Parallel Discrepancy-Based Search

Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper

FORAC research Consortium, Université Laval, Québec, Canada
Thierry.Moisan.1@ulaval.ca, Jonathan.Gaudreault@ift.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 that was used on a massively parallel supercomputer with thousands of cores. PDS improved the best known performance on an industrial problem.

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 good scaling: adding additional processors can never slow down the global process, unlike approaches using communication.

These properties make PDS a scalable algorithm that we used to solve industrial problems from the forest products industry (see [4, 5]) using a massively parallel supercomputer called Colosse deployed at Université Laval.

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 statistically the performance of the algorithm in order to illustrate different characteristics. Section 6 describes the experiments run on industrial problems and their results. Section 7 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 more processors to 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, it shows really good performance in practice. Shylo et al. [15] greatly improves the method 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.

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 kth iteration of LDS visit leaves that have considerably less expected probability of success than those in the first iterations. 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 experiencing 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, 27] 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

Harvey and Ginsberg [6] describe LDS with binary search trees, i.e. trees where each non-leaf node has two children. 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 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.

The value ordering heuristic is a function that orders the children of a node from the most likely one to lead to a solution to the least likely one. When represented graphically, the left child is the most likely one to lead to a solution and the right child is the least likely one. 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.

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.

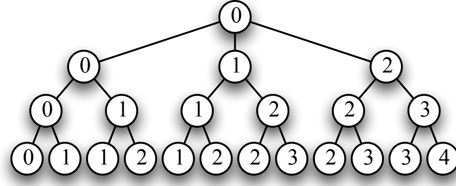

Fig. 1. Search tree. The discrepancy of each node is written inside the node.

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 as 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 must be reassigned 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. We label ρ processors with an integer between 0 and $\rho - 1$ called the *processor id*. There is exactly one process

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([dom(X_1), ..., 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 the heuristic.
d_ ← max(0, k − Σ_{X_a ∈ Candidates\{X_i}} (|dom(X_a)| − 1))
d¯ ← min(|dom(X_i)| − 1, k)
for d = d_..d¯ 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 ∅
```

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 centralized LDS. 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.

Let $C(X_1, \dots, X_n, k)$ be the number of leaves with exactly k discrepancies in a search tree formed by the variables $X_1, \dots, X_n$. The function $C(X_1, \dots, X_n, k)$ is recursively defined as follows.

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

When all domains have cardinality two, the recursion becomes $C(X_1, \dots, X_n, k) = C(X_1, \dots, X_{n-1}, k) + C(X_1, \dots, X_{n-1}, k - 1)$. This recursion is the same that appears in Pascal's triangle to compute the binomial coefficients. We therefore have $C(X_1, \dots, X_n, k) = \binom{n}{k}$ when $|\text{dom}(X_i)| = 2$. 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 $|\text{dom}(X_n)| - 1$. 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.

In equation (1), it seems that we consider a fixed ordering of the variables $X_1, \ldots, X_n$. However, 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 $c_0, \ldots, c_{|\text{dom}(X_n)|-1}$. 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.

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

We now have all the tools to present how the search strategy PDS proceeds. Each call to Algorithm 4 corresponds to the visit of a node in the search tree. The parameter k corresponds to the number of discrepancies that must lie on the path between this node and the leaves. 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.

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

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

```
l ← 0
for k = 0..n do
    Candidates ← {X_i | |dom(X_i)| > 1}
    z ← C(Candidates \ {X_i}, k)
    if (currentProcessor − l) mod ρ < z then
        s ← PDS-Probe([dom(X_1), ..., dom(X_n)], k, l)
        l ← l + C({X_1, ..., X_n}, k) mod ρ
        if s ≠ ∅ then return s
return ∅
```

Algorithm 4: PDS-Probe([dom(X_1), ..., dom(X_n)], k, l)

