Optimizing resource sharing on cooperative execution of algorithms

Alfredo Goldman, Yanik Ngoko, Denis Trystram

▶ **To cite this version:**

Alfredo Goldman, Yanik Ngoko, Denis Trystram. Optimizing resource sharing on cooperative execution of algorithms. [Research Report] RR-LIG-021, 2011. hal-00796872

HAL Id: hal-00796872
https://inria.hal.science/hal-00796872

Submitted on 15 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Les rapports de recherche du LIG

Optimizing resource sharing on cooperative execution of algorithms

Alfredo GOLDMAN, Associate Professor, University of Sao Paulo, Brasil
Yanik NGOKO, Post Doc., University of Sao Paulo, Brasil
Denis TRYSTRAM, Professor, LIG, Grenoble University(Grenoble-INP), France

RR-LIG-021
Septembre 2011

http://rr.liglab.fr
ISSN 2105-0422

L I G

Optimizing resource sharing on cooperative execution of algorithms [1]

ALFREDO GOLDMAN and YANIK NGOKO [2], University of São Paulo, Brazil
and DENIS TRYSTRAM, Grenoble Institute of Technology and Institut Universitaire de France

Solving hard combinatorial problems has always been a challenge. The constant progress in algorithm design and the emergence of powerful large-scale parallel platforms allow to solve a larger number of instances of such problems. As a consequence, many efficient algorithms are available for solving the same problem. However, none of them strictly dominates the others on all the instances.

An important problem is to determine an adequate selection of algorithms for solving a given set of instances in order to minimize the total completion time. However, little is known in cooperation between algorithms for an improved global objective. The execution model of algorithms portfolio is a nice framework for studying this question. It consists in executing all the available algorithms concurrently and interrupt them as soon as a solution is found. This model of execution raises however the question of the resources repartition among the algorithms. Interesting directions in this way have been proposed through parallel algorithms portfolio problem. The idea is that it is possible to learn about the resource sharing based on the observation of executions of the algorithms on a set of instances. Parallel algorithms portfolio problems are generally intractable and thus, many heuristics have been proposed to solve them. However, these solutions do not take into account the possibility of sequential algorithms or unknown speed-ups.

We propose in this paper a two phases approach for learning resource sharing in the context of a portfolio. The first phase consists in estimating what may be the optimal useful execution workload for each algorithm. This estimation is done through a clustering approach inspired by the resolution of the set cover problem. The second phase consists in minimizing the estimated workload by optimal resource allocation. We propose for this purpose a solution based on dynamic programming.

This approach is evaluated experimentally on two settings. In the first setting we simulate the concurrent execution of sequential SAT algorithms. In the second case, we implement a parallel algorithm for CSP based on algorithms portfolio. The experiments clearly exhibit that the proposed approach is suitable for defining an efficient concurrent model of execution.

Categories and Subject Descriptors: XXX [**XXX**]: XXX—*XXX*; XXX [**XXX**]: XXX—*XXX*; XXX [**XXX**]: XXX—*XXX*

General Terms: Algorithms, Experimentations, Performances

Additional Key Words and Phrases: resource sharing, algorithm portfolio, concurrent execution, cooperative problem solving

ACM Reference Format:
Goldman, A. and Ngoko, Y. and Trystram, D. C. 2011.Optimizing resource sharing on cooperative execution of algorithms. ACM Jour. Experi. Algo. X, X, Article 1 (August 2011), 21 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

[1]Special Issue on Multi-core Algorithms
[2]Corresponding author

This work is supported by a FAPESP Grant at the University of São Paulo. Denis Trystram is partially supported by a Google research award.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2011 ACM 1539-9087/2011/08-ART1 $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

This work focuses on the resolution of hard computational problems. Following the advances in algorithms design and the evolution of computing platforms, there exist many heuristics that can be used for solving the same problem. Such heuristics usually have their own characteristics on the execution time depending on the target instances and they behave differently. It is commonly well-admitted that on most problems, no single algorithm dominates all the others (this is for instance the case in scientific contests like the SAT competition [3]). Thus, a possible approach for efficient implementation consists in performing concurrent executions of algorithms [Huberman et al. 1997; Gomes and Selman 2001]. For example in such executions, all the algorithms run on an instance and are stopped as soon as a solution is found. Obviously, there is an overhead to pay for the wasted time lost while computing on the whole set of heuristics, however, redundancy might still be interesting. Positive applications of this idea can be found in cooperative search techniques [El-Abd and Kamel 2005]. Moreover, the idea of concurrent-cooperative executions is particularly suitable in the context of new parallel architectures (large scale multi-core machines). However, it implies a smart resource sharing beyond the algorithms (memory, processor, bandwidth) that is not easy to determine because of resource limitations and the behaviors of the algorithms that depend on the amount of allocated resources. In this paper, we propose to study efficient strategies for the resource sharing problem in this context. We target two basic problems, namely SAT (Satisfiability problem) and CSP (Constraint Satisfaction Problem).

1.1. Related works

The idea of using multiple algorithms for solving a problem has been formalized inside various techniques for designing algorithms. Beyond them, we distinguish between algorithms selection, algorithms cascading and algorithm portfolio techniques. In algorithm selection [Houstis et al. 2000; Burke et al. 2010] the idea is to deal with the choice in a short amount of time of the best algorithm among a set of known algorithms for solving an instance. In algorithm cascading [Matteo Frigo 1998], the problem is expressed by a generic recursive algorithm whose recursive calls can be muted in call to another algorithm. The best choice of these mutations is the key point of the technique and it depends mainly on the sizes of the sub-problems. We focus in this work on algorithms portfolio [Huberman et al. 1997] which proposes to run concurrently the different algorithms. Moreover, we consider parallel computing platforms for running these algorithms with various relative amount of resources.

Two types of concurrencies can be considered: The first one (resource sharing) consists in fixing the number of processors shared between the algorithms while the second (time sharing) assumes that the resource sharing might change and can be adapted during the execution. The optimization of time sharing has mainly been studied in [Sayag et al. 2006; Streeter et al. 2007] in a theoretical perspective and in [Xu et al. 2008; Bhowmick et al. 2003] where it is applied to the resolution of the SAT problem and on solving non-linear numerical systems. We focus in this work on the first type of concurrency. For optimizing resource sharing, the main idea proposed in [Sayag et al. 2006] is to learn it from a finite set of executions on a given testbed. It is formalized as the Resource Sharing Scheduling Problem (RSSP) which consists in determining the best proportion of resources to allocate to each algorithm for solving a finite set of instances as fast as possible. In this model, all the algorithms are executed (with an allocation that has to be determined) on all the instances concurrently until an algo-

[3] http://www.satcompetition.org

rithm finds a solution. The complexity of the problem relies on the combinatorics of the various resources allocation which have different completion times for each instance. The usage of a testbed for defining resource sharing is particular well-suited for solving hard computational problems where the algorithm performance on a set of hard instances is a well accepted qualitative metric. The original model defined in [Sayag et al. 2006] assumes a fractional number of processors. Fractional resources sharing is hard to implement in practice. Alternative solutions based on discrete resources have then been proposed in [Bougeret et al. 2011; Ngoko and Trystram 2009b]. The study proposed in this paper is based on these works.

However, it improves the two following aspects: the first aspect is that until now, the resource sharing has been studied excluding sequential algorithms or the possibility to not select a resource to execute an algorithm. The second aspect is that the resource sharing has generally been studied assuming linear speed-up on parallel algorithms (or quasi-linear speed-up). This might not always be the case in practice. Moreover, increasing resources in a parallel algorithm might even deteriorate the execution time.

1.2. Contributions

In this work, we propose a general solution for designing algorithms portfolio which takes into account the new features described at the end of the previous section. The construction is based on a two phases approach. In a portfolio execution, there is one algorithm that leads to the interruption of the others. The first phase corresponds to an estimation of the leading algorithms, giving a set of instances. This is achieved by using a heuristic inspired by the resolution of the set cover problem. The second phase consists in assigning the right amount of resources to the algorithms in order to minimize the cost induced by the leading algorithm repartition. We show that this later problem is similar to the search of the shortest path in a graph [Cormen et al. 2001]. We propose an efficient solution based on dynamic programming for solving it. Then, we present a quantitative assessment on two settings through a series of experiments. In the first setting, we simulate the concurrent execution of actual sequential SAT algorithms. In the second one, we implement a parallel algorithm for CSP based on algorithm portfolio. Experimental evaluations show that with representative testbeds, the proposed approach is suitable for defining a concurrent model of execution.

