation

Other approaches can be thought of, typically based on the parallelization of one key algorithm of the solver, for instance constraint propagation (Nguyen & Deville, 1998; Hamadi, 2002; Rolf & Kuchcinski, 2009). However, parallelizing propagation is challenging (Kasif, 1990) and the scalability is limited by Amdahl's law. Some other approaches focus on particular topologies or make assumptions on the problem.

2.3.6 Concluding Remarks

Note that 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 be 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.

3. Embarrassingly Parallel Search

In this section, we present the details of our embarrassingly parallel search. First, Section 3.1 introduces the key concepts that guided our design choices. Then, Section 3.2 introduces several search space splitting strategies implemented *via* the top-down or bottom-up decomposition procedures presented in Section 3.3. Section 3.4 gives details about the architecture and the communication. Section 3.5 explains how to manage the queue of subproblems in order to obtain a deterministic parallel algorithm. Section 4.1 gives more details about the implementation.

3.1 Key Concepts

We introduce the key concepts that guided our design choices: massive static decomposition; loose communication; non-intrusive implementation; toward a deterministic algorithm.

3.1.1 Massive Static Decomposition

The master decomposes the problem into p subproblems once and for all which are then solved in parallel and independently by the workers. So, the solving process is equivalent to the real-time scheduling of p jobs on w parallel identical machines known as $P||C_{max}$ (Korf &

Schreiber, 2013). Efficient algorithms exists for $P||C_{max}$ and even the simple list scheduling algorithms (based on priority rules) are a $(2 - \frac{1}{w})$-approximation. The desirable properties defined in Section 3.2 should ensure low precision processing times that makes the problems easier. If we hold the precision and number of workers fixed, and increase the number of subproblems, then problems get harder until perfect schedules appear, and then they get easier. In our case, the number p of subproblems should range between one and three orders of magnitude larger than the number of workers w. If it is too low, the chance of finding perfect schedules, and therefore obtain good speedups, are low. If it is too large, the decomposition takes longer and becomes more difficult. If these conditions are met, then it is unlikely that a worker will be assigned more work than any other, and therefore, the decomposition will be statistically balanced. Beside, to reach good speedups in practice, the total solving time of all subproblems must be close to the sequential solving time of the problem.

An advantage is that the master and workers are independent. They can use different filtering algorithms, branching strategies, or even underlying solvers. The decomposition is a crucial step, because it can be a bottleneck of the computation and its quality also greatly impacts the parallelization efficiency.

3.1.2 Loose Communication

p subproblems are solved in parallel and independently by the w workers. As load balancing must be statistically obtained from the decomposition, we do not allow work stealing in order to drastically reduce communication. Of course, some communication is still needed to dispatch the subproblems, to gather the results and possibly to exchange useful additional information, like objective bound values. Loose communication allows to use a star network without risk of congestion. A central node (foreman) is connected to all other nodes (master and workers).

3.1.3 Non-intrusive Implementation

For the sake of laziness and efficiency, we rely as much as possible on the underlying solver(s) and on the computing infrastructure. Consequently, we modify as little as possible the underlying solver. We do not consider nogoods or clauses exchanges because these techniques are intrusive and increase the communication overhead. Additionally, logging and fault tolerance are respectively delegated to the underlying solver and to the infrastructure.

3.1.4 Toward Determinism

A *deterministic algorithm* is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. If the determinism is already challenging for sequential B&B algorithms due to their complexity (randomization, restarts, learning, optimization), it is still more difficult for parallel B&B algorithms.

Here, we will always guarantee *reproducibility* if the real-time assignment of subproblems to workers is stored. Reproducibility means that it is always possible to replay the solving process. With some restrictions detailed later, our parallel algorithm can be made deterministic with no additional cost. Moreover, the parallel algorithm should be able to mimic the sequential algorithm, *i.e.* they produce identical solutions. It requires that the parallel algorithm visits the tree leaves in the same order as the sequential algorithm. More generally, it would be useful for debugging, performance evaluation, or incremental problem solving that the parallel algorithm may produce identical solutions no matter how many workers are present or which computing infrastructure is used.

Conversely, any real-time scheduling algorithm can be applied to subproblems. It would allow to improve diversification by using more randomization, or to exploit past information provided by the solving process. In the experiments, we will only use FIFO scheduling of the subproblems, because other scheduling policy would change the shape and size of the search tree and, therefore, reduces the relevance of speedups. Unlike EPS, work stealing approaches are not deterministic and offer no control on subproblem scheduling.

3.2 Search-Space Splitting Strategies

Here, we extend the approach to search-space splitting proposed by Bordeaux et al. (2009), called splitting by hashing. Let us recall that $\mathcal{C}$ is the set of constraints of the problem. To split the search space of a problem into p parts, one approach is to assign each subproblem i ($1 \leq i \leq p$) an extended set of constraints $\mathcal{C} \cup H_i$ where H_i is a hashing constraint, which constrains subproblem i to a particular subset of the search space. Hashing constraints must necessarily be sound and should be effective, nontrivial, and statistically balanced.

Sound Hashing constraints must partition the search space: $\cup_{i=1}^{p} H_i$ must cover the entire initial search space (*completeness*), and the mutual intersections $H_i \cap H_j$ ($1 \leq i < j \leq p$) should preferably be empty (*non-overlapping*).

Effective The addition of the hashing constraints should effectively allow each worker to efficiently skip the portions of the search space not assigned to its current subproblem. Each subproblem must be significantly easier than the original problem. This causes overhead, to which we refer to as *recomputation overhead*.

Nontrivial The addition of the hashing constraints should not lead to an immediate failure of the underlying solver. Thus, generating trivial subproblems might be paid by some *exploration overhead*, because many of them would have been discarded by the propagation mechanism of the sequential algorithm.

Statistically Balanced All workers should be given about the same amount of work. If the decomposition is appropriate, then the number p of subproblems is significantly larger than the number w of workers. It is thus unlikely that a given worker would be assigned significantly more work than any other worker by any real-time scheduling algorithm. However, it is possible that solving one subproblem requires significantly more work than another subproblem.

Bordeaux et al. (2009) defined hashing constraints by selecting a subset X of the variables of the problem and stating H_i ($1 \leq i \leq p$) as follows: $\sum_{x \in X} x \equiv i \mod p$. This effectively decomposes a problem into p problems if p is within reasonable limits. For $p = 2$, it imposes a parity constraints over the sum of the variables. Splitting can be repeated to scale-up to an arbitrary number of processors. This splitting is obviously sound, but less effective for

CP solvers than for SAT solvers. Here, we study assignment splitting and node splitting to generate a given number $p^\star$ of subproblems.

3.2.1 Assignment Splitting

Let us consider a non empty subset $X \subseteq \mathcal{X}$ of d ordered variables: $X = (x_1, \ldots, x_d)$. A vector $\tau = (v_1, \ldots, v_d)$ is a *tuple* on X if $v_j \in D(x_j)$ $(j = 1, \ldots, d)$. Let $H(\tau) = \wedge_{j=1}^{d} (x_j = v_j)$ be the hashing constraints which restrict the search space to solutions extending the tuple τ. A *total decomposition* on X splits the initial problem into $\prod_{i=1}^{d} D(x_i)$ subproblems, *i.e.* one subproblem per tuple. A total decomposition is clearly sound and effective, but not efficient in practice. Indeed, Régin et al. (2013) showed that the number of trivial subproblems can grow exponentially.

In a *table decomposition*, a subproblem is defined by a set of tuples that allows to reach exactly the number $p^\star$ of subproblems. Let $\mathcal{T}$ be an ordered list of tuples on X such that $|\mathcal{T}| > p^\star$. Then, the first subproblem is defined by the first $k = \left\lfloor \frac{|\mathcal{T}|}{p^\star} \right\rfloor$ tuples, the second subproblem is defined by the following k tuples, and so on. So, all subproblems are defined by the same number of tuples possibly with the exception of the last.

A tuple τ is *solver-consistent* if the propagation of the extended set of constraints $\mathcal{C} \wedge H(\tau)$ by the underlying solver does not detect unsatisfiability. In order to obtain nontrivial decompositions, total and table decompositions are restricted to solver-consistent tuples.

3.2.2 Node Splitting

Node splitting allows the parallel algorithm to exploit domain-specific strategies for the decomposition when a good strategy is known. Let us recall some concepts on search trees (Perron, 1999) that are the basis of the decomposition procedures introduced later. To decompose the problems, one needs to be able to map individual parts of the search tree to hashing constraints. These parts are called open nodes. Once open nodes are defined, we present how a search tree is decomposed into a set of open nodes.

Open Nodes and Node Expansion The search tree is partitioned into three sets, open nodes, closed nodes, and unexplored nodes. Here, we do not make any assumption about the arity of the search tree, *i.e.* the maximal number of children of its nodes. These subsets have the following properties.

- All the ancestors of an open node are closed nodes.
- Each unexplored node has exactly one open node as its ancestor.
- No closed node has an open node as its ancestor.

The set of open nodes is called the *search frontier* as illustrated in Figure 1. The search

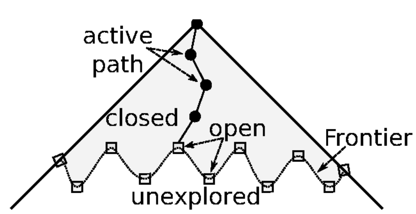


Figure 1: Node status and search frontier of a search tree.

frontier evolves simply through a process known as node expansion. It removes an open node from the frontier, transforms the removed node into a closed node, and adds its unexplored children to the frontier. Node expansion is the only operation that happens during the search. It corresponds to the branch operation in a B&B algorithm.

At any point of the search, the search frontier is a *sound* and *nontrivial* decomposition of the original problem where each open node is associated to a subproblem. The decomposition is effective if and only if the branching strategy is effective. Let us remark that assignment splitting can be seen as a special case of node splitting in which a static ordering is used for variables and values.

Active Path and Jumps in the Search Tree Expanding one node after another may require changing the state (at least the variables domains) of the search process from the first node to the second. So, the worker in charge of exploring an open node must reconstruct the state it visits. This is done using an active path and a jumping operation.

When going down in the search tree, our search process builds an active path, which is the list of ancestors of the current open node, as illustrated in Figure 1. When a worker moves from one node to another, it has to jump in the search tree. To make the jump, it simply recomputes every move from the root until it gets to the target node. This causes overhead, to which we refer to as *recomputation overhead*. Recomputation does not change the search frontier because it does not expand a node.

3.3 Decomposition Procedures

The decomposition challenge is to find the depth at which the search frontier contains approximately $p^\star$ nodes. The assignment splitting strategy is implemented by a top-down procedure which starts from the root node and incrementally visits the next levels, whereas the node splitting strategy is implemented by a bottom-up procedure which starts form a level deep enough and climbs back to the previous levels.