```
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 the heuristic.
d_ ← max(0, k − Σ_{X_j ∈ Candidates \ {X_i}} (|dom(X_j)| − 1))
d¯ ← min(|dom(X_i)| − 1, k)
for d = d_..d¯ do
    z ← C(Candidates \ {X_i}, k − d)
    if (currentProcessor − l) mod ρ < z then
        s ← PDS-Probe([dom(X_1), ..., dom(X_{i-1}), {v_d},
                       dom(X_{i+1}), ..., dom(X_n)], k − d, l)
        if s ≠ ∅ then  return s
    l ← l + z mod ρ
return ∅
```

the whole tree, the round-robin assignation of the processors ensures that the difference between the number of leaves visited by two processors is at most one. Workload balancing is easy to achieve when considering a 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. □

5.1 Overhead

We do an overhead comparison of PDS, LDS, and DFS by counting the number of times a node is visited in a complete search tree associated to n binary variables.

A DFS in a tree with n variables visits the root and performs a DFS on two subtrees of $n-1$ variables. Let DFS(n) be the number of visited nodes.

$$\mathrm{DFS}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2\mathrm{DFS}(n) + 1 & \text{if } n > 0 \end{cases} \quad (2)$$

This non-homogeneous linear recurrence of first order solves to $\text{DFS}(n) = 2^{n+1} - 1$, i.e. the number of nodes in a complete binary tree of height n.

We consider a PDS with ρ processors. Let $\text{PDSprobe}_\rho(n, k, j)$ be the number of nodes visited by processors $j \in \{0, 1, \ldots, \rho - 1\}$ on a tree with n binary variables for which we seek leaves of k discrepancies. We assume that the left-most leaf must be visited by processor 0. If the left-most leaf has to be visited by processor a, one can retrieve the number of visited nodes by relabeling the processors and computing $\text{PDSprobe}_\rho(n, k, j - a \bmod \rho)$. When $k \in \{0, n\}$, the tree has a unique leaf with k discrepancies and only the processor $j = 0$ visits the $n + 1$ nodes between the root and the leaf. If the number of leaves with k discrepancies, $\binom{n}{k}$, is smaller than or equal to j, then the processor j does not have to visit the tree. In all other cases, the number of visited nodes depends on the number of visited nodes in the left and right subtrees. We have the following recurrence.

$$\text{PDSprobe}_\rho(n, k, j) = \begin{cases} n + 1 & \text{if } j = 0 \wedge k \in \{0, n\} \\ 0 & \text{if } \binom{n}{k} \leq j \\ \text{PDSprobe}_\rho(n - 1, k, j) & \text{otherwise} \\ +\text{PDSprobe}_\rho(n - 1, k - 1, & \\ \qquad j - \binom{n-1}{k} \bmod \rho) + 1 & \end{cases}$$

Let $\text{PDS}_\rho(n)$ be the total number of nodes visited by the ρ processors.

$$\begin{aligned} \text{PDS}_\rho(n) =& \sum_{k=0}^{n} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n, k, j) = \sum_{k=1}^{n-1} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n, k, j) + 2(n + 1) \\ =& \sum_{k=1}^{n-1} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n - 1, k, j) \\ &+ \sum_{k=1}^{n-1} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n - 1, k - 1, j - \binom{n-1}{k} \bmod \rho) \\ &+ \sum_{k=1}^{n-1} \sum_{j=0}^{\min(\rho, \binom{n}{k})-1} 1 + 2(n + 1) \end{aligned}$$

One can replace $j - \binom{n-1}{k} \bmod \rho$ by j since we sum over $j = 0..\rho - 1$. We also perform a change of indices for k in the same summation.

$$\begin{aligned} \text{PDS}_\rho(n) =& \sum_{k=1}^{n-1} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n - 1, k, j) + \sum_{k=0}^{n-2} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n - 1, k, j) \\ &+ \sum_{k=1}^{n-1} \min(\rho, \binom{n}{k}) + 2n + 2 \\ =& 2\text{PDS}_\rho(n - 1) + \sum_{k=1}^{n-1} \min(\rho, \binom{n}{k}) + 2 \end{aligned}$$

Using backward substitutions solves the recurrence.

$$\mathrm{PDS}_\rho(n) = 2^n + 2^n \sum_{i=1}^{n} \sum_{k=0}^{i} \frac{1}{2^i} \min(\rho, \binom{i}{k})$$

When solved for $\rho = 1$, we retrieve the number of visited nodes with LDS. We also simplify for $\rho \in \{2, 3\}$ assuming $n \geq 3$.

$$\mathrm{LDS}(n) = 2^{n+2} - n - 3 \quad \mathrm{PDS}_2(n) = 5 \cdot 2^n - 2n - 4 \quad \mathrm{PDS}_3(n) = \frac{23}{4} 2^n - 3n - 5$$

