A Load Balancing Procedure for Parallel Constraint Programming

Simon Boivin
Bernard Gendron
Gilles Pesant

July 2008

CIRRELT-2008-32

A Load Balancing Procedure for Parallel Constraint Programming

Simon Boivin[1,2,*], Bernard Gendron[1,3], Gilles Pesant[1,2]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT)

[2] Département de génie informatique, École Polytechnique de Montréal, C.P. 6079, succursale Centre-Ville, Montréal, Canada H3C 3A7

[3] Department of Computer Science and Operations Research, Université de Montréal, C.P. 6079, succursale Centre-Ville, Montréal, Canada H3C 3J7

Abstract. In this paper, we design a parallel Constraint Programming (CP) method to solve Constraint Satisfaction Problems (CSP). We use some attributes induced by the CP model of a CSP to improve the load balancing procedure embedded in the parallel tree-based search algorithm. Load balancing is improved by using specialized branching heuristics and workload estimators based on the CP model. More precisely, solution counting is used as an approximation of the computational size of the tasks in the parallel CP solver. This approximation of the workload of a task improves the work decomposition or work splitting procedure as well as the distribution of the tasks in the parallel algorithm. Experimental results indicate that this information speeds up the parallel exploration of the search tree.

Keywords. Parallel computing, constraint programming, constraint satisfaction problems, load balancing.

Acknowledgements. The authors would like to thank the members of the Quosséça lab at École Polytechnique de Montréal especially Claude-Guy Quimper and Allessandro Zanarini for their precious help.

---

* Corresponding author: Simon.Boivin@cirrelt.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec,
Bibliothèque et Archives Canada, 2008

1 Introduction

Constraint Programming (CP) is a useful computational approach to solve combinatorial problems. It is used by several exact and heuristic solvers and has shown good performance when solving hard Constraint Satisfaction Problems (CSP)[37]. CP explores a search tree to find one or many solutions to the problem. In this paper, we use parallel computing to speed up the search for solutions. Since CP uses the well known Backtracking or Depth-First Search (DFS) procedure, the speedups obtained by the parallel CP solver depend on the idle time due to load imbalance, on the search overhead and on the communication overhead [23]. Load imbalance appears in a parallel CP solver since it is difficult to estimate the workload of the tasks and equally distribute them to the processors. The search overhead is the amount of extra work performed by the parallel algorithm in terms of the number of nodes explored. More precisely, it is defined as the ratio between the number of nodes explored by the parallel algorithm and the one explored by the sequential algorithm. The communication overhead is the amount of time wasted by the processors in communication and synchronization. In this paper we focus on the load imbalance problem and propose new branching heuristics and workload estimators to improve load balancing in a parallel CP solver.

We use some attributes of the CP model to improve the decomposition of the search space and the load balancing. More precisely, we use solution counting to approximate the size of the tasks in a parallel CP solver. Solution counting was already used by Zanarini and Pesant as a constraint-centered heuristics to guide the search for a solution to a CSP [39]. These heuristics exploit solution counting information extracted from individual constraints in order to guide the search toward parts of the search tree that are more likely to contain a large number of solutions.

The main contribution of this paper is to show that solution counting can be used as an estimator of the size of a task in a parallel CP solver and that it provides an efficient procedure to split the workload on parallel and distributed architectures. Even though we focus on solution counting for individual constraints and not for the overall problem, experimental results show that this information leads to a much more balanced workload compared to traditional tree splitting based on standard branching heuristics.

Another contribution is to show that the search strategy used in the CP solver can be modified to better fit on a parallel architecture. We propose specialized heuristics which use solution counting to decompose the problem into a set of disjoint tasks of approximately the same computational size. In every other parallel CP solver proposed [25, 33, 20], the search strategy attempts to reproduce the serial search.

Finally, we design a parallel CP solver which can be used on top of any sequential CP solver. Most of the parallel CP solvers presented in the literature were developed from their sequential CP solver designers, giving them direct access to the internal data structures. In our work, we use parallel computing on top of the CP solver without direct access to its internal data structures.

We apply our method to the problem of finding all the solutions of a combinatorial problem. Finding all the solutions to a CSP can be interesting in applications where the user wants to compare many of them and make his own choice. Finding all the solutions to a CSP is also interesting for metaheuristics algorithms. These algorithms use local moves to evaluate the neighborhood of a solution. Some local moves imply exploring a large neighborhood [34]. This neighborhood can be modeled as a CSP and solved by enumerating all solutions to this model [26]. Another application appears in the hardware and software verification where the designers want to verify their conformities with the specifications [5].

In Section 2 we present related work about tree based search parallelization strategies and CP. Section 3 presents the task generation and distribution procedures used in our parallel CP-based search. Section 4 describes the specialized branching heuristics and the workload estimators based on the CP model proposed. Section 5 presents some implementation details about the parallel architecture used. Section 6 presents and analyzes computational results on the Quasigroup With Holes problem. Section 7 concludes and presents some extensions of this work.

2 Related Work

2.1 Tree Based Search Parallelization Strategies

Several tree based search algorithms have been parallelized. Backtracking [23], Depth-First Search (DFS) [22, 15, 31], Iterative Deepening [6] and Branch-and-Bound algorithms (B&B) [32, 24, 35, 9, 18, 3, 29] were studied to find effective parallel counterparts. Most of the solutions proposed use a decomposition of the search tree search. Our method is inspired by these parallel algorithms and particularly by the parallel B&B algorithm and by the parallel DFS.

The Branch and Bound algorithm consists of an implicit enumeration method for solving optimization problems. It proceeds by a decomposition or a subdivision, which is done by the *branching* operation, of the set of feasible solutions into several subsets. Each of these subsets define a subproblem that is solved by the *bounding* operation which gives a bound on the optimal value of each of these subproblems. This bound is used by the algorithm to fathom some subproblems if they cannot improve the solution quality according to the best solution known (the incumbent).

Parallel B&B algorithms can be classified according to three strategies [9]. First, the operations on the subproblems can be done in parallel. Bounding and branching operations are then decomposed and sub-operations are done simultaneously on several processors. The second parallel strategy is to build the search tree in parallel by computing several subproblems simultaneously. Thirdly, several B&B trees, defined by different branching or bounding operations, can be traversed in parallel. In this paper, we focus on the second parallel strategy.

Several strategies were also developed to parallelize the DFS algorithm. According to the literature on parallel DFS [22, 15, 31], two important issues must be answered when parallelizing DFS: how to generate the tasks and how to distribute them?

The *task generation procedure* defines *what* a task is in the parallel algorithm. The procedure also defines *when* the tasks are generated. Two examples found in the literature on parallel DFS are the tree-splitting (also known as stack-splitting) [22] and the search-frontier splitting (also known as fixed-packet DFS) [31, 17].

As depicted in Figure 1, in the tree-splitting procedure, every pending node in the search tree is a task of the parallel tree-based search algorithm. The tasks are created through the branching procedure of the tree-based search algorithm. Task generation is then dynamically applied on each node of the search tree.

In a search-frontier splitting the tasks are defined by exploring a part of the search tree until a given number of tasks are created. The task generation procedure is performed at the beginning of the parallel tree-based search. The tasks generated by this procedure are the sharing units of the parallel algorithm. The number of tasks is then statically fixed and a task cannot be re-split. Search-Frontier splitting is represented in Figure 2. A possible strategy for the task generation procedure is to explore the search tree in a breadth-first search manner until a given depth is reached [31, 17]. The depth of the tasks generated is then equal. After the task generation procedure, the task solving operation is asynchronously performed on the processors.

The *task distribution procedure* defines how the workload is balanced during the parallel tree-based search. The workload can be balanced only at the beginning of the search: this strategy is known as static load balancing. The workload can also be balanced at the beginning and during the search when a processor becomes idle: this strategy is known as dynamic load balancing.

