Parallel Discrepancy-based Search: An efficient and scalable search strategy for massively parallel supercomputers providing intrinsic load-balancing without communication

Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper

Université Laval, Québec, Canada
Thierry.Moisan.1@ulaval.ca, Jonathan.Gaudreault@forac.ulaval.ca,
Claude-Guy.Quimper@ift.ulaval.ca

Abstract. Backtracking strategies based on the computation of discrepancies have proved themselves successful at solving large problems. They show really good performance when provided with a high-quality domain-specific branching heuristic (variable and value ordering heuristic), which is the case for many industrial problems.
We propose a novel approach (PDS) that allows parallelizing a strategy based on the computation of discrepancies (LDS). The pool of processors visits the leaves in exactly the same order as the centralized algorithm would do. The implementation allows for a natural/intrinsic load balancing to occur (filtering induced by constraint propagation would affect each processor pretty much in the same way), although there is no communication between processors. These properties make PDS a scalable algorithm to be used on massively parallel supercomputer with thousands of cores.

1 Introduction

Constraint solvers have been used for decades and were successful at solving numerous operations research problems. For instance, these solvers are used for optimizing computer networks by better routing the traffic [1, 2], and for planning and scheduling problems [3] in different industries, among them the forest products industry [4, 5]. A solver accepts as input a combinatorial problem defined by a set of variables and a set of constraints posted on these variables. The solver usually explores the candidate solutions by doing a backtracking search in a tree.

With the rise of multi-core servers, there has been an increase in research for parallelizing constraint solvers. Parallelization is not trivial as there is need for a trade-off between the workload balance, the communication cost, and the duplication (redundancy) of work between the processors.

The choice of an efficient search strategy is instrumental in solving large industrial problems, even in a centralized environment (for performance reasons, it

is essential to explore the most promising leaves first). Among others, backtracking strategies based on the analysis of discrepancies such as LDS [6], DDS [7], and DBDFS [8] have proved themselves successful at solving large problems. They show really good performance when provided with a high-quality branching heuristic (that is, variable and value ordering heuristic), which is the case for many industrial problems (e.g. [5]).

In this article, we propose a novel approach (PDS) that allows parallelizing a strategy based on the computation of discrepancies (i.e. LDS). The proposed approach shows the following characteristics:

- The pool of processors globally visits the leaves in exactly the same order as the centralized version of LDS would do.
- There is no need for communication between the processors.
- The implementation allows for a natural/intrinsic load balancing to occur (filtering induced by constraint propagation would affect each processor pretty much in the same way).
- The method provides robustness (if a processor dies, it can be replaced by a new one that must however restart the work allocated to this processor).
- It offers pretty good scaling: adding additional processors can never slow down the global process as happens with approaches using communication (however, redundancy between processors increases with the number of processors used).

These properties make PDS a scalable algorithm that we intend to use to solve industrial problems from the forest products industry (see [4, 5]) using a massively parallel supercomputer called COLOSSE deployed at Université Laval (7680 cores). However, as this is an ongoing work, at this stage we do not provide a comprehensive performance analysis using industrial data. The focus of this paper is rather to provide a description of the proposed algorithm, and a discussion and analysis of its properties using synthetic data.

The remainder of this paper is organized as follows. Section 2 reviews basic concepts related to parallel tree search. Sections 3 and 4 describe the original algorithm and the parallel version. Section 5 reports theoretical results and evaluates the performance of the algorithm on synthetic trees in order to illustrate different characteristics of the algorithm. Section 6 concludes the paper.

2 Basic Concepts

This section provides an overview of the main approaches regarding parallel tree search. We then give an overview of previous attempts that were made in order to parallelize discrepancy-based strategies.

2.1 Search space in shared memory

The simplest method for parallel tree search is implemented by having many cores share a list of open nodes (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). A comprehensive framework based on this idea was proposed in [9]. Good performance is often reported, as in [10] where a parallel Best First Search was implemented, and evaluated 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. For this reason, the approach cannot easily be adapted for massively parallel supercomputers with thousands of cores.

2.2 Search space splitting / work stealing

This family of approaches is often reported as the most frequently seen in the literature [11]. The main idea is to have the search tree split into different regions allocated to processors (e.g. one processor branches to the left, the other processor branches to the right). As it is unlikely those subtrees will be of equal size, a *work stealing* mechanism (see [12, 13]) is needed. Because it uses both communication and computation time, this cannot easily be scaled up to thousands of processors. In practice, we observe a decrease in performance when reaching a certain number of processors. However, interesting work was reported in [14]; the authors allocated specific processors to coordination tasks, allowing an increase in the number of processors that can be used before performance starts to decline.

Another promising approach is reported in [11]. The authors used a search space splitting mechanism allowing good load balancing without needing 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, which solves the load balancing problem. However, like previous approaches, leaves are globally visited 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 domain-oriented search strategy, a strategy that the parallel algorithm failed to exploit to its full potential.

2.3 Las Vegas algorithms / portfolios

This approach consists in allocating the same search space to each processor. Each processor explores it using a different strategy, leading to a different visiting order of the leaves. No communication is required and an excellent level of load balancing is achieved (they all search the same search space). Even if this approach causes a high level of redundancy between processors, the approach shows really good performance in practice. In [15] the approach was greatly improved by using randomized restart [16–18] on each processor.