1.3. Paper organization

The paper is organized as follows: Section 2 is devoted to introduce the model of resource sharing and the description of the portfolio approach. Section 3 presents the algorithms for solving the resource sharing problem. Section 4 consists in applying the resource sharing algorithms for the construction of parallel solvers for the SAT and CSP problems. We conclude and briefly discuss some extensions of this work in Section 5.

2. MODEL

2.1. Parallel portfolio problem

For defining the resource sharing, we consider a setting where there are a finite set of candidates composed of algorithms and a shared-memory parallel homogeneous architecture (typically a multi-core machine). We assume a representative set of instances that captures the behavior of the candidate algorithms. In this setting the construction of an optimal resource sharing has been formalized through the *dRSSP* problem introduced in [Bougeret et al. 2009]. We recall below the formal definition of the decision version of this problem:

discrete Resource Sharing Scheduling Problem (dRSSP)

Instance: A finite set of instances $\mathcal{I} = \{I_1, \ldots, I_n\}$, a finite set of algorithms $\mathcal{A} = \{A_1, \ldots, A_k\}$, a set of m identical computing resources, a cost $C(A_i, I_j, p) \in R^+$ for each $I_j \in \mathcal{I}$, $A_i \in \mathcal{A}$ and $p \in \{1, \ldots, m\}$, a real value $T \in R^+$.

Question: Is there a vector $S = (S_1, \ldots, S_k)$ with $S_i \in \{0, \ldots, m\}$ and $0 < \sum_{i=1}^{k} S_i \leq m$ such that $\sum_{j=1}^{n} \min_{1 \leq i \leq k} \{C(A_i, I_j, S_i) | S_i > 0\} \leq T$?

In order to illustrate the interest of this problem, suppose that we want to solve two instances (I_1, I_2) of the same problem. There are two available parallel algorithms (namely, A_1, A_2) that have each a linear speed-up. Let suppose that we have 2 resources and that the following execution cost matrix gives the execution times of each algorithm on the entire set of resources. $C = \begin{pmatrix} 2 & 10 \\ 10 & 1 \end{pmatrix}$.

Observe that instance I_1 is solved in 10 time units by algorithm A_2 and in 2 time units by A_1. Instances I_2 is solved in 1 time unit by A_2 and 10 time units by A_1. Thus, if we consider only I_1 to solve, we can just execute A_1. However, if A_1 is again used for solving I_2, the total cost is 12. Now, if we give one resource to both algorithms A_1 and A_2, the total time required becomes $4 + 2 = 6 \leq 12$.

dRSSP is derived from the original problem proposed by Sayag [Sayag et al. 2006] adapted to a parallel homogeneous context. The NP-completeness and inapproximability of *dRSSP* has been proved in [Bougeret et al. 2009]. In the following, we will use following simplified notations: $C^*(I_j) = \min_{1 \leq i \leq k} C(A_i, I_j, 1)$ and $C(A_i, I_j, 1) = C(A_i, I_j)$

2.2. Discussion about dynamic setting and cooperation

We can make at least two criticisms to *dRSSP*: this model suggests a static resource sharing between the algorithms, it does not take into account the fact that while executing some algorithms during a few steps, it is possible to identify some algorithms that should not lead quickly to a solution. Another criticism about *dRSSP* is that it does not take into account cooperation between the algorithms (for instance, for removing redundant computations).

About the static character of *dRSSP*, we can notice that it can be used for defining a dynamic resource sharing model. An example of this usage was suggested in [Ngoko and Trystram 2009a; Gebruers et al. 2005; Gagliolo and Schmidhuber 2008] where a benchmark of instances is considered. For each incoming instance I to solve, using a learning mechanism, we determine the subset of benchmark instances $\mathcal{I}' \subseteq \mathcal{I}$ whose behavior might *a priori* be close to that of I. Then, we can use the resource sharing given by the *dRSSP* solution for the instances $\mathcal{I}'$ to solve I.

About the lack of cooperation, we can remark that even if we assume that the different algorithms can share informations about their executions. The resource sharing defined by *dRSSP* can be used as an approximation for optimizing resources exploitation.

3. SOLVING ALGORITHM PORTFOLIO

As previously said, *dRSSP* can not be approximated within a constant factor. This suggests that we should deal with heuristics for designing polynomial time solutions.

In *dRSSP*, we are searching for a fixed allotment to define an optimal concurrent execution of algorithms for a set of known instances. All the algorithms with at least one resource are run until a solution is found. The resolution of an instance with this model comprises useful executions (those that lead to determine a solution for an instance) and useless ones (that will be interrupted without providing a solution). We will formalize useful executions through the notion of *workload distribution*. The ob-

jective of such a distribution is to determine which algorithm should solve the various instances in the concurrent setting. Then, it provides an approximation for each algorithm of the load to optimize in the resource sharing. Based on this analysis, our methodology decomposes the *dRSSP* resolution into two phases: the prediction of the workload distribution and the resource allocation. We will present in the next section a more detailed point of view on the workload distribution.

3.1. Workload distribution

We define a workload distribution by an injection $\sigma : \mathcal{I} \longrightarrow \mathcal{A}$. In this application, each instance is assigned exactly to one algorithm. There can be an algorithm for which there is no assigned instance. Given a distribution σ and an instance I_j, $\sigma(I_j)$ gives the algorithm that has a useful execution on I_j. In this definition, we suppose that there is just one algorithm with a useful execution per instance. This choice is natural since we stop the concurrent execution of algorithms on any instance as soon as one algorithm ends.

Given a workload distribution σ, we define the workload denoted by A_i as $W(\sigma, A_i) = \sum_{I_j \in \sigma^{-1}(A_i)} C(A_i, I_j)$. It captures the sum of sequential execution times that A_i can take for solving instances assigned to it from the σ distribution.

The ignorance of algorithm workload is one point that makes *dRSSP* different from the other classical load balancing problems. Indeed, they generally assume the knowledge of the load for each algorithm in a sequential context. For *dRSSP* the workload is certainly a good estimation of charges. The next section deals with its estimation.

3.2. Workload estimation

We organize the workload estimation in two parts. We first fix the objectives that must be pursued by the optimal distribution and based on them, we propose an algorithm.

3.2.1. Defining workload estimation problem. Let us denote by $\mathcal{D}$ the set of all possible workload distributions. We formulate the problem of finding the optimal workload distribution through the Workload Estimation Problem (*WEP*) defined in Figure 1.

Minimize $\left(\sum_{j=1}^{n} W(\sigma, I_j), \lvert A^+ \rvert\right)^t$
1. $\sigma \in \mathcal{D}$
2. $A^+ = \{A_i \in \mathcal{A} \lvert \sigma^{-1}(A_i) \neq \emptyset\}$

Fig. 1. Workload Estimation Problem (WEP)

The various objectives of *WEP* are motivated as follows: it is well-known that for achieving optimal performance in parallel, the sequential execution time should also be minimized. Given a distribution σ, the sum $\sum_{j=1}^{n} W(\sigma, I_j)$ is the minimal amount of time needed for solving instances in a sequential setting. By reducing this amount, one can expect to reach good performances for the resource sharing. The reduction of the sequential execution time here must not ignore that the more there are algorithms such that $\sigma^{-1}(A_i) \neq \emptyset$, the fewer there are resources to share per algorithm. Thus, we will have less benefit from the parallelism perspective. Indeed, if all the algorithms have a positive workload in a distribution, it is sure that $\sum_{j=1}^{n} W(\sigma, I_j)$ will be minimal. However, this must not help if for example on all resources, there is an algorithm that completely dominates the other on each instance.

We formulated the search of the optimal workload as a bi-objective problem. Many concepts can be used for defining the solution for a such problem. We will use here the

pareto front concept [Dutot et al. 2009] (an illustration is provided in Figure 2). From the NP completeness result of *dRSSP* in [Bougeret et al. 2009], one can easily show that this problem is NP-complete and inapproximable if the second criteria is fixed. Then, we intend to approximate optimal solutions for *WEP* in the next section.

