

Parallelizing Combinatorial Search in Shared Memory

Zineb HABBAS	Michaël KRAJECKI	Daniel SINGER
LITA	LERI	LITA
Ile du Saulcy	BP 1039	Ile du Saulcy
F-57045 Metz Cedex	F-51687 Reims Cedex2	F-57045 Metz Cedex
zineb@iut.univ-metz.fr	michael.krajecki@univ-reims.fr	singer@lita.univ-metz.fr

Abstract

The Langford's problem is a typical hard combinatorial problem. The last open instance to be solved took one week CPU time on a pool of 3 PCs with a quite specific search algorithm in 2002. It represents a real *challenging* problem especially for Parallel Combinatorial Search. The paper presents a general framework for parallel resolution of combinatorial problems formulated as Constraint Satisfaction Problems (CSP). A first step consisting in a simple decomposition strategy of the Tree Search is sketched. This enables the choice of initial variables to generate independent tasks. The scalability of this approach is studied within the shared memory model using the standard OpenMP library. Because of its irregularity the load balancing question is crucial for parallel combinatorial search. Different static and dynamic policies offered by OpenMP are studied. The experiments were carried out running on the Silicon Graphics Origin'3800 500MHz R14K parallel machine. For the Langford's problem we obtain close to linear speed-ups until 256 processors with some breakdown depending of the granularity.

Keywords: combinatorial search, Langford's problem, OpenMP, parallel processing, shared memory.

1 Introduction

In Computer Science many problems can be represented as Constraint Satisfaction Problems (CSP) especially in Artificial Intelligence. Typically these problems are NP-Complete and they require extensive search to find a solution. In this paper the considered problem is known as the Langford's problem named for Scottish mathematician C. Dudley Langford who once observed his son playing with colored blocks. He noticed that the child had arranged three pairs of colored blocks such that there was one block between the red pair, two between the blue pair, and three between the yellow pair, like so:

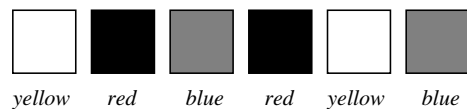


Figure 1: $L(2, 3)$: Arrangement solution for 6 blocks with three colors: yellow, red and blue

From this simple observation the problem has been generalized to any number of colors and any number of blocks taking the same color. More generally, $L(s, n)$ denotes the problem of finding the number of solutions for the Langford's Problem where n indicates the number of colors and s the number of blocks having the same color. Martin Gardner posed $L(2, 4)$ in a collection of short

problems in Mathematical Games of Scientific American (Nov. 1967) and explained that $L(2, n)$ has been shown to be solvable when n is any multiple of four or one less. Practically, the problem instances which can be solved by a specific combinatorial program until now are limited to a small number of colors. For example, it took 2,5 years to solve the $L(2, 19)$ instance problem on a single 300MHz DEC Alpha in 1999. In 2002, the $L(2, 20)$ instance has been solved in one week thanks to the use of a pool of 3 PCs with a specific search algorithm. The Langford's Problem has been approached in different ways (discrete mathematics results, specific algorithms, specific encoding, ...) [7].

Recently, Toby Walsh and Barbara Smith formulated this problem as a Constraint Satisfaction Problem [14, 12]. They presented it as a prototype for the Permutation Problems class. Here we go further, and because of its hardness, we consider it as a real interesting *challenging* constraint satisfaction problem thus improving the CSP methodology. Independently of the specific Discrete Mathematics approach, we claim to solve the hardest open instance within the CSP Resolution framework especially with parallel computation. Consequently, the first contribution of the paper is to propose an efficient CSP formalization of the problem. We define another encoding we named "compact" encoding because it divides by two the number of variables and significantly reduces the resolution time compared to the basic one. We have experimented these two encodings by using the basic FC-MRV algorithm for $L(2, n)$ where n lies between 10 and 16 colors and we notice that the second one reduces the computational cost by a factor of 10. But in any way we encode the Langford's problem with CSP, the computational cost remains prohibitory. It consumes approximately 1 hour and 20 minutes with the compact encoding in the sequential case for $L(2, 15)$. Another observation is that solving $(L2n + 1)$ is at most 10 times harder than solving $L(2, n)$. So, we can bound the sequential time to solve $L(2, 16)$ by 13 hours, and the first open instance $L(2, 20)$ is still far away for CSP resolution! For this challenge the parallel resolution framework is the second contribution of this paper.

Although parallelization seems to be a good candidate to obtain further practical improvements, the research in this direction is not very developed. In a previous work, we have studied a domain decomposition method defined by Chmeiss and Jegou for parallel resolution of CSP with a shared memory [4, 1]. We can sum up it by the three following observations. First, the domain decomposition method behavior is not the same for consistent¹ and for inconsistent problems. Second, the parallel version of this method can give super-linear speed-up for consistent problems and linear speed-up for inconsistent ones. Third, the main drawback of the method is that the number of sub-problems is only proportional to the domain size thus it does not give enough tasks to exploit all the potentiality of parallel resolution.

In [9], we explored another approach for the parallel resolution of CSP. It consists in distributing the search tree itself in the message passing model. The main drawback of this last approach is the cost induced by dynamic load balancing management.

To overcome these drawbacks, this paper presents a *shared memory* parallel framework for a search tree distributed algorithm. We investigate the programming feasibility of the method and its performances with the new emergent standard of shared memory management tool: the OpenMP directive based parallel computation environment [11]. This study may be relevant for the more general framework of the *irregular applications* not suited for parallelization and especially for shared memory implementation. Moreover we take the Langford's problem as a CSP challenging problem.