As there is no communication between processors, this approach is fully scalable, although on small multi-core computers some authors increase the efficiency of the method by allowing processors to share information learned during the search (e.g. nogoods, see [19]).

In general, the main advantage of the algorithm portfolio approach is that one does not need to know a good search strategy beforehand: many strategies will be automatically tried at the same time by the parallel system, thanks to randomization. This is very useful because, as mentioned by [20] and [21], defining good domain-specific labelling strategies (that is, variable and value ordering heuristic) is a difficult task.

However, for complex applications where general strategies are inefficient and where very good domain-specific strategies are known (e.g. [4, 5]) one would like to have the parallel algorithm exploit the domain-specific strategy. In the next section, we will describe a classic backtracking algorithm (LDS) that is known to be efficient in centralized context when a good variable/value selection heuristic is provided. We will then propose in Section 4 a parallel implementation.

To the best of our knowledge, it is the first time that LDS is parallelized this way. In [14] LDS was used locally by processors to search in the trees allocated to them (by a tree splitting / work stealing algorithm) but the global system did not replicate an LDS strategy. The original centralized LDS being an iterative algorithm, Boivin [22] tried running the first k iterations at the same time on k processors. The approach did not prove to be efficient for the following reason: when LDS is provided with a good labelling strategy, the k^{th} iteration of LDS visit leaves that have considerably less expected probability of success than those in the first iterations. Therefore, for domain-specific problems where centralized LDS is known to be good, only the first few processors were really helpful in the parallel implementation. Moreover, they were experimenting load balancing problems.

Finally, LDS was adapted for distributed optimization in [23, 24]. However, distributed problems (DisCSP [25], DCOP [26] and HDCOP [27]) refers to a different context than parallel computing. These are problems that are distributed by nature; different agents are responsible for establishing the value of distinct variables and communication/coordination are inherent to those approaches. Therefore, the algorithm called MacDS we proposed in [23, 24] could not serve as a basis for a scalable parallel LDS algorithm.

The next section provides a comprehensive description of the centralized version of LDS that will be parallelized in Section 4.

3 LDS

The search space of a problem can be represented as a tree where each node corresponds to a partial assignment. The root is the empty partial assignment and the leaves are complete assignments (also called solutions). Each child has one more variable assigned than its parent. In their description of LDS, Harvey and Ginsberg [6] consider binary search trees, i.e. trees where each non-leaf node has two children. In this section we present a generalization of LDS to n-ary trees which includes a modification by Walsh [7] that prevents visiting a leaf more than once.

The value ordering heuristic is a function that orders the children of a node from the child that will most likely lead to a solution to the child that is less likely to lead to a solution. When represented graphically, the left-child is the most likely one to lead to a solution and the right-child is the less likely one to lead to a solution. A *discrepancy* is a deviation from the first choice of the heuristic. We say that the first choice of the heuristic has zero discrepancy, the second choice has one discrepancy, the third choice has two discrepancies and so on. The discrepancy of a node is the sum of the discrepancies associated to each choice on the path from the root of the tree to the node. Figure 1 shows a search tree where the number of discrepancies is shown for each node.

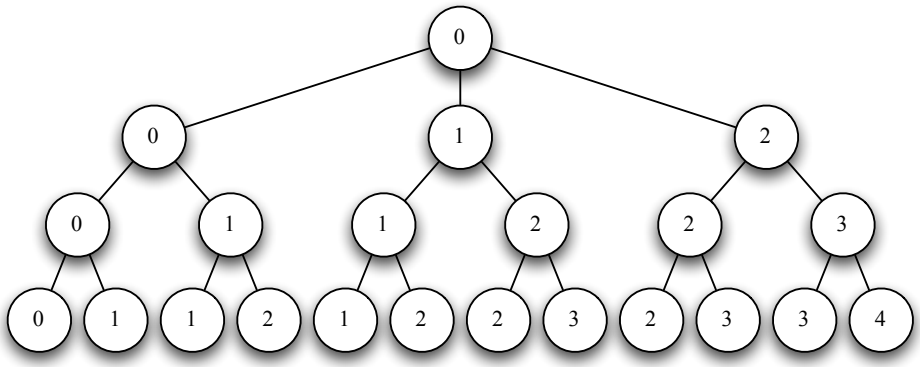

Fig. 1. Search tree. The discrepancy of each node is written inside the node.

Harvey and Ginsberg demonstrated that, with a good value ordering heuristic, the expected quality of a leaf decreases as the number of discrepancies increases. For that reason, they proposed to visit the leaves with the fewest discrepancies first and to keep the leaves with the most discrepancies for the end. Algorithm 2 visits all the leaves that have exactly k discrepancies. Algorithm 1 launches the search to visit all leaves in increasing number of discrepancies.