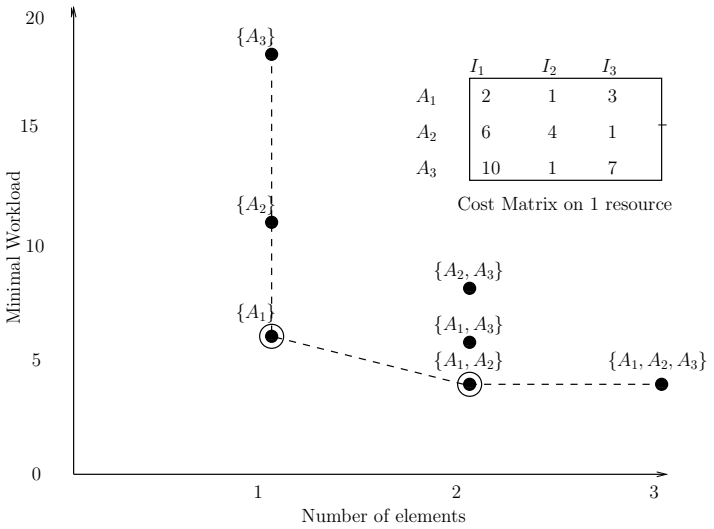

Fig. 2. Illustration of possible distributions. We denote each distribution by a subset A^+ such that instances are assigned to algorithm in order to minimize the workload. For instance, the minimal workload for solving all the instances with $\{A_1, A_3\}$ is $2 + 1 + 3 = 6$. The sets that are circled correspond to Pareto dominant solutions

3.2.2. Set cover Formulation. For solving bi-objective problems, a common approach consists in fixing one objective as a constraint and optimizing the second one. In our case, this approach is adequate since possible values of the second criteria ($\min\{|A^+|\}$) are discrete and finite. Let us now suppose that we are looking for the optimal workload where at most r algorithms ($1 \leq r \leq \min\{k, m\}$) have a positive workload. The estimation of the optimal workload distribution for a fixed r can be stated as a set covering problem from the following observation : For a distribution σ, the inverse σ^{-1} is given by $\sigma^{-1}(A_1), \ldots, \sigma^{-1}(A_k)$ where each $\sigma^{-1}(A_i) \subseteq \mathcal{I}$ is a subset of instances. Since σ is an injection, $\bigcup_{1 \leq i \leq k} \sigma^{-1}(A_i) = \mathcal{I}$. The obtention of the minimal workload can be done in selecting between all possible subsets for each $\sigma^{-1}(A_i)$, the ones leading to the minimal workload.

For approximating σ^{-1}, we do the following steps: We generate all possible subsets $\sigma^{-1}(A_i)$ for each algorithm A_i. Since each of these subsets is included in $\mathcal{I}$, we have a total of $k.2^n$ sets $G = \{G_1, \ldots G_{k.2^n}\}$ that are partial assignments of instances to algorithms. Let suppose that for each subset G_u, $alg(G_u)$ is the algorithm to which we refer in the assignment (for example the subset G_1 can refer to a possible assignment for $alg(G_1) = A_1$). Given G, we choose between all possible selections $Sel(G) \subseteq G$ such that $\bigcup_{G_j \in Sel(G)} = \mathcal{I}$ and $|A(Sel)| \leq r$ where $A(Sel) = \{alg(G_j) | G_j \in Sel(G)\}$, the selection of subsets with minimal workload. The condition $|A(Sel)| \leq r$ ensures that we do not have more that r distinct algorithms.

For determining the complete Pareto front approximation, we repeat the set cover process with different values of r. At the end, only the solutions that are not Pareto dominated are considered.

The formulation above is based on a set cover representation of the distribution function. It proposes to solve $\min\{k, m\}$ problems of covering for approximating the workload distribution. However, the number of sets is exponential: thus, we have to reduce this number.

3.2.3. Approximating cover. To reduce the number of sets, we observe that each set $G_u \in G$ gives a partial assignment for a workload distribution in which the sequential resolution time of an instance $I_j \in G_u$ is $C(alg(G_u), I_j)$. This time is not necessary the optimal possible sequential time $C^*(I_j)$ if we consider the time of all algorithms on I_j. We will say that in the set G_u, the distance at which the instance I_j is solved is $\frac{C(alg(G_u),I_j)}{C^*(I_j)}$.

For each set, there is a maximal distance value $g_m = \max_{I_j \in G_u} \{\frac{C(alg(G_u),I_j)}{C^*(I_j)}\}$. This is the biggest distance at which an instance of the set is solved. From a depreciative evaluation of the quality, let us emphasize that if a partial assignment leads to solve one instance at the maximal distance g, it can also handle the possibilities of solving any other instances at g. In this way, the number of sets is restricted to g-OPT sets.

A g-OPT set is an element of G that contains any instances that it can solve at a given maximal distance value. More precisely, a g-OPT set G_i^g corresponds to a set G_u such that $alg(G_u) = A_i$ and contains all instances I_j such that $C(alg(G_u), I_j) \leq gC^*(I_j)$ (*i.e.* the instances that are solved with a distance at most $g \in [1 + \infty[$).

The idea of g-OPT sets is to consider that in the optimal solution, if there is an algorithm that might not solve one instance optimally, this can be the case for many other instances. Thus, it is not necessary to consider all the possible partial assignments. The advantage of this restriction is that we just need to consider a polynomial number of sets. This remark is summarized in the following proposition.

PROPOSITION 3.1. *We can restrict distance values g_m for defining all g-OPT sets $G^i_{g_m} = \{I_j | C(A_i, I_j) \leq g_m C^*(I_j)\}$ to factors $\frac{C(A_i,I_j)}{C^*(I_j)}$.*

This result is straightforward from the definition of g-OPT sets. One of its consequence is that the number of g-OPT sets is at most $((n-1)k + 1)k$.

Let us now formalize the selection of sets. For this purpose, we adopt a solution inspired on the greedy algorithms for solving the set cover problem. A dynamic price is associated to the sets at each step. Throughout the execution, the sets are selected with the most interesting price and then the covered elements are removed from the remaining sets. The algorithm ends when we have a number of sets corresponding to r distinct algorithms or a set cover. For adapting this algorithm to our case, we have to define precisely the notion of *price*.

Pricing sets : For pricing sets, we propose to promote sets which have an optimal solution on heavy instances (*i.e.* the instances with a big $C^*(I_j)$). Indeed, the impact of a non-optimal resolution of the heaviest instances might be in general more important in the workload than the non-optimal resolution of other instances. We use this observation to fix the price of a g-OPT set G_i^g as $P_i^g = \frac{\sum_{I_j \in G_i^g} C^*(I_j)}{g}$. The price in this definition takes into account the minimal workload processed in the set but also the quality (g). One might think that for having high price, we simply need to have many instances or heavy instances in the set. However, we take into account the quality in introducing g in the price. Thus, the prices advantage the sets that solve a lot of instances optimally

ALGORITHM 1: HIF(r)

$G \leftarrow$ Generate_gOPTsets()
$S \leftarrow \emptyset$
$C \leftarrow \emptyset$
for $G_i^g \in G$ **do**
  $P_i^g \leftarrow \left(\frac{\sum_{I_j \in G_i^g} C^*(I_j)}{g} \right)$
end
while $(|S| < r \; and \; |C| < n)$ **do**
  $(X^r, Z^r) \leftarrow \{(A_i, G_i^g) \notin S | P_i^g \text{ is maximal}\}$
  for $G_i^g \in G$ **do**
    $G_i^g \leftarrow G_i^g \backslash Z^r$
    $P_i^g \leftarrow \left(\frac{\sum_{I_j \in G_i^g} C^*(I_j)}{g} \right)$
  end
  $S \leftarrow S \cup (X^r, Z^r)$
  $C \leftarrow C \cup Z^r$
end

(in these cases $\sum_{I_j \in G_i^g} C^*(I_j)$ is big and g minimal). One advantage of the adopted pricing strategy is that since set corresponds to algorithms, we will be inclined to select algorithms in function of the set of instances on which they have a good execution time. This suggest the complementary of the selected algorithms.

Deriving the greedy algorithm: the problem here is to select r sets corresponding to different algorithms with minimal workload. We give in the algorithm 1 the greedy solution (called **HIF** for Heaviest Instances First) derived from the previous pricing strategy assuming that there are at most r sets corresponding to different algorithms to select.

At the beginning of the algorithm, the set of partial assignments (G) is initialized and prices are computed. At each step, **HIF(r)** selects a set corresponding to a g-OPT set and then, it removes this set from the sets of partial assignments. Then, it removes the instances covered by this set from the other sets and then, it recomputes the prices. S is the main output of **HIF(r)**. It contains sets of instances with associated algorithm reference.

