PARALLEL TREE SEARCH FOR COMBINATORIAL PROBLEMS: A COMPARATIVE STUDY BETWEEN OPENMP AND MPI

MICHAËL KRAJECKI, CHRISTOPHE JAILLET, ALAIN BUI

Abstract. In this paper, a general approach for solving combinatorial problems in parallel is proposed. This study is done using the Constraint Satisfaction Problems (*CSPs*) formalism. The tasks are generated *a priori* by considering the subtrees at a particular depthlevel and represent a partition of the search space. They may be independent or the algorithms may take advantage of a collaboration mechanism between the processors. The parallel algorithm introduces few modifications to the sequential one. The tasks arrangement between the processors is studied with different load balancing strategies, comparing shared memory and a message passing scheme.

Then the Langford problem and optimal Golomb ruler construction are studied and parallelized. The former is a combinatorial problem and the latter a combinatorial optimization one. The applications associated with these problems are written in C, using the standard OpenMP library or the MPI message passing interface. The parallelization of these two applications proved efficient on up to 128 processors and opens up some new perspectives for these particular problems, such as solving the already solved instances of the problems more quickly and solving further open instances in the future.

Keywords: combinatorial problems, optimization, CSPs, tree search, Langford, Golomb ruler, parallel algorithm, load balancing, shared memory, OpenMP, message passing, MPI.

Introduction

This paper presents combinatorial problems as CSPs (Constraint Satisfaction Problems) and proposes to solve them with a tree search approach. The algorithms are parallelized using shared memory and message passing, and we particularly focus on the load balancing of the generated tasks. Two particular problems are presented which can be formalized as CSPs: the Langford problem is a combinatorial problem; the construction of optimal Golomb rulers consists in a combinatorial optimization one.

The first part of this paper presents CSPs and a general framework for their parallel resolution. In a previous work, we have studied parallel resolution of CSPs with a shared memory [HKS00]. We focused on a simple decomposition strategy of the Search Tree which enables the choice of initial variables to generate independent tasks. The scalability of this approach is studied within the shared memory model and compared to a message passing approach. Because of the tasks irregularity the load balancing question is crucial for parallel combinatorial search. Different static and dynamic policies are examined and their shared memory and message passing implementations are studied.

The second part is dedicated to two particular problems. The Langford problem is a combinatorial problem and can be presented as a 4-ary to binary CSP. Two different approaches are studied: Miller's one provides a simple tree search enumerative algorithm; the algebraic method introduced by M. Godfrey counts the solutions without constructing them and consists in evaluating a polynomial sum.
On the other hand, the optimal Golomb rulers construction is a combinatorial optimization problem which can be described as a 4-ary CSP that consists in minimizing a cost function over all the possible Golomb rulers.

The next section proposes experimental results for these two problems: specific algorithms are proposed and their parallelization is discussed, shared memory being implemented with OpenMP and the message passing scheme with MPI. We particularly focus on the impact of the granularity, the load balancing strategies provided, and the possibility of introducing a collaboration scheme between the tasks.

The paper ends with some conclusions on this work, and some perspectives are introduced.

1. Parallel Tree Search for combinatorial problems

In this section we first present Constraint Satisfaction Problems (*CSPs*), and show that combinatorial problems can be described using this formalism. Then we

focus on the parallelization of the classical tree search algorithms and especially on the ways to balance the tasks between the processors.

1.1. CSP formalism

The Constraint Satisfaction Problem model is widely used to represent and solve various AI related problems such as Computer Aided Design, Theorem Proving, Scheduling or Optimization.

A CSP is defined by a set of variables and a set of constraints. A set of allowed values (the *domain*) is associated to each variable. Solving a CSP means finding an assignment for each variable that satisfies all the constraints. *Finite Domains* constraint solving is now a well established technique. Typically, these problems require extensive computation to find a solution. Most of the suggested algorithms are enhancements of the basic search algorithm *Backtrack* (*BT*). Its main drawback is the computational cost. Since CSP resolution is NP-Complete, an efficient (polynomial) general search algorithm is unlikely to exist and parallelization seems to be a good way to achieve further practical improvements.