Algorithm 1: LDS($[\text{dom}(X_1), \ldots, \text{dom}(X_n)]$)

for $k = 0..n$ **do**
  $s \leftarrow$ LDS-Probe($[\text{dom}(X_1), \ldots, \text{dom}(X_n)], k$)
  if $s \neq \emptyset$ **then return** s
return $\emptyset$

Algorithm 2: LDS-Probe($[\text{dom}(X_1), \ldots, \text{dom}(X_n)]$, k)

```
Candidates ← {X_i | |dom(X_i)| > 1}
if Candidates = ∅ then
    if dom(X_1), ..., dom(X_n) satisfies all the constraints then
        return dom(X_1), ..., dom(X_n)
    return ∅
Choose a variable X_i ∈ Candidates
Let v_0, ..., v_{|dom(X_i)|-1} be the values in dom(X_i) sorted by preference of the
heuristic.
d ← max(0, k − Σ_{X_j ∈ Candidates \ {X_i}} (|dom(X_j)| − 1))
d̄ ← min(|dom(X_i)| − 1, k)
for d = d..d̄ do
    s ← LDS-Probe([dom(X_1), ..., dom(X_{i−1}), {v_d},
                   dom(X_{i+1}), ..., dom(X_n)], k − d)
    if s ≠ ∅ then return s
return ∅
```

Harvey and Ginsberg showed, by analyzing binary search trees from different problems, that the quality of a heuristic can be approximated/described by the probability p of finding a solution in the left subtree if no mistakes were made in the current partial assignment. Similarly, we say that the probability of finding a solution in the right subtree is q. If the solution is unique, we have $p + q = 1$. If there is more than one solution, we have $p + q \geq 1$ since there is a probability of having a solution both in the left subtree and the right subtree.

The better a heuristic is, the greater the ratio $\frac{p}{q}$ is. The probability that a leaf with k discrepancies is a solution is $p^{n-k}q^k$ since it involves branching k times on the right and $n - k$ times on the left. Actually, that probability decreases according to constant rate $\beta = -ln\frac{p}{q}$. Therefore, the probability is $p^{n-k}q^k = p^n e^{-\beta k}$. The parameter β characterizes the quality of the heuristic. The higher β is, the more likely the solutions will be concentrated in leaves having few discrepancies. In contrast, the extreme situation where $\beta = 0$ corresponds to a heuristic that does no better than random variable/value selection (all leaves share the same probability of being a solution, and using an LDS would not be a logical choice).

Finally, the *solution density* D of a problem is the ratio between the expected number of solutions and the total number of leaves. Equation 1 gives the solution density for binary search trees.

$$D = \frac{1}{2^n} \sum_{k=0}^{n} (p^n e^{-\beta k}) \binom{n}{k} \tag{1}$$

4 PDS

We want to run an LDS search over multiple processors. Parallelization can be achieved in multiple ways but we set four goals that will influence our choices.

1. **Search strategy preservation** We want the leaves of the search tree to be visited in the same order than they are on a single processor. Suppose that we mark each leaf of the tree with the time as it appears on a wall clock at the moment the leaf is visited. We assume that the clock is precise enough to break any ties. The ordering of the leaves by their visiting time should be the same regardless of the number of processors used.
2. **Workload balancing** We want the amount of work assigned to each processor to be evenly spread. This goal is particularly difficult to reach when the constraints filter the variable domains and make the search tree unbalanced.
3. **Robustness** We aim at running the search on a large cluster of computers. It is frequent on those computers that a processor fails for different reasons and that the program must be restarted on another processor. It must be possible to identify which part of the search tree was assigned to the failing processor so that we can reassign this part of the search tree to another processor.
4. **Minimizing the communication** We aim at minimizing the communication between the processors. We actually want to avoid any communication. We make no assumptions about the geographical location of the processors and their ability to communicate. Communication should be limited to the broadcast of a solution.

We define a variation of LDS that we call PDS. Given there are ρ processors, each processor is labelled with an integer between 0 and $\rho - 1$ called the *processor id*. There is exactly one process running on each processor. The number of processors ρ and the processor id are given as input to each process. These two parameters are sufficient to identify which nodes of the search tree will be explored by each process.

We label each leaf s of the search tree by its visit time $t(s)$ in a standard LDS run on one processor. The first leaf to be visited has a visit time of $t(s_0) = 0$, the second leaf has a visit time of 1 and so on. We assign each leaf to a processor in a round-robin way by assigning a leaf s to processor $t(s) \bmod \rho$. A processor j is only allowed to visit a leaf s that satisfies $t(s) \bmod \rho = j$ or an ancestor of such a leaf. Consequently, before branching on a child node, a processor j has to check whether this child leads to a leaf it can visit. We show how to perform this test.

Each call to Algorithm 4 corresponds to the visit of a node a in the search tree. The parameter k corresponds to the number of discrepancies that must lie on the path between the node a and the leaves. Let $C(X_1, \ldots, X_n, k)$ be the number of leaves that have exactly k discrepancies in a search tree formed by the variables $X_1, \ldots, X_n$. The function $C(X_1, \ldots, X_n, k)$ is recursively defined

as follows.

$$C(X_1,\ldots,X_n,k) = \begin{cases} 0 & \text{if } k < 0 \\ 1 & \text{if } k = 0 \\ \sum_{i=0}^{|\mathrm{dom}(X_n)|-1} C(X_1,\ldots,X_{n-1},k-i) & \text{otherwise} \end{cases} \quad (2)$$

When all domains have cardinality two, the recursion becomes $C(X_1,\ldots,X_n,k) = C(X_1,\ldots,X_{n-1},k) + C(X_1,\ldots,X_{n-1},k-1)$. This recursion is the same that appears in Pascal's triangle to compute the binomial coefficients. We therefore have $C(X_1,\ldots,X_n,k) = \binom{n}{k}$ when $|\mathrm{dom}(X_i)| = 2$. Intuitively, since each variable generates at most one discrepancy, the number of solutions with k discrepancies is the number of ways one can choose k variables among the n variables. When the domains have cardinalities greater than two, the recursion can be understood as follows: the variable X_n can generate a number of discrepancies i between 0 and $|\mathrm{dom}(X_n)|-1$. For each possible value of i, we count the number of solutions in the subtree of height $n-1$ that have exactly $k-i$ discrepancies.