3.3.1 Top-Down Decomposition

The challenge of the top-down decomposition is to find d ordered variables which produce approximately $p^\star$ solver-consistent tuples. Algorithm 1 realizes a solver-consistent table decomposition by iterated depth-bounded depth-first searches with early removals of inconsistent assignments (Régin et al., 2013). The algorithm starts at the root node with an empty list of tuples (line 1). It computes a list $\mathcal{T}$ of $p^\star$ tuples that are a solver-consistent table decomposition by iterativly increasing the decomposition depth. Let us assume that there exists a static order of the variables. At each iteration, it determines a new lower bound (line 4) on the decomposition depth d, *i.e.* the number of variables involved in the decomposition. This lower bound uses the Cartesian product of the current domains of the next variables $x_{d+1}, x_{d+2}, \ldots$ Then, a depth-bounded depth-first search extends the decomposition to its new depth and updates the list of tuples (line 5). The current tuples are added to the constraints of the model during the search (line 5) in order to reduce redundant work. After the search, the extended tuples are propagated (line 7) to reduce the domains, and to improve the next lower bound on the decomposition depth. Each tuple was solver-consistent (not proven infeasible) during the last search. The process is repeated until the number $|\mathcal{T}|$ of tuples is greater or equal to $p^\star$. At the end, tuples are aggregated

Algorithm 1: Top-down decomposition.

Data: A CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ and a number of subproblems $p^\star$
Result: A list of tuples $\mathcal{T}$
1 $d \leftarrow 0$;
2 $\mathcal{T} \leftarrow \emptyset$;
`/* Simulate a breadth-first search: iterated depth bounded DFSs. */`
3 **repeat**
 `/* Determine a lower bound on the decomposition depth. */`
4 $d \leftarrow \min \left\{ l \;\middle|\; \max(1, |\mathcal{T}|) \times \prod_{i=d+1}^{l} |D(x_i)| \geq p^\star \right\}$;
 `/* Extend the current decomposition with new variables. */`
5 $\mathcal{T} \leftarrow$ `depthBoundedDFS` $(\mathcal{X}, \mathcal{C} \cup \{\vee_{\tau \in \mathcal{T}} H(\tau)\}, \mathcal{D}, \{x_1, \ldots, x_d\})$;
6 **if** $\mathcal{T} == \emptyset$ **then break**;
 `/* Propagate the tuples (without failure). */`
7 $\mathcal{D} \leftarrow$ `propagate` $(\mathcal{X}, \mathcal{C} \cup \{\vee_{\tau \in \mathcal{T}} H(\tau)\}, \mathcal{D})$;
8 **until** $|\mathcal{T}| < p^\star$;
`/* Aggregate tuples to generate exactly` $p^\star$ `subproblems */`
9 $\mathcal{T} \leftarrow$ `aggregateTuples`$(\mathcal{T})$ `/* All subproblems become simultaneously available. */`
10 **foreach** $\tau \in \mathcal{T}$ **do** `sendSubProblem` $(\mathcal{X}, \mathcal{C} \cup H(\tau), \mathcal{D})$;

to generate exactly $p^\star$ subproblems. In practice, consecutive tuples of $\mathcal{T}$ are aggregated. All subproblems become simultaneously available after the aggregation.

Sometimes, the sequential decomposition is a bottleneck because of Amdahl's law. So, the parallel decomposition procedure increases the scalability (Régin et al., 2014). Only two steps differ from Algorithm 1. First, instead of starting at depth 0 with an empty list of tuples (line 1 of Algorithm 1), a first list is quickly generated with at least five tuples per worker.

1 $d \leftarrow \min \left\{ l \;\middle|\; \prod_{i=1}^{l} |D(x_i)| \geq 5 \times w \right\}$;
2 $\mathcal{T} \leftarrow \prod_{i=1}^{d} D(x_i)$;

Second, at each iteration, each tuple is extended in parallel instead of extending sequentially all tuples (line 5 of Algorithm 1). The parallel decomposition can change the ordering of $\mathcal{T}$ compared to the sequential one. Again, all subproblems only become available at the end of the decomposition.

5 $\mathcal{T}' \leftarrow \emptyset$;
6 **run in parallel**
7 **foreach** $\tau \in \mathcal{T}$ **do**
  `/* extend each tuple in parallel */`
8 $\mathcal{T}' \leftarrow \mathcal{T}' \cup$ `depthBoundedDFS` $(\mathcal{X}, \mathcal{C} \cup H(\tau), \mathcal{D}, \{x_1, \ldots, x_d\})$;
9 $\mathcal{T} \leftarrow \mathcal{T}'$;

Both top-down procedures assume that the variable ordering used in the decomposition is static. The next decomposition procedure bypasses this limitation and handles any branching strategy.

Algorithm 2: Bottom-up decomposition.

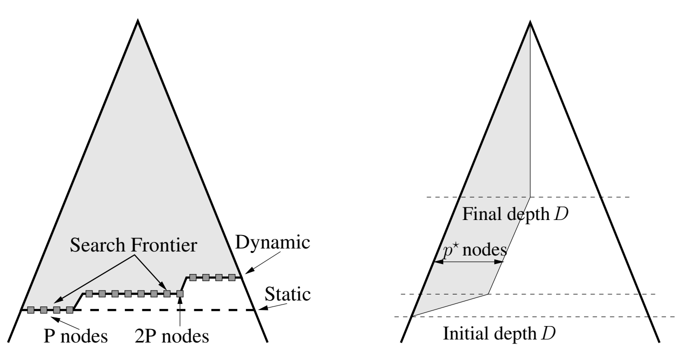
Data: A CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, a decomposition depth $d^\star$, and a subproblem limit P.

1 $p \leftarrow 0$;
`/* Generate subproblems by visiting the top of the real tree. */`
2 **Before Node Callback** *decomposition(node)* **is**
3   **if** `depth`$(node) \geq d^\star$ **then**
4   `sendSubProblem` $(node)$;
5     $p \leftarrow p + 1$;
6   **if** $p \geq P$ **then**
      `/* Decrease dynamically the depth. */`
7       $d^\star \leftarrow \max(1, d^\star - 1)$;
8       $P \leftarrow 2 \times P$;
9   **backtrack**;
10 `DFS` $(\mathcal{X},\mathcal{C},\mathcal{D})$;

3.3.2 Bottom-Up Decomposition

The bottom-up decomposition explores the search frontier at the depth $d^\star$ with approximately $p^\star$ nodes. In its simplest form, the decomposition depth $d^\star$ can be provided by a user with good knowledge of the problem. Algorithm 2 explores the search frontier at depth $d^\star$ using a depth-first search as illustrated in Figure 2(a). A search callback identifies each node at level $d^\star$ (line 2), and sends immediately its active path, which defines a subproblem, so that the subproblem will be solved by a worker. If the decomposition depth is *dynamic*, then it is reduced if the number of subproblems becomes too large (line 6). It aims to compensate a poor choice of the decomposition depth $d^\star$. In practice, the depth is reduced by one unit if the current number of subproblems exceeds a given limit P. This limit is initially set up to $P = 2 \times p^\star$ and is doubled each time it is reached. On the contrary, the depth is *static* ($P = +\infty$) if it never changes whatever be the number of subproblems.

In practice, it is not common that the user provides the decomposition depth, and an automated procedure without any user's intervention is needed. Algorithm 3 aims at identifying the topmost search frontier with approximately $p^\star$ open nodes by sampling and estimation. The procedure can be divided into three phases: build a partial tree by sampling



(a) Decomposition. (b) Estimation.

Figure 2: Bottom-up decomposition and estimation.

Algorithm 3: Bottom-up estimation.

Data: A CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ and a number of subproblems $p^\star$.
Data: A time limit t, a node limit n, and a maximum depth D large enough
Result: A decomposition depth $d^\star$

```
   /* Set counters for the width of the levels */
1  foreach d ∈ [1, D] do width[d] ← 0;
   /* Build a partial tree by sampling.  */
2  Before Node Callback estimation(node) is
3      d ← depth(node);
4      if d ≤ D then
5          width[d] ← width[d] + 1;
6          if width[d] ≥ p* then D ← d − 1 ;
7      else backtrack;
8      if hasFinished (t,n) then break;
9  DFS (X,C,D);
   /* Estimate the level widths of the tree and the decomposition depth.  */
10 width ← estimateWidths(width);
11 d* ← estimateDepth(width, p*);
```

the top of the real search tree; estimate the level widths of the real tree; and then determine the decomposition depth $d^\star$ 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 $p^\star$ open nodes on a level using a callback of a depth-first search. The number of open nodes at each level of the partial tree is counted by the callback. The maximum depth D is reduced each time $p^\star$ nodes are opened at a given level (line 6). If the sampling ends within its limits, then the top of the tree has been entirely visited and no estimation is needed. Otherwise (line 8), one needs to estimate the widths of the topmost levels of the tree depending on the partial tree. The estimation is a straightforward adaptation of the one proposed by Cornuéjols, Karamanov, and Li (2006) to deal with n-ary search tree (line 10). In practice, the main issue is that the higher the arity is, the lower is the precision of the estimation. Therefore, a greedy heuristic determines the decomposition depth based on the estimated number of nodes per level, but also on the number of nodes in the partial tree (line 11). The heuristics minimizes the absolute deviation between the estimated number of nodes and the expected number $p^\star$. If several levels have an identical absolute deviation, then the lowest level with an estimated number of subproblems greater than or equal to $p^\star$ is selected.

3.4 Architecture and Communication

We describe messages exchanged by the actors depending on the problem's type. Then, a typical use case illustrates the solving process for an optimization problem. Briefly, the communication network is a star network where the *foreman* acts as a pipe to transmit messages between the *master* and *workers*.

3.4.1 Actors and Messages

The total number of messages depends linearly of the number of workers (w) and the number of subproblems (p). All messages are synchronous for sake of simplicity which means that work must wait until the communications have completed (Barney & Livermore, 2016). Interleaving computation with communication is the single greatest benefit for using asynchronous communications since work can be done while the communications are taking place. However, asynchronous communications complicate the architecture, for instance if a message requests a answer.

Master is the control unit which decomposes the problem and collects the final results. It sends the following messages: **`create`** the foreman; **`give`** a subproblem to the foreman; **`wait`** for the foreman to gather all results; **`destroy`** the foreman. The master only deals with the foreman. The decomposition time is the elapsed time between the `create` and `wait` messages. The workers time is the elapsed time between the first `give` and `destroy` messages. The wall-clock time is the elapsed time from the creation to the destruction of the master.

Foreman is the central node of the star network. It is a queuing system which stores subproblems received from the master and dispatches them to workers. It also gathers results collected from the workers. The foreman allows the master to concentrate on the problem's decomposition, which is a performance bottleneck, by handling all communications with the workers. It sends the following messages: **`create`** a worker; **`give`** a subproblem to a worker; **`collect`** (send) the final results to the master; **`destroy`** a worker. When the foreman detects that the search has ended, it sends a `collect`-message containing the final results to the master.