1.1.1. Definitions

This section gives the basic definitions and useful notations of the Constraint Satisfaction Problems framework.

Definition 1 ([Mon74])**.** *A* **Constraint Satisfaction Problem** P *is given as a tuple* $P = (X, D, C, R)$*, where:*

- $X = \{X_1, \ldots, X_n\}$ *is a set of* n variables.
- $D = \{D_1, \ldots, D_n\}$ *is a set of* n domains *where each* D_i *is associated with* X_i.
- $C = \{C_1, \ldots, C_m\}$ *is a set of* m constraints *where each constraint* C_i *is defined by a set of variables* $\{X_{i_1}, \ldots, X_{i_{n_i}}\} \subseteq X$.
- $R = \{R_1, \ldots, R_m\}$ *is a set of* m relations *where each relation* R_i *defines a set of* n_i*-tuples on* $D_{i_1} \times \cdots \times D_{i_{n_i}}$ *compatible w.r.t.* C_i.

A *binary CSP* is a problem where all constraints are sets of two variables at most. In a n-*ary CSP* the constraints link at most n variables.

An *instantiation* of a set of variables A is a k-tuple $(a_1, \ldots, a_k)$, representing an assignment of $a_i \in D_i$ to x_i, for all $x_i \in A$. 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 called a *solution* of the CSP and a CSP is said *consistent* if it has at least one solution.

The two particular problems we will present further in this paper are combinatorial ones and can be defined as CSPs: the Langford problem as a 4-ary to binary CSP, and the Golomb ruler construction as 4-ary one.

1.1.2. Solving a CSP

Solving a CSP means either deciding if it admits a solution, finding a solution if any, finding or counting all the solutions. The main complete method to find a solution for a CSP is the basic *Backtrack* algorithm (in the following, we will refer to it as BT). Unfortunately, it presents an important drawback: it is exponential in the number n of variables. A lot of works have been done to improve this algorithm ([HE80, SF94]). We can roughly classify them into the following categories:

- performing constraints propagation before or during search with a number of different filtering techniques generally based on *Arc-Consistency* [Wal93] or *Path-Consistency* [Mac77] for binary CSPs, and *Generalized Arc-Consistency* [MM88] for n-ary CSPs.
- improving the search by choosing good variable or value ordering heuristics for the next variable to be instantiated and value to be assigned [BvR95, FD95].
- improving the search by a more intelligent backtracking (BJ, CBJ), some look-ahead strategies (FC, MAC) or a combination of them[Pro93], or some *nogood* recording and learning [Dec90, Ter01].
- performing subproblems decomposition [Chm96, DP89, FH95, GLS99].

1.1.3. Parallel resolution of CSPs

There are mainly two different approaches to use parallel computation for constraint solving. The first one consists in parallelizing the filtering methods ([Kas89, RAASR99]) and are specific to CSPs. The second one induces the parallelization of the resolution itself using some decomposition technique [LHB92, RK87, HHMS97, Mér98], but most of these approaches are specific to some cases of CSPs. We presented an overview on decomposition techniques for a parallel resolution of CSPs in [HKS04].

The method we chosen mainly distribute the search space between the processors by dividing the search tree at a particular depthlevel.

1.2. Solving a combinatorial problem with a tree search algorithm

In [HKS02], we proposed to formalize the combinatorial search as a CSP and showed that an efficient parallel resolution is possible. This approach is generic for combinatorial problems and combinatorial optimization ones.

The CSP formalism allows to define the space of a combinatorial search as a tree (see figure 1.1):

- a node corresponds to a value of a variable: at depthlevel i we consider the different possible values of the variable x_i (in its domain D_i);
- every leaf of the tree symbolizes a sequence which is a solution if it respects all the constraints defined in the set C.

1.2.1. Generating tasks for parallelism

The tree traversal induced by the explicit construction of all the solutions can be made in parallel while introducing the following definition for the notion of task: it is associated to the traversal of a particular subtree. While choosing to develop all subtrees to a depth k, at most n^k independent tasks can be defined and are accessible using a unique identifier.