By looking at equation (2), it seems that we consider a fixed ordering of the variables $X_1,\ldots,X_n$. However, the function returns the same value for any ordering. Consequently, the ordering used in Equation (2) does not need to be the same ordering as the one used by the search heuristic. In fact, the variable ordering imposed by the heuristic does not need to be static, but is required to be deterministic.

Consider a node a where a value is going to be assigned to X_n and none of the variables $X_1,\ldots,X_n$ are assigned. The node a has for children the nodes $c_0,\ldots,c_{|\mathrm{dom}(X_n)|-1}$. Let $l(a,k)$ be the processor assigned to the left-most leaf with k discrepancies in the subtree rooted at a. From this construction, we have $l(a,k) = l(c_0,k)$ since branching from a to c_0 adds no discrepancies to the partial assignment and that both expressions refer to the same leaf. There are $C(X_1,\ldots,X_{n-1},k)$ leaves with k discrepancies in the subtree rooted at c_0. Since each of these leaves are assigned to the processors in a round-robin way, the processor assigned to the first leaf in the subtree rooted at c_1 is therefore $(l(c_0,k) + C(X_1,\ldots,X_{n-1},k)) \bmod \rho$. The same reasoning applies for the other children leading to the following recursion.

$$l(c_i,k-i) = \begin{cases} l(a,k) & \text{if } i = 0 \\ (l(c_{i-1},k-i+1) + C(X_1,\ldots,X_{n-1},k-i+1)) \bmod \rho & \text{otherwise} \end{cases} \quad (3)$$

We now have all the tools to present how the search strategy PDS proceeds. Each processor visits only the nodes that lead to one of its assigned leaves. For each node a with children $c_0, c_1, \ldots$, Algorithm 4 computes which processor will treat the left-most leaf of the subtree rooted at c_i. This allows computing a range of processors that will visit each child. If the current processor is among that range, then it branches to the child.

Algorithm 3: PDS($[\text{dom}(X_1), \ldots, \text{dom}(X_n)]$)

$l \leftarrow 0$
for $k = 0..n$ **do**
  $s \leftarrow$ PDS-Probe($[\text{dom}(X_1), \ldots, \text{dom}(X_n)], k, l$)
  $l \leftarrow l + C(\{X_1, \ldots, X_n\}, k) \bmod \rho$
  if $s \neq \emptyset$ **then return** s
return $\emptyset$

Algorithm 4: PDS-Probe($[\text{dom}(X_1), \ldots, \text{dom}(X_n)]$, k, l)

$Candidates \leftarrow \{X_i \mid |\text{dom}(X_i)| > 1\}$
if $Candidates = \emptyset$ **then**
  if *dom*$(X_1), \ldots,$ *dom*(X_n) *satisfies all the constraints* **then**
    return $\text{dom}(X_1), \ldots, \text{dom}(X_n)$
  return $\emptyset$
Choose a variable $X_i \in Candidates$
Let $v_0, \ldots, v_{|\text{dom}(X_i)|-1}$ be the values in $\text{dom}(X_i)$ sorted by preference of the heuristic.
$\underline{d} \leftarrow \max(0, k - \sum_{X_j \in Candidates \setminus \{X_i\}} (|\text{dom}(X_j)| - 1))$
$\overline{d} \leftarrow \min(|\text{dom}(X_i)| - 1, k)$
for $d = \underline{d}..\overline{d}$ **do**
  $z \leftarrow C(Candidates \setminus \{X_i\}, k - d)$
  if $z < \rho$ **then**
    if $l < l + z \bmod \rho$ **then**
      $processors \leftarrow \{l, \ldots, l + z - 1 \bmod \rho\}$
    else
      $processors \leftarrow \{0, \ldots l + z - 1 \bmod \rho\} \cup \{l, \ldots, \rho - 1\}$
  else
    $processors \leftarrow \{0, \ldots, \rho - 1\}$ // All processors
  if $currentProcessor \in processors$ **then**
    $s \leftarrow$ PDS-Probe($[\text{dom}(X_1), \ldots, \text{dom}(X_{i-1}), \{v_d\},$
      $\text{dom}(X_{i+1}), \ldots, \text{dom}(X_n)], k - d, l$)
    if $s \neq \emptyset$ **then return** s
  $l \leftarrow l + z \bmod \rho$
return $\emptyset$

5 Analysis

This section provides an analysis of PDS in order to illustrate different properties of the algorithm. Section 4 showed how parallel cores can globally visit the leaves in the same order as the centralized algorithm would do. We now demonstrate the quality of the intrinsic workload balance that is achieved. First, when exploring the whole tree, the round-robin assignation of the processors insures that the difference between the number of leaves visited by two processors is at most one. Workload balancing is easy to achieve when considering complete search tree. However, it becomes harder to evenly divide the work among the processors when the tree is unbalanced. Search trees are often unbalanced when domain filtering and consistency technique are applied. We prove that when a value is filtered out of a variable domain and that a branch is cut from the tree, the workload is evenly reduced among all processors.