There are two conditions that might end the execution of **HIF(r)**. The first is that r distinct algorithms have been selected. The second is that we have a workload distribution covering all the instances ($|C| = n$). It is important to notice that these criteria are not necessarily inclusive.

On two different steps $u < v$ of **HIF(r)**, some selections (X^u, Z^u) and (X^v, Z^v) where $X^u = X^v$ can occur. We then have a selection of a set Z^v associated to an algorithm that is already included in S. In such cases, by making the operation: $S \leftarrow S \cup (X^r, Z^r)$, we will just add the set Z^r to Z^u in the pair $(X^u, Z^u) \in S$.

At the end of **HIF(r)**, we have a solution $(X^1, Z^1), \ldots (X^r, Z^r)$ where each X^u corresponds to an algorithm of $\mathcal{I}$ associated to a set of instances Z^u belonging to G. For the purpose of notations, let suppose that for each algorithm A_i the set associated is G_i. If an algorithm A_u is not selected, we set G_u to $\emptyset$. The workload of each algorithm A_i can

be computed as follows:

$$W(\textbf{HIF(r)}, A_i) = \begin{cases} 0 & \text{if } A_i \text{ is not selected in } S \\ \sum_{I_j \in G_i} C(A_i, I_j) & \text{otherwise.} \end{cases}$$

The computed workload distribution given by the pairs (A_i, G_i) might not cover all the instances. However, the strategy of prioritizing sets solving heavy instances has the advantage of leading to a distribution for instances that are the most important for the resource sharing. Moreover, we propose to complete it by adding all the non-covered instances I_j to the set G_i such that $W(\textbf{HIF(r)}, A_i) > 0$ and $C(A_i, I_j) = \min_{A_u \in A, W(\textbf{HIF(r)}, A_u) \neq 0} C(A_u, I_j)$ (this place I_j in the set of minimal workload for it between the selected sets). Finally, we make vary r to obtain a Pareto front approximation of the workload distribution. At the end of **HIF(r)**, S contains the estimated workload distribution.

3.3. Resource allocation

We now deal with the question of resource allocation. At this level, **HIF(r)** produced a distribution which associates a set of instances G_i for each algorithm A_i. The available resources must be allocated to all algorithms in order to minimize the time required to process the sets of instances G_i. Considering a limited number of resources m, the decision version of this problem can be stated as follows:

Instance: A finite set of k instances subsets $= \{G_1, \ldots, G_k\}$, a finite set of algorithms $\mathcal{A} = \{A_1, \ldots, A_k\}$, a set of m identical resources, a cost $C(A_i, I_j, p) \in R^+$ for each $I_j \in G_i$, $A_i \in \mathcal{A}$ and $p \in \{1, \ldots, m\}$, a real value $T \in R^+$.

Question: Is there a vector $S = (S_1, \ldots, S_k)$ with $S_i \in \{0, \ldots, m\}$ and $0 < \sum_{i=1}^{k} S_i \leq m$ such that $\sum_{i=1}^{k} \sum_{I_j \in G_i} \{C(A_i, I_j, S_i)\} \leq T$?

The objective function is explained as follows: on the chosen distribution, attributing S_i resources to A_i leads to a processing time of $\sum_{I_j \in G_i} \{C(A_i, I_j, S_i)\}$ for processing instances G_i. The sum of these processing times must be minimized taking into account that there are m resources. Obviously, if $S_i = 0$ and $G_i \neq \emptyset$ then $C(A_i, I_j, S_i) = +\infty$. This formulation differs from *dRSSP* because we fixed the right execution for solving the instances. We present how to solve this allocation problem in the next section.

3.3.1. Dynamic programming for the allocation problem. The determination of the minimal resource allocation is similar to the search of a shortest path in a weighted graph. We consider each graph node at depth u as a partial allocation $(A_1, S_1), \ldots, (A_u, S_u)$. It has a weight denoted by $Cost(u, m_u) = \sum_{i=1}^{u} \sum_{I_j \in G_i} \{C(A_i, I_j, S_i)\}$. One can notice that nodes at depth k correspond to a complete allocation. The edges between nodes at depth u and $u + 1$ have a weight which corresponds to the amount of resources assigned for A_{u+1} from the a partial allocation $(A_1, S_1), \ldots, (A_u, S_u)$ to $(A_1, S_1), \ldots, (A_u, S_u), (A_{u+1}, S_{u+1}))$. Our objective is to find the path that leads to the node of smallest weight at depth k.

We adapt the Dijkstra algorithm [Cormen et al. 2001] to the resource allocation as follows: We consider k steps where each step u consists of choosing any assignment for the algorithm A_u that leads to a total utilization of 0 to m resources and will have a minimal load. The cost at step u for a total utilization of u resources is denoted by $Cost(u, m_u)$. This function obeys the following equations:

$$Cost(u, m_u) = \begin{cases} \min\limits_{0 \leq m_{u-1} \leq m_u} \{Cost(u-1, m_{u-1}) + \sum\limits_{I_j \in G_u} C(A_u, I_j, m_u - m_{u-1})\} & \text{if } u > 1 \\ \sum\limits_{I_j \in G_1} C(A_1, I_j, m_1) & \text{if } u = 1. \end{cases}$$

The relations above can be easily derived for the the case $u = 1$. The explanation of the case where $u > 1$ is that if we have m_u resources already assigned at step u, they have been totally or partially assigned on steps $1, \ldots, u$. Then, the minimal cost that we can be obtained with m_u resources assigned at step u is derived by considering all possible assignments at the previous step added to the workload induced by the assignment of remaining resources to A_u.

It is easy to establish that the optimal allocation is determined by computing the previous relations all the possible values $Cost(u, m_u)$, $1 \leq u \leq k, 0 \leq m_u \leq m$ and then, choosing the resource allocation corresponding to $\min\limits_{1 \leq u \leq m} \{Cost(k, u)\}$.

Finally, to find the optimal resource allocation, we proceed in generating all the Pareto solutions from **HIF**. For each of these solutions, we compute the optimal resource sharing, estimate its cost and then select the resource sharing with the smallest cost.

3.4. Complexity analysis

In this part, we proceed at a theoretical evaluation of our approach. We have the following result:

PROPOSITION 3.2. *The two phases approach proposed has a complexity in* $O(\min\{k, m\}(n^2k^2 + km^2))$

PROOF. In algorithm **HIF(r)**, the key operation is the reinitialization of g-OPT sets prices. Since a g-OPT set has at most n instances, we can compute its price in $O(n)$. Assuming that there are n_g g-OPT sets, the reinitialization of prices in **HIF(r)** can be done in $O(r.n_g.n)$. Since we have $n_g = ((n-1)k + 1)k$ from proposition 3.1, we deduce that the complexity of **HIF(r)** is in $O(r(nk)^2)$.

For the resource allocation, we construct a matrix $Cost$ of size $(m + 1) \times k$. From the recurrence equation of $Cost$, we deduce that computing $C(u, m_u)$ requires m_u comparisons. This implies that each column of the matrix yields to $\frac{m(m+1)}{2}$ comparisons. Thus, the computation of the whole matrix requires $k\frac{m(m+1)}{2}$ comparisons. Since for estimating the workload with **HIF** we make vary the size r in **HIF(r)**, we obtain the result. □

4. APPLICATIONS

For assessing the proposed approach, we consider two case studies. The first deals with the construction of a portfolio of sequential algorithms solving the well-known satisfiability problem (SAT). In this case, we did not implement a real concurrent execution of sequential algorithms, but a simulator based on the executions of actual algorithms. The interest of this study is to evaluate how close the workload distribution function issued from HIF is to the optimal. In the second case, we used our solution for defining a parallel algorithm for the Constraint Satisfaction Problem (CSP).

4.1. Designing a parallel algorithm for SAT

Given a set of clauses expressed as disjunction of boolean variables, the SAT problem consists of deciding if there is an assignment of boolean values to variables that satisfies all the clauses (makes all them true). There exist many sequential algorithms

for solving this problem. We used our approach here for the development of a parallel solver based on concurrent execution of sequential algorithms and simulate the performance that could be expected.

The data for building the portfolio come from a SAT database (SatEx [Simon and Chatalic 2001][4]) which gives the CPU time for the execution of a set of 23 sequential heuristics (SAT solvers) on 1303 SAT instances.