So, the sequential algorithm can be summarized in C-like mode by:

```
nbTasks = generateTasks(n,k);
for(task=0 ; task<nbTasks ; task++)
```

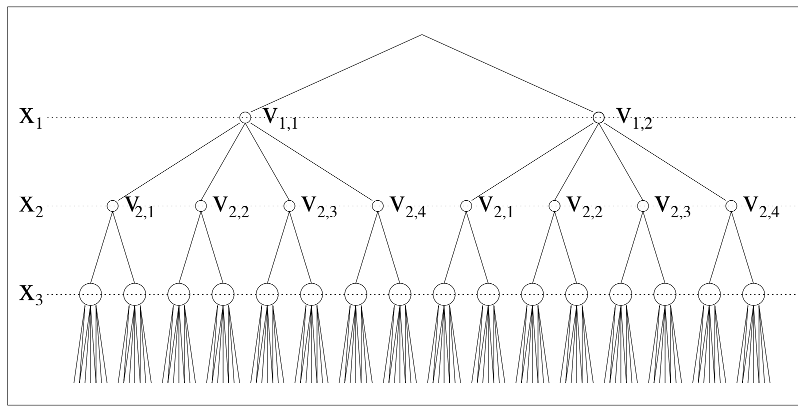

Figure 1.1: Search Tree; tasks generation.

```
solveTask(task);
```

where `nbTasks` is the number of tasks deduced by the development of all subtrees to the depth `k` by the function `generateTasks`. The function `solveTask` is in charge of traversing the subtree associated with the task numbered `task` and of accumulating its result (for example sum it if we plan to count the number of solutions of a given problem).

Note that except some initializations and the accumulation after the subproblem is solved, the tasks use the same algorithm as the global one.

1.2.2. Backtracking effects

It is noticed that, when introducing a backtracking scheme on the inconsistent branches (for which the first instantiated variables already violate one of the constraints), we observe that the computation times associated to the tasks are especially irregular.

To be efficient, any recursive tree traversal must be avoided in the parallel algorithms and the use of arrays as elementary data structure is strongly recommended.

After the tasks generation, a parallel version of the tree search consists in traversing the tasks. Thus, the main interest of the parallelization is the balancing of the tasks between the processors. We will bring this point into focus after we

consider the particularities of the combinatorial optimization problems compared to combinatorial problems.

1.2.3. Particular case of the combinatorial optimization problems

All the particularities presented here are illustrated by the optimal Golomb rulers construction, introduced in section 3.

Combinatorial optimization problems *vs* combinatorial problems

In a combinatorial problem the tasks are independent and can be solved in any order (the application is called *FIIT*: Finite Independent Irregular Tasks application [Kra99]).

When considering a combinatorial optimization problem, we define a new constraint based on the fact that our goal is to optimize a given quantity: if this quantity is improved, an instantiation that would have been consistent with the old value of this quantity (consistent and improving the quantity) might not be consistent with its new value (even if globally consistent with the problem, it might not improve the optimal value anymore). This is the reason why the tasks generated in the case of combinatorial optimization problems are not *really* independent one from another, even though the global result does not differ if the order of the tasks is changed[1].

Forward Checking based on incremental evaluation

We showed in [Jai05] that it is possible to introduce an incremental function, deduced from the objective function, to evaluate partial instantiations instead of global ones only. This offers the possibility of knowing the best possible cost of any eventual solution extending a given partial instantiation, and therefore to skip the instantiations with no possible improvement.

This can be used to cut branches in the search tree before reaching the leaves. Added to the initial tree search traversal based on constraints, it introduces some

[1] and even though any improvement of the quantity to optimize may be ignored

filtering strategy based on optimization, that belongs to the variety of Forward Checking techniques[2].

Collaboration

When running a program in parallel, the tasks are balanced over the processors and each processor commonly has several tasks to compute. So there are four different levels of information: *global information*, *processor information*, *solver information* and *task information*.