Workers are search engines. They send the following messages: **`find`** a subproblem (the foreman must answer by a `give`-message); **`collect`** (send) results to the foreman. The results contain essential information about the solution(s) and the solving process. Workers only know the foreman. When a worker acquires new work (receives a `give`-message from the foreman), the acquired subproblem is recomputed which causes *recomputation overhead*. In work stealing context, Schulte (2000) noticed that the higher the node is in the search tree, the smaller is the recomputation overhead. By construction, only the topmost nodes are used here.

3.4.2 Problem's Types

We discuss the specificities of first solution, all solution, and best solution searches.

First Solution Search The search is complete as soon as a solution has been found. Other workers must be immediately terminated as well as the decomposition procedure.

All Solution Search The search is complete when all subproblems have been solved.

Best Solution Search The main design issue in best-solution search is to maintain the so-far best solution. The sequential B&B algorithm always knows the so-far best solution. This is difficult to achieve in a concurrent setting with several workers. Maintaining the best solution for each worker could lead to large communication and synchronization overheads. Instead we prefer a solution where both the foreman and workers maintain the so-far best solution as follows. By default, the `give` and `collect` messages between the foreman and the workers carry the objective information. Additionally, a worker can send `better` messages to the foreman with an intermediate solution, or the foreman can send its best solution to all workers. For instance, when a worker finds a new solution, it informs the foreman by sending a `better` message if the solution is accepted by a threshold function. Similarly, when the foreman receives a new solution through a `collect` or `better` message, it checks whether the solution is really better. If the solution is accepted by the threshold function, the foreman sends another `better` message to all workers. The architecture sketched above entails that a worker might not always know the so-far best solution. In consequence, some parts of the search tree are explored, and they should have been pruned away if the worker had had exact knowledge. Thus, the loose coupling might be paid by some *exploration overhead*.

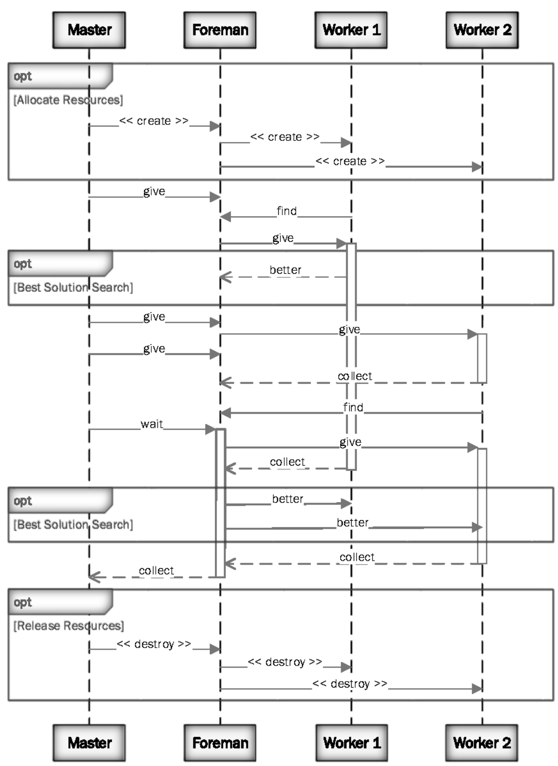


Figure 3: Sequence diagram of the solving process with two workers.

3.4.3 Use Case

Figure 3 is a sequence diagram illustrating the solving process for an optimization problem with two workers. It shows how actors operate with each other in chronological order.

The first horizontal frame is the resource allocation. The master creates the foreman. The foreman creates the workers. Immediately after creation, the master and each worker load the original problem. The foreman transparently manages a concurrent queue of subproblems produced by the master and consumed by workers. After that, workers will only jumps in the search tree.

After the foreman creation, the master starts the decomposition of the original problem into $p = 3$ subproblems. As soon as a subproblem is generated, the master gives it to the foreman. Here, the `give` and `find` messages are interleaved as in the node splitting decomposition proposed in Section 3.3.2. The assignment splitting decomposition proposed in Section 3.3.1 would produce a unique `give` message with all subproblems. When the decomposition is finished, the master sends a `wait` message to the foreman and waits for a `collect` response containing the final result. This last `collect` message triggers the resource deallocation.

Each time a worker is starving, it asks the foreman for a subproblem and waits for it. Here, the first subproblem is assigned to the first worker while the second worker waits for the second subproblem. The “Best Solution Search” frames correspond to specific messages for optimization problems. The first worker quickly finds a good solution and sends it to the foreman *via* a `better` message. A second subproblem is generated by the master and then given to the foreman. In turn, the foreman gives the second subproblem and updated objective information to the second worker. The second problem is quickly solved by the second worker which sends a `collect` message to the foreman. The `collect` message also stands for a `find` message. Then, the third, and last, subproblem is assigned to the second worker.

The foreman broadcasts a `better` message because of the good quality of the solution received from the first worker. Note that this message is useless for the first worker. The foreman detects the termination of the solving process and sends the `collect` message to the master if the three following conditions are met: the master is waiting; the subproblem's queue is empty; and all workers are starving. The last horizontal frame is the resource deallocation.

3.5 Queuing and Determinism

The foreman plays the role of a queuing system which receives subproblems from the master and dispatches them to the workers. In this section, we show that EPS can be modified to return the same solution than the sequential algorithm which can be useful in several scenarios such as debugging or performance evaluation. Generally, any queuing policy can be applied to select the next subproblem to solve.

Let us assume that the subproblems $P_1, P_2, \dots, P_p$ are sent to the foreman in a fixed order which is the case for the sequential top-down procedure and the bottom-up procedure. Otherwise, a fixed order of subproblems can be obtained by sorting the subproblems.

The first solution found by the sequential algorithm belongs to the satisfiable subproblem P_i with the smallest index, *i.e.* the leftmost solution. Let us assume that the parallel algorithm finds a 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.

It can easily be extended for optimization problems by slightly modifying the cutting constraints. Usually, a cutting constraint is stated when a new solution is found that only allows strictly improving solution. 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 solution for subproblems $k < j$.

So, the parallel algorithm returns the same solution than the sequential one if the subproblem 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. In our experiments, the queuing policy is the FIFO policy that ensures that subproblems are solved in the same order so that the speedups are relevant. However, there is no guaranty that the sequential and parallel algorithms return the same solution.

4. Experimental Results

Here, we describe experiments on EPS and carry out a detailed data analysis. We aim to answer the following questions. Is EPS efficient? With different number of workers? With different solvers? On different computing platforms? Compared to other parallel approaches? What is the influence of the different components (decomposition procedures, search strategies, constraint models)? Is EPS robust and flexible? Which anomalies can occur?

Section 4.1 presents the benchmark instances, execution environments, parameters settings, and the different implementations. First, in Section 4.2, we analyze and evaluate the top-down and bottom-up decomposition procedures as well as the importance of the search strategy, especially for the decomposition. Then, we evaluate the efficiency and scalability of parallel solvers on a multi-core machine (Section 4.3), on a data center (Section 4.4), and on a cloud platform (Section 4.5). In these sections, we compare our implementations of EPS with work stealing approaches whenever it is possible. In Section 4.4, we also analyze the efficiency of a parallel solver depending on the search strategy. In Section 4.6, we transform with reasonable effort a parallel solver into a distributed parallel solver by using the batch scheduler provided by the data center. Some anomalies of a parallel solver are explained and resolved by its distributed equivalent. Last, Section 4.7 discusses the performance of parallel solvers compared with static portfolios built from the underlying sequential solvers on the data center.

4.1 Experimental Protocol

In this section, we introduce the benchmark instances, execution environments, metrics and notations. We also give more details about the implementations.

4.1.1 Benchmark Instances

A lot of benchmark instances are available in the literature. We aim to select difficult instances with various models that represent problems tackled by CP. Ideally, an instance is difficult if none of the solvers can solve it quickly. Indeed, parallel solving is relevant only if it shortens a long wall-clock time. Here, we only consider unsatisfiable, enumeration and optimization problem's instances. We will 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 will consider optimization problems for which the same variability can be observed, but at a lesser extent because the optimality proof is required. The variability for unsatisfiable and enumeration instances is lowered, and therefore, they are often used as a test bed for parallel computing. Besides, unsatisfiable instances have a practical importance, for instance in software testing, and enumeration is important for users to compare various solutions.

The first set called `fzn` is a selection of 18 instances selected from more than 5000 instances either from the repository maintained by Kjellerstrand (2014) or directly from the Minizinc 1.6 distribution written in the FlatZinc language (NICTA Optimisation Research Group, 2012). Each instance is solved in more than 500 seconds and less than 1 hour with `Gecode`. The selection is composed of 1 unsatisfiable, 6 enumeration, and 11 optimization instances.

The set `xcsp` is composed of instances from the categories ACAD and REAL of XCSP 2.1 (Roussel & Lecoutre, 2008). It consists of difficult instances that can be solved within 24 hours by `Choco2` (Malapert & Lecoutre, 2014). A first subset called `xcsp1` is composed of 5 unsatisfiable and 5 enumeration instances whereas the second subset called `xcsp2` is composed of 11 unsatisfiable and 3 enumeration instances. The set `xcsp1` is composed of instances easier to solve than those of `xcsp2`.

Besides, we will consider two classical problems, the n-queens and the Golomb ruler problems which have been widely used in the literature (Gent & Walsh, 1999).

4.1.2 Implementation Details

We implemented EPS method on top of three solvers: `Choco2` 2.1.5 written in Java, `Gecode` 4.2.1 and `OR-tools` rev. 3163 written in C++. We use two parallelism implementation technologies: Threads (Mueller et al., 1993; Kleiman, Shah, & Smaalders, 1996) and MPI (Lester, 1993; Gropp & Lusk, 1993). The typical difference between both 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++, we use Threads implemented by `pthreads`, a POSIX library (Mueller et al., 1993; Kleiman et al., 1996) used by Unix systems. In Java, we use the standard Java Thread technology (Hyde, 1999).

There are many implementations for MPI like OpenMPI (Gabriel, Fagg, Bosilca, Angskun, Dongarra, Squyres, Sahay, Kambadur, Barrett, Lumsdaine, et al., 2004), Intel MPI (Intel Corporation, 2015), MPI-CH (MPI-CH Team, 2015) and MS-MPI (Krishna, Balaji, Lusk, Thakur, & Tiller, 2010; Lantz, 2008). MPI is a standard API, so the characteristics of the machine are never taken into account. So, the machine providers like Bull, IBM or Intel provide their own MPI implementation according to the specifications of the delivered machine. Thus, the cluster provided by Bull has a custom Intel MPI 4.0 library, but OpenMPI 1.6.4 is also installed, and Microsoft Azure only supports its own MS-MPI 7 library.

`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 foreman schedules the jobs in FIFO to mimic as much as possible the sequential algorithm so that speedups are relevant. When needed, the master and the workers read the model from the same file. We always take the value selection heuristic which selects the smallest value whatever be the variable selection heuristic.

4.1.3 Execution Environments

We use three execution environments that are representative of computing platforms available nowadays.

