Parallel Depth-bounded Discrepancy Search

Thierry Moisan, Claude-Guy Quimper, and Jonathan Gaudreault

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

Abstract. Search strategies such as Limited Discrepancy Search (LDS) and Depth-bounded Discrepancy Search (DDS) find solutions faster than a standard Depth-First Search (DFS) when provided with good value-selection heuristics. We propose a parallelization of DDS: Parallel Depth-bounded Discrepancy Search (PDDS). This parallel search strategy has the property to visit the nodes of the search tree in the same order as the centralized version of the algorithm. The algorithm creates an intrinsic load-balancing: pruning a branch of the search tree equally affects each worker's workload. This algorithm is based on the implicit assignment of leaves to workers which allows the workers to operate without communication during the search. We present a theoretical analysis of DDS and PDDS. We show that PDDS scales to multiple thousands of workers. We experiment on a massively parallel supercomputer to solve an industrial problem and improve over the best known solution.

1 Introduction

Parallelization has been of growing interest in recent years, including in the optimization community. de la Banda et al. [1] consider parallelization as one of the three main challenges in the future of optimization technologies. Search is at the core of optimization and constraint solvers. If one wants to parallelize a solver, it is natural to consider parallelizing search strategies.

Parallelization of constraint programming solver is a hard problem mainly due to communication between workers. When the number of workers is large, the time each worker spends communicating with the other workers often exceeds the time spent at solving the original problem.

We have recently seen good parallel algorithms without communication that are based on centralized algorithms. Régin et al. [2] split the problem into multiple subproblems. These subproblems are then given to workers that solve them using classic centralized algorithm.

Parallel Limited Discrepancy-based Search (PLDS[1]) [3] is based on Limited Discrepancy-based Search (LDS) [4]. LDS has a huge advantage over traditional search strategies such as Depth-First Search (DFS) when a good value ordering heuristic is used. The PLDS parallel version keeps this advantage by preserving

[1] In the original article this algorithm was named PDS. In this paper, we name it PLDS for clarity concerns.

the node visit ordering of the centralized algorithm. Each leaf of the search tree is implicitly assigned to a worker. Every worker branches in the search tree while making sure there is at least one leaf in the subtree of the current node that is assigned to it. An important property of this approach is that, upon pruning the search tree, workload balance difference can be theoretically bounded. PLDS scales to thousands of workers.

In this paper, we show that the parallelization mechanism used by PLDS can also be used to parallelize other search strategies while keeping the same properties. We parallelize the Depth-bounded Discrepancy Search algorithm (DDS) [5] to obtain Parallel Depth-bounded Discrepancy Search (PDDS). Our motivation lies in the observation that in a centralized environment, DDS is generally more efficient than LDS when it is provided with good value ordering heuristics. We also show how the same parallelization mechanism can be applied to DFS which becomes Parallel Depth-First Search (PDFS). The theoretical analysis of PDFS will simplify the analysis of PDDS.

The rest of this paper is divided as follows. Section 2 describes the DDS algorithm and reviews related parallel computing work. Section 3 details the PDFS algorithm while section 4 details the PDDS algorithm. Section 5 presents a theoretical analysis of the algorithms. Finally, we experiment with an industrial problem coming from the wood-products industry in Section 6.

2 Literature review

We review the related works by presenting different parallelization approaches. Then, we describe the original DDS algorithm that we parallelize in Section 4.

2.1 Shared memory

It is possible to parallelize a search strategy by sharing, through a *shared memory* space, a list of open nodes, i.e. the visited nodes for which there are still values to branch on. Each worker can select an open node and process it until no more work can be done from that point. Then, the worker comes back to the pool of open nodes to obtain more work.

Perron [6] proposes a framework based on this idea. Good performances are often reported, as in [7] where a parallel Best First Search is implemented and evaluated up to 64 processors. However, this approach cannot easily scale up to thousands of processors due to communication overload.

