Elsevier Editorial System(tm) for Artificial Intelligence
Manuscript Draft

Manuscript Number: ARTINT-D-14-00105

Title: PDS: A scalable, communication-free search strategy for massively parallel supercomputers that provides intrinsic load-balancing

Article Type: Fast Track Invited Paper

Keywords: search strategy; discrepancy; parallel computing; constraint programming

Corresponding Author: Mr. Claude-Guy Quimper,

Corresponding Author's Institution:

First Author: Thierry Moisan

Order of Authors: Thierry Moisan; Claude-Guy Quimper; Jonathan Gaudreault

Abstract: Backtracking strategies based on the computation of discrepancies, such as Limited Discrepancy Search (LDS) and Depth-bounded Discrepancy Search (DDS), have proved themselves successful at solving large problems. We propose a parallelization approach called Parallel Discrepancy-based Search (PDS). The implicit round-robin assignment of leaves to the workers allows them to operate without communication during the search. It provides an intrinsic load-balancing mechanism: pruning a branch of the search tree affects each worker's workload in a limited way. We show how the PDS approach can be used to parallelize backtracking algorithms based on the computation of discrepancies (LDS, DDS) as well as Depth-First Search (DFS). The resulting algorithms visit the nodes of the search tree in the same order as their centralized version. We show that these parallel algorithms scale to multiple thousands of workers. We present a theoretical analysis of these algorithms and experiment on a massively parallel supercomputer to solve industrial problems and improve over the best known solutions.

Opposed Reviewers:

PDS: A scalable, communication-free search strategy for massively parallel supercomputers that provides intrinsic load-balancing ☆

Thierry Moisan, Claude-Guy Quimper*, Jonathan Gaudreault

FORAC Research Consortium, Département d'informatique et de génie logiciel, Université Laval, Québec, Canada

Abstract

Backtracking strategies based on the computation of discrepancies, such as Limited Discrepancy Search (LDS) and Depth-bounded Discrepancy Search (DDS), have proved themselves successful at solving large problems. We propose a parallelization approach called Parallel Discrepancy-based Search (PDS). The implicit round-robin assignment of leaves to the workers allows them to operate without communication during the search. It provides an intrinsic load-balancing mechanism: pruning a branch of the search tree affects each worker's workload in a limited way. We show how the PDS approach can be used to parallelize backtracking algorithms based on the computation of discrepancies (LDS, DDS) as well as Depth-First Search (DFS). The resulting algorithms visit the nodes of the search tree in the same order as their centralized version. We show that these parallel algorithms scale to multiple thousands of workers. We present a theoretical analysis of these algorithms and experiment on a massively parallel supercomputer to solve industrial problems and improve over the best known solutions.

Keywords: Search Strategy, Discrepancy, Parallel computing, Constraint Programming

☆This is an invited extended version of a paper entitled *Parallel Discrepancy-based Search* which was awarded the Best Paper award at the CP-2013 conference. It also includes some results from the paper *Parallel Depth-bounded Discrepancy Search* which was presented at the CPAIOR-2014 conference.

*Corresponding author

Email addresses: Thierry.Moisan.1@ulaval.ca (Thierry Moisan), Claude-Guy.Quimper@ift.ulaval.ca (Claude-Guy Quimper), Jonathan.Gaudreault@forac.ulaval.ca (Jonathan Gaudreault)

1. Introduction

Constraint solvers have been used for decades and were successful at solving numerous operations research problems. For instance, they 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]. Such 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 performs a backtracking search in a tree to explore the solution space. With the rise of multi-core servers, there is an increase in research for parallelizing constraint solvers. Parallelization is not trivial as there is a need for a trade-off between the workload balance, the communication cost, and the duplication (redundancy) of work between the workers.

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 computation of discrepancies such as Limited Discrepancy Search (LDS) [6] and Depth-bounded Discrepancy Search (DDS) [7] have proved themselves successful at solving large problems. They show good performances 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 (called PDS), based on an implicit round-robin assignment of leaves to workers, that allows parallelizing search strategies. The proposed approach shows the following characteristics:

- The pool of workers globally visits the leaves in exactly the same order as the centralized version.
- There is no need for communication between the workers.
- The implementation allows for a natural/intrinsic load balancing to occur: filtering induced by constraint propagation affects each worker pretty much in the same way: the workload balance difference can be theoretically bounded.
- The method provides robustness: if a worker dies, it can be replaced by a new one. However, it must restart the work previously allocated to the dead worker.

- It offers good scalability: adding additional workers can never slow down the global process, unlike approaches using communication.

The remainder of this paper is organized as follows. Section 2 reviews basic concepts related to parallel tree search and describes the original centralized algorithms that are later parallelized. Section 3 describes our parallelization scheme and applies it to obtain parallel versions of LDS, DDS, as well as DFS. Section 4 reports theoretical analysis and statistically evaluates the performance of the algorithms in order to illustrate their different characteristics. In Section 5, the algorithms are used to solve industrial problems from the forest-products industry using a massively parallel supercomputer called Colosse deployed at Université Laval (several thousand cores). Section 5 describes the experiments run on industrial problems and their results. Section 6 concludes the paper.

2. Literature review

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

2.1. Search space in shared memory

The simplest method for parallel tree search is implemented with many cores sharing a list of open nodes (nodes for which there is at least one child that is still unvisited). Starved workers just pick up the most promising node in the list and expand it. By defining different node evaluation functions, one can implement different strategies (Depth-First Search, Breadth-First Search and others). Perron [9] proposes a comprehensive framework based on this idea. Good performances are often reported, as in [10] where a parallel Best-First Search was implemented, and evaluated up to 64 workers.

Although this kind of mechanism intrinsically provides excellent load balancing, it is known not to scale beyond a certain number of workers; 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. Portfolios

This approach consists in allocating the same search space to each worker. Each worker explores it using a different set of solvers, parameters and/or search strategies, 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 workers, it shows really good performance in practice. Shylo et al. [11] greatly improves the method using randomized restarts [12, 13, 14] on each worker.

As there is no communication between workers, this approach is fully scalable, although on small multi-core computers some authors increase the efficiency of the method by allowing workers to share information learned during the search (e.g. nogoods, see [15]).

Finding good alternative configurations for a specific problem can be a difficult problem by itself. Xu et al. [16] use machine learning to find appropriate SAT solver configurations to a new problem based on a set of learned examples.

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. This is very useful because, as mentioned by [17] and [18], defining a good domain-specific branching heuristic (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.

2.3. Search space splitting and work stealing

Search space splitting and work stealing approaches are often reported as the most frequently seen in the literature [19]. The main idea is to split the search tree into different regions allocated to workers (e.g. one worker branches to the left, the other worker branches to the right).

As it is unlikely that those subtrees are of equal size, a *work stealing* mechanism (see [20, 21]) is needed. Menouer et al. [22] parallelize the constraint programming solver

OR-Tools using a framework based on work-stealing. Because it uses both communication and computation time, these approaches cannot easily be scaled up to thousands of workers. At some point, the communication monopolizes the majority of the computing power. In this case, reducing the amount of communication speeds up the search. Interesting work was reported in [23]; the authors allocated specific workers to coordinate the tasks, allowing more workers to be used before performance starts to decline.

Yun and Epstein [24] combined the use of portfolios with work-stealing. They start by launching a portfolio phase by making a choice of solver configuration. Then, the search space is divided and work is distributed among the workers. During the search, information about the success (or lack thereof) is transmitted from the workers to the manager inducing a change in the future choices among the portfolio of solvers.

Recent work showed how to implicitly balance the workload while minimizing the communication during the search. Bordeaux et al. [19] report a promising approach that uses a search space splitting mechanism allowing good load balancing without needing work stealing. They use a hashing function that implicitly allocates the leaves to the workers. Each worker 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-worker system. This prevents replicating the performance of a a good domain-oriented search strategy over multiple workers.

Régin et al. [25] split the problem into a large number of subproblems. Some are quicker to explore their search space than others, as pruning occurring during the search does not affect each subproblem equally. However, since a large number of subtrees is assigned to each worker, their workload tends to balance out. The exclusion of any communication during the search is also the solution we advocate in this paper.

2.4. Discrepancy-based search strategies

This section provides a comprehensive description of discrepancy-based search strategies such as LDS and DDS. These are the strategies that will be parallelized in Sections 3.2 and 3.3.

Harvey and Ginsberg [6] introduce the concept of *discrepancy*. 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 assignment and the leaves are complete assignments (also called solutions). Each child has one more variable assigned than its parent. Each time a solver needs to assign a value to a variable, a value-ordering heuristic is used to select the value that will most likely lead to a solution. As a convention, when a binary search tree is graphically represented, the left branch under a node corresponds to the recommendation of the value ordering heuristic while the right one does not. In the case where the variables have a cardinality greater than two, the values are sorted based on their likeliness to lead to a solution. They are then represented from left to right based on this ordering, the leftmost being the recommendation of the value ordering heuristic. Figure 1 shows such a search tree. Each time the solver does not branch to the left in this tree, it goes against the value-ordering heuristic recommendation. Such branching is called a *discrepancy*. Leaves on Figure 1 are labeled with the total number of discrepancies one must follow to go from the root of the tree to that leaf.