We observe that, as the number of variables grows, a LDS visits twice the number of nodes than a DFS. Therefore, when DFS finishes to visit the entire tree, LDS visited half of the leaves. However, these leaves have fewer than $\frac{n}{2}$ discrepancies. So if the heuristic makes no mistakes at least half of the time, LDS finds a solution by the time DFS visits the entire tree. The overhead of LDS compared to DFS is therefore compensated by the search of more promising parts of the search tree.

As n grows, the ratios $\frac{\mathrm{PDS}_2(n)}{\mathrm{LDS}(n)}$ and $\frac{\mathrm{PDS}_3(n)}{\mathrm{LDS}(n)}$ tend to 1.25 and 1.43. These overheads of 25% and 43% grow slower than the number of processors and implies that 2 and 3 processors will visit the search tree in 62% and 48% of the time taken by one processor. Should the search visit the entire search tree, Figure 2 shows the speedup of PDS over LDS as the number of processors increases. We see that the speedup grows linearly except in the degenerate case where the number of leaves equal the number of processors.

To get a more accurate idea of the speedup, one needs to consider the quality of the solution (in an optimization problem) or the probability of finding a solution (in a satisfaction problem). This is done in the next section.

5.2 Statistical analysis

We provide statistical results showing that the performance of the algorithm never declines, except in the degenerated case where there are more processors than leaves. It is therefore a worst-case analysis where the entire tree is explored.

Harvey and Ginsberg [6] showed, by analyzing binary SCP 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 extreme situation where $\frac{p}{q} = 1$ 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). The probability that a leaf

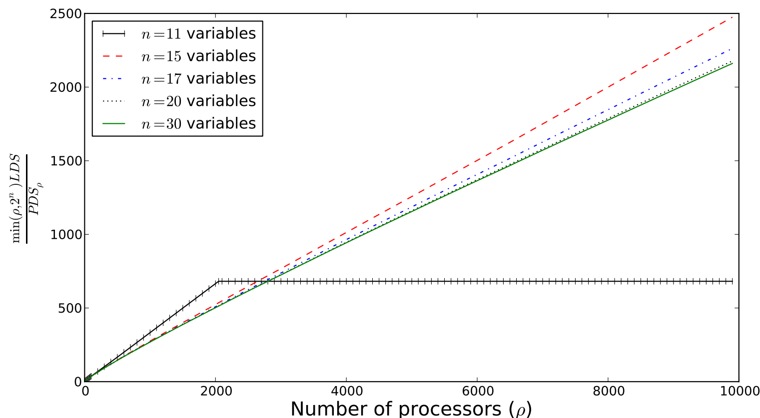

Fig. 2. Speedup for some number of processors

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.

Figure 3 shows the probability that a solution is found according to the number of visited nodes per processor. The probability that a leaf s_i with k deviations is a solution is $P(s_i) = p^{n-k}q^k$. The probability of finding a solution after visiting the leaves $s_1, \ldots, s_m$ is $1 - \prod_{i=1}^{m}(1 - P(s_i))$. For a given computation time, increasing the number of processors increases the probability that a solution has been found.

This clearly illustrates that increasing the number of processors increases the performance until ρ reaches the number of leaves in the search tree. From that point there is no more gain.

As the expected quality of a leaf decreases exponentially with its number of discrepancies (recall Section 3), adding more processors makes us visit additional leaves in the same computation time, but those leaves have smaller probability of success than the previous ones. 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.

Figure 4 presents the speedup obtained for some probability that a solution is found. A lot of variation is present for low probability values. This is due to the use of the heuristic that points toward good solutions quickly. The speedup then converges toward a single value as the probability that a solution is found increases.

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 $\frac{p}{q}$ ratio is, the more likely the solutions will be concentrated in leaves having few discrepancies. In contrast, the extreme situation where $p = q$ simulates the use of a heuristic that does no better than random

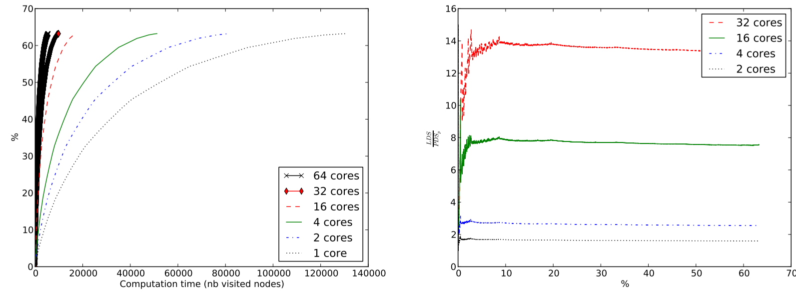

Fig. 3. Probability that a solution is found after some computation time [$n = 15$ vars; $p = 0.6$; $q = 0.4$]

Fig. 4. Speedup for some probability that a solution is found [$n = 15$ vars; $p = 0.6$; $q = 0.4$]

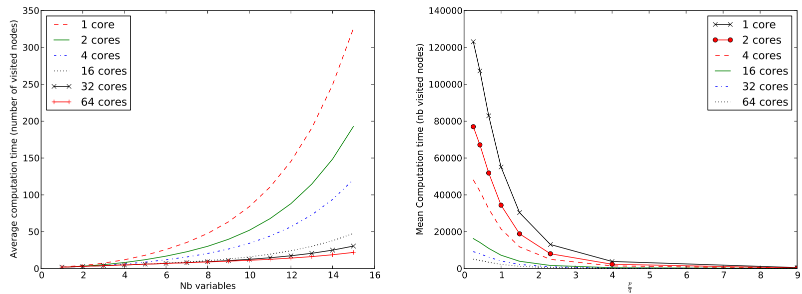

Fig. 5. Average computation time to find a solution according to the number of variables [$p = 0.6$; $q = 0.4$]

Fig. 6. Average computation time to find a solution according to the $\frac{p}{q}$ ratio where $p + q = 1$ with 15 variables.