In a **static** load balancing procedure, the task generation procedure must define at least $nProc$ (the number of processors used) tasks. The task distribution procedure will assign them on the processors. For example, in a search-frontier splitting strategy with static load balancing (Figure 2), a first set of tasks is generated by the task generation procedure. This set of tasks is split into $nProc$ subsets and distributed to the processors. The static load-balancing implies that the tasks are solved asynchronously on each processor without communications or work exchanges. It means that an idle processor cannot request work from a busy one.

In a **dynamic** load balancing procedure, the workload of a busy processor can be shared with an idle one. A more sophisticated protocol than the one used in static load balancing must be defined to perform these work exchanges. First, when an idle processor requests work from a busy one, a *work-partitioning* procedure defines how to split the workload of the busy processor into two parts. This procedure often uses an approximation of the workload of a task to equally split the work pool. After work partitioning, the first part of the workload is kept by the busy processor and the second part is sent to the idle processor.

Work partitioning depends on the parallel strategy used. In the tree-splitting strategy (Figure 1), the donor or the busy processor will split in two parts its set of pending nodes. Additional features can be used to improve this work-partitioning procedure. As shown in Figure 1, a *cutoff depth* can be added to avoid sharing pending nodes which represent few amount of work [22]. In a tree-splitting with cutoff depth, only the pending nodes with a depth lower than the cutoff are candidates to be shared with an idle processor. For example, in Figure 1 the nodes 3, 4, 5 and 6 are the pending nodes of processor p_0 which receives a work request from processor p_1. Since the cutoff depth is three, 5 and 6 are the only candidates for sharing with p_1.

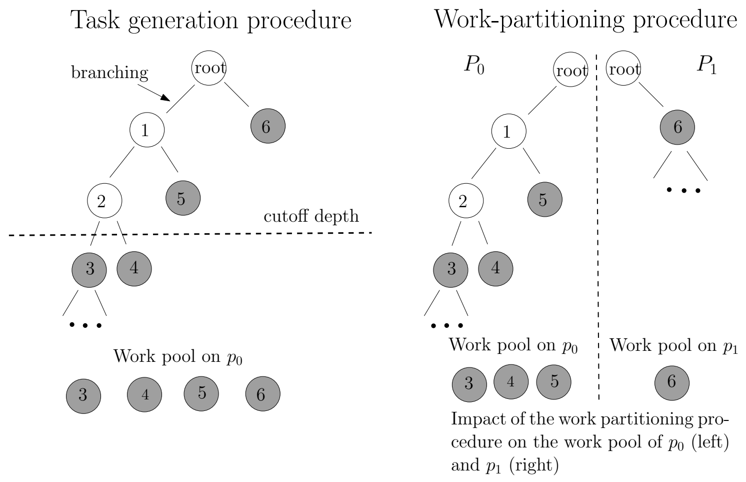
Task generation procedure
Work-partitioning procedure
branching
root
cutoff depth
Work pool on p_0
P_0
P_1
Work pool on p_0
Work pool on p_1
Impact of the work partitioning procedure on the work pool of p_0 (left) and p_1 (right)

Figure 1: Tree-splitting procedure and cutoff depth

In search-frontier splitting with dynamic load balancing, the donor or the busy processor will split its set of tasks (generated at the beginning by the task generation) into two equal parts. Figure 2 presents this procedure. In this example, processor p_0 receives a work request from processor p_1. The work pool of p_0 is $\{i, k, m, o, q\}$ and the approximation of the workload is equal for each task. The five tasks are then split into two subsets: three tasks for processor p_0 and two tasks for processor p_1.

Another important issue in a dynamic load-balancing procedure is the *initiation of the work exchanges*. The dynamic load-balancing procedure must define the communication rules between the idle and the busy processors. In the literature [14], this choice can be made through a *target* variable which can be global (global round robin) or local (asynchronous round robin) that defines the processor to request work from. After each request the *target* variable is incremented ($(target + 1) \bmod nProc$). The choice of the busy processor can also be random (random polling). These communication rules can be applied to tree-splitting as well as to search-frontier splitting.

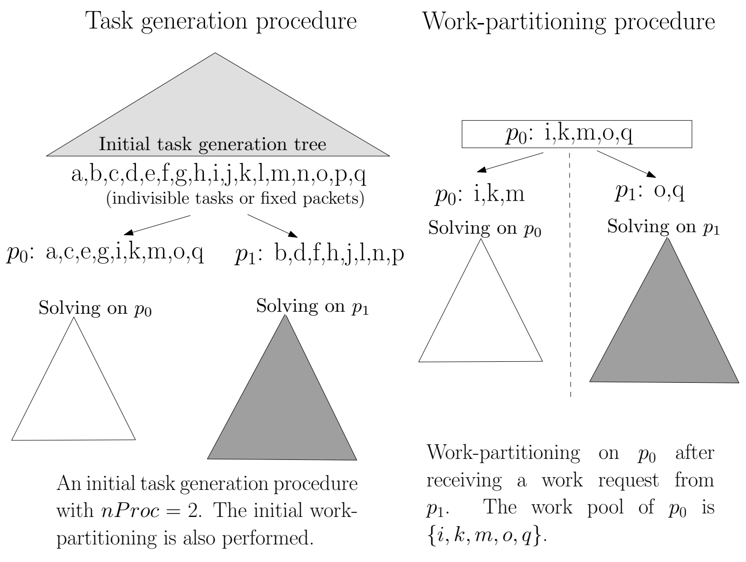
Task generation procedure
Work-partitioning procedure
Initial task generation tree
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q
(indivisible tasks or fixed packets)
p_0: a,c,e,g,i,k,m,o,q
p_1: b,d,f,h,j,l,n,p
Solving on p_0
Solving on p_1
An initial task generation procedure with $nProc = 2$. The initial work-partitioning is also performed.
p_0: i,k,m,o,q
p_0: i,k,m
p_1: o,q
Solving on p_0
Solving on p_1
Work-partitioning on p_0 after receiving a work request from p_1. The work pool of p_0 is $\{i, k, m, o, q\}$.

Figure 2: Search-frontier splitting procedure

2.2 Constraint Programming

Constraint Programming aims to solve combinatorial problems. It uses the constraints of the problem to narrow the search space. CP shows good performances for solving CSPs. For example, CP was already used in scheduling [2], sport scheduling [7], bioinformatics [38], financial portfolio design [8] and software verification [5]. Several CP applications can also be found in Wallace [37]. CP is also a good candidate for hybridizations with other methods, in the field of operations research, like Integer Programming [21] or metaheuristics [36]. It can also be used as a subproblem solver to handle the subproblem part of a column generation procedure [13].

In CP, a problem P is stated as three sets: $P = (X, D, C)$ such that X is the variable set, D the set of the domains of possible values for each variable and C the set of constraints which restrict the variable-value assignments. A CSP and a search procedure are used to describe a CP program. The constraints of the CP program removes inconsistent values $v \in D_i$ from the set of possible values D_i when v cannot be a consistent assignment to X_i according to C and the state of every D_j such that $j \neq i$ (inference or constraint propagation). Research made by the CP community on the constraints substructures (binary, linear, global constraints ...) allows CP programs to detect the inconsistencies and will explore a smaller search space than an exhaustive search.

The CP program uses the search procedure to branch on variable-value assignment and, by doing that, builds the search tree. Two choices must be made by the search procedure: which is the next variable to assign and which is the value to assign to this variable? These choices are known as variable and value branching heuristics and have an impact on the time spent to find a solution or all the solutions to the CSP. Some general branching heuristics were developed like choosing the variable with the minimal domain size ($MinDom$) or choosing the value with the minimal number of conflicts with the constraints of the CSP ($MinConflicts$).

Three CP solvers were adapted to parallel computing. Notice that these solvers build the search tree in parallel by computing several subproblems simultaneously. These parallel CP solvers use the tree splitting decomposition procedure.