As evoked just before, the tasks generated in the case of a combinatorial optimization problem may not be independent. Because of the search space reduction when the optimization spreads out, these may take advantage of the results of the previous ones (by upgrading their value of the quantity to be optimized, if such a collaboration scheme is used).

The goal of the application is to compute the (global) optimal value of an objective quantity over a given set of tasks; it can be reached in four different ways:

- by keeping the optimal value private to each task: when a processor gets a new task, it restores its previous value (but keeps the best of them for the final collecting of the results);
- by sharing the best value between all the tasks of each solver (the last values being gathered before the program concludes): this enables us to cut branches in the search tree, and even to directly avoid some of the tasks;
- by sharing information at the processors level, without taking into account the sets of solvers the processors could support;
- by sharing the optimal value between all the tasks and updating it after each of them: this solution provides an efficient way to decrease the quantity used by each task; it may reduce the search space more quickly, but may impose an exchange overhead.

The use of one of these collaboration strategies can be setup in the application and the effects of this choice can be measured, as mentioned in the section dealing with the experimental results.

[2] *FC* in the field of the CSPs algorithms

Note that we intend to use parallel machines: since the working processors are equivalent, we attribute exactly one solver to each of them. So we limit our investigations to three information levels, skipping *solver* one.

1.3. Load balancing

Dividing the initial problem into subproblems induces quite no computational overhead, except the effort of balancing the load among the processors. In fact, each task corresponds to a subtree and the set of tasks defines a partition of the original search tree. So the accent has to be set on the load balancing strategies. Different options are possible, which may be more or less interesting for each class of problem: whereas static repartitions may prove efficient for regular problems (with regular tasks), the dynamic schemes may be more interesting for irregular ones, among which the combinatorial problems.

Some of the parallel programming languages implement some built-in load balancing strategies: using this possibility puts the systems in charge to balance the tasks over the processors with no programming effort, but this does not allow to develop any fine strategy that may be more adapted to a given class of problem. So the user will certainly have to develop some explicit load balancing strategies by himself.

1.3.1. Static repartitions

The static load balancing strategies are such that the tasks allocation to the processors is computed once at the compiling time. Different possibilities are offered:

1. *static* repartition: each processor is in charge of $\frac{Nbtasks}{Nbproc}$ consecutive tasks. This solution seems to be inadequate in our case because taking consecutive tasks is equivalent to computing at once a larger one (which is divided into the subproblems of the set of minimal tasks).
2. *modulo Nbproc* static repartition: each processor receives $\frac{Nbtasks}{Nbproc}$ different subproblems following a repartition modulo the number of available processors, $Nbproc$. In this solution, processor P_i begins with task T_i, continues

with $T_{i+Nbproc}$, and so on. Its main advantage is to distribute the so-called search tree irregularity among the processors.

The static repartitions cannot be efficient if the tasks are too irregular because, for example, one of the processors may be in charge of a set of tasks consuming on average twice as much as those given to another processor.

1.3.2. Dynamic schedules

For the dynamic schedules the tasks are dynamically allocated to processors at the execution time; there is no guarantee on which thread the tasks are executed.

Two major load balancing schemes have to be presented, which differ by the work distribution initiator.

- the **client-server** strategies are centralized ones:
  A processor is considered as a client if it is working, and as a server if it proposes to distribute tasks. A server (or a set of servers) is in charge of all the tasks and only has to distribute them over the clients. As a client becomes idle, it asks the server for a new task or a new set of tasks. The server may distribute the tasks in a given order and "one by one" or several at the same time, and different distribution strategies can be used.
- the **server initiated** strategies are distributed ones:
  The processors are considered as servers when they are idle or as source ones when they are busy. All of them begin as source processors, and are in charge of a given set of tasks. In this strategy, the initiative of the redistribution is in charge of the idle processors: when a processor becomes a server, it proposes the source processors to take a part of their tasks queue. The different possibilities are determined by the various ways the tasks are aither initially divided or redistributed during the execution: there are different matching policies for a server to choose the source processor whose tasks to take, and even to choose how many tasks (and which ones) to take from this processor.