Multi-core is a Dell computer with 256 GB of RAM and 4 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 2 Intel E5-2670 2.60 GHz processors (8 cores). The cluster is managed by OAR (Capit, Da Costa, Georgiou, Huard, Martin, Mounie, Neyron, & Richard, 2005), *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 to deploy applications on Windows Server technology (Li, 2009). Each node has 56 GB of RAM and Intel Xeon E5-2690E 2.6 GHz processors (8 physical cores) We were allowed to simultaneously use 3 nodes (24 cores) managed by the Microsoft HPC Cluster 2012 (Microsoft Corporation, 2015).

Some computing infrastructures provide hyper-threading technologies. Hyper-threading 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 provides hyper-threading, whereas it is deactivated on the cluster, and not available on the cloud.

4.1.4 Setting up the Parameters

The time limit for solving each instance is set to 12 hours whatever be the solver. If the number of workers is strictly less than the number of cores ($w < c$), then there will always be unused cores. Usually, one chooses $w = c$, so that all workers can work simultaneously. On the multi-core computer, we use two workers per physical core ($w = 2c$) because hyper-threading is efficient as experimentally demonstrated in Appendix A. The target number

$p^\star$ of subproblems depends linearly on the number w of workers ($p^\star = 30 \times w$) that allows statistical balance of the workload without increasing too much the total overhead (Régin et al., 2013).

In our experiments, the network and RAM memory loads are low in regards to the capacities of the computing infrastructures. Indeed, the total number of messages depends linearly of the number of workers and the number of subproblems. RAM is pre-allocated if the computing infrastructure allows it. Last, workers almost produce no input/output or disk access.

4.1.5 Metrics and Notations

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 geometrical means, abbreviated GM, are computed for speedups and efficiency. Missing values, *i.e.* dashes and question marks, are ignored when computing statistics.

We also use a scoring procedure based on the Borda count voting system (Brams & Fishburn, 2002). Each benchmark instance is treated like a voter who ranks the solvers. Each solver scores points related to the number of solvers that it beats. More precisely, a solver s scores points on problem P by comparing its performance with each other solver s' as follows:

- if s gives a better answer than s', it scores 1 point;
- else if s did not answer or gives a worse answer than s', it scores 0 point;
- else scoring is based on execution time comparison (s and s' give indistinguishable answers).

Let t and t' respectively denote the wall-clock times of solvers s and s' for a given problem's instance. In case of indistinguishable answers, s scores $f(t, t')$ according to the Borda system used in the Minizinc challenge. But, the function f does not capture user's preferences very well. Indeed, if the solver s solves n problems in 0.1 seconds and n others in 1000 seconds whereas the solver s' solves the first n problems in 0.2 seconds and the n others in 500 seconds, then both solvers obtain the same score n whereas most users would certainly prefer s'. So, we use another scoring function $g(t, t')$ in which $g(t)$ can be interpreted as the utility function for solving the problem's instance within t seconds. The function $g(t)$ is strictly decreasing from 0.5 toward 0. The remaining points are shared using the function f.

$$f(t,t') = \frac{t'}{t+t'} \qquad g(t,t') = g(t) + (1 - g(t) - g(t')) \times f(t,t') \qquad g(t) = \frac{1}{2 \times (\log_a(t+1)+1)}$$

Using the function g ($a = 10$) in the previous example, solvers s and s' are respectively scored $0.81 \times n$ and $1.19 \times n$ points.

4.2 Analysis of the Decomposition

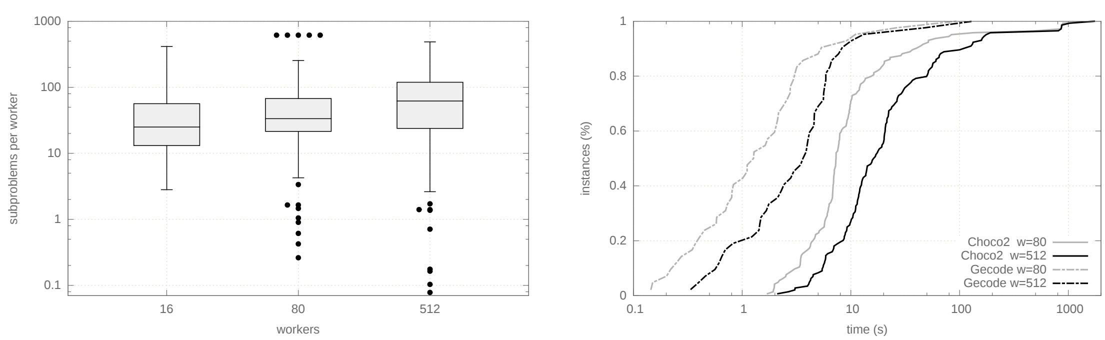
In this section, we compare the quality and performance of the top-down and bottom-up decomposition procedures introduced in Section 3.3.

4.2.1 Decomposition Quality

The top-down decomposition always returns the target number $p^\star = 30 \times w$ of subproblems whereas it is not guaranteed with the bottom-up decomposition. Figure 4(a) is a boxplot of the number of subproblems per worker (p / w) with the bottom-up decomposition of `Choco2` depending on the number of workers. Boxplots display differences among populations without making any assumptions of the underlying statistical distribution: they are non-parametric. The box in the boxplot spans the range of values from the first quartile to the third quartile. The whiskers extend from each end of the box for a range equal to 1.5 times the interquartile range. Any points that lie outside the range of the whiskers are considered outliers: they are drawn as individual circles.

For each number of workers $w \in \{16, 80, 512\}$, the decompositions of `xcsp` instances using one variable selection heuristic among `lex`, `dom`, `dom/ddeg`,`dom/wdeg`, `dom/bwdeg`, and `impact`, combined with `minVal`, are considered. The bottom-up decomposition obtains satisfying average performance (mostly between 10 and 100 subproblems per worker) while respecting as much as possible the branching strategy. However, a few anomalies occur. First, the decomposition is sensitive to the shape of the search tree. Sometimes, the model only contains a few variables with large domains which forbid an accurate decomposition. For instance, the first and second levels of the `knights-80-5` search tree respectively contain more than 6000 and 50000 nodes. There can also be a significant underestimation of the tree size, especially if the branching has high arity. For instance, the width of the second level of `fapp07-0600-7` is estimated around 950 nodes while it contains more than 6000 nodes. On the contrary, an underestimation can occur if top nodes are eliminated from a search tree with a low arity. Apart for a few underestimation, the decomposition is accurate for search trees with low arity.

The top-down decomposition is accurate, but requires a fixed variable ordering, whereas the bottom-up decomposition is less accurate, but handles any branching strategy.



(a) Number of subproblems per worker.

(b) Decomposition time.

Figure 4: Analysis of the decomposition procedures ($w = 16, 80, 512$).

4.2.2 Decomposition Time

Figure 4(b) gives the percentage of decompositions done within a given time. The `Choco2` times are reported for all variable selection heuristics on `xcsp` instances. The `Gecode` times are reported for `lex` on `xcsp` and `fzn` instances.

Because of the implementation differences, times reported for `Choco2` and `Gecode` are slightly different. Indeed, the decomposition time alone is given for `Gecode`. The `Choco2` times take also into account the estimation time, the time taken by the foreman to fill the queue of subproblems, and the time taken by workers to empty the queue. Let us also remind that subproblems only become available after the top-down decomposition is complete whereas they become available on the fly during the bottom-up decomposition. In both cases, the reported time is a lower bound on the solving time.

The top-down decomposition is faster than the bottom-up decomposition because of its parallelism. In fact, the `Gecode` decomposition is often faster than the estimation time alone. One compelling example is the instance `knights-80-5` which has the highest time (around 800 seconds) as well as a poor quality because the structure of the problem is unsuited for the bottom-up decomposition: there are only a few variables with large domains (more than 6000 values); there is almost no domain reduction in the top of the tree; and the propagation is very long.

To conclude, the parallel top-down decomposition of `Gecode` is fast and accurate while the bottom-up decomposition offers greater flexibility, but less robustness.

4.2.3 Influence of the Search Strategy

To analyze the influence of search strategies on the decomposition and the resolution, we apply a variable selection heuristic during the decomposition (master) and another one during the resolution (workers). Table 1 gives the solving times for the combinations of `lex` or `dom` when solving the instances `xcsp1`. Results are not reported if there is no significant differences among solving times. The choice of the variable selection heuristic is more critical for the decomposition than for the resolution. Indeed, initial choices made by the branching are both the least informed and the most important, as they lead to the largest subtrees and the search can hardly recover from early mistakes. From now on, the master and workers will use the same variable selection heuristic.

Instances	Worker	`lex`		`dom`	
	Master	`lex`	`dom`	`lex`	`dom`
`costasArray-14`		**191.2**	240.9	**191.4**	240.0
`latinSquare-dg-8_all`		479.4	**323.8**	470.6	**328.1**
`lemma-100-9-mod`		**109.7**	125.9	**101.8**	123.4
`pigeons-14`		1003.8	**956.3**	953.2	**899.1**
`quasigroup5-10`		182.2	**125.3**	188.5	**123.5**
`queenAttacking-6`		872.4	**598.3**	867.8	**622.5**
`squares-9-9`		**126.8**	1206.5	**127.8**	1213.0

Table 1: Solving times with different search strategies (`Choco2`, multi-core, $w = 2c = 80$).

4.3 Multi-core

In this section, we use parallel solvers based on Thread technologies to solve the instances of `xcsp1` or the n-queens problem using a multi-core computer. Let us recall that there is two worker 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 Schulte (2000), and Nielsen (2006).

4.3.1 Performance Analysis

Table 2 gives the solving times and speedups of the parallel solvers using 80 workers for the `xcsp1` instances. `Choco2` is tested with `lex` and `dom` whereas `Gecode` and `OR-tools` only use `lex`. They are also compared to a work stealing approach denoted `Gecode`-WS (Schulte, 2000; Nielsen, 2006). 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 hours time limit using the work stealing whereas they are using the sequential solver.