Theorem 1. *Let n be the number of variables in the problem. If a branch is cut from the search tree, the number of leaves removed from the workload of each processor differs by at most $n + 1$.*

Proof. The round-robin affection of the leaves with k discrepancies in a subtree guarantees that the number of leaves for each processor differs by at most one. Since we explore a subtree $n+1$ times for solutions with 0, 1, ..., n discrepancies, the difference of workload between the processors is at most $n + 1$ leaves. □

In addition to this, we provide empirical results showing that the performance of the algorithm never declines, even if we have more cores than there are leaves in the tree. This was done using a Monte Carlo approach reminiscent of what was done in [6] and [7].

We first generated very small trees for which we can easily have more cores than leaves (we chose $n = 5$, so we have $2^5 = 32$ leaves). A thousand trees were randomly generated for a given $\frac{p}{q}$ ratio (represented by the parameter β) and a given expected density D (we used "moderate" values for those parameters - we will provide results for other values later in this section). We then searched each tree using PDS, emulating the work of different numbers of cores and noted the "computation time" (number of nodes visited by the first core that finds a solution). [1]

Figure 2 shows the probability that a solution is found according to the computation time. For a given computation time, increasing the number of cores increases the probability that a solution has been found. With 32 cores, we have one processor per leaf. Passed that point, we have neither improvement nor deterioration. Figure 3 shows the same information in a different way.

It shows the average reduction of the computation time achieved when providing additional cores (in comparison with a single core).

[1] We didn't include pruning in these experimentations. However, theorem 1 shows that we can define an upper bound on the amount of work removed from each processor when pruning is done.

This clearly illustrates that we always increase the performance when we provide more cores until we reach the number of leaves. From that point there is no more gain. It also illustrates well that the speedup as a function of the number of processors is far from being linear. This is caused by redundancy between cores visiting the same nodes although reaching different leaves.

Figures 4 and 5 provide the results for the same experiment, but for trees with $n = 15$ variables. Although we did not execute the algorithm using 2^{15} cores, it shows similar results. For a given solution density, solutions tend to be concentrated on the leaves having few discrepancies.

As the expected quality of a leaf decreases exponentially with its number of discrepancies (recall Section 3), adding more cores makes us visit additional leaves in the same computation time, but those leaves have smaller probability of success. This is a natural (and desired) consequence of using a good variable/value selection heuristics and a backtracking strategy visiting leaves in order of expected quality.

The next experiment (Figures 6 and 7) illustrates that, for a given number of cores, the redundancy between cores decreases as the size of the problems (number of variables) increases.

The previous result tends to show that huge problems would greatly gain from this parallelization. The next experiment studies the performance of the algorithm according to the quality of the variable/value selection heuristics used. We recall from Section 3 that the higher the β ratio is, the more likely the solutions will be concentrated in leaves having few discrepancies. In contrast, the extreme situation where $\beta = 0$ simulates the use of a heuristic that does no better than random variable/value selection (all leaves share the same probability of being a solution, and using an LDS would not be a logical choice).

On Figure 8, the curve for 1 core shows that computation time decreases exponentially when β increases (the results we would get using the centralized version of LDS). Other curves show that when we provide additional cores, the computation time still decreases exponentially, but much more quickly.

Finally, the last experiment evaluates the algorithm according to problem density (Figures 9 and 10). The harder the problems are (low density), the more it pays to increase the number of cores. The easier they are (high density), the less useful it is to provide more cores.

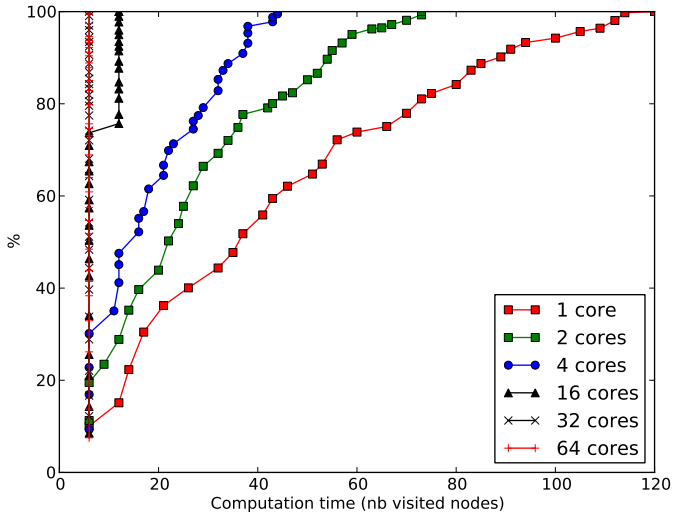

Fig. 2. Probability that a solution is found after some computation time [$n = 5$ vars; $\beta = 0.4$; $p^n = 0.04$; density $D = 0.0163$]