ILOG Parallel Solver was presented in 1999 by Perron [25] and is a parallel implementation of the search procedure used in ILOG Solver. Unexplored ("open") nodes of the search tree are distributed over a set of local work pools, one for each processor. A communication layer prevents and solves the starvation problem (which appears if a processor becomes idle) and balances the load by exchanging nodes between the processors' local work pools when starvation appears. It also detects termination. Few details are given about when and how the nodes exchanges are performed. This parallel solver uses a shared memory architecture and the use of a distributed computing architecture is one of the perspectives of this work.

A parallel implementation of the *Mozart* solver was presented in 2000 by Schulte [33]. It uses a manager / worker architecture such that the manager initiates the work, collects the solutions, receives the job requests, sends the shared requests and detects termination. Each worker explores its subtree, sends its new solutions, and shares its work with an idle processor. A processor chooses the highest node in its search tree when sharing its work with an idle one. The depth of a node is then the estimation of its computational size. In this case, a manager is dedicated to load balance the parallel search.

Parallel COMET was presented in 2007 by Michel *et al.* [20] and is a parallel implementation of the search procedure used in COMET. The authors used a shared memory architecture. A local work pool is assigned to each processor and a global work pool allows work stealing between processors. Work stealing is used as a communication protocol between an idle processor and a busy one. A shared variable *steal* is used to notify that a processor is idle. When this variable becomes *true*, a processor generates some new nodes and puts them on the global work pool for sharing with the idle one. The choice of the subproblem to share is based on its depth in the search tree.

3 The Parallel CP-Based Search Architecture

The parallel CP-based search architecture used is inspired by the search-frontier splitting parallel strategy for DFS. However, we made our own parallelization strategy to better fit with some particularities of our parallel CP search. The parallelization strategy generated an initial set of tasks and distributed them on the processors as in search-frontier splitting. However, a task is not an indivisible piece of work and can be re-splitted during the search. Several particularities of our parallel CP search have motivated this strategy.

First, remember that we want to perform parallelism on top of the CP solver without accessing its internal elements like the search tree, the variables, the domains of possible values and the constraint propagators. In this case, using a tree-splitting strategy is definitely prohibited since we do not have access to the tasks or the sharing units of the parallel search (node of the search tree). Secondly, remember that we want to use solution counting of individual constraints of the problem to generate and to distribute the tasks. Solution counting procedure implies additional computations which are not always useful to solve the problem. Considering this, defining a huge number of tasks at the beginning of the search, as performed in search-frontier splitting, would imply additional parallel cost which increases the parallel runtime and decreases the parallel speedup.

For these reasons, we performed a task generation procedure which dynamically generates the tasks during the parallel CP-based search exploration. In our parallel CP-based search procedure, some tasks or subtrees rooted at a node (or a task) are completely explored by the CP solver which is used as a black-box as in search-frontier splitting. During this operation, the CP solver handles the constraint propagation and the search procedure. Some other tasks are kept available for sharing with an eventually idle processor. As in tree-splitting, these tasks are created by a branching heuristic. This specialized branching heuristic aims to generate tasks with an approximately equal workload and can be different from the one used in the task solving procedure.

In the following, we describe our task generation procedure and our task distribution procedure. Two procedures generate the parallel tasks: the initial task generation procedure and the task solving procedure. Notice that, each task t is represented by a set of k variable-value equalities $\{x_1(t) = d_1, \dots, x_k(t) = d_k\}$ which represent its path from the root node.

The *initial task generation procedure* performs a breadth-first search on the search tree to generate at least $nProc$ tasks. This operation is performed only on one processor. During this operation, a n-ary tree is traversed. The CP solver is used as a black-box to propagate the constraints of the problem. A specialized branching heuristic, which will be defined in Section 4, chooses the next variable to expand. The goal of this heuristic is to generate tasks with an approximately equal workload.

Figure 3 represents the initial tasks generation. Gray nodes are pending nodes or tasks generated. From the root node, a variable $x_i \in X$ is chosen according to a heuristic h. Since the domain of x_i is $\{a, b, c, d\}$ four tasks are generated. According to the number of processors $nProc$ the task generation will explore the tasks $\{x_i = a\}$ and $\{x_j = b\}$ until enough tasks are available.

The *task solving procedure* also deals with the generation of new tasks. Its responsibility is to keep work available for sharing with an eventually idle processor. We handle this problem by performing work-partitioning before computing the last task in a work pool. In this case, work-partitioning is performed by applying the specialized branching heuristic on this task. After performing work-partitioning, the processor will compute one of the new tasks generated.

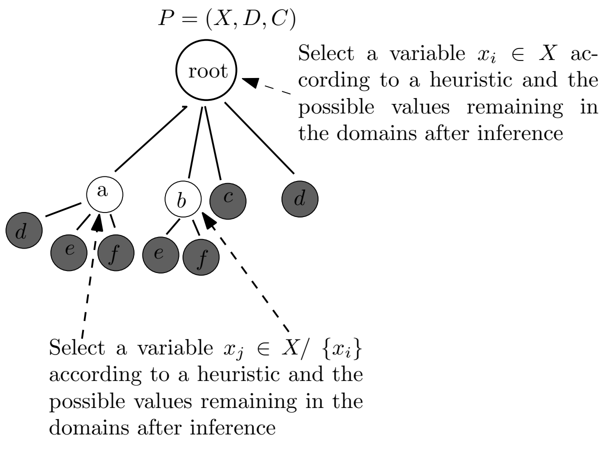
$P = (X, D, C)$
root
Select a variable $x_i \in X$ according to a heuristic and the possible values remaining in the domains after inference
a
b
c
d
d
e
f
e
f
Select a variable $x_j \in X/\ \{x_i\}$ according to a heuristic and the possible values remaining in the domains after inference

Figure 3: Initial Task Generation

Figure 4 represents this operation on a last task which is $\{(x_i = b), (x_j = f)\}$. In this example, the branching operation is performed and two new tasks are created. The first task is sent to the CP solver which explores the entire subtree rooted at this node. The second one is kept as “available for sharing”.

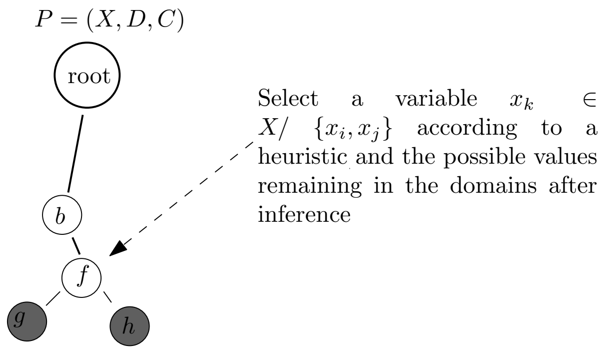
$P = (X, D, C)$
root
Select a variable $x_k \in X/\ \{x_i, x_j\}$ according to a heuristic and the possible values remaining in the domains after inference
b
f
g
h

Figure 4: Last task computation before solving

Briefly, we can state that a task is defined as an open node generated by a specialized branching heuristic. The task generation procedure is performed at the beginning of the search and during the search by branching on the last task available in a work pool before solving it.

The *task distribution procedure* uses a bin packing procedure to split the initial set of tasks into $nProc$ subsets (beginning of the search) or into two subsets (when a processor receives a work request from an idle one). Indeed, the task distribution problem can be modeled as a bin packing problem where the bins are the processors, the objects are the tasks, and an object volume corresponds to a task weight [19]. A weight function $w(t)$, which will be defined later, is used to compute the size of a task (Section 4). The set of tasks assigned to processor p or its work pool is denoted by Ω_p and the sum of the weights of these tasks is denoted by $L(\Omega_p) = \sum_{t \in \Omega_p} w(t)$. The objective is to find an assignment of the objects (tasks) to the bins (processors) in such a way that the volume (weight) in each bin is about the same. We use the minimization of the standard deviation over the workload of the processors as the objective function. A similar problem, where the objective is to minimize the makespan, is the *minimum multiprocessor scheduling problem* [11, 1, 4].