For `Choco2`, `dom` is more efficient in parallel than `lex` but remains slightly slower in average. Decomposition is a key of the bad performance on the instances `knights-80-5` and `lemma-100-9-mod`. As outlined before, the decomposition of `knights-80-5` takes more than 1100 seconds and generates too much subproblems, which forbids 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 heuristic because their

Instances	`Choco2-lex`		`Choco2-dom`		`Gecode`		`OR-tools`		`Gecode`-WS	
	t	su	t	su	t	su	t	su	t	su
`costasArray-14`	191.2	31.4	240.0	**38.8**	62.3	19.1	**50.9**	33.4	594.0	2.0
`knights-80-5`	1138.3	1.2	1133.1	1.5	**548.7**	**37.6**	2173.9	18.5	–	–
`latinSquare-dg-8_all`	479.4	39.0	328.1	39.2	251.7	**42.0**	**166.6**	35.2	4488.5	2.4
`lemma-100-9-mod`	109.7	4.0	123.4	4.1	6.7	10.1	**1.8**	**22.9**	3.0	22.3
`ortholatin-5`	248.7	30.0	249.9	36.0	421.7	13.5	**167.7**	**38.1**	2044.6	2.8
`pigeons-14`	1003.8	13.8	899.1	15.5	**211.8**	**39.1**	730.3	18.5	–	–
`quasigroup5-10`	182.2	30.7	123.5	**32.5**	18.6	26.4	**17.0**	**36.9**	22.8	21.5
`queenAttacking-6`	872.4	23.4	**622.5**	**28.5**	15899.1	?	–	–	–	–
`series-14`	39.3	29.9	39.3	32.9	**11.3**	**34.2**	16.2	28.7	552.3	0.7
`squares-9-9`	126.8	19.0	1213.0	16.1	**17.9**	18.4	81.4	**35.0**	427.8	0.8
AM (t) or GM (su)	439.2	15.9	497.2	17.4	1745.0	24.0	378.4	28.7	1161.9	3.3
Borda score (rank)	20.6 (3)		19.7 (4)		26.1 (1)		22.8 (2)		9.8 (5)	

Table 2: Solving times and speedups (multi-core, $w = 2c = 80$). `Gecode` and `OR-tools` use the `lex` heuristic.

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

4.3.2 Variations About the N-Queens Problem

Here, we verify the effectiveness of EPS in classic CSP settings. We consider four models for the well-known n-queens problem ($n = 17$). The n-queens puzzle is the problem of placing n chess queens on an $n \times n$ chessboard so that no two queens threaten each other. Here, we enumerate all solutions and the heuristics `lex` or `dom` are reasonable choices. The models are: `allDifferent` global constraints which enforce arc-consistency (AC); `allDifferent` constraints which enforce bound-consistency (BC); arithmetic inequalities constraints (NEQ); and a dedicated global constraint (JC) (Milano & Trick, 2004, ch. 3).

Table 3 gives the solving times and speedups of `Choco2` with 80 workers when the decomposition depth is either 3 or 4. What is striking for this result is that our splitting technique gives excellent results, with a linear speedup for up to 40 processors with the exception of the JC model. It is unfortunate since the JC model is clearly the best model for the sequential solver. Here, `dom` is always a better choice than `lex`. The number of subproblems for `dom` is the same whatever the model whereas the total number of nodes changes. It indicates that the filtering is weak at the top of the search tree.

Most other works report good results, and often linear speedups for the n-queens problem. Bordeaux et al. (2009) reported linear speedups up to 30 cores for the 17 queens, but no more improvement until 64 cores, whereas Machado et al. (2013) scales up to 512 workers using their hierarchical work stealing approach. Menouer and Le Cun (2014) reported speedups around 8 using 12 cores for the 16 queens, and Pedro, Abreu, Pedro, and Abreu (2010) reported speedups around 20 using 24 cores. Zoeteweij and Arbab (2004) reported linear speedups up to 16 cores for the 15 queens, Pedro et al. (2010) reported a speedup of 20 using 24 cores, So, the EPS efficiency is slightly above the average, because similar results are observed with 15 and 16 queens.

The previous experimental setting is in favor of EPS because we are exploring a search space exhaustively, and the problem is highly symmetric. Indeed, the variance of the subproblem's solving time is low, especially with higher levels of consistency. Note that the lower speedups of the JC model are probably not caused by load balancing issues because the subproblems of the NEQ model have a greater mean and variance.

Model	`lex`				`dom`			
	$d = 3$		$d = 4$		$d = 3$		$d = 4$	
	t	su	t	su	t	su	t	su
BC	838.8	38.3	835.1	38.5	640.4	38.7	**635.5**	**39.0**
AC	3070.2	38.8	3038.9	**39.3**	2336.2	38.8	**2314.7**	39.2
NEQ	280.7	31.8	241.9	36.9	188.8	36.4	**181.1**	**37.9**
JC	202.4	20.4	196.9	21.0	**140.6**	**24.2**	148.8	22.9

Table 3: Variations about the 17 queens problem (`Choco2`, multi-core, $w = 2c = 80$).

Instances	`lex`		`dom`		`dom/ddeg`		`dom/bwdeg`		`dom/wdeg`		`impact`	
	t	su	t	su	t	su	t	su	t	su	t	su
`cc-15-15-2`	1947.1	**5.2**	25701.7	**?**	–	–	**1524.9**	4.6	2192.1	2.1	31596.1	**?**
`costasArray-14`	**500.4**	**12.4**	641.9	12.1	895.3	8.6	4445.0	2.4	649.9	11.4	652.2	10.5
`crossword-m1`[1]	506.1	4.4	–	–	–	–	492.0	1.9	204.6	5.1	**179.5**	2.9
`crossword-m1c`[2]	2376.9	0.6	**1173.9**	0.6	1316.2	0.5	1471.3	0.7	1611.9	0.6	4689.6	**1.9**
`fapp07-0600-7`	–	–	–	–	–	–	**1069.5**	2.1	2295.7	1.8	–	–
`knights-20-9`	359.3	17.2	353.9	17.3	357.4	14.9	5337.5	1.3	491.3	**17.5**	**215.4**	16.5
`knights-25-9`	855.3	17.8	840.6	**18.0**	986.1	13.3	13264.8	1.3	1645.2	14.1	**550.8**	16.9
`knights-80-5`	**708.5**	2.0	726.9	2.0	716.4	2.1	1829.5	0.9	1395.6	**3.4**	896.3	2.5
`langford-3-17`	38462.7	**?**	**708.3**	**12.5**	5701.6	2.5	6397.5	1.9	3062.2	3.7	5995.6	**?**
`langford-4-18`	40465.2	**?**	**148.2**	**14.4**	1541.9	2.2	1307.1	2.1	538.3	4.8	1041.5	10.0
`langford-4-19`	–	–	**747.2**	**16.9**	–	**0.0**	7280.1	2.3	2735.3	5.6	4778.9	**?**
`latinSquare-dg`[3]	1161.7	14.3	903.2	12.2	812.0	**14.4**	416.9	4.2	294.8	11.3	**28.7**	5.0
`lemma-100-9-mod`	**110.5**	**4.1**	117.6	3.7	180.4	3.7	154.4	3.5	145.3	3.5	226.8	2.5
`ortholatin-5`	572.6	13.5	558.9	13.5	475.5	11.5	453.1	11.6	**362.4**	**13.7**	641.7	10.6
`pigeons-14`	**1330.1**	9.8	1492.6	8.3	1471.6	**11.8**	6331.1	2.6	2993.3	5.1	3637.2	4.2
`quasigroup5-10`	397.2	12.6	**277.3**	12.9	1156.6	3.6	733.5	5.2	451.5	7.9	308.4	**27.8**
`queenAttacking-6`	2596.7	7.3	1411.8	**10.6**	4789.7	4.2	2891.0	1.9	706.4	5.4	**427.1**	6.2
`queensKnights`[4]	–	–	–	–	–	–	**1517.8**	0.2	5209.5	**1.0**	–	–
`ruler-70-12-a3`	137.4	16.8	2410.6	**17.5**	–	–	51.5	2.4	42.8	6.7	**24.8**	12.1
`ruler-70-12-a4`	6832.0	3.9	4021.1	4.7	7549.2	2.2	1412.0	0.9	1331.3	2.3	**102.9**	**24.4**
`scen11-f5`	–	–	–	–	–	–	38698.7	**0.0**	–	**0.0**	–	–
`series-14`	**77.8**	**14.8**	89.1	12.4	9828.6	3.4	1232.2	2.6	338.9	9.9	346.5	8.5
`squares-9-9`	220.7	10.5	1987.4	7.2	129.7	9.4	697.2	2.2	**115.9**	**11.0**	138.9	10.8
`squaresUnsat`[5]	–	–	–	–	–	–	3766.1	1.2	**3039.8**	**2.9**	–	–
AM (t) or GM (su)	5243.1	7.5	2332.2	**8.6**	2369.3	4.8	4282.3	1.6	**1385.0**	4.9	2823.9	7.7
Borda score (rank)	65.1 (6)		72.8 (5)		49.6 (8)		91.2 (2)		100.3 (1)		81.4 (3)	

[1]`crossword-m1-words-05-06` [2]`crossword-m1c-words-vg7-7_ext` [3]`latinSquare-dg-8_all`
[4]`queensKnights-20-5-mul` [5]`squaresUnsat-19-19`

Table 4: Detailed speedups and solving times depending on the variable selection heuristics (`Choco2`, data center, $w = 16$).

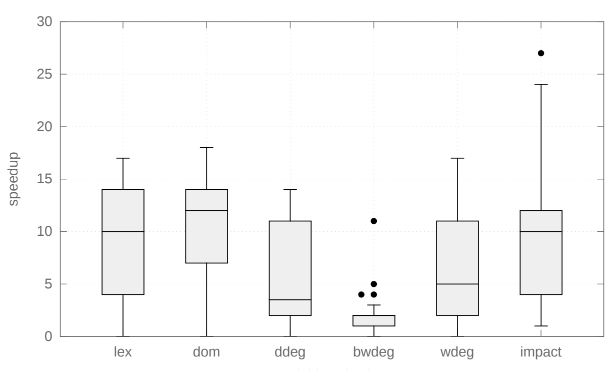


Figure 5: Speedups of the variable selection heuristics (`Choco2`, data center, $w = 16$).

4.4 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.

4.4.1 Influence of the Search Strategy

We study the performance of `Choco2` using 16 workers for solving the `xcsp` instances using the variable selection heuristics presented in Section 2.1. Figure 5 is a boxplot of the speedups for each variable selection heuristic. First, speedups are lower for `dom/bwdeg` because the decomposition is not *effective*. The binary branching states the constraint $x = a$ in the left branch and $x \neq a$ in the right branch. So, the workload between the left and right branches is imbalanced. In this case, only the positive decisions in the left branches should be taken into account. Second, without learning (`lex` and `dom`), the parallel algorithm is more efficient and robust in terms of speedup. With learning (`dom/bwdeg`, `dom/wdeg`, `impact`), the parallel algorithm may explore a different search tree than the sequential one. Indeed, the master only explores the top of the tree which changes the learning, and possibly the branching decisions. The worker also learns only from their subproblems, and not from the whole search tree. This frequently causes exploration overhead as for solving `queensKnights-20-5-mul` (twelve times more nodes using `dom/wdeg`) or, sometimes gives a super-linear speedup as for solving `quasigroup5-10` (three times less nodes using `impact`). Last, low speedups occur for all variable selection heuristics.

Table 4 gives solving times and speedups obtained for the different variable selection heuristics. Borda scores are computed for `Choco2` (Table 4) and `Gecode` (Table 5). First, no variable selection heuristics strictly dominates the others either in sequential or parallel. However, `dom/wdeg` is the most robust as outlined by the Borda scores. In fact, the variability of the solving times between the different heuristics is reduced by the parallelization, but remains important. Second, in spite of low speedups, `dom/bwdeg` remains the second best variable selection heuristic for parallel solving because it was the best one in sequential. In average, using advanced variable selection heuristics such as `dom/bwdeg`, `dom/wdeg`, or `impact` gives lower solving times than `lex` or `dom` in spite of lower speedups. It highlights the fact that decomposition procedures should handle any branching strategy. In Section 4.6.1, we will investigate the very low speedups for the instance `crossword-m1c-words-vg7-7` that are not caused by the variable selection heuristics.