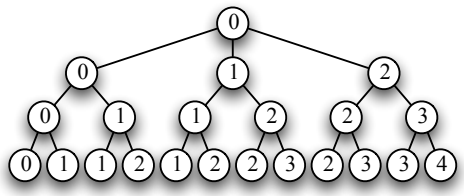


Figure 1: A binary search tree with the number of discrepancies of each node.

2.4.1. Limited Discrepancy-based Search (LDS)

Harvey and Ginsberg [6] describe Limited Discrepancy-based Search (LDS) for binary search trees. The authors 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 propose visiting the leaves with the fewest discrepancies first and keeping the leaves with the most discrepancies for the end.

We present a generalization of LDS to n-ary trees which includes a modification by Korf [26] that prevents visiting a leaf more than once. Branching on the n^{th} value of the variable, ordered by the value-ordering heuristic, adds $n - 1$ discrepancies. The discrepancy of a node is still the sum of the discrepancies in the path leading to this node.

Algorithm 1 launches probes to visit all leaves in increasing number of discrepancies. Algorithm 2 visits all the leaves that have exactly k discrepancies.

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

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

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

$Candidates \leftarrow \{X_i \mid |\mathrm{dom}(X_i)| > 1\}$
if $Candidates = \emptyset$ **then**
  if $\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_n)$ *satisfies all the constraints* **then**
    return $\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_n)$
  return $\emptyset$
Choose a variable $X_i \in Candidates$
Let $v_0, \ldots, v_{|\mathrm{dom}(X_i)|-1}$ be the values in $\mathrm{dom}(X_i)$ sorted by the heuristic.
$\underline{d} \leftarrow \max(0, k - \sum_{X_a \in Candidates \setminus \{X_i\}} (|\mathrm{dom}(X_a)| - 1))$
$\overline{d} \leftarrow \min(|\mathrm{dom}(X_i)| - 1, k)$
for $d = \underline{d}..\overline{d}$ **do**
  $s \leftarrow$ LDS-Probe($[\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_{i-1}), \{v_d\},$
    $\mathrm{dom}(X_{i+1}), \ldots, \mathrm{dom}(X_n)], k - d$)
  if $s \neq \emptyset$ **then return** s
return $\emptyset$

2.4.2. Depth-bounded Discrepancy Search (DDS)

Depth-bounded Discrepancy Search (DDS) [7] makes the following assumption: it is more probable that the value ordering heuristic will make a mistake at the top of search tree than at the bottom. A value-ordering heuristic can make better decisions lower in

the search tree since it has more information about the problem. Hence, it is more likely that the heuristic will make a mistake at top the of the tree. Based on this assumption, if the search has to reconsider the choices it made, it is better to reconsider choices made at the top of the search tree rather than at the bottom.

Given a search tree of depth n, DDS performs $n+1$ iterations. At iteration $k=0$, DDS visits the leftmost leaf of the tree. At iteration k, for $1 \leq k \leq n$, DDS visits all the branches in the search tree above level $k-1$. At level k, the algorithm visits all value assignments that do not respect the value ordering heuristic. Beyond level k, DDS visits all value assignments that respect the value ordering heuristic and therefore have no discrepancies. For example, for $k=2$, DDS visits all nodes down to level 1, branches once on a value that does not respect the heuristic, and then always branches on the left-most child until it reaches a leaf.

Algorithms 3 and 4 are a generalization of the original DDS algorithm [7] for n-ary variables. In the following description, we suppose that the variable ordering heuristic is deterministic and only depends on the variable domains. Hence, under the same conditions, the algorithm will always make the same choices (otherwise, the variable-ordering heuristics could cause the search strategy to visit some leaves multiple times and ignore other leaves). This supposition was also made in [6] and [7].

There are other strategies based on the computation of discrepancies such as Discrepancy-Bounded Depth First Search (DBDFS) [8] and Limited Discrepancy Beam Search (BULB) [27].

2.4.3. Discrepancies in distributed and parallel context

In this section we describe various applications of discrepancies in distributed and parallel computing.

In [23] LDS was used locally by workers to search 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 et al. [28] ran the first k iterations at the same time on k workers. The approach did not prove to be efficient for the following reason: when LDS is provided with a good value selection heuristic, the k^{th} iteration of LDS visits 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 workers were really helpful in the parallel implementation. Moreover, they were experiencing load balancing problems.

LDS was adapted for distributed optimization in [29, 30]. However, distributed problems (DisCSP [31], DCOP [32] and HDCOP [33]) refers to a different context than parallel computing. These problems 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 proposed in [29, 30] cannot serve as a basis for a scalable parallel LDS algorithm.

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

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

3. Parallel discrepancy-based search (PDS): implicitly assigning leaves to workers

This section introduces PDS, a general scheme for the parallelization of deterministic search strategies. Parallelization can be achieved in multiple ways but we set four goals to guide our approach.

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 worker. 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 workers used.
2. **Workload balancing** We want the amount of work assigned to each worker to be evenly spread. This goal is particularly difficult to reach when the constraints filter the variable domains and make the search tree unbalanced.

Algorithm 4: DDS-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 the heuristic.
if k = 1 then d̲ ← 1 else d̲ ← 0
if k = 0 then d̄ ← 0 else d̄ ← |dom(x_i)| − 1
for d = d̲..d̄ do
    s ← DDS-Probe([dom(X_1), ..., dom(X_{i-1}), {v_d},
                   dom(X_{i+1}), ..., dom(X_n)], max(0, k − 1))
    if s ≠ ∅ then return s
return ∅
```

3. **Robustness** We aim at running the search on a large cluster of computers. On those computers, it is frequent that a worker fails for different reasons and that the program must be restarted on another worker. It must be possible to identify which part of the search tree must be reassigned to another worker.
4. **Minimizing the communication** We aim at minimizing the communication between the workers. We actually want to avoid any communication. We make no assumptions about the geographical location of the workers and their ability to communicate. Communication should be limited to the broadcast of a solution.

In the remainder of this section, we describe the general parallelization scheme. Then, we apply it in order to produce parallel versions of DFS, LDS and DDS.

The parallelization works as follows. We label ρ workers with an integer between 0 and $\rho - 1$ called the *worker id*. There is exactly one process running on each worker. The number of workers ρ and the worker 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 order $t(s)$ in the corresponding centralized search strategy. The first leaf to be visited has a visit order of 0, the second leaf has a visit order of 1 and so on. We assign each leaf to a worker in a round-robin way by assigning a leaf s to worker $t(s) \bmod \rho$. A worker 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 worker j has to check whether this child leads to a leaf it can visit. We show how to perform this test.

Let $C([X_1, \ldots, X_n])$ be the number of leaves in a search tree formed by the variables $X_1, \ldots, X_n$. Consider a node a where a value is going to be assigned to X_n when none of the variables $X_1, \ldots, X_n$ are assigned. The node a has for children the nodes $c_0, \ldots, c_{|\operatorname{dom}(X_n)|-1}$. Let $l(a)$ be the worker assigned to the leftmost leaf in the subtree rooted at a. From this construction, we have $l(a) = l(c_0)$.

There are $C([X_1, \ldots, X_{n-1}])$ leaves in the subtree rooted at c_0. Since each of these leaves are assigned to the workers in a round-robin way, the worker assigned to the first leaf in the subtree rooted at c_1 is therefore $(l(c_0) + C([X_1, \ldots, X_{n-1}])) \bmod \rho$.

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

During the search, for each node a with children $c_0, c_1, \ldots$, one can compute the id $l(c_i)$ of the worker that will treat the left-most leaf of the subtree rooted at c_i. This allows computing the range of workers that visit the child c_i, i.e. $l(c_i)..(l(c_i)+C([X_1, \ldots, X_{n-1}])-1) \bmod C$. If the current worker j is among that range, that is if $(j - l(c_i)) \bmod \rho < C([X_1, \ldots, X_n])$ then it branches to the child c_i.

This approach creates a parallelization that is robust to hardware failure. If such failure occurs during the experiments, it is possible to restart a single worker while leaving the other ones running.

3.1. Parallel Depth-First Search (PDFS)

We show in this section how this mechanism can be applied to a Depth-First Search (DFS) which becomes a Parallel Depth-First Search (PDFS).

Let $C_{\text{DFS}}([X_1, \ldots, X_n])$ be the number of leaves visited by a DFS in a search tree defined by the variables $X_1, \ldots, X_n$.

$$C_{\text{DFS}}([X_1, \ldots, X_n]) = \prod_{i=1}^{n} |\operatorname{dom}(X_i)| \tag{1}$$

If all variable domains have cardinality δ, Equation 1 simplifies to Equation 2.

$$C_{\text{DFS}}([X_1, \ldots, X_n]) = \delta^n \tag{2}$$

Algorithm 5: PDFS($[\operatorname{dom}(X_1), \ldots, \operatorname{dom}(X_n)]$, l)