The distribution of the tasks to the processors is not a part of the CSP to solve. It also represents additional computations which increase the parallel runtime and decrease the parallel speedup. Consequently, we chose to use a heuristic algorithm. Algorithm 1 describes this task distribution procedure. As found in the literature on the minimum multiprocessor scheduling problem, the *Largest Processing Time algorithm* (LPT) can be used to solve the tasks distribution problem. Moreover, this algorithm gives a sharp bound $(\frac{4}{3} - \frac{1}{3 \times nProc})$ to the optimal value of the minimum multiprocessor scheduling problem [11]. The particularity of our method is to allow task generation by performing the branching operation on the heaviest task to better balance the workload. The procedure sorts the tasks in the work pool of a busy processor in decreasing order of their weights. It assigns each of them to the least loaded processor according to their current workload $L(\Omega_p)$. The algorithm selects the least loaded processor since it will be the first idle processor according to the approximation of the workload. When an assignment of the tasks to the processors is defined, the algorithm computes the standard deviation (σ) on the weights of all processors. The standard deviation must be less than or equal to a given threshold th. Notice that if the bin packing procedure is not able to distribute the tasks equally enough, according to the weight of the tasks generated and the threshold used, it calls the specialized branching operation on the heaviest task in the work pool and re-executes the bin packing procedure until the tasks can be distributed (line 10-12).

4 CP Based Branching Heuristics and Workload Estimators

A question remains open when considering the parallel CP based architecture: which heuristic is used to generate the tasks (branching or node expansion) and which workload estimator can be used in the distribution procedure? In the next section we present the concept of solution counting which is proposed as an answer.

4.1 Solution Counting

Counting solutions of CSPs at the constraint level was proposed by Pesant [28] and adaptation to a *constraint-centered heuristic* was proposed by Zanarini and Pesant [39]. The main idea is to use information about the number of solutions

Algorithm 1 Task distribution procedure

```
Require: a processor $k$ s.t. the work pool $\Omega_k$ has to be split
         a set of idle processors $V$
         a threshold $th$
Ensure: a work pool $\Omega_p$ for all $p \in V$ s.t. $\sigma(\{L(\Omega_p)$ s.t. $p \in V\}) \leq th$
1: $T \leftarrow \Omega_k$
2: $\Omega_k \leftarrow \emptyset$
3: $V \leftarrow V \cup k$
4: loop
5:   for all $t \in T$ sorted decreasingly by $w(t)$ do
6:     choose a processor $p \in V$ s.t. $L(\Omega_p)$ is minimal
7:     $\Omega_p \leftarrow \Omega_p \cup t$
8:   end for
9:   if $\sigma(\{L(\Omega_p)$ s.t. $p \in V\}) > th$ then
10:    Select task $t \in T$ s.t. $w(t)$ is maximal
11:    Split $t$ into $t_1, \ldots, t_l$
12:    $T \leftarrow T \cup \{t_1, \ldots, t_l\}$
13:  else
14:    return $\Omega_p$ s.t. $p \in V$
15:  end if
16: end loop
```

of individual global constraints to guide the search procedure. The concepts of *solution count* and *solution density* are presented in the following:

Definition 1 *(solution count) Given a constraint $\gamma(x_1, \ldots, x_n)$ and respective finite domains D_i $1 \leq i \leq n$, we denote by $\#\gamma(x_1, \ldots, x_n)$ (or simply $\#\gamma$) the number of solutions of constraint γ.*

Definition 2 *(solution density) Given a constraint $\gamma(x_1, \ldots, x_n)$ and respective finite domains D_i $1 \leq i \leq n$, a variable x_n in the scope of γ and a value $d \in D$, we will call*

$$\tau(x_i, d, \gamma) = \frac{\#\gamma(x_1, \ldots, x_{i-1}, d, x_{i+1}, \ldots, x_n)}{\#\gamma(x_1, \ldots, x_n)}$$

the solution density of pair (x_i, d) in γ.

The authors presented several procedures to compute $\#\gamma(x_1, \ldots, x_n)$ for the AllDifferent constraint [30] and for the Regular constraint [27]. Since counting for the AllDifferent constraint is *#P-complete*, the authors presented a heuristic algorithm based on sampling. In the case of the Regular constraint, the filtering algorithm based on dynamic programming provides data structures which are used to exactly count the number of solutions [39].

They used three search heuristics to guide the search for a solution. These heuristics use solution count to follow a "fail-first principle." The Maximum Solution Density (*MaxSD*) heuristic chooses the variable-value assignment (x_i, d)

such that $\tau(x_i, d, \gamma)$ is maximum for some constraint γ. The Minimum Solution Count-Maximum Solution Density (*MinSC:MaxSD*) heuristic chooses the variable-value assignment (x_i, d) such that the variable x_i is in the scope of the constraint γ which minimizes $\#\gamma$ and the value d maximizes $\tau(x_i, d, \gamma)$. Finally, the Smallest Domain-Maximum Solution Density (*MinDom:MaxSD*) heuristic chooses in the set of minimum domain size variables the variable-value assignment (x_i, d) such that $\tau(x_i, d, \gamma)$ is maximum for some constraint γ.

The authors tested these heuristics on the Quasigroup with Holes problem (Partial Latin Square) and the Nonogram problem. The heuristic MinSC:MaxSD performs well on the nonogram problem and MaxSD performs well on the Quasigroup with Holes problem. They reduce significantly the average number of backtracks and the average solution time in comparison with classical branching heuristics. These results motivate the use of solution counting as a workload estimator in the load balancing procedure of a parallel CP search.

4.2 Task Generation and Workload Estimators Based on Solution Counting

In this section, we describe the heuristic used to generate the tasks and the weight function used to estimate and to balance the workload during the task distribution to the processors. We suggest specialized heuristics used in the initial task generation phase and before to solve the last node in a work pool. These heuristics do not follow the typical branching heuristics used in CP. The branching heuristics used in CP aim to find solutions quickly, not to split the tree into subtrees of similar computational size. A novelty of our work is to modify the branching heuristic and use solution count and solution density in order to generate tasks of approximately equal size.

We propose three heuristics that will be described in the following. Two heuristics exploit solution count to choose a constraint γ of the problem with a maximum or a minimum solution count $\#\gamma$. Maximizing the solution count aims to choose a part of the model in which a lot of solutions are found, possibly making it easier to distribute them among processors. Minimizing the solution count can be seen as a “fail-first choice” of the constraint.

After choosing this constraint γ, the heuristics choose a variable x_i, in the scope of γ, which minimizes the standard deviation σ over the densities $\tau(x_i, d, \gamma)$ of all values $d \in D_i$ (see Algorithms 2 and 3, where $\sigma(W)$ denotes the standard deviation over all numbers in set W). The aim of minimizing the standard deviation is to generate subtrees with approximately the same size. $MinConstMinSTD$ denotes the heuristic which chooses the constraint with the minimum number of solutions and the minimum standard deviation over the densities. $MaxConstMinSTD$ denotes the heuristic which chooses the constraint with the maximum number of solutions and the minimum standard deviation over the densities.