2.2 Portfolios

Portfolio-based methods use a set of different solvers, parameters and/or search strategies. Workers are using different configurations to solve the exact same problem in parallel, increasing the probability of quickly finding a good solution. The approach can be improved by making use of randomized restarts [8] and nogoods learning [9].

Finding good alternative configurations for a specific problem can be a difficult problem by itself. Xu et al. [10] use machine learning to find appropriate SAT solver configurations to a new problem based on a set of learned examples.

2.3 Search space splitting and work stealing

Space splitting divides the search tree into small subtrees that are assigned to the workers. As it is unlikely that those subtrees are of equal size, a *work stealing* mechanism (see [11,12]) allows busy workers to share their workload with idle workers and therefore evenly balance the workload among all workers. In [13], Menouer et al. parallelize the constraint programming solver OR-Tools using a framework based on work-stealing.

Communication may cause issues when there are too many workers. At some point, the communication monopolizes the majority of the computing power. Reducing the amount of communication speeds up the search. For example, Xie and Davenport [14] allocate specific workers to coordinate the tasks, allowing more processors to be used before performance starts to decline.

Yun and Epstein [15] 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. Régin et al. [2] split the problem into a large number of subtrees. Some are quicker to explore than others, as pruning occurring during the search does not equally affect each part of the search tree. However, since a large number of subtrees is assigned to each worker, their workload tend to balance.

The exclusion of all communication during the search is also the solution we advocate in our previous work [3] where we introduced PLDS, a parallel version of Limited Discrepancy Search (LDS) [4]. The parallelization is done by implicitly assigning leaves of the search tree to workers. We showed how to test whether a worker has any work assigned to it in the subtree under the current node. This parallel algorithm has the property to keep the same node visit ordering as the sequential version. This is the approach we will generalize in this paper.

2.4 Depth-bounded Discrepancy Search

Harvey and Ginsberg [4] introduce the concept of *discrepancy*. 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 represented graphically, the left branch under a node corresponds to the recommendation of the heuristic while the right one does not. Figure 1 shows such binary search tree. Therefore, each time the solver branches to the right in this tree, it goes against the heuristic recommendation. Such branching

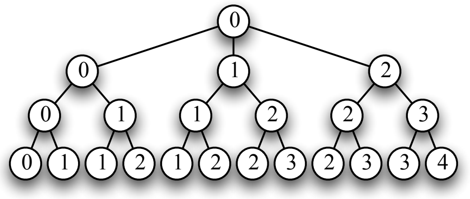


Fig. 1: A binary search tree with the number of discrepancies of each node.

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. Harvey and Ginsberg show that, when using a good value ordering heuristic, the expected quality of a leaf decreases as the number of discrepancies increases.

Limited discrepancy search (LDS) [4] is the first search strategy based on discrepancies. It visits the leaves of the tree in order of discrepancies. Improved Limited Discrepancy Search (ILDS) [16] is an improvement over LDS since it visits each leaf at most once (the original LDS has redundancy). There are other search strategies that take advantage of discrepancies such as Discrepancy-Bounded Depth First Search (DBDFS) [17] and Limited Discrepancy Beam Search (BULB) [18].

Depth-bounded Discrepancy Search (DDS) [5] makes the following assumption: it is more probable that the value ordering heuristic makes a mistake at the top of the 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 makes a mistake at top the of the tree. Based on this assumption, if the search has to reconsider the choices it made, it better 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 values that do not respect the heuristic, and then always branches on the leftmost child until it reaches a leaf.

Algorithms 1 and 2 are a generalization of the original DDS algorithm [5] for n-ary variables.

In the following description, we suppose that the variable ordering heuristic is deterministic and only depends on the states of the 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 multiple times some leaves and ignore other leaves). This supposition was also made in [4] and [5].