Another contribution of this paper is to prove the applicability of a shared memory parallel environment for coarse grain parallelism. Using a shared memory scheme, it seems to be easier to solve the load balancing problem.

In [5], the first experiments we have made on the Langford's problem have led to linear speed-up until 32 processors. In this paper, it will be shown that, thanks to a more accurate task generation, we

¹Consistent problem: problem with at least one solution.

can efficiently exploit until 256 processors.

The paper is organized as follows. Section 2 presents the Langford's problem and introduces its CSP modeling. Section 3 outlines a parallel resolution framework with search tree distribution. Section 4 is dedicated to the parallel resolution of CSP using a shared memory. Section 5 gives the first results of this approach. Finally, we conclude by giving some perspectives for this study.

2 The Langford's problem

2.1 Background

Langford's problem is named for Scottish mathematician C. Dudley Langford who once observed his son playing with colored blocks. He noticed that the child had arranged three pairs of colored blocks such that there was one block between the red pair, two between the blue pair, and three between the yellow pair. If we add a fourth color (green e.g) four blocks have to stand between the two greens. In this case the solution is unique. Reversing the order is not significant, because all you have to do is walk around to the other side of a given arrangement and view it from that side. Generalizing from colors to numbers, the above example became 4 1 3 1 2 4 3 2. In 1958, Langford's submission to Mathematical Gazette gave one arrangement for the 7, 8, 11, 12, and 15 pairs of numbers. He noted that arrangements did not seem to be possible for 5, 6, 9, or 10 pairs, so he called for a theoretical treatment: what values of n were solvable? In a later paper, Roy O. Davies wondered how many solutions there were for different n 's. The stage was set for a popular computing problem. The Web site of John E. Miller [7] presents more results, variants and pointers for this problem.

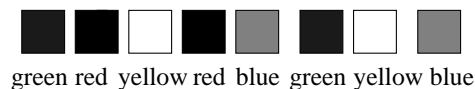


Figure 2: $L(2, 4)$: Arrangement solution for 8 blocks with four colors: green, yellow, red and blue

This problem was first formulated as a constraint satisfaction problem by the APES research group² especially Toby Walsh and Barbara Smith. In particular, they used it as a permutation problem prototype: *i. e.* it has the same number of values as variables, all variables have the same domain and each variable must be assigned a different value. Different ways of modeling this problem and finding all solutions have been investigated.

T. Walsh performs an extensive study of different models for general permutation problems [14]. He defines a measure of constraint tightness parameterized by the level of local consistency being enforced. It assumes constraints are defined over the same variables and values or, are bijectively related. He illustrates a methodology to compare different constraint models. In a permutation problem, we have as many values as variables, and each variable takes an unique value. We can therefore swap variables for values. Many assignment, scheduling and routing problems are permutation problems.

In the Dual model studied by B. Smith the roles of the variables and values are interchanged [12]. The Dual model is useless by itself for solving most instances of this problem because its constraint propagation is so poor. This is the reason why it is interesting to introduce "channelling constraints" linking primal and dual variables. B. Smith observes that channelling constraints remove the need for the primal all-diff constraint. It would be worthwhile to combine models (primal and dual) if channelling constraints are added. On Langford's problem, B. Smith found that MAC algorithm with the channelling constraints is often the most competitive model for finding all solutions. She predicts that these results will transfer over to other permutation problems but it is hard to derive coherent conclusions from the results for finding the first solution.

²<http://www.dcs.st-and.ac.uk/apes/>

2.2 The CSP modeling of the Langford's problem

In the sequel, the classical definition of a CSP $\mathcal{P} = (X, D, C, R)$ given by Montanari will be used [10]. A *binary CSP* is a CSP where all the constraints are sets of two variables at most. An *instantiation* of a set of variables A is a k -tuple (a_1, \dots, a_k) , representing an assignment of $a_j \in D_j$ to X_j , for all $X_j \in A$. A *consistent instantiation* of A is a set of assignments that satisfies all the constraints C_k such that $C_k \subseteq A$. A consistent instantiation on X is a *solution* of the CSP.

In this paper, we consider that solving a CSP is *finding all the solutions*. In fact the Langford's problem is to count the number of solutions. We also restrict the study to binary CSP on Finite Domains where the relations are given *in extension* by all the admissible value pairs.

We present below the *basic* and *compact* CSP formulations of the Langford's problem. The basic one corresponds to the primal one presented in [12] and we called the second one compact because the number of variables is reduced and all the information concerning the constraints, especially the *alldiff* one is compacted inside the relations. This is much more clear when the number of blocks of the same colors s is growing and we let it as an exercise for the reader. First the 4 colors example is detailed and then the general case is described.

2.2.1 The 4 colors example : $L(2, 4)$

Basic encoding

The basic 4 colors Langford's problem is specified as the following CSP $\mathcal{P} = (X, D, C, R)$ where