4.1.1. Brief description of the database. We present here a brief description of the benchmark and refer to [Simon and Chatalic 2001] for a more exhaustive one. The 1303 instances of the SatEx database are issued from many domains where SAT is encountered. Some of them are: Logistic planning, formal verification of microprocessors, Scheduling. Some instances are also issued from many challenging benchmarks for SAT which are used in one of the most popular annual SAT competition [5]

4.1.2. SAT solvers. The SatEx database contains three main SAT solvers families:

— The DLL family with the following heuristics: $asat$, $csat$, $eqsatz$, $nsat$, $sat-grasp$, $posit$, $relsat$, $sato$, $sato-3.2.1$, $satz$, $satz-213$, $satz-215$, $zchaff$,
— the DP family with : $calcres$, dr, $zres$,
— the randomized DLL family with : $ntab$, $ntab-back$, $ntab-back2$. $relsat-200$

It contains also some other algorithms: $heerhugo$, $modoc$, $modoc-2.0$ non classified in a family above. Beyond the set of algorithms and instances, the algorithm that solves all the instances in minimum time is $zchaff$. Our goal is to show that we can do better with a parallel portfolio.

4.1.3. Experiments plan and results. The general setting of our experiments on SAT is the following: we consider a multi-core parallel machine with p cores. A parallel SAT algorithm can be designed by running on each core one of the 23 sequential SAT solvers and stop when an algorithm finds a solution. In a multi-core context, the necessary synchronization for stopping all algorithms can be easily implemented by using the shared memory. The question is to know which algorithm to choose when $p \leq 23$. As claimed in this paper, this choice can be learnt with a benchmark of SAT instances. Since the SAT solvers are all sequential algorithms, an optimal execution of concurrent algorithms on the benchmark can be derived from the optimal workload distribution where we have at most p SAT solvers to which an instance is assigned. We do not need to use the resource sharing phase of our approach since solvers are sequential.

This setting allows us to evaluate the first part of our two phases approach (HIF vs Optimal distribution). We associate in the evaluation other meaningful choices of distributions; Mainly:

— The Winners distribution (WIN) which assigns a positive workload only to the first (p) fastest algorithms. In this distribution, each instance is solved by the algorithm with the smallest sequential execution time between the p selected ones. It is obvious that the workload of this distribution is better than those of the best single algorithm;
— The Random distribution (RAND) which arbitrarily assigns a positive workload to (p) algorithms. As for the previous one, each instance is solved by an algorithm with the smallest sequential execution time.

We use the 1303 SatEx instances as a benchmark and we assume that the execution time provided by each algorithm for each instance in SatEx is what we will have in executing the same algorithm on a single processor on the parallel system.

[4] `http://www.lri.fr/~simon/satex/satex.php3`
[5] `http://www.satcompetition.org`

In the setting described above, we run two series of experiments. In the first one, we make vary p in order to compare the distributions on workload and the execution times. In the second one, we select multiple SAT solvers from the 23 available and build multiple distribution functions from them. The objective is to make a more large evaluation of the quality of the distribution function produced by HIF. The size of selected SAT solvers in these experiments vary between 1 and 20. For each size k_m, we select $5(k_m - 2)$ distinct subset of SAT solvers. For each subset of size k_m, we build a distribution function for values of p between 1 and $k_m - 2$. We can notice that if $p = k_m - 1$ or $p = k_m$, the construction of a distribution is trivial.

Figure 3 depicts the workload given by the different distributions on the SatEx instances when we select p solvers among the 23 available ones. For the random distribution, the workload is estimated over 100 experiments.

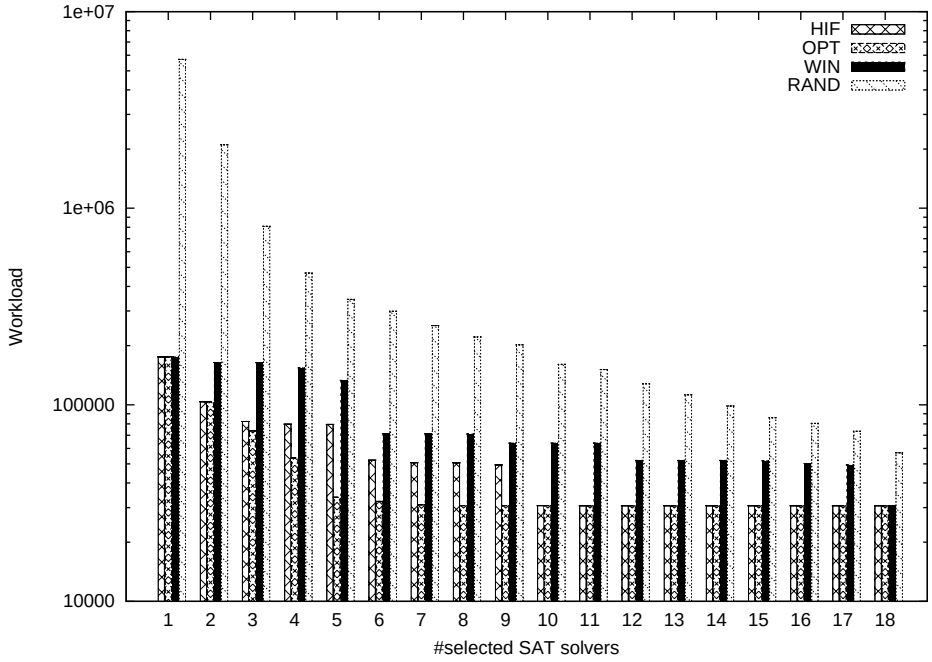

Fig. 3. Distribution workload on SatEx

The figure shows that an increase on p improves any distribution. Since this is the case even for the random distribution, we conclude that the complementarity between solvers in SatEx is such that even in taking any subsets of solvers and run them concurrently, the resulting parallel solver could have a better performance that sequential ones on which it is based.

On all SAT solver selections, the HIF distribution is better that the WIN and RAND distributions. An explanation is suggested by the change between HIF and WIN workloads from $p = 1$ to $p = 2$. For selecting two solvers, HIF will choose first the best solver like WIN. While for the second solver WIN will select the one that performs well in the remaining ones, HIF will look more for a solver that is complementary with what was already chosen. In Table I we give a better view of the differences between HIF and WIN through the ratio between their workloads and the optimal distribution. HIF

Table I. Average ratio between HIF, Winners, Random and the optimal for different values of p

p	HIF	WIN	RAND
1	1.00	1.00	32.64
2	1.00	1.58	20.26
3	1.11	2.22	10.98
4	1.48	2.87	8.71
5	2.34	3.93	10.15
6	1.62	2.21	9.27
7	1.63	2.29	8.15
8	1.65	2.31	7.21
9	1.61	2.08	6.57
10	1.00	2.08	5.23
11	1.00	2.08	4.92
12	1.00	1.69	4.17
13	1.00	1.69	3.66
14	1.00	1.69	3.21
15	1.00	1.69	2.80
16	1.00	1.63	2.62
17	1.00	1.61	2.39
18	1.00	1.00	1.86

Table II. Execution time of HIF and the optimal algorithm for different values of p (here $k = 23$)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
HIF	6	6.4	6.4	6.4	6.6	6.4	6.8	6.4	6.6	6.8	6.8	6.8	7	7.4	7	7	7	7.2
OPT	0.0	0.0	0.2	1.4	5.6	18	45.4	94.8	162	232.4	284.4	296.2	263.4	195.4	121.8	63	27	9.2

is near-optimal when $p \geq 10$. In some cases, for smaller values of p its ratio to the optimal solution can reach 2.34. This is because HIF does not adapt dynamically its selection mechanism depending on the value of p. Indeed, while with $p = 1$, one must give a positive workload to the best solver, when $p = 2$ the best solver is not necessarily the one that could be combined with a second one for having the best solution. By the same way, some solvers selected when $p = 2$ might not be taken when $p > 3$ for having optimal performances.

HIF does not lead to the optimal distribution always. However, its provides a much more shorter time. This is reported in Table II which gives the optimal distribution with the optimal algorithm of [Knuth 1998] for computing the different possibilities of choosing p algorithms beyond 23. These measures are taken from executions on a 4 cores hyperthreaded architectures where each core has a frequence of $2661MHz$. A schematic view of the memory organization of this architecture is given at Figure 7.

The measures of execution time reveal an explosion of the optimal distribution time when $p \geq 6$ mainly due to the number of combinations that should then be taken into account. This explosion reaches a pic and then slows down when there are less combinations. The execution time of HIF is in general higher than those of the WIN or RAND distributions that are lower than 1. We explain this by the manipulation of g-OPT sets whose number is important when the number of instances is large. In our experiments, we limited the precision of maximal distance values (factors $\frac{C(A_i,I_j)}{C^*(I_j)}$) to 10^{-3}. Even if this certainly contributed to reduce the execution time, better solutions might be envisioned.