```
Candidates ← {X_i | |dom(X_i)| > 1}
Choose a variable X_i ∈ Candidates
z ← C_DFS(Candidates \ {X_i})
for v_d ∈ dom(X_i) do
    if (worker_id − l) mod ρ < z then
        s ← PDFS([dom(X_1), ..., dom(X_{i−1}), {v_d}, dom(X_{i+1}) ..., dom(X_n)], l)
        if s ≠ ∅ then return s
    l ← (l + z) mod ρ
return ∅
```

Algorithm 5 describes PDFS. The first call to PDFS is done with the original variable domains and $l = 0$.

3.2. Parallel Limited Discrepancy Search (PLDS)

We want to execute a LDS search over multiple workers using the same parallelization scheme. We name the resulting parallel algorithm Parallel Limited Discrepancy Search (PLDS).

Let $C_{\text{LDS}}([X_1, \ldots, X_n], k)$ be the number of leaves with exactly k discrepancies in a search tree formed by the variables $X_1, \ldots, X_n$. The function $C_{\text{LDS}}([X_1, \ldots, X_n], k)$ is recursively defined as follows.

$$C_{\text{LDS}}([X_1, \ldots, X_n], k) = \begin{cases} 0 & \text{if } k < 0 \\ 1 & \text{if } k = 0 \\ \sum_{i=0}^{|\operatorname{dom}(X_n)|-1} C_{\text{LDS}}([X_1, \ldots, X_{n-1}], k-i) & \text{otherwise} \end{cases} \tag{3}$$

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

In equation (3), 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.

Algorithm 6: PLDS($[\operatorname{dom}(X_1), \ldots, \operatorname{dom}(X_n)]$)

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

We now have all the tools to present how PLDS proceeds. Algorithm 6 launches each iteration k of the search where all leaves with k discrepancies are visited. Each call to Algorithm 7 corresponds to the visit of a node in the search tree.

3.3. Parallel Depth-bounded Discrepancy Search (PDDS)

As for PLDS and PDFS, the parallelization of DDS is achieved by assigning the leaves of the search tree to each worker in a round-robin fashion.

Algorithms 8 and 9 show how PDDS operates. Algorithm 8 visits all the leaves for which the last discrepancy is on the k^{th} branching. Algorithm 9 launches an iteration to visit all those leaves following a DFS search.

As for PLDS and PDFS the algorithm requires a function C_{DDS} that counts the number of leaves under the current node that should be visited during the current iteration

Algorithm 7: PLDS-Probe($[\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_n)]$, 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))
d̄ ← min(|dom(X_i)| − 1, k)
for d = d..d̄ do
    z ← C_LDS(Candidates \ {X_i}, k − d)
    if (worker_id − l) mod ρ < z then
        s ← PLDS-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 ∅
```

of DDS. The next subsection shows how to implement this function.

3.3.1. PDDS counting functions

To count the number of leaves in a subtree that have to be visited in the current iteration of DDS, we first suppose a static ordering. We then show under which conditions we can relax this assumption and what other options exist if we can't make those assumptions. Such counting function takes as input the variables to be explored in this subtree and the number of levels k where discrepancies are allowed.

Function (4) assumes that the variables are selected in a static variable ordering, $X_1, X_2, \ldots, X_n$, which is used in every branch of the search. Under this assumption, variable domains can have different cardinalities.

Algorithm 8: PDDS($[\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_n)]$)

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

$$C_{\text{DDS}}([X_1, \ldots, X_n], k) = \begin{cases} 1 & \text{if } k = 0 \\ |\operatorname{dom}(X_1)| - 1 & \text{if } k = 1 \\ (|\operatorname{dom}(X_k)| - 1) \prod_{i=1}^{k-1} |\operatorname{dom}(X_i)| & \text{if } k > 1 \end{cases} \tag{4}$$

Function (5) assumes that all variable domains have cardinality δ. In this case, one can count the number of leaves as follows. At iteration k, DDS performs a DFS over a tree of height $k-1$. For each leaf of this tree, DDS explores the $\delta - 1$ solutions in the subtree that cause one or more discrepancies to occur.

$$C_{\text{DDS}}([X_1, \ldots, X_n], k) = \begin{cases} 1 & \text{if } k = 0 \\ \delta^{k-1}(\delta - 1) & \text{if } k > 0 \end{cases} \tag{5}$$

Interestingly, when all domains have the same size, the number of leaves depends only on the iteration number k and the cardinality of the domains δ, but not on the number of variables.

If the variable ordering is not static and the variable domains have different cardinalities, one needs to provide a new function to count the number of leaves in a subtree that integrates the knowledge of the branching heuristic. Another possibility is to extend the