1. Variables X correspond to the positions of the 8 blocks: $X = \{X_1, X_2, \dots, X_8\}$
2. Domains $D = \{D_1 = \{1, \dots, 6\}, D_2 = \{1, \dots, 5\}, D_3 = \{1, \dots, 4\}, D_4 = \{1, 2, 3\}, D_5 = \{3, \dots, 8\}, D_6 = \{4, \dots, 8\}, D_7 = \{5, \dots, 8\}, D_8 = \{6, 7, 8\}\}$
3. Constraints $C = \{C_{1,5} = \{X_1, X_5\}, C_{2,6} = \{X_2, X_6\}, C_{3,7} = \{X_3, X_7\}, C_{4,8} = \{X_4, X_8\}, C_{alldiff} = \{X_1, \dots, X_8\}\}$
4. Relations $R = \{R_{1,5}, R_{2,6}, R_{3,7}, R_{4,8}, R_{alldiff}\}$ where :
 - $R_{1,5} \Leftrightarrow \{X_1 + 2 = X_5\} = \{(1, 3), (2, 4), (3, 5), (4, 6), (5, 7), (6, 8)\}$
 - $R_{2,6} \Leftrightarrow \{X_2 + 3 = X_6\} = \{(1, 4), (2, 5), (3, 6), (4, 7), (5, 8)\}$
 - $R_{3,7} \Leftrightarrow \{X_3 + 4 = X_7\} = \{(1, 5), (2, 6), (3, 7), (4, 8)\}$
 - $R_{4,8} \Leftrightarrow \{X_4 + 5 = X_8\} = \{(1, 6), (2, 7), (3, 8)\}$
 - $R_{alldiff} \Leftrightarrow \{X_1 \neq X_2 \neq \dots \neq X_8\}$

Remark: it is important to note the *alldiff* constraint will neither be implemented as a n -ary constraint, nor as binary constraints. The matrix data structure will permit us to manage it very efficiently in implicit way.

Compact encoding

The compact 4 colors Langford's problem is specified as the following CSP $\mathcal{P} = (X, D, C, R)$ where

1. Variables X correspond to the first positions of the 4 blocks $X = \{X_1, X_2, X_3, X_4\}$
2. Domains $D = \{D_1 = \{1, \dots, 6\}, D_2 = \{1, \dots, 5\}, D_3 = \{1, \dots, 4\}, D_4 = \{1, 2, 3\}\}$
3. Constraints $C = \{C_{1,2} = \{X_1, X_2\}, C_{1,3} = \{X_1, X_3\}, C_{1,4} = \{X_1, X_4\}, C_{2,3} = \{X_2, X_3\}, C_{2,4} = \{X_2, X_4\}, C_{3,4} = \{X_3, X_4\}\}$

4. Relations $R = \{R_{1,2}, R_{1,3}, R_{1,4}, R_{2,3}, R_{2,4}, R_{3,4}\}$ where

- $R_{1,2} \Leftrightarrow \{X_1 \neq X_2, X_1 + 2 \neq X_2 + 3, X_1 \neq X_2 + 3, X_2 \neq X_1 + 2\} = \{(1, 2), (1, 4), (1, 5), (2, 3), (2, 5), (3, 1), (3, 4), (4, 2), (4, 5), (5, 1), (5, 3), (6, 1), (6, 2), (6, 4)\}$
- $R_{1,3} \Leftrightarrow \{X_1 \neq X_3, X_1 + 2 \neq X_3 + 4, X_1 \neq X_3 + 4, X_3 \neq X_1 + 2\} = \{(1, 2), (1, 4), (2, 1), (2, 3), (3, 2), (3, 4), (4, 1), (4, 3), (5, 2), (5, 4), (6, 1), (6, 3)\}$
- $R_{1,4} \Leftrightarrow \{X_1 \neq X_4, X_1 + 2 \neq X_4 + 5, X_1 \neq X_4 + 5, X_4 \neq X_1 + 2\} = \{(1, 2), (2, 1), (2, 3), (3, 1), (3, 2), (4, 2), (4, 3), (5, 1), (5, 3), (6, 2)\}$
- $R_{2,3} \Leftrightarrow \{X_2 \neq X_3, X_2 + 3 \neq X_3 + 4, X_2 \neq X_3 + 4, X_3 \neq X_2 + 3\} = \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 1), (3, 4), (4, 1), (4, 2), (5, 2), (5, 3)\}$
- $R_{2,4} \Leftrightarrow \{X_2 \neq X_4, X_2 + 2 \neq X_4 + 5, X_2 \neq X_4 + 5, X_4 \neq X_2 + 3\} = \{(1, 2), (1, 3), (2, 1), (2, 3), (3, 2), (4, 1), (4, 3), (5, 1), (5, 2)\}$
- $R_{3,4} \Leftrightarrow \{X_3 \neq X_4, X_3 + 4 \neq X_4 + 5, X_3 \neq X_4 + 5, X_4 \neq X_3 + 4\} = \{(1, 2), (1, 3), (2, 3), (3, 1), (4, 1), (4, 2)\}$

2.2.2 The general case : $L(2, n)$

The previous 4 coloring example can be easily generalized to the n coloring instance for both encodings.

Basic encoding

The Langford's problem with n colors is specified as a CSP $\mathcal{P} = (X, D, C, R)$ as follows.

1. Variables $X = \{X_1, X_2, \dots, X_{2n}\}$
2. Domains $D = \{D_1 = \{1, \dots, 2n-2\}, D_2 = \{1, \dots, 2n-3\}, \dots, D_{n+1} = \{3, \dots, 2n\}, \dots, D_{2n} = \{n+2, \dots, 2n\}\}$
3. Constraints $C = \{C_{1,n+1} = \{X_1, X_{n+1}\}, C_{2,n+2} = \{X_2, X_{n+2}\}, \dots, C_{n,2n} = \{X_n, X_{2n}\}, C_{alldiff} = \{X_1, \dots, X_{2n}\}\}$
4. Relations $R = \{R_{1,n+1} = \{(1, 3), \dots, (2n-2, 2n)\}, \dots, R_{n,2n} = \{(1, n+2), \dots, (n-1, 2n)\}, R_{alldiff}\}, R_{i,n+i} \Leftrightarrow \{X_i + (i+1) = X_{n+i}\}$

Compact encoding

The Langford's problem with n colors is specified as a CSP $\mathcal{P} = (X, D, C, R)$ as follows.