With these two heuristics, the weight of a variable-value assignment $x_i = d$

is equal to the solution density for this assignment:

$$w(\{x_i = d_i\}) = \tau(x_i, d_i, \gamma)$$

Algorithm 2 MinConstMinSTD - $\min \#\gamma$ and $\min \sigma(\tau(x_i, d, \gamma))$

input: $X' \subseteq X$, $D' \subseteq D$ and C the set of constraints
output: a variable branching decision var

```
1: $minSTD \leftarrow \infty$
2: Choose $\gamma \in C$ such that $\#\gamma$ is minimal
3: Let $S = \{x_i \in X' \mid |D'_i| > 1$ and $x_i$ is in the scope of $\gamma\}$
4: for all variables $x_i \in S$ do
5:    $W \leftarrow \{\tau(x_i, d, \gamma) \mid d \in D'_i\}$
6:    if $\sigma(W) < minSTD$ then
7:       $var \leftarrow x_i$
8:       $min \leftarrow \sigma(W)$
9:    end if
10: end for
11: return $var$
```

Algorithm 3 MaxConstMinSTD - $\max \#\gamma$ and $\min \sigma(\tau(x_i, d, \gamma))$

input: $X' \subseteq X$, $D' \subseteq D$ and C the set of constraints
output: a variable branching decision var

```
1: $minSTD \leftarrow \infty$
2: Choose $\gamma \in C$ such that $\#\gamma$ is maximal
3: Let $S = \{x_i \in X' \mid |D'_i| > 1$ and $x_i$ is in the scope of $\gamma\}$
4: for all variables $x_i \in S$ do
5:    $W \leftarrow \{\tau(x_i, d, \gamma) \mid d \in D'_i\}$
6:    if $\sigma(W) < minSTD$ then
7:       $var \leftarrow x_i$
8:       $minSTD \leftarrow \sigma(W)$
9:    end if
10: end for
11: return $var$
```

We propose a third heuristic, $MinDomMinSTD$, which minimizes the size of the domain of a variable as a first selector, and uses the standard deviation over the densities of the possible values of a variable as a tie breaker. Algorithm 4 describes this heuristic.

In this case, the weight of each variable-value assignment is equal to :

$$w(\{x_i = d_i\}) = \frac{\sum_{\gamma \in \Gamma} \tau(x_i, d_i, \gamma)}{|\Gamma|}$$

such that Γ is the set of constraints on the variable x_i.

These three heuristics compute the weight of a task t as the product of the weights of the assignments representing the task:

$$w(t) = \prod_{j=1}^{k} w(\{x_j(t) = d_j\})$$

These three heuristics are compared with the *MinDom* heuristic which chooses to expand the variable with the minimum domain size. In this case, the weight of each variable-value assignment $x_i = d$ is equal to $1/|D_i|$ and the weight of a task is also computed as the product of each variable-value assignment in the task.

Algorithm 4 MinDomMinSTD - $\min |D_i|$ and $\min \sigma(\tau(x_i, d, \gamma))$

```
input: $X' \subseteq X$, $D' \subseteq D$ and $C$ the set of constraints
output: a variable branching decision $var$
1: $minSTD \leftarrow \infty$
2: Let $S = \{x_i : |D'_i| > 1$ and minimum $\}$
3: for all $x_i \in S$ do
4:   for all $\lambda \in C : x_i$ is in the scope of $\lambda$ do
5:     $W \leftarrow \{\tau(x_i, d, \lambda) \mid d \in D'_i\}$
6:     if $\sigma(W) < minSTD$ then
7:       $var \leftarrow x_i$
8:       $minSTD \leftarrow \sigma(W)$
9:     end if
10:  end for
11: end for
12: return $var$
```

5 Implementation of the Parallel CP-Based Architecture

Computations are distributed over a set of task pools; one for each processor. The processors communicate by a message passing interface and a processor initiates the search by performing the initial task generation and by using the Bin Packing procedure to distribute the tasks on the processors.

A ring topology is used to handle termination and work exchanges but other topologies could be used depending on the number of processors and the speed of the network. In our case, we tested the algorithm on a 16 processor architecture which is organized as four nodes of quad processors. We use a ring since the communication is fast on this architecture and the number of processors is low. Termination detection is handled by adding the state (running, waiting and stop) of each processor into the token exchanged. The waiting state of a processor is used to notify the other processors of its starvation and to initiate

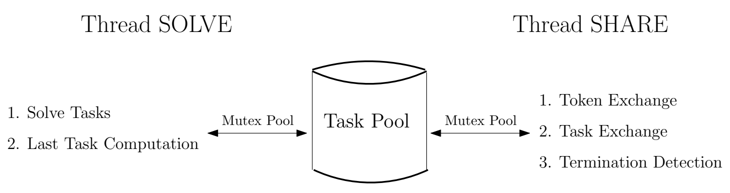

Figure 5: Thread architecture and pool management

work exchanges. When a processor receives the token with waiting as its own state it detects termination. The information about the approximated workload on the processors is also added in the tokens exchanged. This way, an idle processor will receive work from the most busy processor according to this approximation.

Since we used the CP solver as a black-box, we have no guarantee on when a given task will be completely solved. A processor can be requested for work while it performs the task solving procedure. This is the reason why we isolate the parallel communication from the CP solving procedure. The multi-threaded architecture used is shown in Figure 5. Parallelism is handled by a communication thread named $SHARE$. Another thread $SOLVE$ is used to perform the solving procedure on each task. Since a thread is dedicated to communicate with other processors, the processor can share available work with an idle processor even if it already called the computation of a task. The algorithm is then completely asynchronous.

The two threads are created on the same processor and communicate together by a shared *Pool* of tasks. The $SHARE$ thread adds tasks when it receives work from another processor. It also performs the distribution procedure when receiving a work request from another processor. The $SOLVE$ thread performs the computation of the parallel CP tasks by using the CP solver as a black-box and returns the number of solutions found. Mutual exclusion is handled by *mutex* in the pthread protocol.

6 Experimental Results

The objective of our computational experiments is to show that the specialized branching heuristics and the workload estimators proposed have a positive impact on the running time of the parallel algorithm. These experimentations will show the improvement in the parallel speedups.

We tested our parallel CP solver on the Quasigroup with Holes (QWH) problem. The QWH problem is a challenging benchmark which is well-studied by the CP community [10]. In the QWH problem, the solver must fill an $n \times n$ grid with a value between 1 and n such that each value appears exactly once in each row and once in each column. The problem is modeled by using an *AllDifferent* constraint for each row and for each column. We solved 60 QWH instances with a grid order of 12 and 40% of pre-filled cells. We generated

30 completely random instances and 30 balanced instances following Gomes et *al.* [10]. Random instances are generated by randomly removing some values from a complete QWH problem. Balanced instances are generated by removing some values from a complete QWH problem in such a way the number of holes is approximately the same for each row and for each column.

Following Michel et *al.* [20], we evaluate the parallel CP solver by enumerating all the feasible solutions of each of these instances. Rao and Kumar also used an enumeration of all the feasible solutions of the *hacker's* problem to evaluate efficiency of parallel backtracking [23].

Our CP solver for the QWH problem uses the minimum domain as variable branching heuristic and the minimum conflict as a value selector during the solving phase. Following preliminary experiments, the threshold used in the bin packing algorithm of the task distribution procedure is equal to 0.02. We tested the algorithm with 2, 4, 6, 8, 12 and 16 processors. We use the message passing interface (MPI) and parallel architecture with 16 processors (4 Quad-Core, Intel Xeon X7350, 2.93 GHz) and 16 GB DIMMs. The operating system used is Centos 5.0 and we use ILOG Solver 6.2 as the CP solver. We performed each test ten times since we run an asynchronous algorithm.

We use several performance measures to rank the search heuristics described in Section 5. Recall that the motivation of this paper is to speedup the CP search for solutions by improving load balance and decreasing idle time on the processors. The first performance measure used to rank the specialized branching heuristics and the workload estimators is the percentage of processor utilization. The percentage of utilization for a processor p named $U(p)$ is defined as :

$$U(p) = \frac{T_{tot}(p) - T_{idle}(p)}{T_{tot}(p)}$$

such that $U(p)$ is the percentage of processor utilization, $T_{tot}(p)$ is the elapsed time on processor p and $T_{idle}(p)$ is the idle time on processor p. The *idle time* on a processor ($T_{idle}(p)$) is the sum of each time a processor spends between the moment it sends a work request and the moment it receives some tasks. A timer is added in the $SHARE$ thread to count the idle time.

Another important load balance measure is the *number of work exchanges* needed during the search. A work exchange counter is also added in the $SHARE$ thread.