Algorithm 9: PDDS-Probe($[\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_n)]$, 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.
if k = 1 then d ← 1 else d ← 0
if k = 0 then d̄ ← 0 else d̄ ← |dom(X_i)| − 1
for d = d..d̄ do
    z ← C_DDS(Candidates \ {X_i}, max(0, k − 1))
    if (worker_id − l) mod ρ < z then
        s ← PDDS-Probe([dom(X_1), ..., dom(X_{i−1}), {v_d},
                        dom(X_{i+1}), ..., dom(X_n)], max(0, k − 1), l)
        if s ≠ ∅ then return s
    l ← (l + z) mod ρ
return ∅
```

domains of all variables with dummy values to match the cardinality of the largest domain. The dummy values can be filtered out causing a slight workload imbalance among the workers as will be discussed in Section 4.1.

4. Analysis

This Section provides an analysis of the parallel search strategies presented in Section 3.

4.1. Workload analysis

When exploring a balanced search tree, the round-robin assignment of the workers ensures that the difference between the number of leaves visited by any two workers is

at most one. However, it becomes harder to evenly divide the work among the workers when the tree is unbalanced. Search trees are often unbalanced when domain filtering and consistency techniques are applied. We prove that when a value is filtered out of a variable domain and that a branch is pruned from the tree, the workload difference among all workers is bounded.

Lemma 1 shows the benefit of implicitly assigning leaves to workers in a round-robin fashion. This lemma is based on the concept of *iteration-based search strategies*, i.e., the execution is divided into iterations that are executed one after the other. Corollary 1 is a direct application of Lemma 1 for PDFS. Theorems 1 and 2 improve Lemma 1 for the specific case of PLDS and PDDS.

Lemma 1. *Suppose an iteration-based search strategy is parallelized with a round-robin assignment of the leaves. Let i be the number of iterations necessary to cover the entire search tree. If a branch is pruned from the search tree during the search, the number of leaves removed from the workload of each worker differs by at most i.*

Proof of Lemma 1. If a branch of the tree is pruned, all the nodes under this branch are removed. Each leaf in the removed subtree is associated with a worker w and with an iteration k in which an iteration-based search strategy visits the leaf. The leaves belonging to the same iteration are assigned to the workers in a round-robin fashion. Therefore, for an iteration k, the workload among the workers differs by at most one. Since there are i iterations ($k = 1..i$), one concludes that the accumulated workload gap is bounded by i. □

Corollary 1. *Let n be the number of variables in the problem. If a branch is pruned from the search tree during the PDFS search, the number of leaves removed from the workload of each worker differs by at most* 1.

Proof of Corollary 1. Knowing that PDFS covers the entire search tree in one iteration, we can directly apply Lemma 1. Hence, the maximum workload gap is bounded by 1. □

Theorem 1. *Let n be the number of variables in the problem. If a branch is pruned from the search tree during the PLDS search, the number of leaves removed from the workload of each worker differs by at most n.*

PROOF OF THEOREM 1. We know from Lemma 1 that the workload of each worker differs by at most $n+1$ since there are $n+1$ iterations during a complete PLDS search.

The unique leaf visited in iteration 0 and the unique leaf visited in iteration n of PLDS cannot both be simultaneously filtered without filtering the whole tree. If the whole tree is filtered, then the workload between each worker is the same, as there is no work to do. Otherwise, either iteration 0 or n does not create a workload difference of one. Hence, the maximum workload gap is bounded by n. □

Theorem 2. *Let n be the number of variables in the problem. If a branch is pruned from the search tree during the PDDS search, the number of leaves removed from the workload of each worker differs by at most n.*

PROOF OF THEOREM 2. We know from Lemma 1 that the workload of each worker differs by at most $n+1$ since there are $n+1$ iterations during a complete PDDS search.

The unique leaf visited in iteration 0 and the unique leaf visited in iteration 1 of PDDS cannot both be filtered without filtering the whole tree. If the whole tree is filtered, then the workload between each worker is the same, as there is no work to do. Otherwise, either iteration 0 or 1 does not create a workload difference of one. Hence, the maximum workload gap is bounded by n. □

4.2. Overhead

To compare the overhead of PDFS, PLDS, and PDDS with DFS, LDS, and DDS, we count the number of times a node is visited when completely exploring the tree associated to n binary variables.

4.2.1. DFS as a base case

In a DFS, each node of the search tree is visited once. Since there are $2^{n+1}-1$ nodes in a binary tree of height n, let $\text{DFS}(n) = 2^{n+1}-1$ be the number of node visits in a complete DFS.

4.3. PDFS overhead

We are interested in the the number of node visits done by PDFS. To simplify the analysis, we suppose that the number of workers is a power of two: $\rho = 2^x$. If there

are more workers than leaves ($\rho > 2^n$), then there are 2^n workers that each visits $n+1$ nodes from the root to a leaf. The other other $\rho - 2^n$ workers remain idle. If there are more leaves than workers ($\rho \leq 2^n$), then all nodes at level i, for $n - \log_2 \rho < i \leq n$, are visited by exactly 2^{n-i} workers, i.e. the 2^n leaves are visited by one worker each, the 2^{n-1} parents of the leaves are visited by two workers each, the 2^{n-2} grand parents are visited by four workers each and so on. All nodes in levels 0 to $n - \log_2 \rho$ are visited by all workers. The function $\mathrm{PDFS}(\rho, n)$ returns the number of node visits of PDFS with ρ workers in a tree of n binary variables.

$$\mathrm{PDFS}(\rho, n) = \begin{cases} (n+1)2^n & \text{if } 2^n < \rho \\ \rho \cdot \mathrm{DFS}(n - \log_2 \rho) + \sum_{i=n-\log_2 \rho+1}^{n} 2^i 2^{n-i} & \text{otherwise} \end{cases} \tag{6}$$

$$= \begin{cases} (n+1)2^n & \text{if } 2^n < \rho \\ (2 + \log_2 \rho)2^n - \rho & \text{otherwise} \end{cases} \tag{7}$$

This shows that as the number of workers grows, the computational power grows linearly while the number of node visits grows logarithmically until we reach the degenerate case where the workers outnumber the leaves of the tree.

4.4. LDS overhead

Let $\mathrm{LDS}(n, k)$ be the number of nodes visited in a tree with n binary variables for which we seek leaves having k discrepancies. For $k \in \{0, n\}$, the tree has a unique leaf with k discrepancies and LDS visits $n+1$ nodes between the root and this leaf. In all other cases, the number of visited nodes depends on the number of visited nodes in the left and right subtrees. We obtain the following recurrence.

$$\mathrm{LDS}(n, k) = \begin{cases} n+1 & \text{if } k = 0 \vee k = n \\ \mathrm{LDS}(n-1, k) + \mathrm{LDS}(n-1, k-1) + 1 & \text{otherwise} \end{cases} \tag{8}$$

Theorem 3. *The closed form of recurrence* (8) *is* $\mathrm{LDS}(n, k) = \binom{n+2}{k+1} - 1$.

Proof of theorem 3. We prove, case by case, that $\binom{n+2}{k+1} - 1$ is equal to the recurrence (8).

- If $k = 0$ then $\binom{n+2}{k+1} - 1 = \binom{n+2}{1} - 1 = n + 1 = \text{LDS}(n, 0)$.
- If $k = n$ then $\binom{n+2}{k+1} - 1 = \binom{n+2}{n+1} - 1 = n + 2 - 1 = n + 1 = \text{LDS}(n, n)$.
- Otherwise, $1 \leq k < n$ and we want to prove that

$$\binom{n+2}{k+1} - 1 = \text{LDS}(n-1, k) + \text{LDS}(n-1, k-1) + 1$$

Pascal's rule states that $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$. Using this rule we obtain

$$\begin{aligned}\binom{n+2}{k+1} - 1 &= \binom{n+1}{k+1} - 1 + \binom{n+1}{k} - 1 + 1 \\ &= \text{LDS}(n-1, k) + \text{LDS}(n-1, k-1) + 1\end{aligned}$$