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 6, the curve for 1 processor shows that computation time decreases exponentially when $\frac{p}{q}$ increases. Other curves show that when we provide additional processors, the computation time still decreases exponentially, but much more quickly.

6 Experimentation with industrial data

In a lumber finishing facility, lumbers are planed and sorted according to their grade (i.e. quality). It may be trimmed in order to produce a shorter lumber of a higher grade and value. The operation that improves a piece of lumber only depends on the piece of lumber itself with no consideration for the actual customer demand. This causes the production of multiple finished products at the same time (co-production) from a single raw product (divergence). This

makes the production very difficult to plan according to the customer demand. There is a finite set of processes that can be used to transform one raw product into many finished products. The plant can only process lumber of a single category in a given production shift. Mills prefer long campaigns of a single category as it reduces costs: once the mill is configured for a given setup, they want to stay in this configuration for as many shifts as possible. The plant maintains an inventory of raw and finished products. For each customer order, a given quantity of a finished product has to be delivered at a specific time.

To sum up, the decisions that must be taken in order to plan the finishing operations are the following: (1) select a lumber category to process during a campaign, (2) decide when the campaign starts and for how long it lasts, and (3) for each campaign, decide the quantities of each compatible products to process. It is a single machine planning and scheduling problem. Each planning period corresponds to one "production shift" (approximatively 4 hours). The objective is to minimize orders lateness (modelled as a penalty cost) and production costs. The problem is fully described in [4] which provides a good heuristic for this problem. In [5], the heuristic is used to guide the search using constraint programming (applying LDS) and it outperforms the DFS and the mathematical programming approach.

Industrial instances are huge and there is a need for good solutions in shorter computation time. The instances have 65,142 variables and 50,238 constraints. Among them, there are 42 discrete decision variables whose domains have cardinality 6 and 4200 continuous decision variables. As we have a really good branching heuristic for which LDS works really well, this problem is an ideal candidate for PDS. This search heuristic first branches on variable/values for the integer variables (decisions 1 and 2 in the previous paragraph). Once the values for these variables are known, the remaining continuous variables (3) define a linear program that can be easily solved to optimality using the simplex method. Therefore, each time we have a valid assignment of the integer variables, we consider we have reached a leaf and we solve a linear program to evaluate the value of this solution. This implies that the leaves have a heavier computation time than the inner nodes. This situation differs from Section 5.1 where all nodes have the same computation time.

We implemented PDS and ran it on Colosse, a supercomputer with more than 8000 cores (dual, quad-core Intel Nehalem CPUs, 2.8 GHz with 24 GB RAM). Two Canadian lumber companies involved in the project provided the industrial instances. The four datasets have from 30 to 42 production shifts, from 20 to 133 processes, from 60 to 308 customer orders, from 20 to 68 raw products, and from 60 to 222 finished products.

6.1 Results and discussion

Figures 7 to 10 show the objective value according to the computation time (maximum of one hour) for different numbers of processors. We also computed the best solution for LDS on these instances and the curve is indistinguishable from the PDS(1) curve. In Figure 10, we can see that a solution of quality of

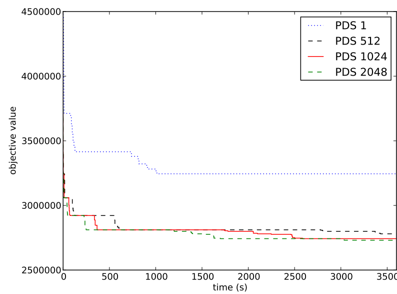

Fig. 7. Best solution found for K1 dataset

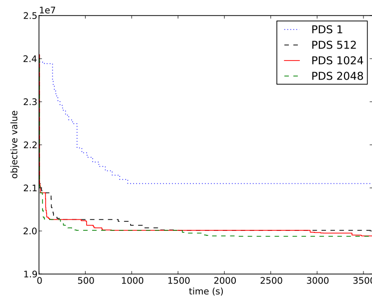