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

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

Algorithm 2: DDS-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.
if $k = 1$ then $\underline{d} \leftarrow 1$ else $\underline{d} \leftarrow 0$
if $k = 0$ then $\overline{d} \leftarrow 0$ else $\overline{d} \leftarrow |\mathrm{dom}(x_i)| - 1$
for $d = \underline{d}..\overline{d}$ do
    $s \leftarrow$ DDS-Probe($[\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_{i-1}), \{v_d\},$
                     $\mathrm{dom}(X_{i+1}), \ldots, \mathrm{dom}(X_n)], \max(0, k-1)$)
    if $s \neq \emptyset$ then return $s$
return $\emptyset$
```

3 PDFS algorithm

PDFS will simplify the theoretical analysis of DDS and PDDS for the following reason. At iteration k, DDS (PDDS) performs a DFS (PDFS) over the first $k-1$ variables of the problem.

To our knowledge, it is the first time that DFS and DDS are parallelized this way.

We parallelize DFS over ρ workers labeled from 0 to $\rho - 1$. Let s be a leaf of the search tree and $v(s)$ its order of visit in DFS, i.e. the first leaf visited by DFS has a visit order of 0, the second leaf visited by DFS has a visit order of 1, and so on. We implicitly assign a leaf s to worker $v(s) \bmod \rho$. Each worker is aware of its label and the total number of workers ρ. A worker w performs a standard DFS but only visits the nodes that have at least one leaf, among their descendants, whose assigned worker is w. Each worker needs to decide whether a node leads to a leaf of interest. This is done as follows.

Let a be the current node and left(a) its left child. The PDFS search keeps track of the worker $l(a)$ assigned to the leftmost leaf, in the subtree rooted at a, to be visited in the current iteration of the centralized search strategy. In the case of PDFS, there is only one iteration but PDDS is run over multiple iterations. We necessarily have $l(a) = l(\text{left}(a))$ since both subtrees have the same leftmost leaf. The function C_{DFS} takes as input a node and returns the number of its descendants that are leaves to be visited in the current iteration of the centralized search strategy.

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

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

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

The list of workers that needs to visit left(a) is given by $l(a), (l(a)+1) \bmod \rho, (l(a)+2) \bmod \rho, \ldots, (l(a) + C_{\text{DFS}}(\text{left}(a)) - 1) \bmod \rho$. Consequently, the worker w only needs to visit the node left(a) if it belongs to this list. This can be tested with the inequality $(w - l(\text{left}(a))) \bmod \rho < C_{\text{DFS}}(\text{left}(a))$. One can apply the same reasoning on the right child right(a) knowing that that $l(\text{right}(a)) = (l(\text{left}(a)) + C_{\text{DFS}}(\text{left}(a))) \bmod \rho$.

Algorithm 3: PDFS($[\operatorname{dom}(X_1), \ldots, \operatorname{dom}(X_n)]$, l)

```
$Candidates \leftarrow \{X_i \mid |\operatorname{dom}(X_i)| > 1\}$
Choose a variable $X_i \in Candidates$
$z \leftarrow C_{\text{DFS}}(Candidates \setminus \{X_i\})$
for $v_d \in \operatorname{dom}(X_i)$ do
    if $(currentProcessor - l) \bmod \rho < z$ then
        $s \leftarrow$ PDFS($[\operatorname{dom}(X_1), \ldots, \operatorname{dom}(X_{i-1}), \{v_d\}, \operatorname{dom}(X_{i+1}) \ldots, \operatorname{dom}(X_n)], l$)
        if $s \neq \emptyset$ then return $s$
    $l \leftarrow (l + z) \bmod \rho$
return $\emptyset$
```

Algorithm 3 describes PDFS. The first call to PDFS is done with the original variable domains and $l = 0$.

4 PDDS algorithm

We show in this section how the same mechanism can be applied to DDS which becomes a Parallel Depth-bounded Discrepancy Search (PDDS). As in PLDS

Algorithm 4: PDDS($[\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_n)]$)