Since all cases of the recurrence (8) are equal to $\binom{n+2}{k+1} - 1$, then it is its closed form. □

When a complete a binary tree is searched we obtain a number of nodes visited equal to equation 9.

$$\begin{aligned}\text{LDS}(n) = \sum_{k=0}^{n} \text{LDS}(n, k) &= \sum_{k=0}^{n} \left(\binom{n+2}{k+1} - 1 \right) \\ &= \sum_{k=0}^{n} \binom{n+2}{k+1} - n - 1 \\ &= \sum_{k=0}^{n+2} \binom{n+2}{k} - n - 1 - \binom{n+2}{n+2} - \binom{n+2}{0} \\ &= 4 \cdot 2^n - n - 3\end{aligned} \tag{9}$$

We observe that, as the number of variables grows, LDS visits twice the number of nodes as DFS. Therefore, when DFS finishes visiting the entire tree, LDS has visited half of the leaves. However, these leaves have fewer than $\frac{n}{2}$ discrepancies. For any heuristic more efficient than a random heuristic, 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.

4.5. PLDS overhead

Here, we consider a PLDS with ρ workers. Let $\text{PLDS}(\rho, n, k, j)$ be the number of nodes visited by workers $j \in \{0, 1, \ldots, \rho - 1\}$ of a tree with n binary variables for which

we seek leaves having k discrepancies. We assume that the leftmost leaf must be visited by worker 0. If the leftmost leaf has to be visited by worker a, one can retrieve the number of visited nodes by relabeling the workers and computing $\text{PLDS}(\rho, n, k, j - a \bmod \rho)$. When $k \in \{0, n\}$, the tree has a unique leaf with k discrepancies and only the worker $j = 0$ 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 worker 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.

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

Let $\text{PLDS}(\rho, n)$ be the total number of nodes visited by the ρ workers.

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

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.

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

Using backward substitutions solves the recurrence.

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

We simplify for $\rho \in \{1, 2, 3\}$ assuming $n \geq 3$.

$$\text{PLDS}(1, n) = 4 \cdot 2^n - n - 3 \quad \text{PLDS}(2, n) = 5 \cdot 2^n - 2n - 4 \quad \text{PLDS}(3, n) = 5.75 \cdot 2^n - 3n - 5$$

One can note that the value of $\text{PLDS}(1, n)$ is consistent with (9), which was obtained by summing $\text{LDS}(n, k)$ over all discrepancies.

As n grows, the ratios $\frac{\text{PLDS}(2,n)}{\text{LDS}(n)}$ and $\frac{\text{PLDS}(3,n)}{\text{LDS}(n)}$ tend to 1.25 and 1.43. These overheads of 25% and 43% grow slower than the number of workers and this implies that two and three workers will visit the search tree in 62% and 48% of the time taken by one worker.

4.6. DDS overhead

Let n be the number of binary variables in a search tree and k the level of the last discrepancy for $k \leq n$. If the level of the last discrepancy is 0, then the search goes directly to the leftmost leaf of the subtree. Hence, the algorithm visits one node per variable left to instantiate, which is equal to n plus the root node which gives $n + 1$. Otherwise, the search does a DFS over the $k - 1$ first variables. For each of the 2^{k-1} leaves of this DFS, $n - k + 1$ nodes are visited down to the bottom of the search tree.

$$\begin{aligned} \text{DDS}(n, k) &= \begin{cases} n + 1 & \text{if } k = 0 \\ \text{DFS}(k - 1) + 2^{k-1}(n - k + 1) & \text{otherwise} \end{cases} \\ &= \begin{cases} n + 1 & \text{if } k = 0 \\ 2^k - 1 + 2^{k-1}(n - k + 1) & \text{otherwise} \end{cases} \end{aligned}$$

The total number of node visits done by the DDS search strategy is given by the sum over all levels $k = 0..n$ in the search tree.

$$\text{DDS}(n) = \sum_{k=0}^{n} \text{DDS}(n, k) \tag{10}$$

$$= 4 \cdot 2^n - n - 3 \tag{11}$$

Surprisingly, this is the same number of node visits as a complete LDS search (the version proposed in [26]) as we show in Section 4.4.

4.7. Analysis of PDDS

An iteration of PDDS can be seen as a PDFS over $k-1$ variables. For each of the 2^{k-1} leaves in this PDFS, PDDS completes the search by instantiating the remaining $n-k+1$ variables. Let $\text{PDDS}(\rho, n, k)$ be the number of node visits in iteration k of PDDS with ρ workers.

$$\text{PDDS}(\rho, n, k) = \begin{cases} n+1 & \text{if } k = 0 \\ \text{PDFS}(\rho, k-1) + 2^{k-1}(n-k+1) & \text{otherwise} \end{cases} \tag{12}$$

which can be expanded to

$$\text{PDDS}(\rho, n, k) = \begin{cases} n+1 & \text{if } k = 0 \\ (n+1)2^{k-1} & \text{if } k > 0 \text{ and } 2^{k-1} \leq \rho \\ (\log_2 \rho + n - k + 3)2^{k-1} - \rho & \text{otherwise} \end{cases} \tag{13}$$

We can further analyze the behavior of PDDS by summing the node visits over all iterations $k = 0..n$.

$$\text{PDDS}(\rho, n) = \sum_{k=0}^{n} \text{PDDS}(\rho, n, k) \tag{14}$$

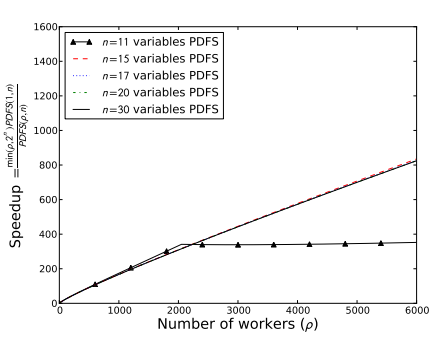
$$= n + 1 + \sum_{k=1}^{\min(\log_2 \rho, n)} \text{PDDS}(\rho, n, k) + \sum_{k=\log_2 \rho + 1}^{n} \text{PDDS}(\rho, n, k) \tag{15}$$

$$= \begin{cases} (4 + \log_2 \rho)2^n - \rho(n - \log_2 \rho + 3) & \text{if } \rho \leq 2^n \\ (n+1)2^n & \text{otherwise} \end{cases} \tag{16}$$

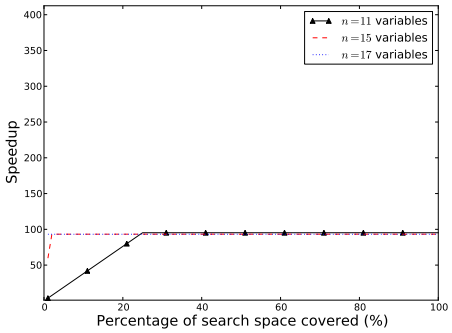
$$\tag{17}$$

4.8. Speedup analysis

The *speedup* is the ratio between the time needed for a single worker to accomplish a task over the time required for ρ workers to accomplish the same task. In this theoretical



(a) Speedup of PDFS exploring a complete binary tree as a function of the number of workers.



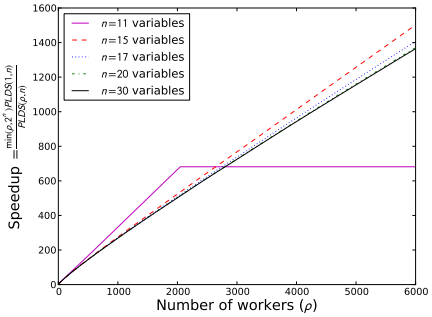
(b) Speedup of PDFS with $\rho = 512$ as a function of the percentage of the tree that is visited.

Figure 2: PDFS speedups

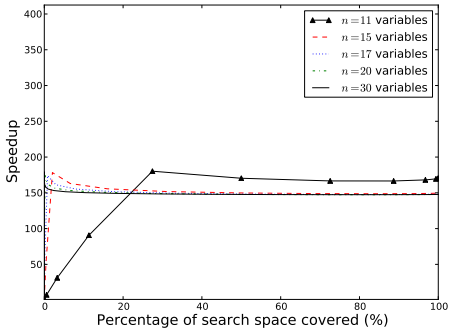
analysis, we measure the time in number of node visits while supposing that all nodes have an equal processing time. A single worker visits $\text{PDFS}(n, 1)$ nodes to explore an entire search tree of n binary variables while ρ workers each visit $\frac{\text{PDFS}(n,\rho)}{\min(\rho,2^n)}$ nodes to collectively explore the entire search tree. We therefore have a speedup of $\frac{\min(\rho,2^n)\text{PDFS}(n,1)}{\text{PDFS}(n,\rho)}$. A similar computation applies for PLDS and PDDS.

Figures 2a, 3a and 4a show the speedup of the parallel algorithms compared to their sequential versions as the number of workers increases. These speedups are computed over a complete search tree. These figures are all visually similar. This was expected since the number of nodes visited in a complete search tree, for the centralized versions, are the same. For $n = 11$ variables, the speedup stops growing after 2048 workers. Beyond this point, there are more workers than leaves in the search tree. Since any additional worker is an idle worker, the speedup reaches a plateau.

One can see that the speedups as a function of n grow linearly. In fact, while analyzing the functions $\text{PLDS}(\rho, n)$ (Equation 10), $\text{PDDS}(\rho, n)$ (Equation 13) and $\text{PDFS}(\rho, n)$ (Equation 7), one sees that the most dominant term, 2^n, is multiplied by $\log_2 \rho$. This shows that the number of nodes to be visited logarithmically increases with the number of workers. However, the computation power increases linearly with ρ. It results in a speedup in $\Theta(\frac{\rho}{\log_2 \rho})$.

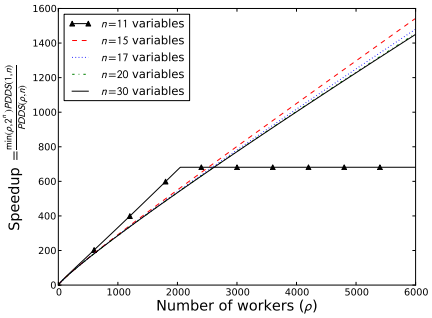


(a) Speedup of PLDS exploring a complete binary tree as a function of the number of workers.

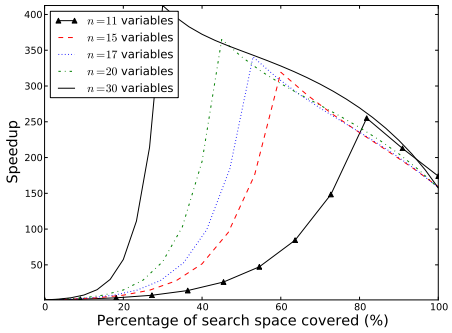


(b) Speedup of PLDS with $\rho = 512$ as a function of the percentage of the tree that is visited.

Figure 3: PLDS speedups



(a) Speedup of PDDS exploring a complete binary tree as a function of the number of workers.



(b) Speedup of PDDS with $\rho = 512$ as a function of the percentage of the tree that is visited.

Figure 4: PDDS speedups