For the experimental study (see section 4), we propose to deal with a simple client-server strategy, and with a server initiated one, for which a server processor

chooses as a peer the first next that still has tasks to compute and takes half of its remaining tasks.

Dynamic jumping for skipping useless tasks

Although these can be adapted to the particular problems to be treated, the previous load balancing strategies are generic. In some cases some particular strategies can be developed that take advantage of the problem's specificities. With the combinatorial optimization problems it is possible to develop a load balancing strategy that benefits from the fact that some or most of the tasks may become useless when they have to be considered (see 3.4 and table 4 in 4.4.3).

This can be used to adapt a server initiated strategy: an idle processor (server) will not take a given set of tasks from the source processor it has chosen, but will take as many as necessary (in a *while* manner) in order to take at least a useful one. But this may increase the parallel overhead, blocking the tasks lists for a longer period.

On the other hand a naive client server strategy can be adapted in the same way. A task is determined by the values given to the first variables and it can reveal useless because of the assignment of any of them. If for example it becomes useless with the value of the third variable[3], then all the tasks beginning with these same 3 values are useless: the first next task to be eventually useful is the next one that does not begin with theses values. This gives a way to know which is the task following a given one if it fails at a given depthlevel: it determines a precedence graph between the tasks (depending on the failure depthlevel), which can be used to skip the useless tasks not one by one but with a dynamic jumping manner.

1.3.3. Shared memory implementation

Whereas the message passing model of programming only offers communication to distribute any information, the shared memory model allows to put the information to be shared in a memory accessible by all the processors. It is a user-friendly

[3] with depthlevel 3 at least

way of programming, that nevertheless may induce some computational overhead, depending on the way the shared memory is implemented on a given architecture.

We briefly present in this section the way to use this model for the implementation of the two main load balancing strategies we proposed previously.

- **Client-server** strategy:
  With this centralized load balancing strategy, a server holds the list of all the tasks and it distributes them through the clients on demand. Using shared memory, the list of tasks can be shared and no centralized server thread is necessary: the clients take their new tasks from the list, respecting mutual exclusion on the `next_task` index.
- **Server initiated** strategy:
  The processors have their own list of tasks. As an idle processor initiates a redistribution, it looks for a non empty list and for instance takes the half of them. In order to enable the exchanges, the lists should be placed in shared memory. Locks are necessary to manage the lists while transferring or consuming tasks, preventing deadlocks especially when the number of remaining tasks decreases.

1.4. Langford and Golomb ruler problems

The Langford problem is a combinatorial one that can be presented as a 4-ary or binary CSP. The goal of the computation is to count all the solutions. We particularly will study different load balancing strategies on the tree search method, implemented in shared memory. The Godfrey algebraic approach will be introduced as a second method and parallelized using a message passing model.

For the optimal Golomb ruler construction problem all the sequences are not equivalent: we have to minimize the length of the constructed solutions as a cost function. Different collaboration levels between the tasks and load balancing strategies will be proposed for this combinatorial optimization problem.

A large part of the study presented in sections 2 and 3 can be found with more details in [JK04b] and [JK04a] respectively.

2. The Langford problem

C. Dudley Langford gave his name to a classic permutation problem [Gar56, Sim83]. While observing his son manipulating blocks of different colours, he noticed that it was possible to arrange three pairs of blocks of different colours (yellow, red, blue) in such a way that only one block separates the red pair, two blocks separate the blue pair and finally three blocks separate the yellow one (see figure 2.2).

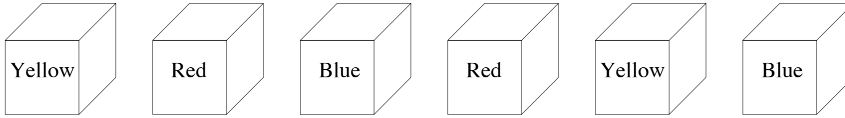

Figure 2.2: L(2,3): arrangement for 6 blocks of 3 colours: yellow, red and blue.