Achieving load balancing is an important goal of the procedure. However, attention must be paid to the *parallel search overhead factor*. The parallel search overhead factor is the ratio between the number of operations performed by the parallel algorithm and the number of operations performed by the sequential algorithm. Parallel search overhead is one cause of the speedup anomalies in parallel Branch-and-Bound algorithms [16] and in parallel DFS algorithms [23]. In this work, the number of operations used to compute the parallel search overhead is the number of choice points explored during the search.

Finally, we also presented the parallel speedups obtained by each branching heuristic and workload estimator. The speedup of a multi-processor system

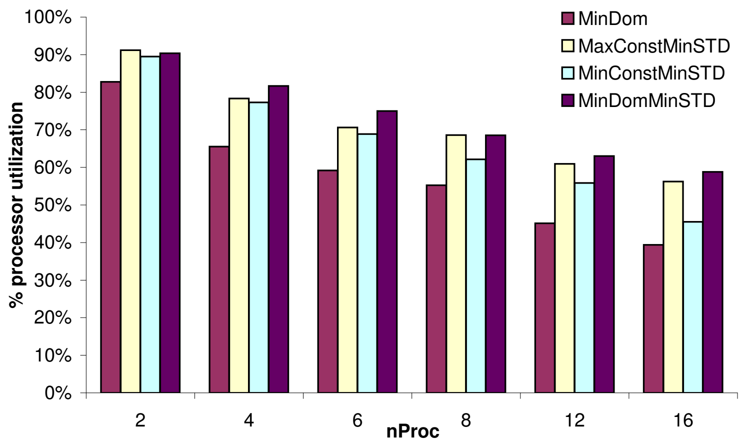

Figure 6: Balanced instances - Processors percentage of utilization

where P is the set of processors used is compute as :

$$S(P) = \frac{T_{SEQ}}{T_P}$$

such that T_{SEQ} is the time spent by the best sequential algorithm known and T_P is the time spent by the parallel algorithm tested.

We performed two different comparisons of the specialized branching heuristic and workload estimators proposed. First, we present the performance measures of the initial task generation procedure without work exchanges (static load balancing). Secondly, we present the performances of the specialized branching heuristic and workload estimators with work exchanges (dynamic load balancing).

6.1 Initial Task Generation Performance Measurement

Figures 6 and 7 present the percentage of processor utilization depending on the branching heuristic used for balanced and random instances respectively. According to this load balancing measures, the three branching heuristics which use solution counting give a better distribution of the workload than the $MinDom$ heuristic.

The percentage of processor utilization has an impact on the parallel speedups as we can observe in Figures 8 and 9. As we expect from a static load balancing procedure, the speedups are sublinear and processor utilization decreases when the number of processors increases. The heuristics $MinConstMinSTD$ and $MinDomMinSTD$, which give better percentage of processor utilization, give better parallel speedups than the $MinDom$ heuristic. This is not the case of the $MaxConstMinSTD$ which gives a better percentage of processor utilization but a lower parallel speedups than $MinDom$.

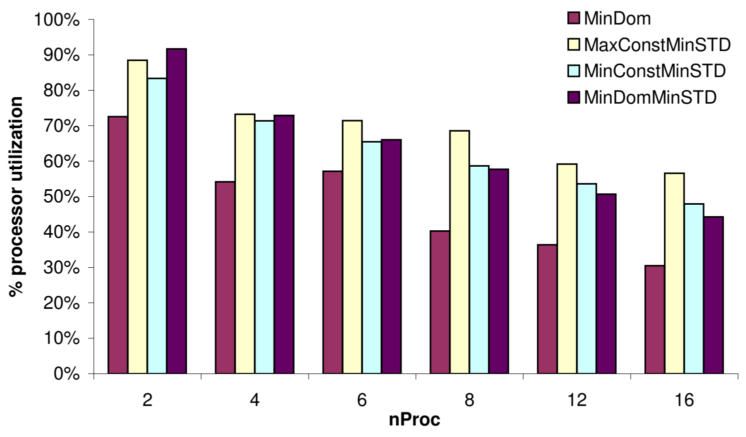

Figure 7: Random instances - Processors percentage of utilization

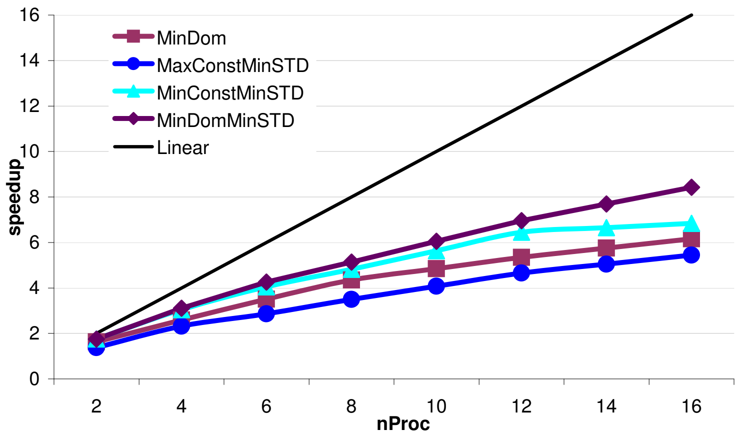

Figure 8: Balanced instances - Parallel Speedups

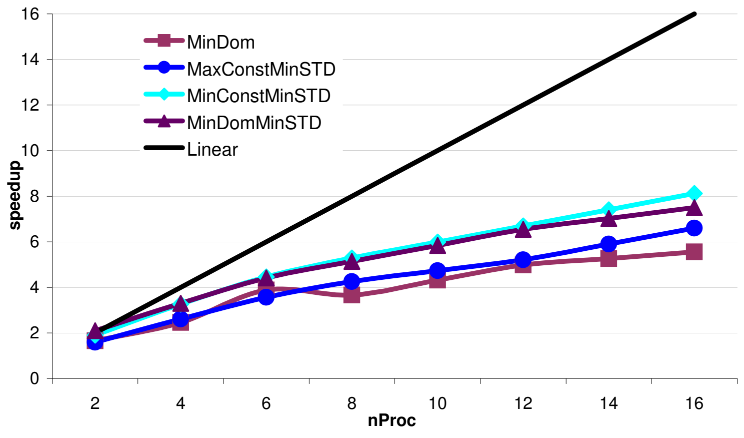

Figure 9: Random instances - Parallel Speedups

This situation can be explained by the parallel search overhead factor. Figures 10 and 11 present the parallel search overhead depending on the branching heuristic used. We observe that the branching choices made by the splitting algorithm have an impact on the size of the search tree explored. In the case of $MaxConstMinSTD$, we did not use a "fail-first" criterion at the top of the tree. A "fail-first" criterion consists of choosing the variable value assignment such that the search is more likely to fail or backtrack. By doing this, the search space (number of choice points explored by the search procedure) would be smaller and fail would appear at the top of the search tree [12]. This is the reason why $MaxConstMinSTD$ gives a better load balance but a worst solving time than the $MinDom$ heuristic. For this reason, the $MaxConstMinSTD$ heuristic will not be used during the dynamic load balancing experimentations.

6.2 Dynamic Load Balancing Performance Measurement

In the case of a dynamic load balancing, the branching heuristic and the workload estimator have an impact on the speedups by increasing the percentage of processor utilization and by decreasing the number of work exchanges needed. The percentage of processor utilization is presented in Figures 12 and 13.

By using solution counting based branching heuristics the processors percentage of utilization is higher than the $MinDom$ heuristic. The choice of a heuristic depends on the kind of instances to solve. $MinDomMinSTD$ works better on balanced ones while $MinConstMinSTD$ gives good results on random ones. The two heuristics also reduces the number of work exchanges performed during the search as shown in Figure 14 and 15.