```
$l \leftarrow 0$
for $k = 0..n$ do
    $Candidates \leftarrow \{X_i \mid |\mathrm{dom}(X_i)| > 1\}$
    $z \leftarrow C_{\text{DDS}}(Candidates, k)$
    if $(currentProcessor - l) \bmod \rho < z$ then
        $s \leftarrow$ PDDS-Probe($[\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_n)], k, l$)
        if $s \neq \emptyset$ then return $s$
    $l \leftarrow (l + z) \bmod \rho$
return $\emptyset$
```

and PDFS, parallelization is done by assigning the leaves of the search tree to each worker in a round-robin fashion.

Algorithms 4 and 5 show how PDDS operates. Algorithm 4 visits all the leaves whose discrepancies appear within the first k variables. Algorithm 5 launches an iteration to visit all those leaves following a DFS search.

As for 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 of DDS. The next subsection shows how to implement the function C_{DDS}.

Algorithm 5: PDDS-Probe($[\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_n)]$, k, l)

```
$Candidates \leftarrow \{X_i \mid |\mathrm{dom}(X_i)| > 1\}$
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.
if $k = 1$ then $\underline{d} \leftarrow 1$ else $\underline{d} \leftarrow 0$
if $k = 0$ then $\overline{d} \leftarrow 0$ else $\overline{d} \leftarrow |\mathrm{dom}(X_i)| - 1$
for $d = \underline{d}..\overline{d}$ do
    $z \leftarrow C_{\text{DDS}}(Candidates \setminus \{X_i\}, \max(0, k-1))$
    if $(currentProcessor - l) \bmod \rho < z$ then
        $s \leftarrow$ PDDS-Probe($[\mathrm{dom}(X_1), \ldots, \mathrm{dom}(X_{i-1}), \{v_d\}$,
                        $\mathrm{dom}(X_{i+1}), \ldots, \mathrm{dom}(X_n)], \max(0, k-1), l$)
        if $s \neq \emptyset$ then  return $s$
    $l \leftarrow (l + z) \bmod \rho$
return $\emptyset$
```

4.1 Counting functions

We provide two functions that count the number of leaves in a subtree that have to be visited in the current iteration of the DDS. Both functions take as input the variables to be explored in this subtree and the number of levels k where discrepancies are allowed. Function 3 assumes that all variable domains have cardinality δ. Function 4 assumes that variables are selected in a static ordering. Without these assumptions, one would need to integrate the knowledge of the branching heuristic into the computation of the number of leaves. However, it is always possible to do a workaround and to extend the 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 processors as it will be discussed in Section 5.6.

If all variable domains have cardinality δ, then 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 that cause one or more discrepancies to occur.

$$C_{\text{DDS}}([X_1, \ldots, X_n], k) = \begin{cases} 1 & \text{if } k = 0 \\ \delta^{k-1}(\delta - 1) & \text{if } k > 0 \end{cases} \tag{3}$$

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.

We can also suppose 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.

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

If the variables do not have the same domain size and their ordering is not static, then the number of leaves in the search tree visited at iteration k depends on the variable ordering. The function C_{DDS} should be redefined according to the branching heuristic.

5 Analysis

This section provides an analysis of DFS, DDS, PDFS and PDDS. To compare these search strategies, we count the number of times each strategy visits a node while exploring an entire search tree of n binary variables.

5.1 Analysis of DFS

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, we define $\text{DFS}(n) = 2^{n+1} - 1$ to be the number of node visits in a complete DFS.