4.4.2 Scalability up to 512 Workers

Table 5 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 the work stealing. With 16 workers, the work stealing is ranked last using the Borda score. With 512 workers, EPS is in average almost 10 times faster than the work stealing. It is also more efficient because they both parallelize the same sequential solver. On the multi-core machine, `Gecode` was faster than `Choco2` on most instances of `xcsp1`. Here, the performance of `Gecode` are more mitigated as outlined by the Borda scores. Five instances that are not solved within the time limit by `Gecode` are not reported in Table 5. Six instances are not solved with 16 workers whereas twelve instances were not solved with the sequential solver. By way of comparison, only five instances are not solved by `Choco2` using the `lex`

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
Borda score (rank)	76.9 (4)		60.3 (7)					

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

Table 5: Speedups and solving times for `xcsp` (`Gecode`, `lex`, data center, $w = 16$ or 512).

heuristics whereas all instances are solved in sequential or parallel when using `dom/wdeg` or `dom/bwdeg`. Once again, it highlights the importance of the search strategy.

Figure 6 is a boxplot of the speedups with different numbers of workers for solving `fzn` instances. The median of speedups are around $\frac{w}{2}$ in average and their dispersion remains low.

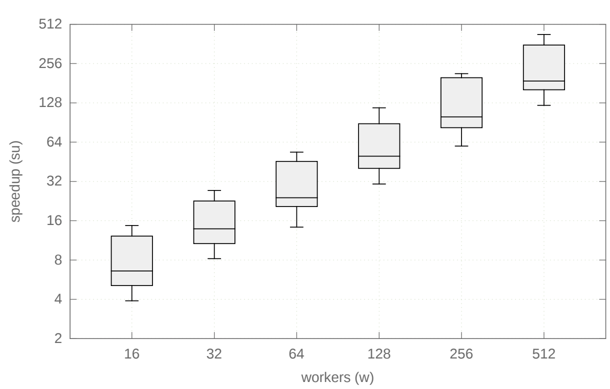


Figure 6: Scalability up to 512 workers (`Gecode`, `lex`, data center).

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 6: Solving times and speedups for `fzn` (`Gecode`, `lex`, cloud, $w = 24$).

4.5 Cloud Computing

EPS can be deployed on the Microsoft Azure cloud platform. The available computing infrastructure is organized as follows: cluster nodes computes the application; one head node manages the cluster nodes; and proxy nodes load-balances communication between cluster nodes. On the contrary to a data center, cluster nodes may be far from each other and communication time may take longer. Proxy nodes requires 2 cores and are managed by the service provider. Here, 3 nodes of 8 cores with 56 GB of RAM memory provide 24 workers (cluster nodes) managed by MPI.

Table 6 compares the `Gecode` implementations of EPS and work stealing for solving the `fzn` instances with 24 workers. Briefly, EPS is always faster than the work stealing, and therefore, more efficient because they both parallelize the same sequential solver. The 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 forbid improvements of the work stealing such as those proposed by Machado et al. (2013), or by Xie and Davenport (2010).

4.6 Embarrassingly Distributed Search

In this section, we transform with reasonable effort a parallel solver (EPS) into a distributed parallel solver (EDPS) by using the batch scheduler OAR (Capit et al., 2005) provided by the data center. In fact, the batch scheduler OAR plays the foreman. The parallel `Choco2` solver is modified so that the workers write the subproblems into files instead of solving them. Then, a script submits the jobs/subproblems to the OAR batch scheduler, waits for their termination, and gathers the results. OAR schedules jobs on the cluster using priority FIFO with backfilling and fair-share based priorities. Backfilling allows to start lower priority jobs without delaying highest priority jobs whereas fair-share means that no user/application is preferred in any way. The main drawback is that a new worker must be created for each subproblem. Each worker process is allocated by OAR with predefined resources. A worker is either a sequential (EDS) or a parallel solver (EDPS).

This approach offers a practical advantage for resource reservation in a data center. Indeed, when asking for an MPI process, one has to wait until enough resources are available before the process starts. Here, resources (cores or nodes) are nibbled as soon as they become available which can drastically reduce the waiting time. Furthermore, it bypasses limitations of the Threads Technology by allowing to use multiple nodes of the data center. However, it clearly increases the recomputation overhead because, a worker solves a single subproblem instead of multiple subproblems. So, the model creation and initial propagation are realized more often. It also introduces a non-negligible *submission overhead* which is the time taken to create and submit all jobs to the OAR batch scheduler.

4.6.1 Anomaly in `crossword-m1c-words-vg7-7_ext`

We investigate the very low speedups for solving the instance `crossword-m1c-words-vg7-7` with any variable selection heuristic (see Table 4). We compare the results of the parallel (EPS, $w = 16$) and distributed (EDS with sequential worker) algorithms for different decomposition depths ($d = 1, 2, 3$). Table 7 gives the solving times, speedups, and efficiencies. The number of distinct cores used by the distributed algorithm is a bad estimator for computing efficiencies, because some of them are used only for a short period of time. Therefore, the number c of cores used to compute the efficiency of EDS or EDPS is estimated as the ratio of the total runtime over the wall-clock time.

First, the parallel algorithm is always slower than the sequential one. However, the speedups of the distributed algorithms are significant even if they decrease quickly as the decomposition depth increases. The fall of the efficiency shows that EDS is not scalable with sequential workers. Indeed, the recomputation, and especially the submission overhead become too important when the number of subproblems increases.

Second, the bad performance of the parallel algorithms are not caused by a statistically imbalanced decomposition because we would observe similar performance for the distributed algorithm. Profiling the parallel algorithm on this particular instances suggests that the bad

		EDS			EPS ($w = 16$)		
d	p	t	su	eff	t	su	eff
2	186	73.0	10.2	0.435	1069.9	0.7	0.044
3	827	229.0	3.3	0.128	1074.2	0.7	0.044
4	2935	797.0	1.1	0.039	1091.8	0.7	0.044

Table 7: EDS and EPS for the `crossword` instance (`Choco2`, `dom`, data center).

performance comes from the underlying solver itself. Indeed, the number of instructions is similar for the sequential and parallel algorithms whereas the numbers of context switches, cache references and cache misses increase considerably. In fact, the parallel algorithms spent more than half of its time in some internal methods of extensional constraints, *i.e.* the relation of the constraint is specified by listing its satisfying tuples. This issue occurred on all computing infrastructure and for different Java virtual machines. Note that other instances use extensional constraints, but they impose fewer consequences. This issue would not happen with an MPI implementation because there is no shared memory. So, it advocates for implementations of EPS based on MPI rather than on the Thread Technology.

4.6.2 Variations About the Golomb Ruler Problem

A Golomb ruler is a set of marks at integer positions along an imaginary ruler such that no two pairs of marks are the same distance apart. The number of marks on the ruler is its order, and the largest distance between two of its marks is its length. Here, we *enumerate the optimal rulers* (minimal length for the specific number of marks) with a simple constraint model inspired from the one by Galinier, Jaumard, Morales, and Pesant (2001) for which the heuristics `lex` or `dom` are a reasonable choice. Table 8 gives the solving times, speedups, and efficiencies for the parallel algorithm ($w = 16$), the distributed algorithm with sequential workers ($w = 1$), and the distributed algorithms with parallel workers ($w = 16$ and the worker decomposition depth is $d_w = 2$) with different master decomposition depths d.

First, EPS obtains almost linear speedup if the decomposition depth is large enough. Without surprise, speedups are lower if there are not enough subproblems. Second, the distributed algorithm EDS with sequential workers is efficient only if the number of subproblems remains low. Otherwise, it can still give some speedups (`dom`), but wastes the resources since the efficiency is very low. In fact, submitting too many jobs to the batch scheduler (`lex`) lead to a high submission overhead (around 13 minutes) and globally degrades the performance. Finally, the distributed algorithms with parallel workers offer a good trade-off between speedups and efficiencies because it allows to use many resources while only submitting a few jobs thus reducing the submission and recomputation overheads. Note that EDS with $d = 1$ is not tested because it is roughly equivalent to EPS with 16 workers, and EDPS with $d = 3$ is not tested because the submission overhead becomes too important.

			EDS			EDPS ($w = 16, d_w = 2$)			EPS ($w = 16$)		
	d	p	t	su	eff	t	su	eff	t	su	eff
`lex`	1	20	–	–	–	572.0	89.7	0.968	11141.7	4.6	0.288
	2	575	769.0	66.8	0.846	497.0	103.3	0.232	4084.2	12.6	0.786
	3	14223	17880.0	2.9	0.005	–	–	–	3502.6	14.7	0.916
`dom`	1	20	–	–	–	1538.0	78,6	0,935	28299,9	4,3	0,267
	2	222	2394.0	50,5	0,989	366.0	330,2	0,742	9703,6	12,4	0,778
	3	5333	3018.0	40,0	0,146	–	–	–	8266,6	14,6	0,914

Table 8: EDS and EPS for Golomb Ruler with 14 marks (`Choco2`, data center).

Most other parallel approaches reported good performance for the Golomb ruler problem. For instance, Michel et al. (2009), and Chu et al. (2009) respectively reported linear speedups for 4 and 8 workers. EDS is more efficient than the work stealing proposed by Menouer and Le Cun (2014) using 48 workers for the ruler with 13 marks and as efficient as the selfsplit by Fischetti et al. (2014) using 64 workers for the ruler with 14 marks.

Last, we enumerated optimal Golomb Rulers with 15 and 16 marks using EDPS. The Master and workers use the `lex` heuristic. The master decomposition depth d is equal to 2 that generates around 800 hundreds subproblems. There are 16 parallel workers with a decomposition depth d_w equal to 2. With this settings, we used more than 700 cores of the data center during the solving process. So, it bypasses the limitations on the number of cores used by MPI imposed by the administrator. Furthermore, the solving process starts immediately because cores are grabbed as soon as they become available whereas a MPI process waits that enough cores becomes simultaneously available. Enumerating optimal rulers with 15 and 16 marks respectively took 1422 and 5246 seconds. To our knowledge, this is the first time where a constraint solver finds these rulers, and furthermore in a reasonable amount of time. However, these optimal rulers have been discovered via an exhaustive computer search (Shearer, 1990). More recently, Distributed Computing Technologies Inc (20) found optimum rulers up to 26 marks. Beside, plane construction (Atkinson & Hassenklover, 1984) allows to find larger optimal rulers.

4.7 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 4 portfolios. The portfolio `CPHydra` (O'Mahony et al., 2008) uses features selection on the 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 (see Section 2.1) as well as randomization and restarts. Details about `Choco2` and CAG can be found in (Malapert & Lecoutre, 2014). 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 medal 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 portfolio (up to 512 workers) is a difficult task and almost no implementation is publicly available.