These load balancing measures have an impact on the parallel speedups as shown in Figures 16 and 17. We observe that the parallel speedups increase by using the branching heuristics and workload estimators based on solution counting. The work exchanges are enough to balance the workload on balanced

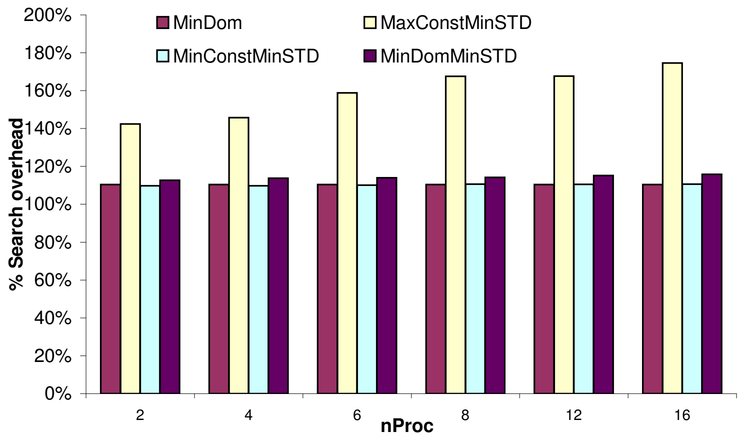

Figure 10: Balanced instances - Parallel Search Overhead

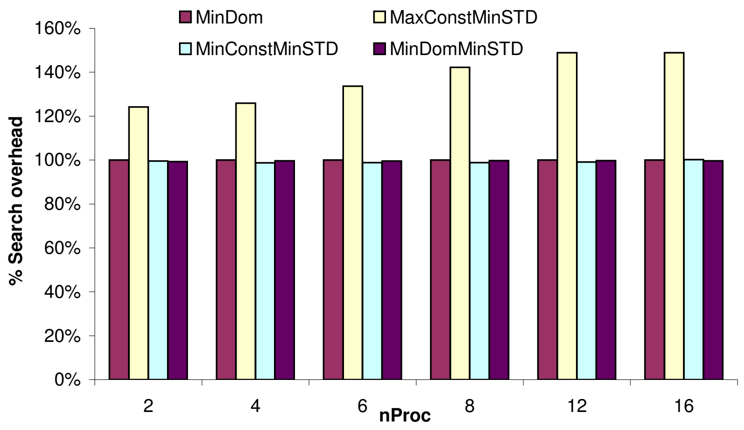

Figure 11: Random instances - Parallel Search Overhead

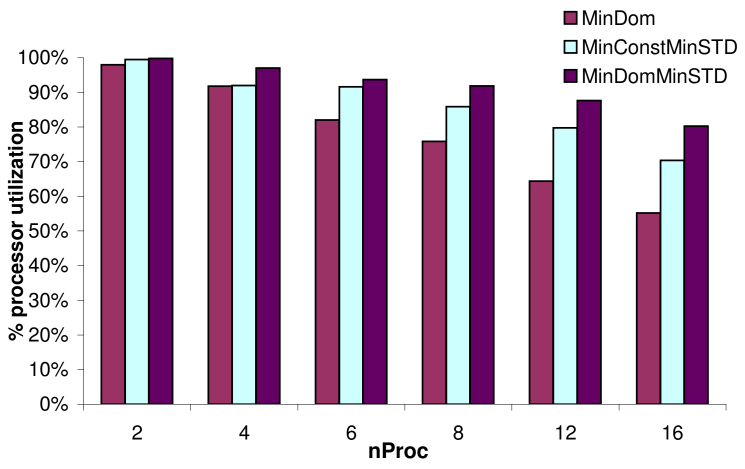

Figure 12: Balanced instances - Processor percentage of utilization with dynamic load balancing

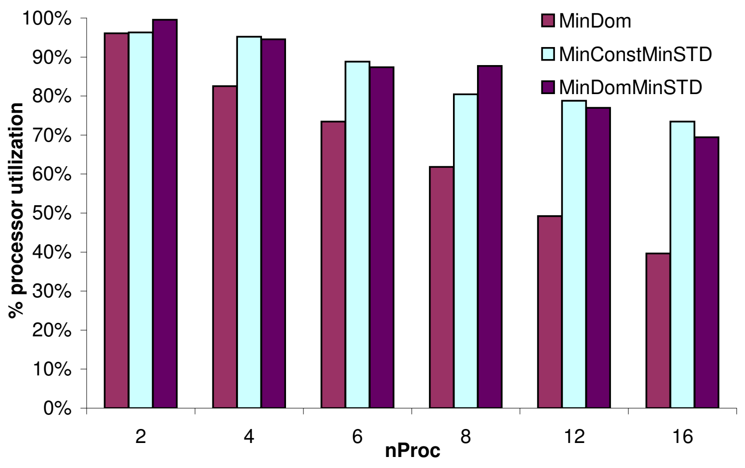

Figure 13: Random instances - Processor percentage of utilization with dynamic load balancing

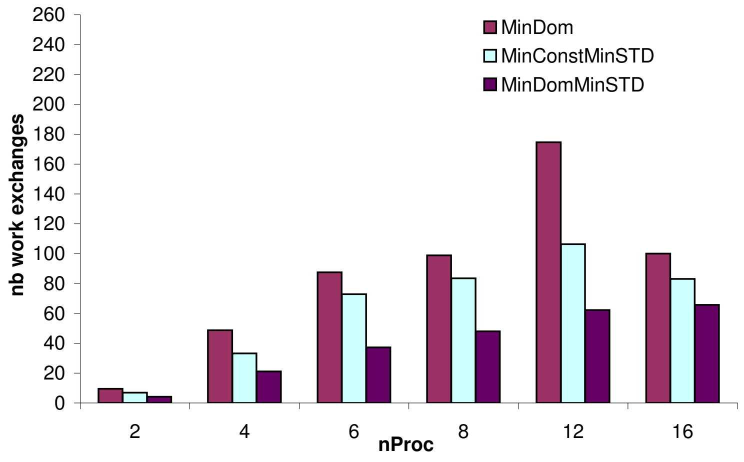

Figure 14: Balanced instances - Number of work exchanges

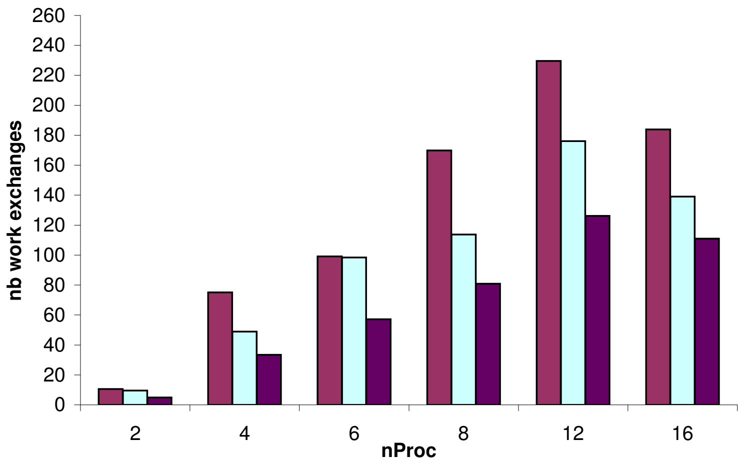

Figure 15: Random instances - Number of work exchanges

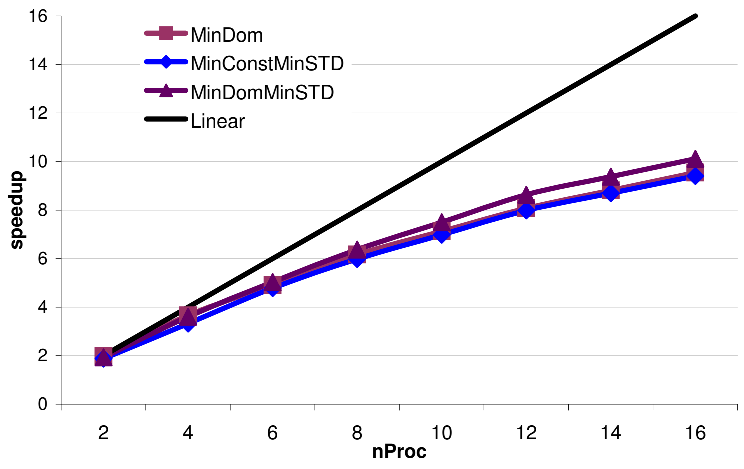

Figure 16: Balanced instances - Parallel speedups

QWH instances. In this particular case, a speedup improvement of 0.19 with 8 processors, 0.56 with 12 processors and 0.57 with 16 processor is observed. The speedups are almost the same when using a number of processors between 2 and 6.