5.2 Analysis of DDS

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.

$$\begin{aligned}\text{DDS}(n) &= \sum_{k=0}^{n} \text{DDS}(n,k) && (5) \\ &= 4 \cdot 2^n - n - 3 && (6)\end{aligned}$$

Surprisingly, this is the same number of node visits as a complete LDS search (the version proposed in [16]). The number of node visits of LDS was previously shown in [3].

5.3 Analysis of PDFS

We are interested in 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 $\rho - 2^n$ 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 2 workers each, the 2^{n-2} grand-parents are visited by 4 workers each and so on until level $n - \log_2 \rho$ where all nodes are visited by all workers. All nodes in levels 0 to $n - \log_2 \rho$ are visited by all processors. The function $\text{PDFS}(\rho, n)$ returns the number of node visits of PDFS with ρ workers in a tree of n binary variables.

$$\mathrm{PDFS}(\rho, n) = \begin{cases} (n+1)2^n & \text{if } 2^n < \rho \\ \rho \cdot \mathrm{DFS}(n - \log_2 \rho) + \sum_{i=n-\log_2 \rho+1}^{n} 2^i 2^{n-i} & \text{otherwise} \end{cases} \tag{7}$$

$$= \begin{cases} (n+1)2^n & \text{if } 2^n < \rho \\ (2 + \log_2 \rho)2^n - \rho & \text{otherwise} \end{cases} \tag{8}$$

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.

5.4 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 $\mathrm{PDDS}(\rho, n, k)$ be the number of node visits in iteration k of PDDS with ρ workers.

$$\mathrm{PDDS}(\rho, n, k) = \begin{cases} n+1 & \text{if } k = 0 \\ \mathrm{PDFS}(\rho, k-1) + 2^{k-1}(n-k+1) & \text{otherwise} \end{cases} \tag{9}$$

which can be expanded to

$$\mathrm{PDDS}(\rho, n, k) = \begin{cases} n+1 & \text{if } k = 0 \\ (n+1)2^{k-1} & \text{if } k > 0 \text{ and } 2^{k-1} \leq \rho \\ (\log_2 \rho + n - k + 3)2^{k-1} - \rho & \text{otherwise} \end{cases} \tag{10}$$

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

$$\mathrm{PDDS}(\rho, n) = \sum_{k=0}^{n} \mathrm{PDDS}(\rho, n, k) \tag{11}$$

$$= n + 1 + \sum_{k=1}^{\min(\log_2 \rho, n)} \mathrm{PDDS}(\rho, n, k) + \sum_{k=\log_2 \rho+1}^{n} \mathrm{PDDS}(\rho, n, k) \tag{12}$$

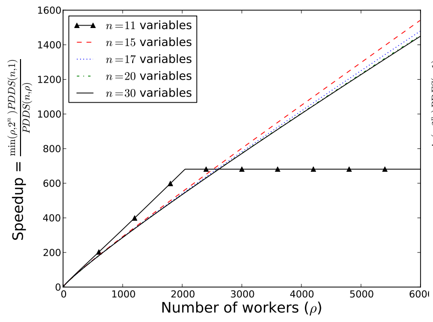
$$= \begin{cases} (4 + \log_2 \rho)2^n - \rho(n - \log_2 \rho + 3) & \text{if } \rho \leq 2^n \\ (n+1)2^n & \text{otherwise} \end{cases} \tag{13}$$

In comparison, as reported in [3], equation (14) shows the number of node visits done by PLDS when searching a complete binary tree.

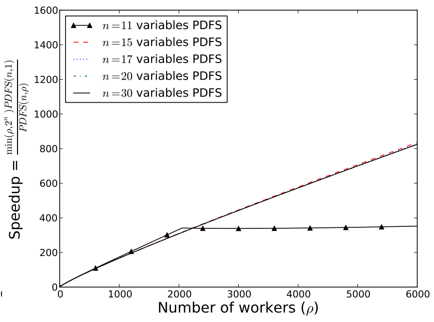
$$\mathrm{PLDS}(\rho, n) = 2^n + 2^n \sum_{i=1}^{n} \sum_{k=0}^{i} \frac{1}{2^i} \min(\rho, \binom{i}{k}) \tag{14}$$

5.5 Speedup analysis

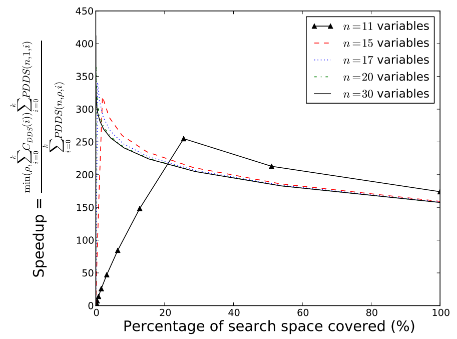
The *speedup* is the ratio between the time for a single worker to accomplish a task over the time required for ρ workers to accomplish the same task. We measure the time in number of node visits while supposing that all nodes have an equal processing time. A single worker visits $\text{PDDS}(n, 1)$ nodes to explore an entire search tree of n binary variables while ρ workers each visits $\frac{\text{PDDS}(n,\rho)}{\min(\rho,2^n)}$ nodes to collectively explore the entire search tree. We therefore have a speedup of $\frac{\min(\rho,2^n)\text{PDDS}(n,1)}{\text{PDDS}(n,\rho)}$. A similar computation applies for PDFS.



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



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



(c) Speedup of PDDS with $\rho = 512$ in function of the percentage of search space covered. Recalls $C_{DDS}(k)$ is the number of leaves processed at iteration k.