PLDS and PDDS show a greater speedup than PDFS when a complete search of the tree is performed. Even in a centralized environment we expect LDS and DDS to find a solution sooner than DFS as we better exploit the value ordering heuristic. However, it is uncommon in practice to completely visit a search tree. Figures 2b, 3b and 4b show the speedup obtained when the search is interrupted after some fraction of the search space is covered. The speedup increases until it reaches a peak from where it decreases. For PLDS and PDDS the peak is reached at iteration $\log_2 \rho$, when the number of visited leaves reaches the number of workers. Since there are few leaves visited from iteration 0 to iteration $\log_2 \rho$, not all workers contribute to these iterations. As k grows, more leaves need to be explored and more workers contribute to these iterations, which explains why the speedup increases. After the peak, there are more leaves to visit than workers. The decrease in the speedup is due to the increase in the redundancy among the workers. Indeed, at iteration k, the redundancy occurs when the workers visit the first $k-1$ levels of the tree. The greater k is, the greater the subtree is in which the redundancy occurs.

The number of variables affects the performance of these algorithms, especially when the problem contains few variables. However, as the search progresses through the tree, the speedup stabilizes toward the value found for a complete search tree. This is a desirable effect since the search tree is rarely explored completely in practice.

For PDDS in Figure 4b, one can see that instances with more variables create a greater speedup at the beginning of the search. This is a surprising result that can be explained by the relationship between the number of variables and the overhead. In the beginning of the search, as the number of variables grows, the number of nodes between the last discrepancy of an iteration of PDDS and the leaves also grows. These nodes are assigned to exactly one worker during an iteration. Hence the overhead is reduced for larger instances during the first iterations of the search which leads to a greater speedup.

One can compare the speedups in Figures 3b and 4b with the speedups on Figure 2b where the same approach is applied to PDFS. Since PDFS contains only one iteration, the speedup quickly stabilizes to the speedup obtained with the search of a complete tree.

4.9. Statistical analysis

To get a more accurate idea of the speedup in a real context, one needs to consider the average time needed to obtain a solution of a given quality (for an optimization problem) or the probability of finding a solution (for a satisfaction problem).

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

Harvey and Ginsberg [6] showed, by analyzing the binary search trees of satisfaction problems, that the quality of a value-selection 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 in both the left and right subtrees.

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 a random variable/value selection (all leaves share the same probability of being a solution, and using LDS or DDS would not be a logical choice).

4.9.1. PLDS statistical analysis

Figure 5 shows the probability that a solution is found as a function of the computation time. The probability that a leaf s_i with k deviations is a solution is $P(s_i) = p^{n-k}q^k$ since it involves branching k times on the right and $n-k$ times on the left. The probability of finding a solution after visiting the leaves $s_1, \ldots, s_m$ is $1 - \prod_{i=1}^{m}(1 - P(s_i))$.

Of course, for a given computation time, increasing the number of workers increases the probability of finding a solution.

As the expected quality of a leaf decreases exponentially with its number of discrepancies, adding more workers makes us visit additional leaves in the same computation time, but those leaves have a 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.

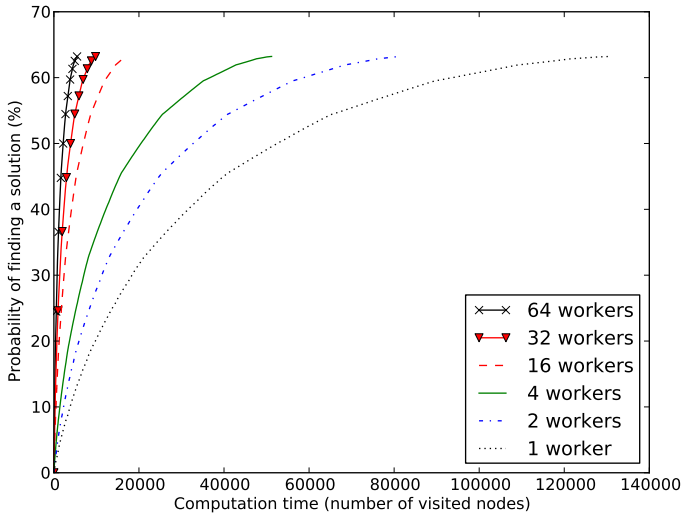


Figure 5: Probability that a solution is found as a function of the computation time with PLDS [$n = 15$ variables; $p = 0.6$; $q = 0.4$]

Figure 6 presents the speedup as a function of the 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.

Figure 7 shows the average computation time to find a solution as the number of variables grows. As expected, for one worker, the average computation time grows exponentially as the number of variables grows. However, as more workers are used, this exponential curve is pushed to the right and allows the search strategy to find a solution in less computation time.

The next experiment studies the performance of the algorithm according to the quality of the value selection heuristics used. We recall that the higher the $\frac{p}{q}$ ratio is, the more likely the solutions will be concentrated in leaves having few discrepancies. On Figure 8, the curve for one worker shows that computation time decreases exponentially when $\frac{p}{q}$ increases. Other curves show that when we provide additional workers, the computation time still decreases exponentially, but much more quickly.

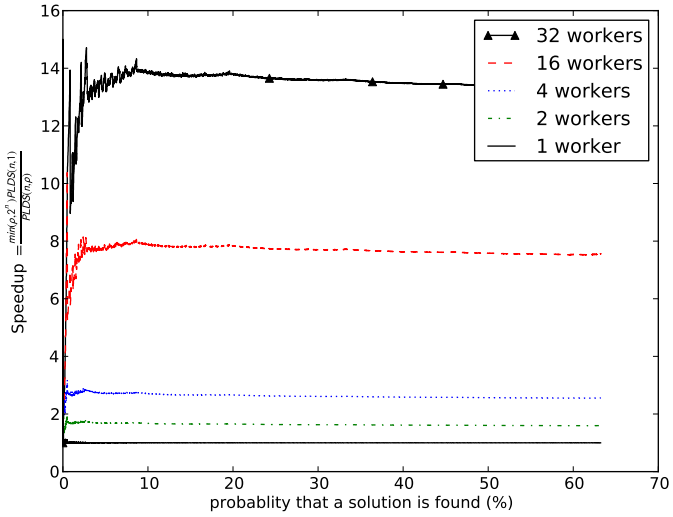


Figure 6: Speedup as a function of the probability that a solution is found with PLDS [$n = 15$ variables; $p = 0.6$; $q = 0.4$]

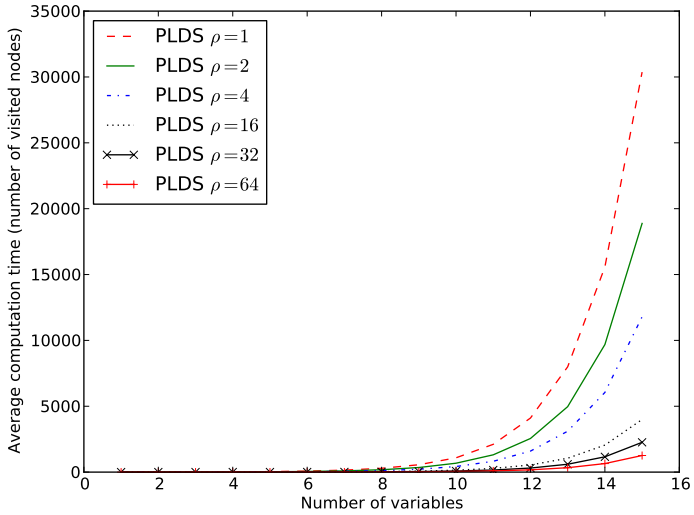


Figure 7: Average computation time to find a solution as a function of the number of variables with PLDS [$p = 0.6$; $q = 0.4$]

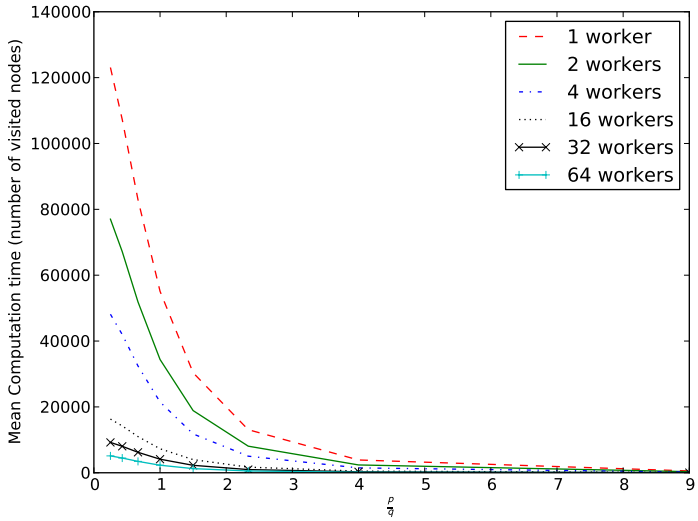


Figure 8: Average computation time to find a solution as a function of the $\frac{p}{q}$ ratio where $p + q = 1$ with 15 variables and PLDS.

4.9.2. PDDS statistical analysis

The statistical analysis used for LDS could be directly reproduced for DDS. However, DDS was developed in a different context: the probability that the value-selection heuristic finds a solution grows as the search goes deeper in the search tree. We prove here that this context also applies to PDDS. To do so, we use a statistical analysis that includes a probability of finding a solution that grows as the search gets deeper in the tree. Following Walsh [7], we suppose that the probability that the heuristic branches towards a solution grows linearly with the depth of the node in the tree. The probability p of finding a solution in the left sub-tree starts at 0.5 at the root and grows by equal steps at each level of the search tree. The probability p reaches a maximum of 0.9 at the last level of the search tree. At each level of the search tree, the probability of finding a solution in the right subtree is given by $q = 1 - p$. This relation ensures that the expected number of solutions in the search tree is one. The probability of a leaf being a solution is given by the product of the probabilities of finding a solution at each branching.

Figure 9 shows a curve similar to the one of PLDS on Figure 5. Finding a solution with

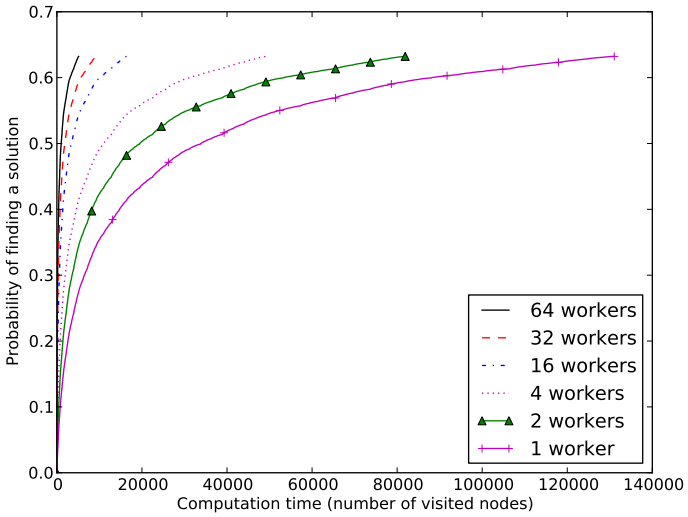


Figure 9: Probability that a solution is found as a function of the computation time with PDDS [$n = 15$ variables, $p \in [0.5, 0.9]$ linearly varying with $q = 1 - p$]

greater probability takes exponentially more computation time. Increasing the number of workers increases the probability of finding a solution for a given computation time.

5. Experiments with industrial data

We carried out experiments with industrial data for a combined planning and scheduling problem from the forest-products industry described in [4].

In a lumber finishing facility, lumbers are planed, and then sorted according to their grade (i.e. quality). They may be trimmed in order to produce shorter lumbers of a higher grade and value. This causes production of multiple finished products at the same time (co-production) from a single raw product (divergence).