The improvement is better on random instances which needs more work exchanges than the balanced ones. In this particular case, $MinConstMinSTD$ gives better results than $MinDomMinSTD$ when using more than 8 processors. These two heuristics also work better than $MinDom$. The sublinear speedups are caused by the communication costs and the parallel task tree management.

7 Conclusion and Future Work

We have studied several heuristics developed to improve load balancing in a parallel CP solver. The experimentations show that the use of solution counting as a workload estimator can improve the work decomposition and the load balance over a multiprocessor architecture. Attention must be paid to the parallel search overhead factor when using a specialized branching heuristic during tasks generation. Also, randomly generated QWH instances take better advantage of the specialized branching heuristics and workload estimators than balanced instances.

As extensions to this work, we propose to use solution counting to estimate the workload when solving combinatorial *optimization* problems. In this case, the workload must be approximated as a function not only of solution density, but also of the bound computed at any node in the search tree.

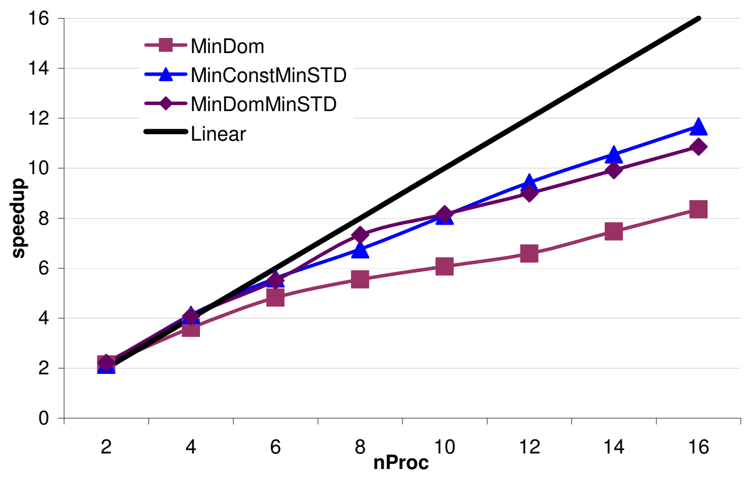

Figure 17: Random instances - Parallel speedups

References

[1] K.R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, USA, 1974.

[2] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems*, volume 39 of *International Series in Operations Research & Management Science*. Springer, 2001.

[3] Mohamed Benaïchouche, Van-Dat Cung, Salah Dowaji, Bertrand Le Cun, Thierry Mautor, and Catherine Roucairol. Building a parallel branch and bound library. In *Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques*, volume 1054 of *Lecture Notes in Computer Science*, pages 201–231, London, UK, 1996. Springer.

[4] E.G. Coffman, M.R. Garey, and D.S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal of Computing*, 7(1):1–17, 1978.

[5] Hélène Collavizza and Michel Rueher. Exploring different constraint-based modelings for program verification. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2007.

[6] D.J. Cook and R.C. Varnell. Adaptive parallel iterative deepening search. *Journal of Artificial Intelligence Research*, 9:139–65, 1998.

[7] Kelly Easton, George L. Nemhauser, and Michael A. Trick. Solving the travelling tournament problem: A combined integer programming and constraint programming approach. In *International Series of Conferences on*

the Practice and Theory of Automated Timetabling (PATAT), volume 2740 of *Lecture Notes in Computer Science*, pages 100–112, Gent, Belgium, 2002. Springer.

[8] Pierre Flener, Justin Pearson, Luis G. Reyna, and Olof Sivertsson. Design of financial cdo squared transactions using constraint programming. *Constraints*, 12(2):179–205, 2007.

[9] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-bound algorithms:survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[10] C.P. Gomes and D. Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of Computational Symposium on Graph Coloring and Generalizations*, Ithaca, NY, USA, 2002.

[11] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[12] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[13] Ulrich Junker, Stefan E. Karisch, Niklas Kohl, Bo Vaaben, Torsten Fahle, and Meinolf Sellmann. A framework for constraint programming based column generation. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 261–274, London, UK, 1999. Springer.

[14] V. Kumar, A. Grama, and V. Nageshwara Rao. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[15] V. Kumar and V. Nageshawara Rao. Parallel depth first search. ii. analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[16] Ten-Hwang Lai and Sartaj Sahni. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27(6):594–602, 1984.

[17] Reinhard Lüling, Burkhard Monien, Alexander Reinefeld, and Stefan Tschöke. Mapping tree-structured combinatorial optimization problems onto parallel computers. In *Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques*, volume 1054 of *Lecture Notes in Computer Science*, pages 115–144, London, UK, 1996. Springer.

[18] B. Mans and C. Roucairol. Performances of parallel branch and bound algorithms with best-first search. *Discrete Applied Mathematics*, 66(1):57–74, 1996.

[19] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations.* Interscience Series In Discrete Mathematics And Optimization. Wiley, New York, NY, USA, 1990.

[20] L. Michel, A. See, and P. Van Hentenryck. Parallelizing constraint programs transparently. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 514–528. Springer, 2007.

[21] Michela Milano. *Constraint and Integer Programming: Toward a Unified Methodology*, volume 27 of *Operations Research and Computer Science Interfaces.* Kluwer Academic Publisher, 2003.

[22] V. Nageshwara Rao and V. Kumar. Parallel depth first search. i. implementation. *International Journal of Parallel Programming*, 16(6):479 – 499, 1987.

[23] V. Nageshwara Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, 1993.

[24] P. Pardalos and Xiaoye Li. Parallel branch-and-bound algorithms for combinatorial optimization. *Supercomputer*, 7(5):23–30, 1990.

[25] Laurent Perron. Search procedures and parallelism in constraint programming. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360, London, UK, 1999. Springer.

[26] G. Pesant and M. Gendreau. A constraint programming framework for local search methods. *Journal of Heuristics*, 5(3):255–279, 1999.

[27] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer, 2004.

[28] Gilles Pesant. Counting solutions of csps: A structural approach. In *International Joint Conferences on Artificial Intelligence*, pages 260–265. Professional Book Center, 2005.

[29] Ted K. Ralphs, Laszlo Ladányi, and Matthew J. Saltzman. A library hierarchy for implementing scalable parallel search algorithms. *Journal of Supercomputing*, 28(2):215–234, 2004.

[30] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence*, volume 1, pages 362–367, CA, USA, 1994. American Association for Artificial Intelligence.

[31] A. Reinefeld and V. Schnecke. Work load balancing in highly parallel depth-first search. In *Proceedings of Scalable High Performance Computing Conference*, pages 773–780, TN, USA, 1994. IEEE Computer Science Press.

[32] Catherine Roucairol. Parallel computing in combinatorial optimization. *Computer Physics Reports*, 11:195–220, 1989.

[33] Christian Schulte. Parallel search made simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems*, Singapore, 2000.

[34] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998.

[35] H. Trienekens and A. de Bruin. Towards a taxonomy of parallel branch and bound algorithms. Technical Report EUR-CS-92-01, Department of Computer Science, Erasmus University Rotterdam, 1992.

[36] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* MIT Press, MA, USA, 2005.

[37] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.

[38] Sebastian Will, Anke Busch, and Rolf Backofen. Efficient sequence alignment with side-constraints by cluster tree elimination. *Constraints*, 13(1):110–129, 2008.

[39] Alessandro Zanarini and Gilles Pesant. Solution counting algorithms for constraint-centered search heuristics. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 743–757. Springer, 2007.