Finally, in Table III, we report the main results of our second series of experiments. We present the ratio between the workload of different distributions with the optimal distribution for different selection of subsets of solvers. We group the ratio per size of subsets of solvers selected beyond the 23 available. This experiment confirms the fact that HIF is in general better than the other distributions (except the optimal).

Table III. Ratio between HIF, Winners and Random for different values of k_m

k_m	HIF	WIN	RAND
4	1	1	8.99
5	1.03	1.02	6.03
6	1.16	1.304	7.76
7	1.26	1.41	11.76
8	1.23	1.45	5.21
9	1.21	1.48	6.15
10	1.25	1.43	7.24
11	1.26	1.46	7.48
12	1.18	1.38	6.21
13	1.32	1.71	5.04
14	1.34	1.60	5.68
15	1.26	1.61	5.45
16	1.33	1.76	5.86
17	1.35	1.8	10.91
18	1.28	1.87	6.03
19	1.272	1.74	6.19
20	1.28	1.75	7.26

For each k_m we selected a total of $5(k_m - 2)$ subsets

The experiments of this part show that the HIF distribution proposes an interesting compromise between the workload produced and the time required to compute it. We will see in the next section how it works while used for designing a portfolio of parallel solvers.

4.2. Designing a parallel algorithm for CSP

A binary Constraint Satisfaction Problem (CSP) is defined as a tuple $CP = (V, D, C, I)$ where:

— $V = \{v_1, \ldots, v_n\}$ is a set of variables
— $D = \{D(v_1), \ldots, D(v_n)\}$ is the set of definition domains for each variable
— $C = \{C_1, \ldots C_m\}$ is a set of constraints where each $C_i = (v_i^1, v_i^2) \in V \times V$
— $N = \{N_1, \ldots N_m\}$ is the set of incompatible assignments with each $N_i = \{(d_i^1, d_i^2) \in D(v_i^1) \times D(v_i^2)\}$ giving for C_i the assignments that will lead to unsatisfiability.

The objective is to find an assignment of values to the variables of V with no incompatible assignments. The CSP generalizes many other computational problems like SAT, graph coloring or the N queens. It is also important to notice that there exists a version of this problem where the objective is to find all the possible valid assignments. We do not consider this variant here. One of the most popular technique for solving CSP is backtracking which is presented below.

4.2.1. Backtracking for CSP. The idea in backtracking is to progressively assign a value to the variables $v_1, v_2, \ldots, v_n$. At each level, we check if the partial assignment realized fulfills all the constraints. Let denote an assignment as a tuple: $(v_1, f(v_1)), \ldots, (v_n, f(v_n))$ where v_i are variables and $f(v_i)$ are the assigned values. Given a partial assignment $(v_1, f(v_1)), \ldots, (v_u, f(v_u)), u < n$, if it is impossible to assign a value $f(v_{u+1})$ to v_{u+1} such that there is an incompatible assignment $(f(v_j), f(v_{u+1}))$, then we come back and reassign another value to v_u. We proceed this way until a complete assignment is obtained.

The progressive checking of partial assignments in backtracking reduces the exploration of useless search space since a solution is not completely generated before being

checked. However, it is important to have a good order for the assignment of values to variables. Such an order may reveal quickly incompatible assignments. This idea motivated the proposition of multiple ordering strategies for variables in backtracking.

We will restrict the study to 9 ordering heuristics:

— Lexicographic (lexicographic order on the variable names);
— Degree ordering with: max-degree and min-degree (dual of max-degree). max-degree prioritizes variables that are the most involved in constraints. The degree of a variable is the number of constraints in which it is involved;
— Forward degree ordering with: max-forward-degree(dynamically computes the maximum degree only for subset of non assigned variables), min-forward-degree (dual of max-forward-degree);
— Domain ordering with: min-domain (prioritizes variables for which we have the less number of values that are not part of incompatible assignments), max-domain (dual of min ordering);
— Domain/degree ordering with: min-domain/degree (prioritizes variables for which we have the smallest domain/degree), max-domain/degree (dual of min-domain/degree).

We illustrate some ordering strategies in Figure 4. We use there a bigraph representation of constraints. Edges between variables mean that there is an assignment of values that is not allowed for the two variables. On this example, the max-degree will take v_1 as the first variable since its degree is the greatest one (4). max-forward-degree will put also v_1 at the first position but will differ on the position of v_5.

In some orderings we might have situations where many variables can be selected for a same position (for example if they have the same degree). We applied the lexicographic order in these cases.

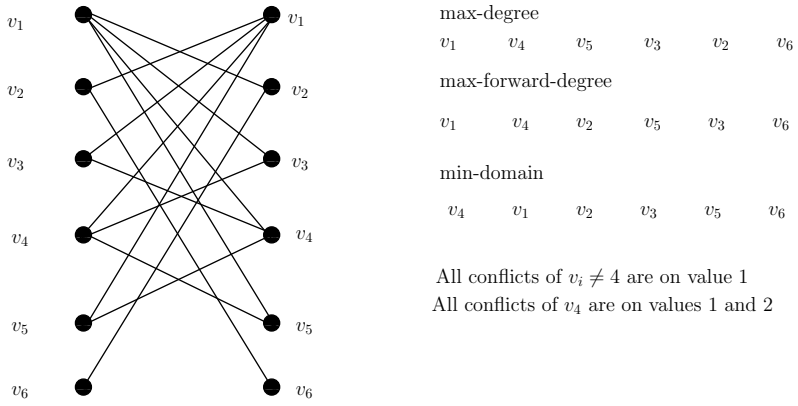

Fig. 4. Example of heuristic ordering for 6 variables v_i. The dual versions will give the reverse order

For each of these sequential backtracking algorithms we provide a parallel implementation.

4.2.2. Parallel Backtracking for CSP. A common approach for parallelizing backtracking consists of distributing the search by splitting the domain of the first variable(v_1) [Platzner and Rinner ; Habbas et al. 2000]. If $D(v_1)$ is splitted in l subdomains, we obtain l sub-problems differing on the domain of the first variable. The

parallel algorithm formulation is derived from the concurrent execution of a sequential algorithms on each sub-problem. This parallelization approach raises the question of efficient domain splitting for balancing the work among processors.

We address this question in a multithreaded context with shared memory. In this setting, concurrent executions for solving sub-problems are done with threads (each thread running a same sequential algorithm). As load balancing mechanism, we employ a dynamic work sharing beyond threads. When a thread does not have something to do, it steals a sub-problem x beyond the $|D(v_1)|$ that have not be considered by any thread. We proceed like this until a stopping criterion is reached (*i.e.* when a solution is found or when the exploration of all possibilities is complete).

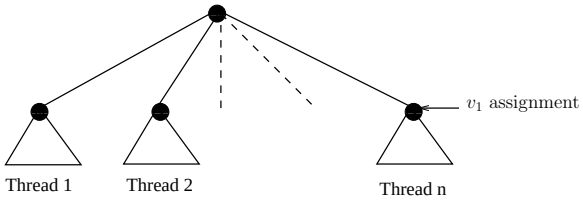

Fig. 5. Illustration of Parallel Backtacking with n threads. The search space is represented as a tree where the nodes correspond to a set of variable assignments. Each thread can steal a sub-tree where v_1 is already assigned.

From the approach described, multiple parallel backtracking can be derived for each ordering heuristic. In each of them, a thread executes a same sequential algorithm (corresponding to backtracking with an ordering heuristic) for solving a sub-problem. Since the threads have the same ordering strategy, they consider the same variable v_1. This simplifies the distribution of the search space.

4.2.3. Parallel Portfolio Backtracking for CSP. In a parallel portfolio of backtracking, we suppose that the different threads can implement different sequential backtracking algorithms in a same concurrent execution. On any instance to solve, all threads are executed until a stopping criterion is reached . We defined two versions of portfolio backtracking. The non-cooperative case where beyond threads of the same family (threads implementing the same sequential algorithms), one entire search space is shared and the cooperative case where there is a unique search space for all threads. In all theses cases the work is shared beyond threads using the dynamic work sharing principle described for parallel backtracking. For computing the resources allocation in the parallel portfolio, we used the *dRSSP* model with a benchmark of CSP instances. The interest of the non-cooperative case is to be more respectful of the solution suggested by *dRSSP*.