Fig. 2: Theoretical speedup of PDDS and PDFS algorithms.

Figure 2a and Figure 2b show the speedup for PDDS and PDFS. For $n = 11$ variables, the speedup stops growing after 2048 workers. Beyond this point, there are more processors than leaves in the search tree. Since any additional worker is an idle worker, the speedup reaches a plateau. The number of variables affects the performance of PDDS, especially when there are few variables. However, as the number of variables grows, the effects become negligible.

One can see from Figure 2a and Figure 2b that the speedups in function of n for PDDS and PDFS are almost linear. In fact, while analyzing the functions $\text{PDDS}(\rho, n)$ (Equation 10) and $\text{PDFS}(\rho, n)$ (Equation 8), 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 $\Theta(\frac{\rho}{\log_2 \rho})$.

PDDS shows a greater speedup than PDFS when a complete search of the tree is performed. However, it is uncommon in practice to completely visit a search tree. Actually, even in a centralized environment we expect DDS to find a solution sooner than DFS as we better exploit the value ordering heuristic.

Figure 2c shows the speedup obtained when the search is interrupted after some fraction of the search space has been covered. The speedup increases until it reaches a peak from where it decreases. 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 is the subtree in which the redundancy occurs.

5.6 Workload analysis

Theorem 1. *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. If a branch of the tree is pruned, all the nodes under this branch are removed. Each leaf in the removed subtree are associated to a worker w and to an iteration k in which DDS visits the leaf. The leaves belonging to the same iteration are assigned to the workers in a round-robin fashion. Therefore, for the same iteration k, the workload among the workers differs by at most one. Since there are $n + 1$ iterations ($k = 0..n$), one concludes that the accumulated workload gap is bounded by $n + 1$.

However, the leaf visited at iteration 0 and the leaf visited at iteration 1 cannot be both 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. $\square$

Theorem 1 shows the benefit of implicitly assigning leaves to workers in a round-robin fashion.

6 Experiments with industrial data

We carried out experiments with industrial data for an integrated planning and scheduling problem from the forest-products industry. Planning and scheduling lumber finishing operations is very challenging for the following reason: (1) the manufacturing operations lead to co-production (we simultaneously produce many different types of products from a single input) and (2) there are many alternative operations that can be used to transform a given raw product (each operation leads to a different basket of products). The result is that each operation contributes to partially fulfill many orders at the same time and each order can be fulfilled by many operations. The lumber finishing problem is fully described in [19] which provides a good heuristic to solve this problem. This heuristic inspired a search procedure (variable/value selection heuristics) [20] that allowed a constraint programming model to outperform standard mathematical programming. In [3], DFS, LDS and PLDS were compared using industrial data. LDS outperformed DFS, and PLDS allowed an impressive speedup and and solution quality that were never obtained before.

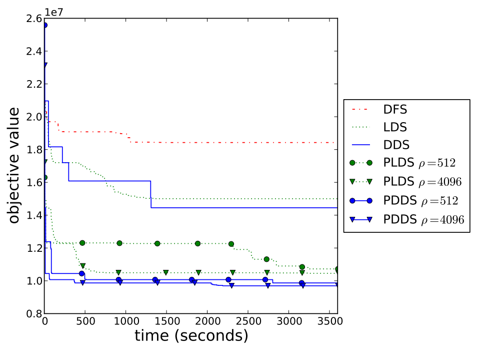