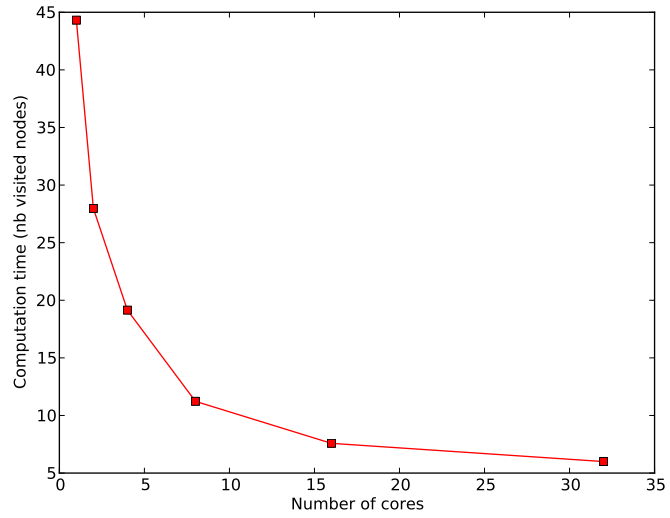

Fig. 3. Computation time according to the number of cores used [$n = 5$ vars; $\beta = 0.4$; $p^n = 0.04$; density $D = 0.0163$]

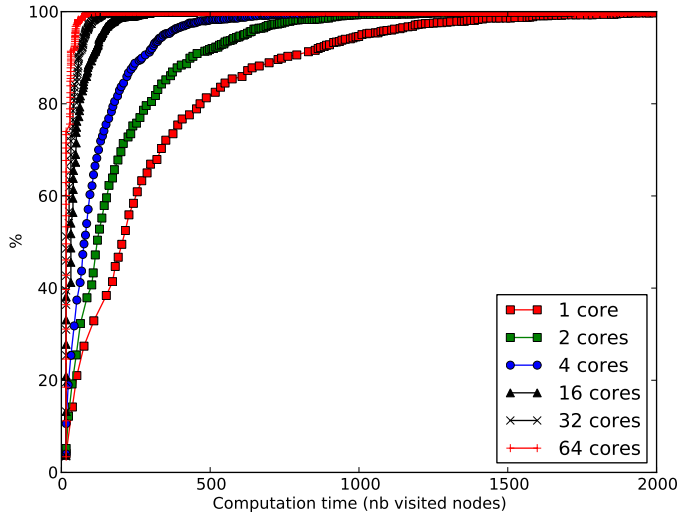

Fig. 4. Probability that a solution is found after some computation time [$n = 15$ vars; $\beta = 0.4$; $p^n = 0.04$; density $D = 0.0027$]

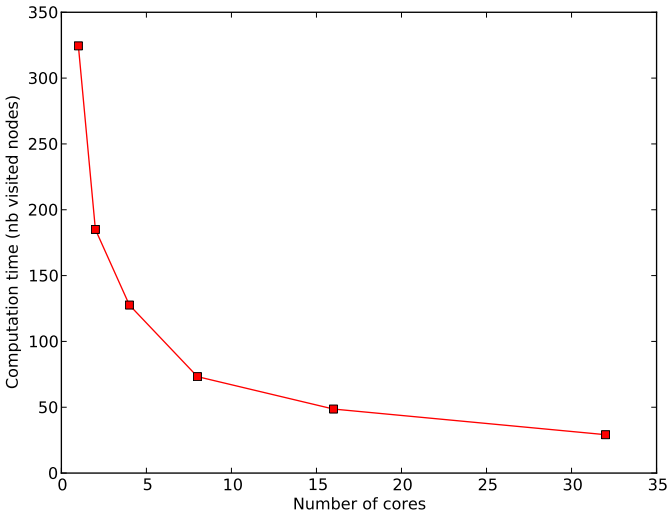

Fig. 5. Computation time according to the number of cores used [$n = 15$ vars; $\beta = 0.4$; $p^n = 0.04$; density $D = 0.0027$]

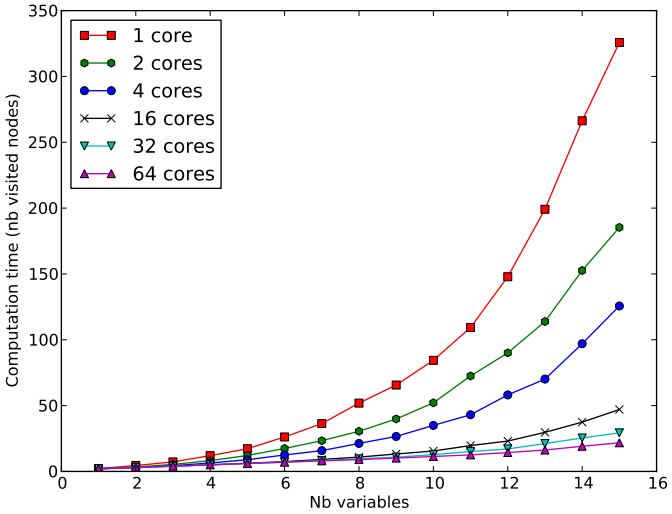

Fig. 6. Expected computation time according to the number of variables [$\beta = 0.4$; $p^n = 0.04$]