In the cooperative case we need to define a policy for distributing the search tree because since different threads have different orderings, the first variable might not be the same for all of them. For managing this, the first variable is taken as the variable with max-degree (for promoting propagation of constraints at the first level).

Figure 6 depicts the differences between parallel, non-cooperative and cooperative backtracking.

4.2.4. Stopping criteria and synchronization. For the considered CSP instances, we decide to stop the execution when a satisfiable assignment or a proof that the problem might not be satisfiable is obtained. This yields to the following conditions for stopping threads execution:

(1) A solution is found by one thread. In this case, we have a satisfiable assignment;

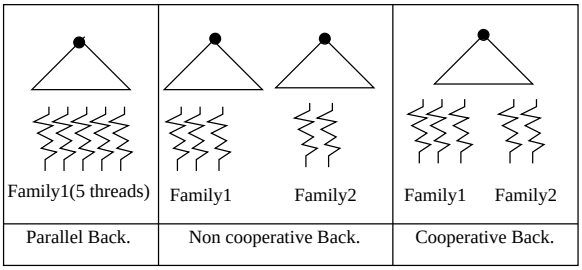

Fig. 6. In parallel backtraking, all threads are of the same family. It might not be the case in cooperative and non-cooperative backtracking. The difference between these two later algorithms is is that computations are not repeated in the cooperative algorithm

Table IV. Classes of CSP instances

class name	Model B parameters
A	$< 30, 8, 0.31, 0.34 >$
B	$< 30, 8, 0.18, 0.5 >$
C	$< 20, 8, 0.315, 0.34 >$
D	$< 25, 10, 0.26, 0.66 >$
E	$< 8, 6, 0.72, 0.45 >$
F	$< 25, 10, 0.105, 0.7 >$
G	$< 50, 10.0, 0.138, 0.33 >$
H	$< 22, 6.0, 0.6, 0.1 >$

(2) The exploration of the search space is completed without finding a solution. In this case, we have an unsatisfiable assignment;
(3) For the special case of non-cooperative backtracking, it is not useful to wait for the end of all threads. If indeed a family of threads ends their execution without success, then we have an unsatisfiable assignment.

Because the execution of a thread interrupt the others, a synchronization policy must be defined. We used a shared memory variable accessed for writing in mutual exclusion and events counters. While the shared memory variable serves to indicate that a satisfiable or unsatisfiable assignment is obtained, events counters serve to count the number of threads per families that ended their execution without satisfiability (useful for non-cooperative cases).

4.2.5. Experiments settings. Benchmark: For the experiments, we used a benchmark of Random CSP instances generated using the model B [Gomes et al. 2004]. In this model, a CSP class of instances is characterized by a tuple $< n, m, d, t >$. n here is the number of variables, m the domain size, d the proportion of constraints (we have exactly $d.\lfloor \frac{n(n-1)}{2} \rfloor$) and t the proportion of unsatisfiable assignments for each constraint (we have exactly $\lfloor t.d^2 \rfloor$ non authorized assignments per constraint). Using this benchmark, 9 classes of instances taken from the review of literature mainly of [Petrovic and Epstein 2007; Gomes et al. 2004] and described in table IV have been considered.

These classes cover different structures of a CSP problem like the situations where there are many variables and relatively small domains or fewer variables but larger domains [Petrovic and Epstein 2007]. Moreover, the experiments show that some of them (like class H) are particularly easy to solve while some others (G for example) are hard to solve.

Architecture: We run the experiments on a parallel multi-core machine with 4 cores. The cores have a frequence of $2661MHz$ and hyperthreading is used in each core. We give in Figure 7 a representation of the memory organization. This organization has

Table V. Cumulative execution times of parallel backtracking on p resources

p	lex.	min-deg.	max-deg.	min-dom	max-dom	min-forward-deg.	max-forward-deg.	min-dom/deg.	max-dom/deg
1	3814	4696	526	3711	3679	3920	3346	3848	3653
2	3513	4653	190	3493	3460	3798	2897	3505	3347
3	3354	4653	70	3374	3318	3657	2438	3329	3190
4	3263	4650	60	3312	3150	3576	2280	3210	3062
5	3245	4650	53	3285	3146	3571	2214	3196	3061
6	3268	4643	60	3307	3157	3513	2162	3272	3030
7	3175	4643	65	3204	3129	3546	2113	3190	2980
8	3170	4660	65	3190	3195	3645	2118	3129	2993

three cache levels. At level $L1$ and $L2$, each core has its own cache. The cache of level $L3$ is however shared between the different cores.

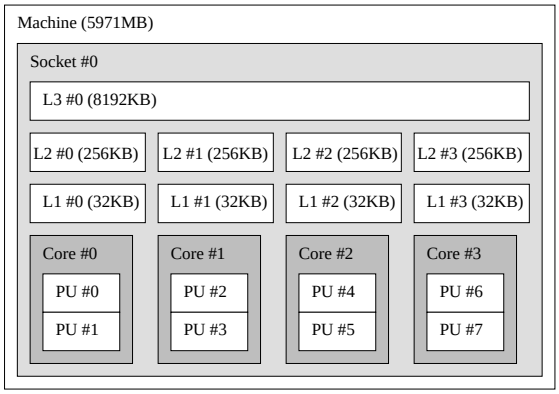

Fig. 7. processors and memory organization

According to the characteristics of this architecture, we consider that the number of concurrent threads that could be executed in parallel is fixed to 8. This is deduced from the theoretical limit of the benefit that can be expected from hyperthreading.

4.2.6. Experiments plan and results. The generation of the instances is based on a random generator [6]. For each of class of instances (A, B ..), we generated 25 instances. Since there are 9 classes, we have a total of 225 instances. For each instance, we measured the execution time for the different parallel backtracking algorithms. Table V reports the total time needed by each algorithm for solving the 225 instances with p threads. Each parallel backtracking algorithm is executed in at most 30 seconds.

The max-degree algorithm dominates clearly on all classes of considered instances. An interesting aspect in this table is the increase of the time of max-degree between $p = 5$ and $p = 6$. This increase is due to the parallelism overhead on the resolution of some instances when the number of threads increases. Let suppose for example that during the execution we have d sub-trees that can dynamically assigned. If for a sub-tree x, we have a satisfiable assignment that is the easiest one to find for all sub-trees, then we have a fast solution with x threads. If the number of threads increases, we do not have any benefit from parallelism but we have to pay an additional time for its management. These times are particular expensive when hyperthreading is used (specially when $p > 4$).

[6] available at http://www.lirmm.fr/~bessiere/generator.html

Table VI. Execution time variation on the number of resources

p	max-deg.	Theo. pred.	Non-Coop. back. (time, factor).	Coop. Back.
8	53	25	$(45, 1.8)$	720
7	65	36	$(58, 1.63)$	744
6	60	141	$(210, 1.34)$	805
5	53	402	$(459, 1.14)$	810
4	60	407	$(447, 1.09)$	812
3	70	423	$(465, 1.09)$	806
2	190	435	$(476, 1.09)$	603
1	526	526	$(526, 1)$	526

The factors give the overhead in comparison to theoretical previsions

The increase of the execution time for max-degree however suggests however a possible interesting situation. Indeed, we have an algorithm that does not need to use the total amount of possible threads. Thus, some of them can be devoted to the concurrent execution of another algorithm. The interest is that on instances where max-degree is not the best one, another algorithm may help to obtain a faster solution.

Using the benchmark data, we build cooperative and non-cooperative backtracking algorithms suggested by the resource sharing of our two phases approach. Table VI presents the execution time of these algorithms. After the resolution of *dRSSP* on p resources, we have a resource sharing $R(p)$ whose estimated execution time is in column *Theo. pred.*. The estimation is given by the objective function of *dRSSP* on the allocation $R(p)$. The execution times obtained from the real execution of the non-cooperative and cooperative backtracking algorithms with allocations $R(p)$ are in columns *Non-Coop. back* and *Coop. Back* . The value *factor* in column *Non-Coop. back* is the ratio between the effective observed time for non-cooperative backtracking and the theoretical estimations.

The non-cooperative backtracking has a better execution time than the best algorithm on $p = 7, 8$ resources. This means that on some instances where max-degree is not optimal, we benefit from the execution of another concurrent algorithm. There is however an important overhead between the theoretical predicted time for the non-cooperative backtracking and the real observed time. One explanation for this phenomenon is that theoretical measures do not take into account hyperthreading overhead. Indeed, the distance with theoretical estimations (factor) decreases when we reduce the number of threads (p) and it becomes more stable when there is no need to use hyperthreading.