The mill can only process lumber of a single category (raw material) in a given production shift. However, they prefer longer campaigns as this reduces costs. All this makes production very difficult to plan according to customers, orders (a set of ⟨date, product, quantity⟩ tuples).

To sum up, the decisions that must be taken in order to plan the finishing operations

are the following:

1. decide when the campaigns start and for how long they last;
2. select a lumber category to process during the campaign;
3. decide, for each campaign, the quantities of each compatible lumber product to process.

The objective is to minimize order lateness and production costs.

The problem is fully described in [4] which provides a good heuristic for this problem. In [5], this heuristic was used to guide the search in a constraint programming model. Provided with this heuristic, LDS outperformed DFS as well as a mathematical programming approach.

Industrial instances are huge and there is a need for good solutions in shorter computation time. The instances have an average of 50,238 constraints and 65,142 variables. Among them, there are 42 discrete decision variables whose domains have cardinality 6 and 4200 continuous decision variables. As we have a very good branching heuristic for which LDS works really well, this problem is an ideal candidate for PLDS and PDDS. 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 4 where all nodes have the same computation time.

We implement PLDS, PDDS and PDFS and run 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 provide the industrial instances. The three 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.

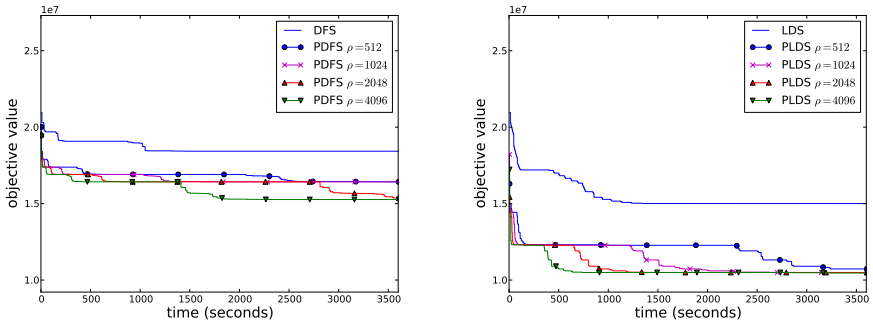
5.1. Results and discussion

Figures 10 to 12 show the objective value according to computation time (maximum one hour) for 1, 512, 1024, 2048 and 4096 workers. DFS and PDFS with one worker showed the same performance. For this reason we omit the latter in the chart. The same comment applies to LDS (PLDS) and DDS (PDDS).

As expected, DFS has the lowest performance and DDS outperformed LDS since we use a specialized value and variable ordering heuristic adapted to this problem. This shows that the assumption of DDS is true in this case: exploring first the discrepancies at the top of the search tree leads to better solutions faster. The centralized DDS even catches up with PLDS running on 512 workers (see Figure 11).

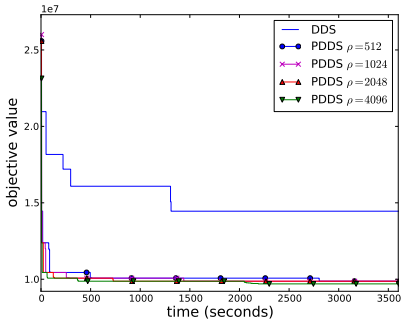
For Figures 10 to 12, the curves for $512, 1024, 2048$, and 4096 workers have the same shape but become more compressed over time as the number of workers increases. This shows that the heuristic and the search strategies remain the same even in their parallelized version. We can see, in these figures, that the quality of the solutions found with the parallel algorithms are far greater than with their sequential version. For example, in Figure 10b, a solution of quality of 1.1×10^7 is not found with one worker even after an hour but can be found in 10 minutes with 4096 workers. Furthermore, one worker obtained a solution of 1.5×10^7 in one hour while the same solution is found in a few seconds with 512 workers. This is a major improvement from an industrial point of view where computation time is the real constraint. PDDS using 4096 workers obtains solutions of quality that were never reached before. PDDS has reduced the backorders by a ratio ranging between 68% and 85% when compared to DDS. Finally, if one needs a solution of a given quality, PDDS finds it with much less computation time than PLDS and PDFS.

Table 1 reports statistics computed during these experiments. Let χ_j be the number of leaves processed by worker j. Let ψ be the number of leaves visited by one worker. The wall clock time of this experiment is fixed to one hour. The speedup is computed as a ratio of the total number of leaves visited by multiple workers in one hour and the number of leaves visited by one worker during the same time: $\frac{\sum_{j=1}^{\rho} \chi_j}{\psi}$. We denote by $\min(\chi)$ the minimum value of χ_j for every worker $j \in 0, 1, \ldots, \rho - 1$ and by $\overline{\chi}$ the average number of leaves visited by a worker. Even if the whole search tree is not visited and no



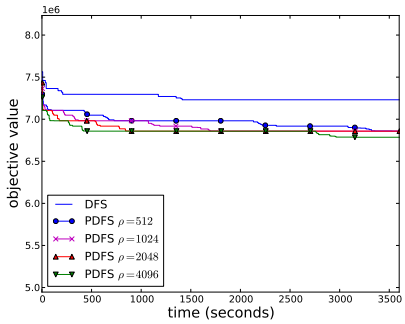
(a) PDFS

(b) PLDS

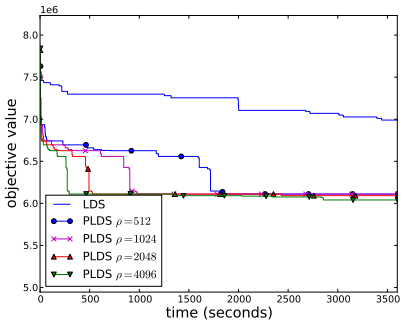


(c) PDDS

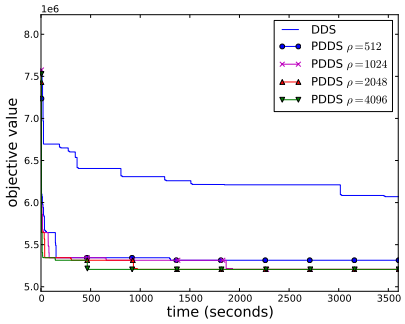
Figure 10: Objective value in function of time (dataset M1)



(a) PDFS

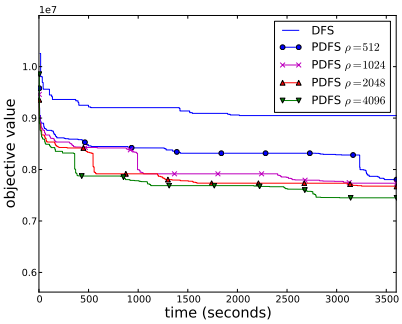


(b) PLDS

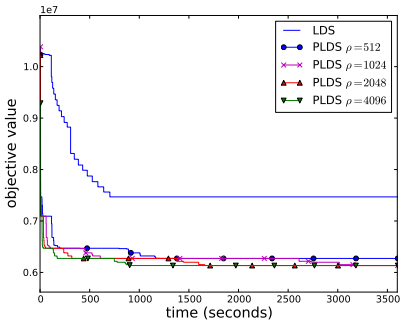


(c) PDDS

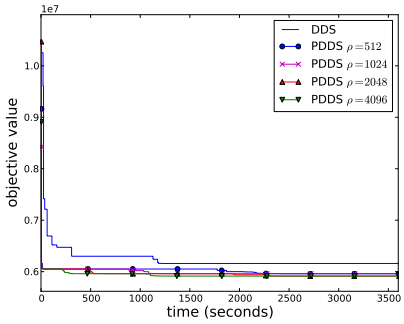
Figure 11: Objective value in function of time (dataset M2)



(a) PDFS



(b) PLDS



(c) PDDS

Figure 12: Objective value in function of time (dataset M3)

dataset	search strategy	ρ	speedup	$\overline{\chi}$	$max(\chi) - min(\chi)$	σ_χ	Leaves per second
M1	PLDS	-	-	-	-	-	0.23
M1	PLDS	512	455.1	735.07	13	2.3	104.54
M1	PLDS	1024	852.7	688.68	27	4.75	195.89
M1	PLDS	2048	1613.9	651.69	29	4.88	370.74
M1	PLDS	4096	3116.8	629.29	31	6.33	715.99
M1	PDDS	-	-	-	-	-	0.17
M1	PDDS	512	517.9	622.04	12	1.94	88.47
M1	PDDS	1024	1001.0	601.21	24	4.95	171.01
M1	PDDS	2048	2008.2	603.06	24	4.68	343.07
M1	PDDS	4096	4087.0	613.66	24	4.37	698.2
M1	PDFS	-	-	-	-	-	0.26
M1	PDFS	512	525.6	975.25	93	19.55	138.7
M1	PDFS	1024	1059.5	982.93	93	17.65	279.59
M1	PDFS	2048	2071.7	960.97	258	62.99	546.69
M1	PDFS	4096	4175.3	968.38	270	68.78	1101.8
M2	PLDS	-	-	-	-	-	0.3
M2	PLDS	512	432.0	922.24	16	2.35	131.16
M2	PLDS	1024	822.9	878.36	12	1.88	249.84
M2	PLDS	2048	1579.2	842.81	13	1.92	479.46
M2	PLDS	4096	2947.1	786.41	18	2.33	894.76
M2	PDDS	-	-	-	-	-	0.23
M2	PDDS	512	475.0	756.11	14	2.68	107.53
M2	PDDS	1024	945.4	752.41	14	2.44	214.02
M2	PDDS	2048	1886.7	750.82	17	2.53	427.13
M2	PDDS	4096	3732.8	742.72	18	2.76	845.06
M2	PDFS	-	-	-	-	-	0.34
M2	PDFS	512	469.0	1109.2	62	17.73	157.75
M2	PDFS	1024	921.7	1090.0	255	28.51	310.04
M2	PDFS	2048	1824.3	1078.73	293	26.76	613.68
M2	PDFS	4096	3608.0	1066.71	84	24.3	1213.68
M3	PLDS	-	-	-	-	-	0.32
M3	PLDS	512	360.6	820.55	20	3.14	116.7
M3	PLDS	1024	689.4	784.38	42	9.8	223.11
M3	PLDS	2048	1246.4	709.0	41	6.65	403.34
M3	PLDS	4096	2436.7	693.06	33	5.03	788.55
M3	PDDS	-	-	-	-	-	0.25
M3	PDDS	512	469.0	830.79	84	11.16	118.16
M3	PDDS	1024	926.5	820.67	90	13.55	233.43
M3	PDDS	2048	1844.2	816.75	85	11.8	464.64
M3	PDDS	4096	3695.4	818.3	113	14.29	931.04
M3	PDFS	-	-	-	-	-	0.37
M3	PDFS	512	433.5	1138.91	221	28.99	161.98
M3	PDFS	1024	846.4	1111.69	612	39.74	316.21
M3	PDFS	2048	1630.9	1071.06	675	40.01	609.31
M3	PDFS	4096	3160.2	1037.72	1183	66.1	1180.7

Table 1: Statistics of the industrial datasets experiments. Column $\overline{\chi}$ is the average number of leaves visited by each worker. Column σ_χ standard deviation of the number of leaves visited by each worker. Column $max(\chi) - min(\chi)$ is the maximum difference of processed leaves between workers.

workers become idle, we wanted to measure the difference of workload between workers, in terms of visited leaves. A worker that visits more leaves is more likely to complete the search and become idle. We measure the standard deviation of the number of leaves visited by a worker that we denote σ_χ. We also measure the difference $\max(\chi) - \min(\chi)$.

As shown on Table 1, the speedup is not linear but continues to grow as we add more workers. This was expected from our theoretical analysis.[1] The number of leaves visited per second also grows at roughly the same rate. The average number of leaves visited by a worker, $\overline{\chi}$, varies slightly for a same dataset and search strategy. This is due to uneven processing times of the leaves. The difference $\max(\chi) - \min(\chi)$ and the standard deviation σ_χ, when compared to $\overline{\chi}$, shows that the workers have a good workload balance and that they have visited roughly the same number of leaves.