1. Variables $X = \{X_1, X_2, \dots, X_n\}$
2. Domains $D = \{D_1 = \{1, \dots, 2n-2\}, D_2 = \{1, \dots, 2n-3\}, \dots, D_n = \{1, \dots, n-1\}\}$
3. Constraints $C = \{C_{1,2} = \{X_1, X_2\}, C_{1,3} = \{X_1, X_3\}, \dots, C_{n-1,n} = \{X_{n-1}, X_n\}\}$
 $C = \{C_{i,j} = \{X_i, X_j\} / i \in \{1..n-1\}, j \in \{2..n\}, i < j\}$
4. Relations $R = \{R_{1,2} = \{(1, 2), (1, 4), (1, 5), \dots, (2n-2, 2n-4)\}, R_{1,3} = \{(1, 2)(1, 4), (2, 1), \dots, (2n-2, 2n-5)\}, \dots, R_{n-1,n} = \{(1, 2), (1, 3), \dots, (n, n-2)\}\}$
 $R_{i,j} \Leftrightarrow \{X_i \neq X_j, X_i + (i+1) \neq X_j + (j+1), X_i + (i+1) \neq X_j, X_i \neq X_j + (j+1)\}$

Remark:

- Notice that in the compact encoding the *alldiff* n-ary constraint is hidden in all the binary constraints as it is the case with the channelling constraints introduced by B.Smith [12].
- The compact encoding can very easily be applied to the most general $L(s, n)$ Langford's problem.
- It is an open question if this compact encoding can be applied to general permutation problems.

Proposition. *The compact encoding is correct with respect to the basic one and conversely.*

Proof. We prove the correctness by showing that a given solution of the compact encoded Langford problem is also a solution of the basic encoded one and conversely.

1. Compact solution \longrightarrow Basic solution

Let be $y_1 = v_1 \dots y_i = v_i \dots y_n = v_n$ a compact solution satisfying :

$\forall i \in \{1 \dots n - 1\}, \forall j \in \{2 \dots n\}, i < j$

$$\left\{ \begin{array}{l} 1) \quad v_i \quad \neq \quad v_j \\ 2) \quad v_i + i + 1 \quad \neq \quad v_j + j + 1 \\ 3) \quad v_i + i + 1 \quad \neq \quad v_j \\ 4) \quad v_j + j + 1 \quad \neq \quad v_i \end{array} \right\}$$

Then we can show by construction that :

$x_1 = v_1, \dots, x_i = v_i, \dots, x_n = v_n, x_{n+1} = v_1 + 2, \dots, x_{n+i} = v_i + i + 1, \dots, x_{2n} = v_n + n + 1$ is a basic solution.

(a) Domains of $x_1 \dots x_i \dots x_n$ are the domains of $y_1 \dots y_i \dots y_n$.

The domains of $x_{n+1} \dots x_{n+i} \dots x_{2n}$ are the following ones:

$v_1 + 2 \in \{3, \dots, 2n\}$ since $v_1 \in \{1, \dots, 2n - 2\}$

$v_i + i + 1 \in \{i + 2, \dots, 2n\}$ since $v_i \in \{1, \dots, 2n - (i + 1)\}$

$v_n + n + 1 \in \{n + 2, \dots, 2n\}$ since $v_n \in \{1, \dots, n - 1\}$

(b) $\forall k \in \{1 \dots n\}$ the list of values (v_1, \dots, v_{2n}) satisfies the relation:

$R_{k, n+k} \Leftrightarrow x_k + (k + 1) = x_{n+k}$.

(c) The *alldiff* constraint is satisfied since

- from hypothesis $\forall i, j \in \{1 \dots n\} v_i \neq v_j$
- $\forall i \in \{1 \dots n\}, \forall j \in \{n + 1 \dots 2n\} v_i \neq v_j$. We show this by refutation. Assume that $\exists i \in \{1 \dots n\}$ and $\exists j \in \{n + 1 \dots 2n\} | v_i = v_j$. If $v_i = v_j$ then $v_j = v_{n+k}$ and then $v_i = v_{n+k} = v_k + (k + 1)$. This contradicts the constraints 3 or 4.
- $\forall i \in \{n + 1 \dots 2n\}, \forall j \in \{n + 1 \dots 2n\} v_i \neq v_j$. Assume that $\exists i \in \{n + 1 \dots 2n\}$ and $\exists j \in \{n + 1 \dots 2n\} | v_i = v_j$. If $v_i = v_j$ then $\exists k \in \{1 \dots n\}$ and $\exists l \in \{1 \dots n\}$ with $i = n + k, j = n + l$ and $v_{n+k} = v_{n+l}$ and then $v_k + (k + 1) = v_l + (l + 1)$. This contradicts the constraint 2.

2. Basic solution \longrightarrow compact solution

Let be $x_1 = v_1 \dots x_i = v_i \dots x_n = v_n, x_{n+1} = v_{n+1} \dots x_{2n} = v_{2n}$ a basic solution satisfying:

$\forall i \in \{1 \dots n - 1\}, \forall j \in \{2 \dots n\}, i < j$.

$$\left\{ \begin{array}{l} \forall i \in \{1 \dots n\} \quad v_{n+i} = v_i + i + 1 \\ \forall i, \forall j \in \{1 \dots n\} \quad v_i \neq v_j \end{array} \right\}$$

then we have to show that $y_1 = v_1, \dots, y_i = v_i, \dots, y_n = v_n$ satisfies all the constraints of the compact solution.