However, when the number of resources is lower than 6 resources, the non-cooperative backtracking does not perform as well as the the best algorithm. This is caused by wrong computations of the distribution function in situations of super linear speed-up. On some instances (mainly for class G), max-degree has a large time units for $p = 1, 2$ resources and then, its time is close to 0 when $p = 3$. Since the workload distribution is decided from the performance of the algorithm on the sequential case, we do not consider while computing resource sharing that max-degree can have a good execution time for the resolution of these instances. The consequence is that it has less charges and when $p = 1, 2, 3, 4, 5$, we assign just one resource to it. Then, this makes the time of the portfolio close to the time of max-degree with one resource.

In all the considered cases, the cooperative backtracking is worse than the other solutions. Dynamic work has a responsibility in this. Typically, there are situations where a thread will take a sub-tree on which it does not know how to find quickly a solution in comparison to another thread of a different family that is fast for the sub-tree.

Table VII. Speed-up of different algorithms

p	max-deg.	Theo. pred.	Non-Coop back.	Coop. Back.
8	9.92	20.23	11.69	0.73
7	8.09	14.61	9.07	0.70
6	8.76	3.73	2.50	0.65
5	9.92	1.30	1.14	0.65
4	8.76	1.29	1.17	0.65
3	7.51	1.24	1.13	0.65
2	2.76	1.20	1.11	0.87

Finally, we present the speed-up achieved by the different algorithms in table VII. For each number of resources, the speed-up is defined as the ratio between parallel and sequential time. This table shows that it is possible to obtain at a speed-up greater than 8 (the number of threads). But, this is a common feature already observed on parallel backtracking algorithms [Habbas et al. 2000].

Experiments on CSP mainly showed that even on a hyperthreaded architecture with an algorithm that highly dominates the others, it is possible to build a combination of algorithms based on our two phases approach with better performances than any single algorithm taken alone. However, this approach can yield to poor performances when there are few resources and when the algorithms have a super linear speed-up. These experiments also show that in a search problem where an algorithm strongly dominates the others, it might be better to duplicate the work for the best algorithm in order to avoid situations where a weaker algorithm has a space containing a solution that the best algorithm can find much faster.

5. CONCLUSION

We have presented in this paper a new approach based on parallel algorithms portfolio for solving a set of instances of hard problems. This approach improves the other existing parallel portfolio approaches since it takes into account more features. It has been tested on two target problems, namely SAT and CSP. The obtained portfolios are better than any other algorithms. The main reason is that the proposed mechanism provides a learning process which can determine the complementary of the algorithms in the portfolio. For SAT, the experimental results are close to the lower bound corresponding to the best parallel solution based on the concurrent executions of sequential solvers. For CSP, we show how efficient distributions of the search space between multiple CSP algorithms can be designed. The corresponding portfolio has been implemented in a multi-threaded parallel multi-core machine on random instances. It is efficient and robust in the sense where the obtained results are good even in the case of a family where an algorithm dominates all the others.

Further improvements are possible. One promising direction is to take into account potential super-linear speed-ups (when the processing time of an algorithm can drastically decrease while the resources increase). This problem is closer to load balancing instead of resource sharing. Another issue is to incorporate some dynamicity in the solution proposed for the theoretical estimation of the loads in order to obtain a more precise approximation. Finally, the overheads induced by the concurrent execution of algorithms could be introduced into the definition of *dRSSP* for providing a more precise theoretical time estimation.

REFERENCES

BHOWMICK, S., MCINNES, L. C., NORRIS, B., AND RAGHAVAN, P. 2003. The role of multi-method linear solvers in pde-based simulations. In *ICCSA (1)*. 828–839.

BOUGERET, M., DUTOT, P., GOLDMAN, A., NGOKO, Y., AND TRYSTRAM, D. 2009. Combining multiple heuristics on discrete resources. In *11th Workshop on Advances in Parallel and Distributed Computational Models APDCM, (IPDPS)*.

BOUGERET, M., DUTOT, P.-F., GOLDMAN, A., NGOKO, Y., AND TRYSTRAM, D. 2011. Approximating the discrete resource sharing scheduling problem. *Int. J. Found. Comput. Sci. 22,* 3, 639–656.

BURKE, E. K., CURTOIS, T., HYDE, M. R., KENDALL, G., OCHOA, G., PETROVIC, S., RODRÍGUEZ, J. A. V., AND GENDREAU, M. 2010. Iterated local search vs. hyper-heuristics: Towards general-purpose search algorithms. In *IEEE Congress on Evolutionary Computation*. 1–8.

CORMEN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. 2001. *Introduction to algorithms*. MIT electrical engineering and computer science series. MIT Press.

DUTOT, P., RZADCA, K., SAULE, E., AND TRYSTRAM, D. 2009. *Multi-objective scheduling*. Chapman and Hall/CRC Press, Chapter 9. ISBN: 978-1420072730.

EL-ABD, M. AND KAMEL, M. 2005. A taxonomy of cooperative search algorithms. In *Hybrid Metaheuristics*. 32–41.

GAGLIOLO, M. AND SCHMIDHUBER, J. 2008. Algorithm selection as a bandit problem with unbounded losses. *CoRR abs/0807.1494*.

GEBRUERS, C., HNICH, B., BRIDGE, D. G., AND FREUDER, E. C. 2005. Using cbr to select solution strategies in constraint programming. In *ICCBR*. 222–236.

GOMES, C. AND SELMAN, B. 2001. Algorithm portfolios. *Artificial Intelligence 126,* 1, 43–62.

GOMES, C. P., FERNNDEZ, C., SELMAN, B., AND BESSIRE, C. 2004. Statistical regimes across constrainedness regions. In *Constraints*. Springer, 32–46.

HABBAS, Z., KRAJECKI, M., SINGER, D., METZ, U., REIMS, U., ARDENNE UNIVERSIT METZ, C., F-METZ, F, C., CEDEX, R., AND CEDEX, F.-M. 2000. Domain decomposition for parallel resolution of constraint satisfaction problems with openmp. In *In Proceedings of the Second European Workshop on OpenMP*.

HOUSTIS, E. N., CATLIN, A. C., RICE, J. R., VERYKIOS, V. S., RAMAKRISHNAN, N., AND HOUSTIS, C. E. 2000. PYTHIA-II: a knowledge/database system for managing performance data and recommending scientific software. *ACM Trans. Math. Softw. 26,* 2, 227–253.

HUBERMAN, B., LUKOSE, R., AND HOGG, T. 1997. An economics approach to hard computational problems. *Science 275,* 5296, 51–54.

KNUTH, D. E. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley second Edition.

MATTEO FRIGO, S. G. J. May 1998. FFTW: An adaptive software architecture for the FFT. In *proceedings of the International Conference on Acoustics, Speech and Signal Processing*. ACM SIGARC, Seattle, Washington.

NGOKO, Y. AND TRYSTRAM, D. 2009a. Combining numerical iterative solvers. In *PARCO*. Lyon, France, 43–50.

NGOKO, Y. AND TRYSTRAM, D. 2009b. Combining SAT solvers on dicrete resources. In *HPCS, International Conference on high performance computing and simulation*. IEEE Computer Society, Leipzig, Germany, 153–160.

PETROVIC, S. AND EPSTEIN, S. L. 2007. Random subsets support learning a mixture of heuristics. In *FLAIRS Conference*. 616–621.

PLATZNER, M. AND RINNER, B. Design and implementation of a parallel constraint satisfaction algorithm.

SAYAG, T., FINE, S., AND MANSOUR, Y. 2006. Combining multiple heuristics. In *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, February 23-25, 2006, Proceedings*, Durand and Thomas, Eds. LNCS Series, vol. 3884. Springer, Marseille, France, 242–253.

SIMON, L. AND CHATALIC, P. 2001. SATEx: a web-based framework for SAT experimentation. *Electronic Notes in Discrete Mathematics 9*, 129–149.

STREETER, M., GOLOVIN, D., AND SMITH, S. 2007. Combining multiple heuristics online. In *proceedings of the national conference on artificial intelligence*. Vol. 22. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, London, 1197.

XU, L., HUTTER, F., HOOS, H. H., AND LEYTON-BROWN, K. 2008. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res. (JAIR) 32*, 565–606.


L I G