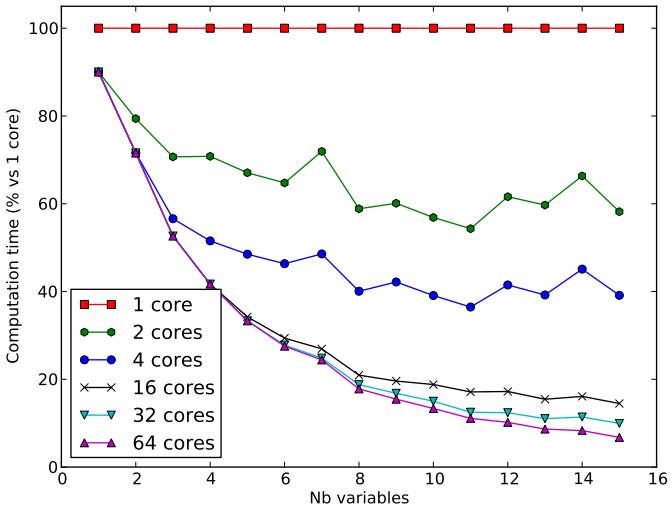

Fig. 7. Reduction of computation time when providing more cores, for different problem sizes [$\beta = 0.4$; $p^n = 0.04$]

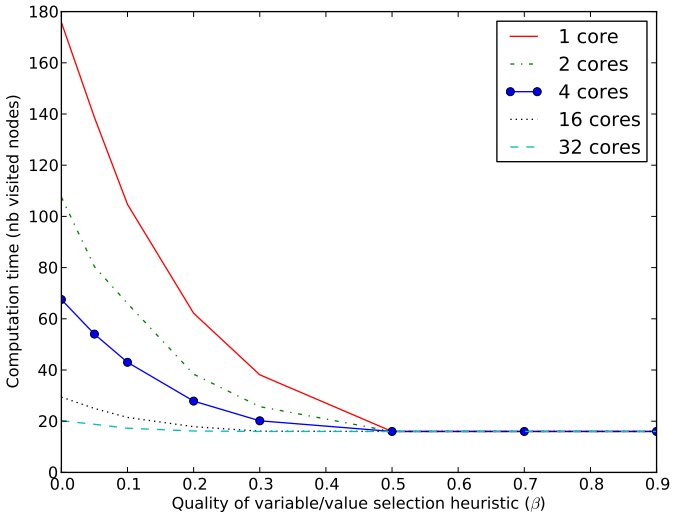

Fig. 8. Performance according to the quality of the variable/value selection heuristic [$n = 15$ vars; density $D = 0.0400$] The higher the β is, the more likely solutions will be concentrated on leaves with fewer discrepancies. In the extreme case where $\beta = 0$, this simulates the use of a heuristic that does no better than random selection (all leaves have the same probability of being a solution).

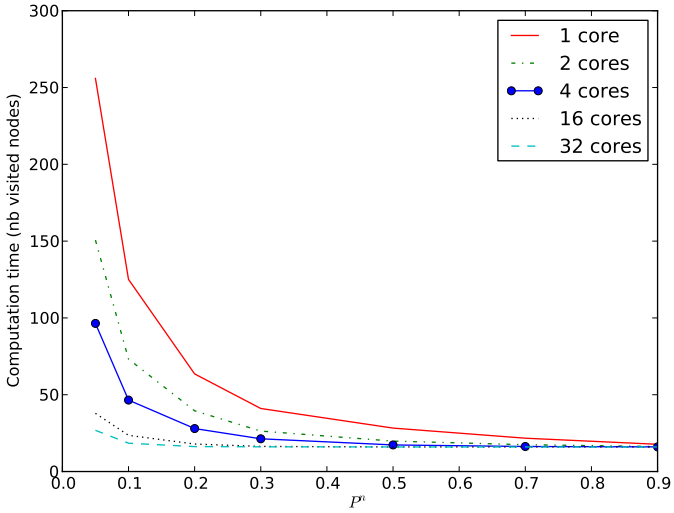

Fig. 9. Computation time according to solution density [$n = 15$ vars; $\beta = 0.4$] The p^n parameter goes from 0.05 to 0.9. This corresponds to trees with a solution density going from 0.0034 to 0.0604

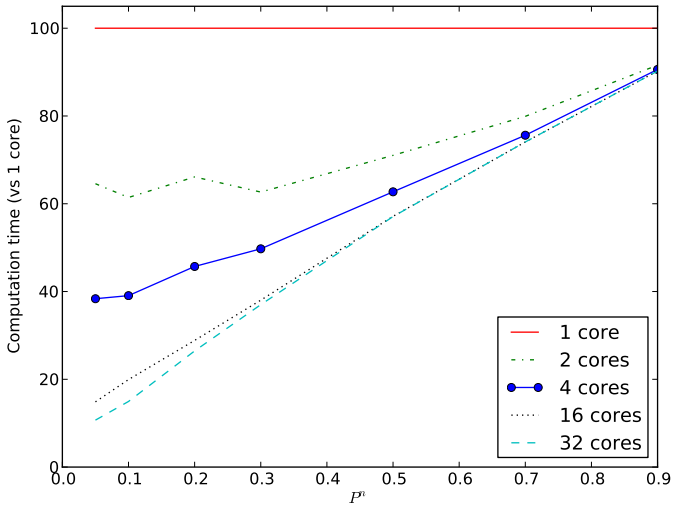

Fig. 10. Computation time according to solution density [$n = 15$ vars; $\beta = 0.4$] The p^n parameter goes from 0.05 to 0.9. This corresponds to trees with a solution density going from 0.0034 to 0.0604