- (a) Domains of $x_1 \dots x_i \dots x_n$ are the domains of $y_1 \dots y_i \dots y_n$
- (b) All the constraints are satisfied since the following propositions are verified by construction:
- Based on the hypothesis, $R_{i,j} \leftrightarrow y_i \neq y_j$
 - $y_i + (i + 1) \neq y_j + (j + 1)$ since if $\exists i \mid v_i + (i + 1) = v_j + (j + 1)$ then $v_{n+i} = v_{n+j}$ with the first basic encoding. As $i \neq j$ this contradicts the alldiff constraint.
 - $y_i + (i + 1) \neq y_j$ since if $\exists i \mid v_i + (i + 1) = v_j$ then $v_{n+i} = v_j$ with the first basic encoding. As $i \neq j$ this contradicts the alldiff constraint.
 - $y_i \neq y_j + (j + 1)$ since if $\exists i \mid v_i = v_j + (j + 1)$ then $v_i = v_{n+j}$ with the first basic encoding. As $i \neq j$ this contradicts the alldiff constraint.

3 Parallel Resolution of CSP

3.1 Related works

The past twenty years have seen a flurry of activity in the area of parallel and distributed computing. Most of the early results were developed for PRAM-like platforms, featuring various degrees of synchronization. While these models proved to be ideal test beds for algorithm development, they ignored key implementation issues, including interprocessor communication, memory access, and synchronization. As a consequence, we are witnessing an increasing interest in more realistic computational models that better address practical concerns. In recent years, novel parallel and distributed computational models have been proposed in the literature, reflecting advances in new computational devices and environments such as Field Programmable Gate Arrays (FPGA), clusters, DNA computing, quantum computing, etc. These new models has lead to significant advances in the resolution of various difficult problems of practical interest.

Concretely, there are mainly two different approaches to use parallel computation for constraints solving (see figure 3).

1. Splitting the initial CSP into a collection of easier subproblems obtained by some decomposition method (domain decomposition, structural decompositions, etc.) to solve them in parallel.
2. Distributing the search tree among the processors.

Notice that both approaches may be seen as initial static work distribution strategies.

Most of the significant developed works are related to the second method, but only within the MIMD and message passing parallelism model. We have investigated this paradigm in previous works [3, 9].

A recent paper explores another paradigm for parallel resolution of CSP based on the no-good (or inconsistent partial instantiations) recording and exchanging between the processors [13]. In this last work the same problem is solved by different serial resolution algorithms based on different heuristics.

Thanks to the emerging OpenMP standard for high level shared memory managing tool, here is the first practical study, in our knowledge, on the parallel resolution of CSP within a shared memory model.

The reader may find more details and results on the first approach with a domain decomposition strategy in shared memory in [4, 6].

In the sequel, we present the results of the second approach obtained for the $L(2, n)$ Langford's problem in shared memory.

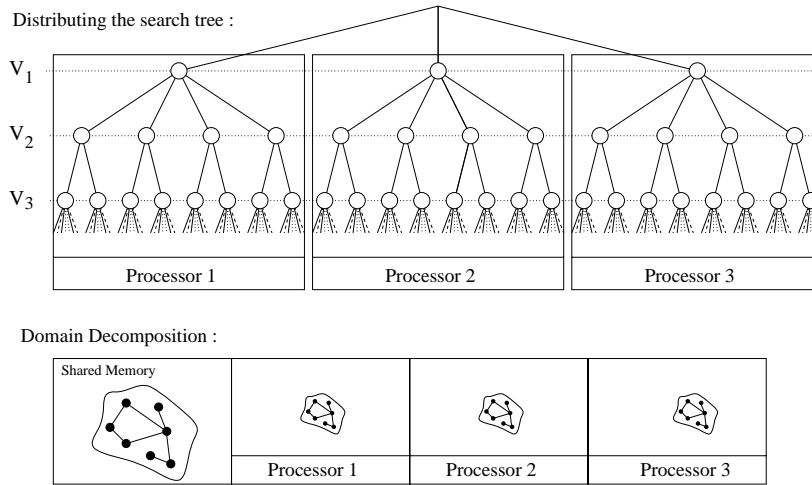


Figure 3: Two parallel CSP resolution approaches.

3.2 Search tree distribution

The domain decomposition method splits the initial problem into a collection of independent subproblems which can be solved in parallel without communication. Unfortunately, in our experiments we found the number of subproblems deriving from this method only proportional to the size of the variable domains. This is why, the domain decomposition is clearly inadequate to a massively parallel architecture.

In order to improve the performance of the parallel resolution of CSP, we have to increase the number of parallel tasks and this can be done by using a partition of the search tree.

The method consists in a preprocessing step of dividing the search tree in independent tasks such that the whole number of tasks is greater than the number of processors. This is done by assigning all the possible values to some k initial variables corresponding. The same technique named *variable elimination* has been studied in the sequential programming framework and thus for a different objective by Dechter, Pearl and by Larrosa [2, 8].

Larrosa's algorithm transforms a problem into a set of subproblems by selecting a variable and considering the assignment of each of its domain values. The subproblems are solved in sequence applying recursively the same transformation rule. It proposes a general solving scheme which combines search and variable elimination. In [2], the algorithm uses adaptive consistency and proceeds by selecting one variable at a time and replacing it by a new constraint which summarizes the effect of the chosen variable.

The main point of our shared memory proposition consists in fixing (and then eliminating) k variables of the CSP. This induces d^k subproblems with $n - k$ variables each where d is the uniform domains size. Hence, each processor is quite easily assigned more than one task. We consider as a main parameter of this procedure the k named *DepthLevel* of the initial tree search subdivision. It is necessary that $DepthLevel \geq \log_d(Nbproc)$ to ensure that the number of generated tasks assigned to each processor be greater than 1. $d^{DepthLevel}$ tasks are generated with this method compared to approximately d tasks for the Domain Decomposition one. Notice that this parameter can be used for tuning the parallel granularity with respect to the size of domains or to the number of processors.