Table 9 gives the solving times of EPS and portfolios for solving the `xcsp` instances on the data center. First, `CPHydra` with 16 workers only solves 2 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 less 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 in 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

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
Borda score (rank)	65.0 (3)	52.2 (5)	77.1 (1)	57.0 (4)	72.8 (2)	20.0 (6)

Table 9: Solving times of EPS and portfolio (data center).

B&B algorithm. This is emphasized for the CAG portfolio that 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, Choco2 and CAG portfolios are more robust thanks to the inherent diversification, but the 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.

5. Conclusion

We have introduced an Embarrassingly Parallel Search (EPS) method for solving constraint satisfaction problems and constraint optimization problems. This approach has several advantages. First, it is an efficient method which matches or even outperforms state-of-the-art algorithms on a number of problems using various computing infrastructures. Second, it involves almost no communication or synchronization and mostly relies on the underlying sequential solver so that the implementation and debugging is made easier. Last, the simplicity of the method allows to propose many variants adapted to specific applications or computing infrastructures. Moreover, under certain restrictions, the parallel algorithm can be deterministic, and even mimic the sequential algorithm which is very important in practice either in production or for debugging.

There are several interesting perspectives around EPS. First, it can be modified in order to provide diversification and to learn useful information when solving subproblems. For instance, it can easily be combined with a portfolio approach in which subproblems can be solved by several search strategies. Second, thanks to its simplicity, the simplest variants of EPS could be implemented as meta-searches (Rendl, Guns, Stuckey, & Tack, 2015), and would offer a convenient way to parallelize applications with a satisfactory efficiency. Last, another perspective is to predict the solution time of a large combinatorial problem, based on known solution times of a small set of subproblems based on statistical or machine learning approaches.

Acknowledgments

We would like to thank very much Christophe Lecoutre, Laurent Perron, Youssef Hamadi, Carine Fédèle, Bertrand Lecun and Tarek Menouer for their comments and advices which helped to improve the paper. This work has been supported by both CNRS and OSEO (BPI France) within the ISI project "Pajero". This work was granted access to the HPC and visualization resources of "Centre de Calcul Interactif" hosted by "Université Nice Sophia Antipolis", and also to the Microsoft Azure Cloud. We also wish to thank the anonymous referees for their comments.

Appendix A. Efficiency of Hyper-Threading

In this section, we show that hyper-threading technology improves the efficiency of EPS for solving the instances of `xcsp1` on a multi-core computer. Figure 7 is a boxplot of the speedups provided by the hyper-threading for each parallel solver among `Choco2`, `Gecode`, `OR-tools`. Here, the speedups indicate how many times a parallel solver using 80 workers ($w = 2c$) is faster than one using 40 workers ($w = c$). The maximum speedup according to Amdahl's law is 2.

`Choco2` is tested with `lex` and `dom` whereas `Gecode` and `OR-tools` only use `lex`. They are also compared to a work stealing approach proposed by Schulte (2000) and denoted `Gecode`-WS. Hyper-threading clearly improves the parallel efficiency of EPS whereas the performance of the work stealing roughly remains unchanged. It is interesting because EPS has a very high CPU demand and resources of each physical core are shared by its two logical cores. Indeed, the performance of hyper-threading are known to be very application-dependent. With the exception of `lemma-100-9-mod` and `squares-9-9`, `Choco2` and `OR-tools` are faster with 80 workers. For `lemma-100-9-mod`, the `Choco2` decomposition for 80 workers takes longer and generates too many subproblems. The instance is solved

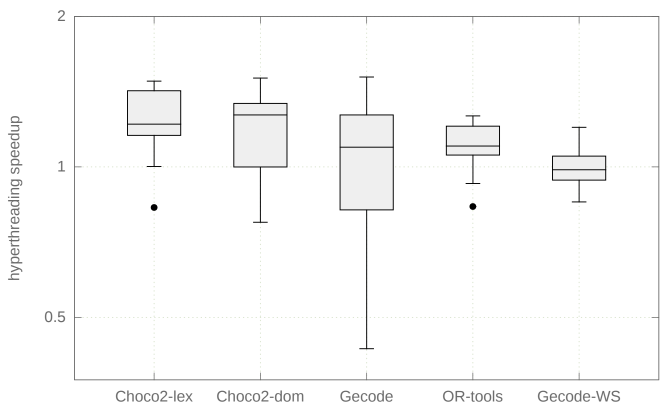


Figure 7: Speedups provided by hyper-threading (multi-core, $w = 40, 80$).

easily by `OR-tools` (less than two seconds) and it becomes difficult to improve its efficiency. For `squares-9-9`, the decomposition changes according to the number of workers, but it cannot explain why hyper-threading does not improve EPS. The parallel efficiency of `Gecode` is reduced for multiple instances and the interest of hyper-threading is less obvious than for `Choco2` or `OR-tools`. To conclude, hyper-threading globally improves the efficiency of EPS while it has a limited interest on the work stealing.

References

Almasi, G. S., & Gottlieb, A. (1989). *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

Amadini, R., Gabbrielli, M., & Mauro, J. (2013). An Empirical Evaluation of Portfolios Approaches for Solving CSPs In Gomes, C., & Sellmann, M.Eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Vol. 7874 of *Lecture Notes in Computer Science*, pp. 316–324. Springer Berlin Heidelberg.

Amdahl, G. (1967). 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, New York, NY, USA. ACM.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., & Werthimer, D. (2002). Seti@home: An experiment in public-resource computing *Commun. ACM*, *45*(11), 56–61.

Atkinson, M. D., & Hassenklover, A. (1984). Sets of Integers with Distinct Differences Tech. Rep. SCS-TR-63, School of Computer Science, Carlton University, Ottawa Ontario, Canada.

Bader, D., Hart, W., & Phillips, C. (2005). Parallel Algorithm Design for Branch and Bound In G, H.Ed., *Tutorials on Emerging Methodologies and Applications in Operations Research*, Vol. 76 of *International Series in Operations Research & Management Science*, pp. 5–1–5–44. Springer New York.

Barney, B., & Livermore, L. (2016). Introduction to Parallel Computing `https://computing.llnl.gov/tutorials/parallel_comp/`.

Bauer, M. A. (2007). High performance computing: The software challenges In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pp. 11–12, New York, NY, USA. ACM.

Beck, C., Prosser, P., & Wallace, R. (2005). Trying Again to Fail-First In *Recent Advances in Constraints*, pp. 41–55. Springer Berlin Heidelberg.

Bordeaux, L., Hamadi, Y., & Samulowitz, H. (2009). Experiments with Massively Parallel Constraint Solving. In Boutilier (Boutilier, 2009), pp. 443–448.

Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting Systematic Search by Weighting Constraints In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS*, pp. 146–150.

Boutilier, C.Ed.. (2009). *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17.*

Brams, S. J., & Fishburn, P. C. (2002). Voting procedures In Arrow, K. J., Sen, A. K., & Suzumura, K.Eds., *Handbook of Social Choice and Welfare*, Vol. 1 of *Handbook of Social Choice and Welfare*, chap. 4, pp. 173–236. Elsevier.

Budiu, M., Delling, D., & Werneck, R. (2011). DryadOpt: Branch-and-bound on distributed data-parallel execution engines In *Parallel and Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 1278–1289. IEEE.

Burton, F. W., & Sleep, M. R. (1981). 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, New York, NY, USA. ACM.

Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounie, G., Neyron, P., & Richard, O. (2005). 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, Washington, DC, USA. IEEE Computer Society.

Choco, T. (2010). Choco: an open source java constraint programming library *Ecole des Mines de Nantes, Research report*, *1*, 10–02.

Chong, Y. L., & Hamadi, Y. (2006). 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, Amsterdam, The Netherlands, The Netherlands. IOS Press.

Chu, G., Schulte, C., & Stuckey, P. J. (2009). Confidence-Based Work Stealing in Parallel Constraint Programming In Gent, I. P.Ed., *CP*, Vol. 5732 of *Lecture Notes in Computer Science*, pp. 226–241. Springer.

Chu, G., Stuckey, P. J., & Harwood, A. (2008). PMiniSAT: A Parallelization of MiniSAT 2.0 Tech. Rep., NICTA : National ICT Australia.

Cire, A. A., Kadioglu, S., & Sellmann, M. (2014). Parallel Restarted Search In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pp. 842–848. AAAI Press.

Cornuéjols, G., Karamanov, M., & Li, Y. (2006). Early Estimates of the Size of Branch-and-Bound Trees *INFORMS Journal on Computing*, *18*, 86–96.

Crainic, T. G., Le Cun, B., & Roucairol, C. (2006). Parallel branch-and-bound algorithms *Parallel combinatorial optimization*, *1*, 1–28.

De Kergommeaux, J. C., & Codognet, P. (1994). Parallel logic programming systems *ACM Computing Surveys (CSUR)*, *26*(3), 295–336.

Distributed Computing Technologies Inc (20). The Distributed.net home page `http://www.distributed.net/`.

Een, N., & Sörensson, N. (2005). MiniSat: A SAT solver with conflict-clause minimization *Sat*, *5*, 1.

Ezzahir, R., Bessière, C., Belaissaoui, M., & Bouyakhf, E. H. (2007). 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.

Fischetti, M., Monaci, M., & Salvagnin, D. (2014). Self-splitting of workload in parallel computation In Simonis, H.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, Cham. Springer International Publishing.

Gabriel, E., Fagg, G., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. (2004). Open MPI: Goals, Concept, and Design of a next generation MPI implementation In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 97–104. Springer.

Galea, Fran c., & Le Cun, B. (2007). 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.

Galinier, P., Jaumard, B., Morales, R., & Pesant, G. (2001). A Constraint-Based Approach to the Golomb Ruler Problem In *3rd International Workshop on integration of AI and OR techniques*.

Gendron, B., & Crainic, T. G. (1994). Parallel branch-and-bound algorithms: Survey and synthesis *Operations research*, *42*(6), 1042–1066.

Gent, I., & Walsh, T. (1999). CSPLIB: A Benchmark Library for Constraints In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, CP '99, pp. 480–481.

Gomes, C., & Selman, B. (1997). Algorithm Portfolio Design: Theory vs. Practice In *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, pp. 190–197.

Gomes, C., & Selman, B. (1999). Search strategies for hybrid search spaces In *Tools with Artificial Intelligence, 1999. Proceedings. 11th IEEE International Conference*, pp. 359–364. IEEE.

Gomes, C., & Selman, B. (2000). Hybrid Search Strategies For Heterogeneous Search Spaces *International Journal on Artificial Intelligence Tools*, *09*, 45–57.

Gomes, C., & Selman, B. (2001). Algorithm Portfolios *Artificial Intelligence*, *126*, 43–62.

Gropp, W., & Lusk, E. (1993). The MPI communication library: its design and a portable implementation In *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pp. 160–165. IEEE.