6 Conclusion

We proposed a new parallelization scheme based on the LDS backtracking strategy. This parallelization does not alter the strategy since the order of visits of the nodes remains unchanged. Moreover, PDS provides an intrinsic workload balancing, it scales on multiple processors, and it is robust to hardware failures.

These properties make PDS a scalable algorithm that we intend to use to solve industrial problems for which excellent problem-specific variable/value selection heuristics are known.This is presently a work in progress and more developments are on their way. PDS will be implemented on a massively parallel supercomputer (7680 cores) in order to solve huge problems from the forest products industry.

References

1. Chabrier, A., Danna, E., Le Pape, C., Perron, L.: Solving a network design problem. Annals of Operations Research **130** (2004) 217–239
2. Le Pape, C., Perron, L., Régin, J.C., Shaw, P.: Robust and parallel solving of a network design problem. In: Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002). (2002) 633–648
3. Le Pape, C., Baptiste, P.: Heuristic control of a constraint-based algorithm for the preemptive job-shop scheduling problem. Journal of Heuristics **5** (1999) 305–325
4. Gaudreault, J., Forget, P., Frayret, J.M., Rousseau, A., Lemieux, S., D'Amours, S.: Distributed operations planning in the lumber supply chain: Models and coordination. International Journal of Industrial Engineering: Theory, Applications and Practice **17** (2010)
5. Gaudreault, J., Frayret, J.M., Rousseau, A., D'Amours, S.: Combined planning and scheduling in a divergent production system with co-production: A case study in the lumber industry. Computers Operations Research **38** (2011) 1238–1250
6. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 1995). (1995) 607–613
7. Walsh, T.: Depth-bounded discrepancy search. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 1997). (1997) 1388–1393
8. Beck, J.C., Perron, L.: Discrepancy-bounded depth first search. In: Proceedings of the Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2000). (2000) 8–10
9. Perron, L.: Search procedures and parallelism in constraint programming. In: Proceedings of Fifth International Conference on Principles and Practice of Constraint Programming (CP 1999). (1999) 346–360
10. Vidal, V., Bordeaux, L., Hamadi, Y.: Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In: Proceedings of the Third International Symposium on Combinatorial Search (SOCS 2010). (2010)
11. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI 2009). (2009) 443–448
12. Michel, L., See, A., Van Hentenryck, P.: Transparent parallelization of constraint programming. INFORMS J. on Computing **21** (2009) 363–382
13. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP 2009). (2009) 226–241
14. Xie, F., Davenport, A.: Massively parallel constraint programming for supercomputers: Challenges and initial results. In: The Seventh International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2010). (2010) 334–338
15. Shylo, O.V., Middelkoop, T., Pardalos, P.M.: Restart strategies in optimization: Parallel and serial cases. Parallel Computing **37** (2010) 60–68
16. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Information Processing Letters **47** (1993) 173–180

17. Gomes, C.P.: Boosting combinatorial search through randomization. In: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence (AAAI '98/IAAI '98). (1998) 431–437
18. Gomes, C.P.: Complete randomized backtrack search. In: Constraint and Integer Programming: Toward a Unified Methodology. (2003) 233–283
19. Hamadi, Y., Sais, L.: Manysat: a parallel sat solver. Journal on Satisfiability, Boolean Modeling and Computation **6** (2009) 245–262
20. Hamadi, Y., Ringwelski, G.: Boosting distributed constraint satisfaction. Journal of Heuristics (2010) 251–279
21. Puget, J.F.: Constraint programming next challenge: Simplicity of use. In: Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004). (2004) 5–8
22. Boivin, S., Gendron, B., Pesant, G.: Parallel constraint programming discrepancy-based search decomposition. Optimization days, Montréal, Canada (2007)
23. Gaudreault, J., Frayret, J.M., Pesant, G.: Discrepancy-based method for hierarchical distributed optimization. In: Nineteenth International Conference on Tools with Artificial Intelligence (ICTAI 2007). (2007) 75–81
24. Gaudreault, J., Frayret, J.M., Pesant, G.: Distributed search for supply chain coordination. Computers in Industry **60** (2009) 441–451
25. Yokoo, M.: Distributed constraint satisfaction: foundations of cooperation in multi-agent systems. Springer-Verlag, London, UK (2001)
26. Modi, P.J., Shen, W.M., Tambe, M., Yokoo, M.: Adopt: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence **161** (2006) 149–180
27. Gaudreault, J., Frayret, J.M., Pesant, G.: Discrepancy-based optimization for distributed supply chain operations planning. In: Proceeding of the Ninth International Workshop on Distributed Constraint Reasoning (DCR 2007). (2007)