Fig. 8. Best solution found for M3 dataset

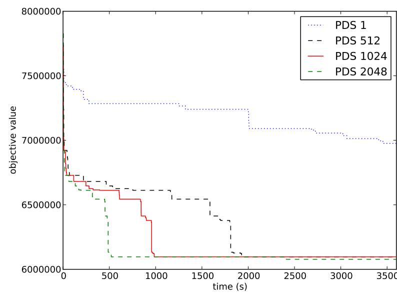

Fig. 9. Best solution found for M2 dataset

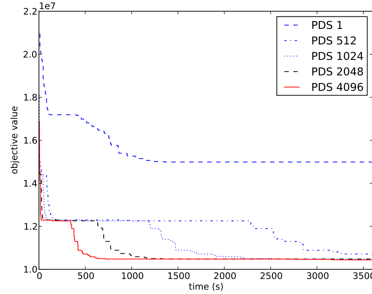

Fig. 10. Best solution found for M1 dataset

1.1×10^7 is not found with 1 processor even after an hour but can be found in 10 minutes with 4096 processors. Furthermore, 1 processor obtained a solution of 1.5×10^7 in one hour while the same solution is found in a few seconds with 512 processors. This is a major improvement from an industrial point of view where computation time is the real constraint.

The harder instances are those where the heuristic has more difficulties and a good solution is obtained later in the search (Figure 7 is the easiest instance, Figure 10 is the hardest one). The absolute time saving is greater on harder instances when using PDS.

For each Figure from 7 to 10, the curves for $512, 1024$, and 2048 processors have the same shape but get more compressed over time as the number of processors increases. This shows that the heuristic and the search strategies remains the same even in its parallelized version.

Table 1 lists statistics we computed during these experiments. The speedup is the ratio of the number of leaves visited by multiple processors over the number of leaves visited by one processor. PDS scales well: even with 4096 processors, the speedup is still increasing almost linearly.

One hour was insufficient to visit the entire search tree. With 4096 processors, we reached solutions with 6 discrepancies but did not visit all of them. Therefore,

dataSet	ρ	speedup	$\overline{\chi}$	$max(\chi) - min(\chi)$	$\frac{\overline{\chi}}{min(\chi)}$ (%)
K1	512	338.9	4668.6	550	8.6
K1	1024	585.9	4036.0	441	7.97
K1	2048	941.9	3244.4	363	7.79
M3	512	446.4	725.4	24	1.89
M3	1024	863.1	701.3	57	6.09
M3	2048	1601.1	650.4	25	2.92
M2	512	432.7	920.3	53	3.98
M2	1024	823.2	875.5	40	3.49
M2	2048	1604.7	853.3	23	1.94
M1	512	447.7	729.3	42	4.03
M1	1024	869.1	707.9	42	4.25
M1	2048	1656.1	674.4	79	11.1
M1	4096	3152.3	641.9	57	6.44

Table 1. Industrial datasets experiments statistics. The column $\overline{\chi}$ is the average number of leaves visited by each processor. The column $max(\chi) - min(\chi)$ is the maximum difference of leaves treated between processors. The column $\frac{\overline{\chi}}{min(\chi)}$ is the average percentage of leaves differences between each processor and the minimal number of leaves treated by one processor.

there is no idle time. However, we want to measure how the workload, in terms of visited leaves, differs between processors. Let χ_j be the number of leaves processed by processor j. Let $min(\chi)$ be the minimum value of χ_j for every $j \in 0, 1, \ldots, \rho-1$. Let $\overline{\chi}$ be the average number of leaves visited by each processor. The relative difference between $min(\chi)$ and $\overline{\chi}$ is $\frac{\overline{\chi}}{min(\chi)}$. This measure shows processors have visited roughly the same number of leaves.

We had hardware failures during the experiments and we have been able to restart a single processor while leaving the other ones running.

7 Conclusion

The contributions of this paper are twofold. First, we proposed a new parallelization scheme based on the LDS backtracking strategy. This parallelization does not alter the strategy since the visit order of the nodes remains unchanged. Moreover, PDS provides an intrinsic workload balancing, it scales on multiple processors, and it is robust to hardware failures. We provided a theoretical analysis that evaluated the performance of PDS based on the quality of the heuristic. This showed that adding more processors always provides a speedup.

Secondly, we experimented with a difficult industrial problem from the forest products industry for which an excellent problem-specific variable/value selection heuristics is known. This has been done by using as many as 4096 processors on a supercomputer. It shows the great potential of constraint programming in a massively parallel environment for which good search strategies are known.

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)