3.3 How to choose the initial variables?

In order to split the search tree into a collection of subtrees we have to assign all the possible values to some *DepthLevel* variables. We present four different strategies in order to fix the first initial

variables.

1. *the naive strategy* chooses the variables corresponding to the first variables for a given dynamic or static ordering.
2. *the search strategy* consists in assigning the variables of the strongest constraints, that is the constraints with the smallest relation. This technique induces few parallelism but reduces the global search effort by relaxing the tightest constraints.
3. *the parallel strategy* consists in assigning the variables of the weakest constraints, that is the constraints with the largest relation. Clearly this strategy induces more parallelism but the global search space is less reduced.
4. *the independent strategy* chooses independent variables (not interconnected) which are the most connected to the rest of the constraint graph.

4 The shared memory implementation framework

The sequential search algorithm is developed in an Object Oriented style with C++. This work is only concerned with the significance of the Parallel Resolution of CSP by distributing the search tree in a shared memory. We only present in this section the first results on parallel experiments compared to sequential ones. All the presented experiments are fulfilled by using a standard Forward Checking algorithm with Minimum Remaining Value ordering heuristic (FC-MRV) as a basic search algorithm. Of course all this general approach may be applied to more efficient and sophisticated serial algorithms.

4.1 The standard OpenMP Library

To evaluate the performance of the parallel resolution of CSP we use OpenMP for shared memory parallelism [11]. OpenMP is a complete API for programming shared memory machine. It is easy to obtain parallel code since it is quite close to the sequential one. In addition OpenMP makes it possible to test different work distribution policies. OpenMP derives from the ANSI X3115 efforts. It is a set of compiler directives and runtime library routines that extend a sequential programming language to express parallelism with a shared memory (see Appendix A). OpenMP conforms to SPMD programming language style. Moreover, from another point of view the development cost of an MPI version is much more important and the portability seems to be lower because of the architecture communication network dependency. It potentially induces numerous additional programming efforts to deal with the crucial load balancing problem and especially for irregular applications which is the case of CSP.

4.2 The tasks allocation and OpenMP

Within the OpenMP environment the tasks allocation to the processors (*threads*) can be done very easily by one compilation directive following three ways :

1. The *Modulo Nbproc* repartition: the figure 4 illustrates this algorithm for a given problem. Here, we consider 7 processors with *DepthLevel* = 3 and the tasks numbered 5, 12 and 19 are affected to processor P_5 . The main advantage of this initial static distribution is to distribute the a priori irregularity of the search tree among all the processors.
2. The *Dynamic* repartition: the different tasks are allocated to processors dynamically by the system at the execution time but there is no guarantee on which thread the tasks are executed.

3. The *Static* repartition: the tasks allocation to the processors is computed once at the compiling time. Each processor receives $\frac{Nbtasks}{Nbproc}$ different subproblems.

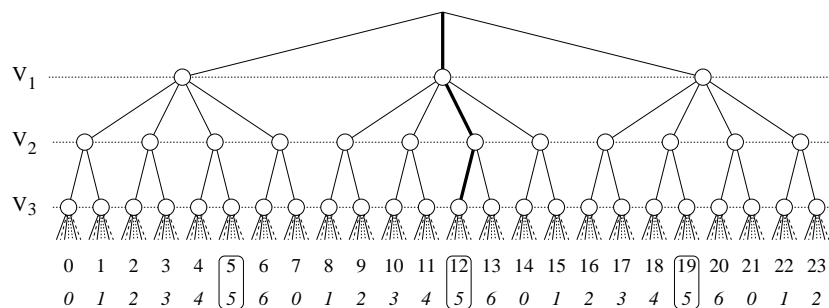


Figure 4: tasks distribution to the processor P_5

4.3 The binary CSP memory representation

We use two data structures to represent a particular CSP: the first one is an adjacency matrix ($Adj[][]$) which enables to check whether a relation between two variables of the problem exists or not. The second one is a list of couples defining the relation when a constraint between X_i and X_j exists. $Adj[X_i, X_j]$ is a reference to the list of allowed couples for the relation R_{ij} . We can now evaluate the space complexity of these data structures. For the adjacency matrix, the space complexity is $O(n^2)$ where n is the variables number. In the worst case, which is the case for the compact encoding, the CSP contains $\frac{n \times (n-1)}{2}$ relations and the length of each list is $O(d^2)$ where d is the maximum domain size.

4.4 The dynamic memory allocation problem

The search algorithm itself needs some working data structures to efficiently investigate the search tree. For example, FC maintains n tables of size d called *currentdomain* to compute the value allowed for each variable while the search is performed. It also uses n stacks to perform the filtering and to restore the tree when inconsistency is discovered.

In C and C++, it is possible to dynamically allocate the memory for each table. Moreover the memory used by the application in this case is exactly equal to the required space to solve the particular CSP. This is the reason why we have decided to use dynamic memory allocation in our first experiments. But doing this we obtain a very poor efficiency of our parallel algorithm and it was especially the case for 8 and more processors. The general following problem appears with OpenMP: when a program allocates dynamically the memory (using `new` or `malloc`), the new object is necessarily shared and there is no way to make it private. In our application, we need private structures for the FC algorithm in order to be as efficient as possible. At the end we have to choose to allocate statically (at the compilation time) the table *currentdomain* and the needed stacks.

5 Experimental results

In this section, we give an experimental evaluation of the parallel resolution of the Langford's problem (expressed as a CSP) based on the search tree distribution with OpenMP on the Silicon Graphics Origin'3800 500MHz R14K parallel machine.