Gupta, G., Pontelli, E., Ali, K. A., Carlsson, M., & Hermenegildo, M. V. (2001). Parallel execution of prolog programs: a survey *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *23*(4), 472–602.

Halstead, R. (1984). Implementation of Multilisp: Lisp on a Multiprocessor In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pp. 9–17, New York, NY, USA. ACM.

Hamadi, Y. (2002). Optimal Distributed Arc-Consistency *Constraints*, *7*, 367–385.

Hamadi, Y., Jabbour, S., & Sais, L. (2008). ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, *6*(4), 245–262.

Haralick, R., & Elliott, G. (1980). Increasing Tree Search Efficiency for Constraint Satisfaction Problems *Artificial intelligence*, *14*(3), 263–313.

Harvey, W. D., & Ginsberg, M. L. (1995). 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.

Heule, M. J., Kullmann, O., Wieringa, S., & Biere, A. (2012). Cube and conquer: Guiding CDCL SAT solvers by lookaheads In *Hardware and Software: Verification and Testing*, pp. 50–65. Springer.

Hirayama, K., & Yokoo, M. (1997). Distributed Partial Constraint Satisfaction Problem In *Principles and Practice of Constraint Programming-CP97*, pp. 222–236. Springer.

Hyde, P. (1999). *Java thread programming*, Vol. 1. Sams.

Intel Corporation (2015). Intel MPI Library `https://software.intel.com/en-us/intel-mpi-library`.

Jaffar, J., Santosa, A. E., Yap, R. H. C., & Zhu, K. Q. (2004). Scalable Distributed Depth-First Search with Greedy Work Stealing In *16th IEEE International Conference on Tools with Artificial Intelligence*, pp. 98–103. IEEE Computer Society.

Kale, L., & Krishnan, S. (1993). *CHARM++: a portable concurrent object oriented system based on C++*, Vol. 28. ACM.

Kasif, S. (1990). On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction networks *Artificial Intelligence*, *45*, 275–286.

Kautz, H., Horvitz, E., Ruan, Y., Gomes, C., & Selman, B. (2002). Dynamic Restart Policies *18th National Conference on Artificial Intelligence AAAI/IAAI*, *97*, 674–681.

Kjellerstrand, H. (2014). Håkan Kjellerstrand's Blog `http://www.hakank.org/`.

Kleiman, S., Shah, D., & Smaalders, B. (1996). *Programming with threads*. Sun Soft Press.

Korf, R. E., & Schreiber, E. L. (2013). Optimally Scheduling Small Numbers of Identical Parallel Machines In Borrajo, D., Kambhampati, S., Oddi, A., & Fratini, S.Eds., *ICAPS*. AAAI.

Krishna, J., Balaji, P., Lusk, E., Thakur, R., & Tiller, F. (2010). Implementing MPI on Windows: Comparison with Common Approaches on Unix In *Recent Advances in the Message Passing Interface*, Vol. 6305 of *Lecture Notes in Computer Science*, pp. 160–169. Springer Berlin Heidelberg.

Lai, T.-H., & Sahni, S. (1984). Anomalies in Parallel Branch-and-bound Algorithms *Commun. ACM*, *27*(6), 594–602.

Lantz, E. (2008). *Windows HPC Server : Using Microsoft Message Passing Interface (MS-MPI)*.

Le Cun, B., Menouer, T., & Vander-Swalmen, P. (2007). Bobpp `http://forge.prism.uvsq.fr/projects/bobpp`.

Léauté, T., Ottens, B., & Szymanek, R. (2009). FRODO 2.0: An open-source framework for distributed constraint optimization. In Boutilier (Boutilier, 2009), pp. 160–164.

Leiserson, C. E. (2010). The Cilk++ concurrency platform *The Journal of Supercomputing*, *51*(3), 244–257.

Lester, B. (1993). *The art of parallel programming*. Prentice Hall Englewood Cliffs, NJ.

Li, H. (2009). *Introducing Windows Azure*. Apress, Berkely, CA, USA.

Luby, M., Sinclair, A., & Zuckerman, D. (1993). Optimal Speedup of Las Vegas Algorithms *Inf. Process. Lett.*, *47*, 173–180.

Machado, R., Pedro, V., & Abreu, S. (2013). On the Scalability of Constraint Programming on Hierarchical Multiprocessor Systems In *ICPP*, pp. 530–535. IEEE.

Malapert, A., & Lecoutre, C. (2014). À propos de la bibliothèque de modèles XCSP In *10èmes Journées Francophones de Programmation par Contraintes(JFPC'15)*, Angers, France.

Mattson, T., Sanders, B., & Massingill, B. (2004). *Patterns for Parallel Programming* (First ed.). Addison-Wesley Professional.

Menouer, T., & Le Cun, B. (2013). 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.

Menouer, T., & Le Cun, B. (2014). 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*.

Michel, L., See, A., & Hentenryck, P. V. (2009). Transparent Parallelization of Constraint Programming *INFORMS Journal on Computing*, *21*, 363–382.

Microsoft Corporation (2015). Microsoft HPC Pack 2012 R2 and HPC Pack 2012 `http://technet.microsoft.com/en-us/library/jj899572.aspx`.

Milano, M., & Trick, M. (2004). *Constraint and Integer Programming: Toward a Unified Methodology*. Springer US, Boston, MA.

Moisan, T., Gaudreault, J., & Quimper, C.-G. (2013). Parallel Discrepancy-Based Search In *Principles and Practice of Constraint Programming*, Vol. 8124 of *Lecture Notes in Computer Science*, pp. 30–46. Springer Berlin Heidelberg.

Moisan, T., Quimper, C.-G., & Gaudreault, J. (2014). Parallel Depth-bounded Discrepancy Search In Simonis, H.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, Cham. Springer International Publishing.

MPI-CH Team (2015). High-Performance Portable MPI `http://www.mpich.org/`.

Mueller, F., et al. (1993). A Library Implementation of POSIX Threads under UNIX. In *USENIX Winter*, pp. 29–42.

Nguyen, T., & Deville, Y. (1998). A distributed arc-consistency algorithm *Science of Computer Programming*, *30*(1–2), 227 – 250. Concurrent Constraint Programming.

NICTA Optimisation Research Group (2012). MiniZinc and FlatZinc `http://www.g12.csse.unimelb.edu.au/minizinc/`.

Nielsen, M. (2006). Parallel Search in Gecode Master's thesis, KTH Royal Institute of Technology.

O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., & O'Sullivan, B. (2008). Using case-based reasoning in an algorithm portfolio for constraint solving In *Irish Conference on Artificial Intelligence and Cognitive Science*, pp. 210–216.

Pedro, V., Abreu, S., Pedro, V., & Abreu, S. (2010). Distributed Work Stealing for Constraint Solving *CoRR*, *abs/1009.3800*, 1–18.

Perron, L. (1999). 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, Berlin, Heidelberg. Springer Berlin Heidelberg.

Perron, L., Nikolaj, V. O., & Vincent, F. (2012). Or-Tools Tech. Rep., Google.

Pruul, E., Nemhauser, G., & Rushmeier, R. (1988). Branch-and-bound and Parallel Computation: A historical note *Operations Research Letters*, *7*, 65–69.

Refalo, P. (2004). Impact-Based Search Strategies for Constraint Programming In Wallace, M.Ed., *Principles and Practice of Constraint Programming, 10th International Conference, CP 2004, Toronto, Canada*, Vol. 3258 of *Lecture Notes in Computer Science*, pp. 557–571. Springer.

Régin, J.-C., Rezgui, M., & Malapert, A. (2013). 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.

Régin, J.-C., Rezgui, M., & Malapert, A. (2014). Improvement of the Embarrassingly Parallel Search for Data Centers In O'Sullivan, B.Ed., *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, Vol. 8656 of *Lecture Notes in Computer Science*, pp. 622–635. Springer International Publishing, Cham.

Rendl, A., Guns, T., Stuckey, P., & Tack, G. (2015). MiniSearch: A Solver-Independent Meta-Search Language for MiniZinc In Pesant, G., Pesant, G., & Pesant, G.Eds., *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31 – September 4, 2015, Proceedings*, Vol. 9255 of *Lecture Notes in Computer Science*, pp. 376–392. Springer International Publishing, Cham.

Rezgui, M., Régin, J.-C., & Malapert, A. (2014). Using Cloud Computing for Solving Constraint Programming Problems In *First Workshop on Cloud Computing and Optimization, a conference workshop of CP 2014*, Lyon, France.

Rolf, C. C., & Kuchcinski, K. (2009). Parallel Consistency in Constraint Programming *PDPTA '09: The 2009 International Conference on Parallel and Distributed Processing Techniques and Applications*, *2*, 638–644.

Rossi, F., Van Beek, P., & Walsh, T.Eds.. (2006). *Handbook of Constraint Programming.* Elsevier.

Roussel, O., & Lecoutre, C. (2008). Xml representation of constraint networks format `http://www.cril.univ-artois.fr/CPAI08/XCSP2_1Competition.pdf`.

Schulte, C. (2000). 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.

Schulte, C. (2006). Gecode: Generic Constraint Development Environment `http://www.gecode.org/`.

Shearer, J. B. (1990). Some New Optimum Golomb Rulers *IEEE Trans. Inf. Theor.*, *36*(1), 183–184.

Stephan, K., & Michael, K. (2011). SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing In *Pragmatics of SAT*.

Sutter, H., & Larus, J. (2005). The free lunch is over: A fundamental turn toward toward Concurrency *Dr. Dobb's Journal*, *30*, 202–210.

Van Der Tak, P., Heule, M. J., & Biere, A. (2012). Concurrent cube-and-conquer In *Theory and Applications of Satisfiability Testing–SAT 2012*, pp. 475–476. Springer.

Vidal, V., Bordeaux, L., & Hamadi, Y. (2010). 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.

Wahbi, M., Ezzahir, R., Bessiere, C., & Bouyakhf, E.-H. (2011). 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.

Wilkinson, B., & Allen, M. (2005). *Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers* (2nd ed.). Prentice-Hall Inc.

Xie, F., & Davenport, A. (2010). Massively Parallel Constraint Programming for Supercomputers: Challenges and Initial Results In *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*, Vol. 6140 of *Lecture Notes in Computer Science*, pp. 334–338, Berlin, Heidelberg. Springer Berlin Heidelberg.

Xu, L., Hoos, H., & Leyton-Brown, K. (2010). Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection In *AAAI Conference on Artificial Intelligence*, Vol. 10, pp. 210–216.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based Algorithm Selection for SAT *Journal of Artificial Intelligence Research*, *32*, 565–606.

Yokoo, M., Ishida, T., & Kuwabara, K. (1990). Distributed Constraint Satisfaction for DAI Problems In *Proceedings of the 1990 Distributed AI Workshop*, Bandara, TX.

Zoeteweij, P., & Arbab, F. (2004). A Component-Based Parallel Constraint Solver In De Nicola, R., Ferrari, G. L., & Meredith, G.Eds., *Coordination*, Vol. 2949 of *Lecture Notes in Computer Science*, pp. 307–322. Springer.