6. Conclusion and future work

We proposed a parallelization scheme (PDS) based on implicit round-robin assignment of the leaves to the workers. We theoretically showed that this approach scales up to the degenerate case where there are more workers than leaves in the search tree. This ability to scale to a large number of workers comes from the absence of communication between the workers. PDS allows keeping the workload of the workers balanced even when the search tree is pruned.

We applied this approach to LDS, DFS and DDS to produce parallel algorithms that keep the node visit order of their sequential version. We theoretically analyzed the numbers of node visits of LDS, PLDS, DFS, DDS, PDFS and PDDS. These numbers of node visits are used to analyze the theoretical speedup of PLDS, PDDS and PDFS. We showed that the number of node visits of DDS is the same as LDS when visiting a complete search tree. These analysis showed that the number of nodes grows logarithmically with the PDS parallelization scheme. Since the number of workers grows linearly, we can conclude that more workers will always lead to a greater speedup. When only a part of the tree is visited, as is most common, the instances with more variables lead to a greater speedup.

[1] The super-linear speedup obtained on instance M1 is explained by the uneven time required to solve the linear programs associated to each leaf. Other instances do not show this behavior.

Finally we used an industrial problem from the forest-products industry to experiment with these parallel algorithms. We showed that PDDS consistently performs better than PDFS and PLDS in our industrial context for which a good variable/value selection heuristic was provided. From an industrial point of view, the computation time is 4087 times faster with 4096 workers than with one worker. Moreover, the order lateness is reduced by up to 67% for a given computation time.

Other sequential algorithms can easily be parallelized using this scheme. This can lead to parallelization of search strategies that have interesting properties and have not had parallel counterparts so far. Multiple avenues are possible for future works based on the concept presented in this paper. A possible improvement for PDS would be the ability to add workers on the fly during optimization. In this case some minimal communication would be needed but the gain from additional workers could outweigh the computation and communication needed for their insertion.

References

[1] A. Chabrier, E. Danna, C. Le Pape, L. Perron, Solving a network design problem, Annals of Operations Research 130 (2004) 217–239.

[2] C. Le Pape, L. Perron, J.-C. Régin, P. Shaw, 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, pp. 633–648.

[3] C. Le Pape, P. Baptiste, Heuristic control of a constraint-based algorithm for the preemptive job-shop scheduling problem, Journal of Heuristics 5 (1999) 305–325.

[4] J. Gaudreault, P. Forget, J.-M. Frayret, A. Rousseau, S. Lemieux, S. D'Amours, Distributed operations planning in the lumber supply chain: Models and coordination., International Journal of Industrial Engineering: Theory, Applications and Practice 17 (3).

[5] J. Gaudreault, J.-M. Frayret, A. Rousseau, S. D'Amours, Combined planning and scheduling in a divergent production system with co-production: A case study in the lumber industry, Computers and Operations Research 38 (2011) 1238–1250.

[6] W. D. Harvey, M. L. Ginsberg, Limited discrepancy search, in: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 1995), 1995, pp. 607–613.

[7] T. Walsh, Depth-bounded discrepancy search, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 1997), 1997, pp. 1388–1393.

[8] J. C. Beck, L. Perron, 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, pp. 8–10.

[9] L. Perron, Search procedures and parallelism in constraint programming, in: Proceedings of Fifth International Conference on Principles and Practice of Constraint Programming (CP 1999), 1999, pp. 346–360.

[10] V. Vidal, L. Bordeaux, Y. Hamadi, 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] O. V. Shylo, T. Middelkoop, P. M. Pardalos, Restart strategies in optimization: Parallel and serial cases, Parallel Computing 37 (1) (2010) 60–68.

[12] M. Luby, A. Sinclair, D. Zuckerman, Optimal speedup of las vegas algorithms, Information Processing Letters 47 (1993) 173–180.

[13] C. P. Gomes, 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, pp. 431–437.

[14] C. P. Gomes, Complete randomized backtrack search, in: Constraint and Integer Programming: Toward a Unified Methodology, 2003, pp. 233–283.

[15] Y. Hamadi, L. Sais, ManySAT: a parallel SAT solver, Journal on Satisfiability, Boolean Modeling and Computation 6 (2009) 245–262.

[16] L. Xu, F. Hutter, H. H. Hoos, K. Leyton-Brown, Satzilla: Portfolio-based algorithm selection for sat., Journal of Artificial Intelligence Research (JAIR) 32 (2008) 565–606.

[17] Y. Hamadi, G. Ringwelski, Boosting distributed constraint satisfaction, Journal of Heuristics (3) (2010) 251–279.

[18] J.-F. Puget, Constraint programming next challenge: Simplicity of use, in: Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004), 2004, pp. 5–8.

[19] L. Bordeaux, Y. Hamadi, H. Samulowitz, Experiments with massively parallel constraint solving, in: Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI 2009), 2009, pp. 443–448.

[20] L. Michel, A. See, P. Van Hentenryck, Transparent parallelization of constraint programming, INFORMS Journal on Computing 21 (2009) 363–382. doi:10.1287/ijoc.1080.0313.
URL http://portal.acm.org/citation.cfm?id=1576269.1576272

[21] G. Chu, C. Schulte, P. J. Stuckey, 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, pp. 226–241.

[22] T. Menouer, B. Le Cun, P. Vander-Swalmen, Partitioning methods to parallelize constraint programming solver using the parallel framework Bobpp, in: Advanced Computational Methods for Knowledge Engineering, Springer, 2013, pp. 117–127.

[23] F. Xie, A. Davenport, 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, pp. 334–338.

[24] X. Yun, S. L. Epstein, A hybrid paradigm for adaptive parallel search, in: Principles and Practice of Constraint Programming, Springer, 2012, pp. 720–734.

[25] J.-C. Régin, M. Rezgui, A. Malapert, Embarrassingly parallel search, in: Proceedings of the 19^{th} International Conference on Principles and Practice of Constraint Programming (CP 2013), Springer, 2013, pp. 596–610.

[26] R. E. Korf, Improved limited discrepancy search, in: Proceedings of the 30th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference, Volume 1, 1996, pp. 286–291.

[27] D. Furcy, S. Koenig, Limited discrepancy beam search, in: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2005), 2005, pp. 125–131.

[28] S. Boivin, B. Gendron, G. Pesant, Parallel constraint programming discrepancy-based search decomposition, Optimization days, Montréal, Canada (May 2007).

[29] J. Gaudreault, J.-M. Frayret, G. Pesant, Discrepancy-based method for hierarchical distributed optimization, in: Nineteenth International Conference on Tools with Artificial Intelligence (ICTAI 2007), 2007, pp. 75–81.

[30] J. Gaudreault, J.-M. Frayret, G. Pesant, Distributed search for supply chain coordination, Computers in Industry 60 (2009) 441–451.

[31] M. Yokoo, Distributed constraint satisfaction: foundations of cooperation in multi-agent systems, Springer-Verlag, London, UK, 2001.

[32] P. J. Modi, W.-M. Shen, M. Tambe, M. Yokoo, Adopt: Asynchronous distributed constraint optimization with quality guarantees, Artificial Intelligence 161 (1–2) (2006) 149–180.

[33] J. Gaudreault, J.-M. Frayret, G. Pesant, Discrepancy-based optimization for distributed supply chain operations planning, in: Proceeding of the Ninth International Workshop on Distributed Constraint Reasoning (DCR 2007), 2007.