Using the same datasets and methodology as in [3], we compare DFS, LDS, PLDS, DDS, and PDDS. The search only considers integer variables. Once the values for these variables are known, the remaining continuous variables define a linear program that can be easily solved to optimality using the simplex method. Therefore, each time a valid assignment of the integer variables is obtained, we consider we have reached a leaf in the search tree and a linear program is solved to evaluate the value of this solution. The linear programs were solved using CPLEX version 12.5.

We used Colosse, a supercomputer with 7680 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.

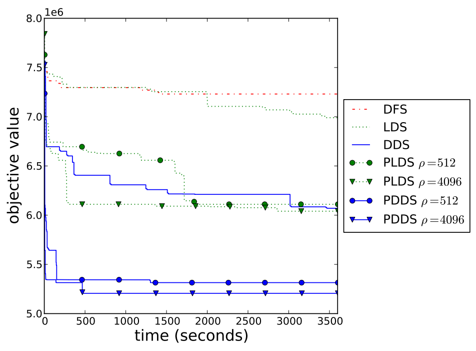
Figures 3a to 3c show the objective value (minimizing backorder costs) according to computation time (maximum one hour) for 1, 512 and 4096 workers. DDS and PDDS with one worker showed the same performance. For this reason we omit the latter in the chart. The same comment applies to LDS (PLDS) and DFS (PDFS).

As expected, 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 discrepancies at the top of the search tree first leads to better solution faster. The centralized DDS even catches up PLDS running on 512 workers (see Figure 3b).

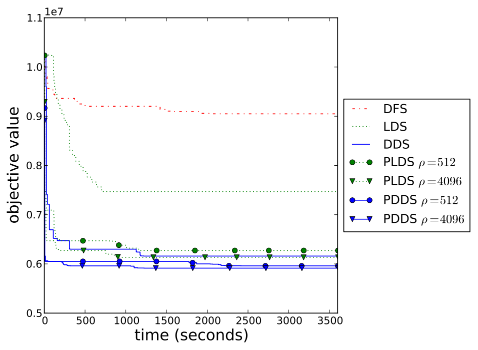
PDDS using 4096 workers obtains solutions of quality that was never reached before. The gap between the solutions obtained with PLDS (4096 workers) and PDDS (4096 workers) is considerable from an industrial point of view. PDDS has reduced the backorders by a ratio ranging between 68% and 85% when compared



(a) M1 dataset



(b) M2 dataset



(c) M3 dataset

Fig. 3: Best objective value found depending on time for various datasets.

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. The speedup is computed as the ratio of the number of leaves visited by multiple workers divided by the number of leaves visited by one worker.[2] The true speedup measure based on wall-clock time is not used since it was not practical from an experimental point of view. For example, with dataset M1, DDS visits 615 leaves in one hour. The same leaves are visited in a few seconds with PDDS 1024 workers while 409 workers are idle. With the same dataset, PDDS 1024 workers visits 614885 leaves in 10 minutes which is equivalent to 110 days of work for a centralized DDS. Experiments with such high difference in task size would not lead to any significant results.

[2] The super-linear speedup obtained on instance M1 with $\rho = 512$ workers is explained by the uneven time required to solve the linear programs associated to each leaf. The average solving time is greater for the leaves both reached by DDS and PDDS than for the additional leaves visited by PDDS. Other instances do not show this behavior.

dataset	ρ	speedup	$\overline{\chi}$	σ_χ	$max(\chi) - min(\chi)$
M1	512	517.9	622.04	1.94	12
M1	1024	1001.0	601.21	4.95	24
M1	2048	2008.2	603.06	4.68	24
M1	4096	4087.0	613.66	4.37	24
M2	512	475.0	756.11	2.68	14
M2	1024	945.4	752.41	2.44	14
M2	2048	1886.7	750.82	2.53	17
M2	4096	3732.8	742.72	2.76	18
M3	512	469.0	830.79	11.16	84
M3	1024	926.5	820.67	13.55	90
M3	2048	1844.2	816.75	11.8	85
M3	4096	3695.4	818.3	14.29	113

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

Even if the whole search tree is not visited, we wanted to measure the difference of workload between workers, in terms of visited leaves. Let χ_j be the number of leaves processed by worker j. Let $min(\chi)$ be the minimum value of χ_j for every $j \in 0, 1, \ldots, \rho - 1$ and $max(\chi)$ the maximum. Let $\overline{\chi}$ be the average number of leaves visited by each worker. The standard deviation of the number of leaves visited by each worker is σ_χ. This measure shows that processors have visited roughly the same number of leaves.