5.1 Comparing dynamic and modulo policies

First we compare both the *Dynamic* and the *Static Modulo* repartition policies for the search decomposition strategy. Notice the *Static* one is not presented here, because its poor efficiency due to the irregularity of our application.

The figure 5 shows the efficiencies of the dynamic strategy (a) and the modulo one (b) from 1 to 256 processors. The dynamic strategy is always superior to the modulo one. Harder is the problem better are the speed-ups but the poor scalability can be explained by the low number of generated tasks: 156, 182, 210 respectively. Nevertheless, it can be concluded that the dynamic directive provided by OpenMP is very efficient for irregular tree search.

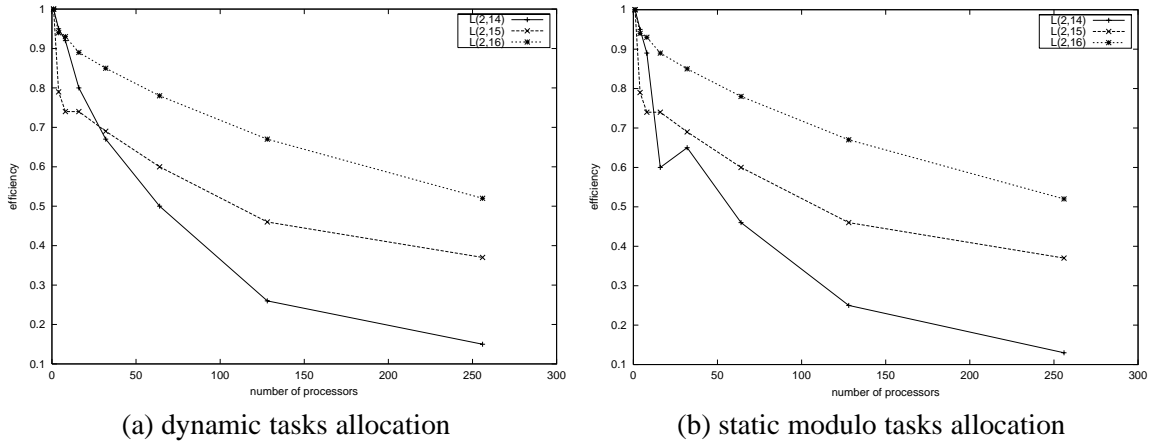


Figure 5: efficiencies for problems 14, 15, 16

5.2 Comparing the two decomposition strategies

We compare the *search* and *parallel* strategies to choose the variables to be initially fixed. The double objective is to reduce the general search effort and at the same time to give enough parallelism.

The table 1 presents the CPU times (in seconds) obtained for the problems 14, 15 and 16 both for the *parallel* and *search* strategies. In the second column we give the number of generated tasks (*nb_tasks*). Clearly the last one improves the CPU time by a factor of 0.3 approximately since it instantiates the more constrained variables. But the efficiency is approximately 0.5 for $L(2, 16)$ with 256 processors because the lack of tasks.

Table 1: CPU times of decomposition strategies

Prob.	nb_tasks	1	4	8	16	32	64	128	256
14, parall	510	810.9	211.5	118.3	64	38.5	31	24	22
14, search	156	547	148	79	45	27	21	20.9	19
15, parall	604	6990	1860	986	491	262	150	99	72
15, search	182	4773	1312	630	330	182	96.8	73	58.5
16, parall	706	64292	16959	8681	4487	2367	1278	743	482
16, search	210	44484	11578	5862	3000	1617	877	568	356

5.3 The parallel granularity: comparing the Depthlevel parameter

The table 2 presents the study of the parallel granularity obtained by tuning the *Depthlevel* parameter on the $L(2, 16)$ instance with the Dynamic tasks allocation and the *search* decomposition strategy. This clearly shows the way to obtain very good scalability for hundreds of processors with 90% of efficiency with 256 processors.

nb_tasks	1	4	E4	8	E8	16	E16	32	E32	64	E64	128	E128	256	E256
210	44484	11578	0.96	5682	0.98	3000	0.92	1617	0.86	877	0.79	568	0.61	356	0.50
2393	44484	11537	0.96	5704	0.97	2841	0.97	1450	0.96	800	0.87	402	0.86	223	0.78
32078	44484	11520	0.96	5864	0.95	2908	0.95	1487	0.93	753	0.92	375	0.92	196	0.89

Table 2: CPU times (in seconds) and efficiencies for the Depthlevels 2, 3 and 4 for problem 16

We can sum up these results in three points:

1. When the number of processors is less than (or equals to) 8, a small set of tasks (*Depthlevel* = 2) gives the better results.
2. When the number of processors lies between 8 and 32, a medium set of tasks (*Depthlevel* = 3) is needed.
3. When the number of processors is gretter than 32, *Depthlevel* = 4 outperforms the other choices.

5.4 The memory granularity: the chunk size parameter

Figure 6 gives an idea of the tasks irregularity for $L(2, 16)$ with *Depthlevel* = 2 (a) 1 and *Depthlevel* = 3 (b). The y axis is expressed in number of checks which is correlated to the CPU time, and the x axis corresponds to the tasks. The task size approximately scaling from 1 to 15. The compiler chooses the good chunk size that corresponds to a cache-line. The size of the requested memory pieces can be reduced by choosing the chunk size. This parameter would give another way to improve the load balancing but this study is still in progress. We found difficulty to adjust this parameter to get improvement for this application.