7 Conclusion

We proposed a parallelization of DDS that we named Parallel Depth-bounded Discrepancy Search (PDDS). We theoretically showed that PDDS scales to unlimited number of workers until there are more workers than leaves in the search tree, thanks to the fact that there is no communication between the workers. When only a part of the tree is searched, as it is most common, the instances with more variables lead to a greater speedup.

We theoretically analyzed the numbers of node visits of DDS and PDDS. These numbers of node visits are used to analyze the theoretical speedup of PDDS. We showed that the number node visits of DDS is the same as LDS when visiting a complete search tree.

Finally we used an industrial problem from the forest-products industry to experiment with PDDS. We showed that PDDS consistently performs better than PLDS in our industrial context for which a good heuristic was provided. From an industrial point of view, the computation time is reduced and the solution quality is enhanced.

References

1. de la Banda, M.G., Stuckey, P.J., Van Hentenryck, P., Wallace, M.: The future of optimization technology. Constraints (2013) 1–13
2. Régin, J.C., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Proceedings of the 19^{th} International Conference on Principles and Practice of Constraint Programming (CP 2013), Springer (2013) 596–610
3. Moisan, T., Gaudreault, J., Quimper, C.G.: Parallel discrepancy-based search. In: Proceedings of the 19^{th} International Conference on Principles and Practice of Constraint Programming (CP 2013), Springer (2013) 30–46
4. 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
5. Walsh, T.: Depth-bounded discrepancy search. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 1997). (1997) 1388–1393
6. 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
7. 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)
8. Shylo, O.V., Middelkoop, T., Pardalos, P.M.: Restart strategies in optimization: Parallel and serial cases. Parallel Computing **37** (2010) 60–68
9. Hamadi, Y., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation **6** (2009) 245–262
10. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for sat. Journal of Artificial Intelligence Research (JAIR) **32** (2008) 565–606
11. Michel, L., See, A., Van Hentenryck, P.: Transparent parallelization of constraint programming. INFORMS Journal on Computing **21** (2009) 363–382
12. 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
13. Menouer, T., Le Cun, B., Vander-Swalmen, P.: Partitioning methods to parallelize constraint programming solver using the parallel framework Bobpp. In: Advanced Computational Methods for Knowledge Engineering. Springer (2013) 117–127
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. Yun, X., Epstein, S.L.: A hybrid paradigm for adaptive parallel search. In: Principles and Practice of Constraint Programming, Springer (2012) 720–734
16. Korf, R.E.: 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) 286–291
17. 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

18. Furcy, D., Koenig, S.: Limited discrepancy beam search. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2005). (2005) 125–131
19. 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)
20. 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 and Operations Research **38** (2011) 1238–1250