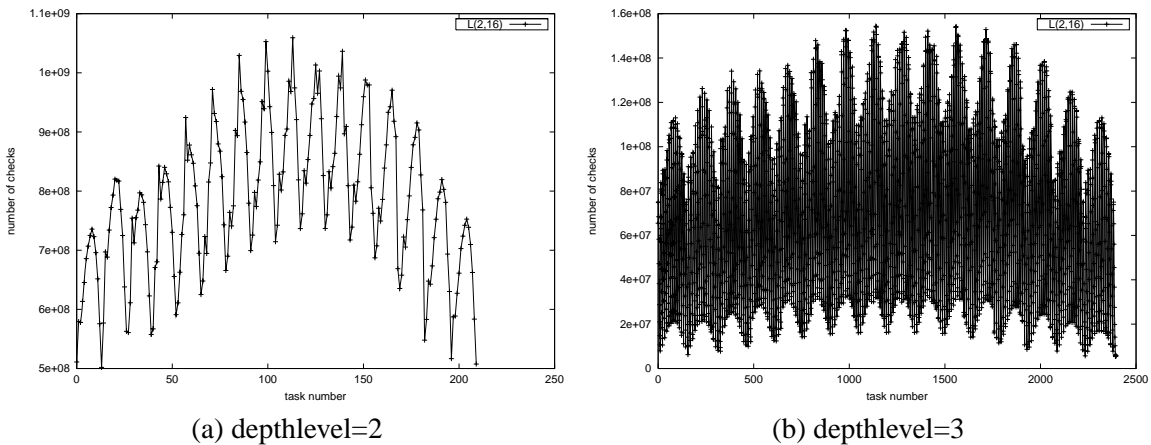


Figure 6: size of tasks for problem $L(2, 16)$

6 Conclusion

The paper presents a simple and general framework to obtain a parallel combinatorial search using a shared memory.

It presents two main contributions. First, the definition of some parameters to deal with the granularity of the parallel implementation and some strategies to fit the problem to be solved. And second, different load balancing policies are studied to prove the feasibility and benefit of a high level shared memory tool such as OpenMP for parallel resolution of combinatorial problems.

It gives practical results of a tree search distribution approach applied to a hard combinatorial problem. The linear speed-up we obtained until 256 processors on an SGI Origin'3800 encourage us to put the largest solved Langford's problem $L(2, 20)$ as a *Challenge* for general parallel solvers. One way to achieve this goal may be to enhance the sequential search algorithm with some *intelligent* backtrack (with some learning technique) and Maintaining Arc Consistency all along the search (MAC).

From a practical point of view, this study demonstrates an easy and efficient use of hundreds processors parallel machines and gives new insight how combining message passing and shared memory to go further.

Acknowledgments

This work was partly supported by the "Centre Lorrain de Calcul Hautes Performances": Centre Charles Hermite (CCH) and the "Centre Informatique National de l'Enseignement Supérieur" (CINES).

References

- [1] CHMEISS, A., AND JÉGOU, P. Décomposition: Vers une érosion du pic de difficulté? In *JNPC'98* (Nantes, France, 1998), pp. 21–29.
- [2] DECHTER, R., AND PEARL, J. Tree clustering for constraint networks. *Artificial Intelligence* 38 (1989), 353–366.
- [3] HABBAS, Z., HERRMANN, F., MÉREL, P.-P., AND SINGER, D. Load balancing strategies for parallel forward search algorithm with conflict based backjumping. In *Proceedings of the International Conference on Parallel and Distributed Systems* (Séoul, 1997).
- [4] HABBAS, Z., KRAJECKI, M., AND SINGER, D. Parallel resolution of csp with openmp. In *Proceedings of the second European Workshop on OpenMP* (Edinburgh, Scotland, 2000), pp. 1–8.
- [5] HABBAS, Z., KRAJECKI, M., AND SINGER, D. The langford's problem: A challenge for parallel resolution of csp. In *Fourth International Conference on Parallel Processing and Applied Mathematics (PPAM'2001)*, vol. 2328 of *Lecture Notes in Computer Science*. Springer-Verlag, Naleczow, Poland, Sept. 2001, pp. 789–797.
- [6] HABBAS, Z., KRAJECKI, M., AND SINGER, D. Shared memory implementation of constraint satisfaction problem resolution. *HLPP2001: International workshop on High-level parallel programming and applications in Parallel Processing Letters* 11, 4 (dec 2001), 487–501.
- [7] J.E. MILLER. *Langford's Problem*. Online, 1999. <http://www.lclark.edu/miller/langford.html>.

- [8] LARROSA, J. Boosting Search with Variable Elimination. In *Proceedings of the 6th. International Conference on Principles and Practice of Constraint Programming, CP'2000, LNCS 1894* (Singapore, 2000), pp. 291–305.
- [9] MÉREL, P.-P. *Les problèmes de satisfaction de contraintes : recherche n -aire et parallélisme – Application au placement en CAO*. PhD thesis, Université de Metz, 1998.
- [10] MONTANARI, U. Networks of constraints: Fundamental properties and applications to pictures processing. *Information Sciences* 7 (1974), 95–132.
- [11] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP C and C++ Application Program Interface*, Oct. 1997. <http://www.openmp.org>.
- [12] SMITH, B. Modelling a Permutation Problem. In *Proceedings of ECAI'2000, Workshop on Modelling and Solving Problems with Constraints, RR 2000.18* available at <http://www.dcs.st-and.ac.uk/apes/2000.html> (Berlin, 2000).
- [13] TERRIOUX, C. Recherche coopérative et nogood recording. In *Proceedings of JFPLC'10, to appear* (Paris, France, 2001).
- [14] WALSH, T. Permutation problems and channelling constraints. Tech. Rep. APES-26-2001, APES Research Group, January 2001. available at <http://www.dcs.st-and.ac.uk/apes/apesreports.html>.