The problem has been generalized to any number of colours n and any number of blocks having the same colour s. $L(s, n)$ consists in searching for the number of solutions to the Langford problem. In November 1967, Martin Gardner presented $L(2, 4)$ (two cubes and four colours) as being part of a collection of small mathematical games and stated that $L(2, n)$ has solutions for all n such that $n = 4k$ or $n = 4k - 1$ for $k \in \mathbb{N} \setminus \{0\}$.

The Langford Problem has been approached in different ways (discrete mathematics results, specific algorithms, specific encoding, ...): see [Mil99].

At the moment the instances solved in practice in a merely combinatorial manner limit themselves to a small number of colours. In this case, one mentions the instance $L(2, 19)$ that was solved in 2 years and a half on a DEC Alpha to 300MHz in 1999. In 2002, $L(2, 20)$ was solved with the help of a new algorithm and the intensive use of a cluster of 3 PCs during one week.

2.1. the Langford Problem as a CSP

Recently, Toby Walsh and Barbara Smith formulated the $L(2, n)$ problem as a 4-ary Constraint Satisfaction Problem [Wal01, Smi00]. In [HKS01] we proposed it as a binary CSP with a compact representation.

In this section, we propose to solve the Langford problem with the general tree search approach introduced in the previous section of this paper.

2.2. Miller's algorithm: a tree search approach

The Langford problem can be modelized as a tree search problem. In order to solve $L(2,n)$, we consider the tree of height n and width $2n-2$ (see figure 2.3):

- every node of the tree corresponds to the place in the sequence of the cubes of a determined colour;
- to the depth p, the first node corresponds to the place of the first cube of colour p in first position and the ith node corresponds to the positioning of the first cube of colour p in position i, where $i \in [1, 2n-1-p]$[4];
- every leaf of the tree symbolizes the positions of all the cubes;
- a leaf is a solution if it respects the colour constraint defined by the Langford problem: all the cubes[5] must be in different places.

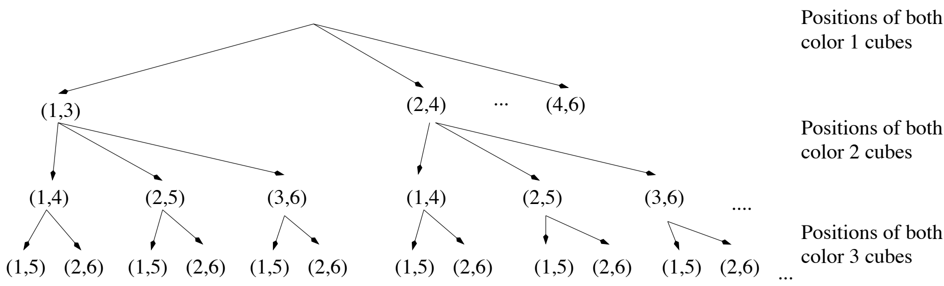

Figure 2.3: Search tree for $L(2,3)$.

It is now sufficient to propose a walk through the search tree, in depth first, to get a simple sequential algorithm solving the Langford problem[6].

It may be taken into consideration that this algorithm explicitly constructs all the solutions to count them, which is not the case in Godfrey's algorithm.

[4] if the first cube of p-th colour is in position i, then the second one is in position $i+p+1$

[5] first and second of each colour

[6] A sequential array-based non-recursive algorithm (written in C) is accessible on page `http://www.lclark.edu/~miller/langford/langford-algorithm.html`

2.3. Godfrey's algorithm: algebraic method

In 2002, an algebraic representation of the Langford problem has been proposed by M. Godfrey.

Consider $L(2,3)$ and $X = (X_1, X_2, X_3, X_4, X_5, X_6)$. It proposes to modelize $L(2,3)$ by $F(X,3) = (X_1X_3 + X_2X_4 + X_3X_5 + X_4X_6) \times (X_1X_4 + X_2X_5 + X_3X_6) \times (X_1X_5 + X_2X_6)$. In this approach, each term represents a position for both cubes of a given colour and a solution to the problem is equal to the polynomial coefficient of $X_1X_2X_3X_4X_5X_6$ in the development. More generally, a solution to $L(2,n)$ can be deduced from $X_1X_2X_3X_4X_5 \dots X_{2n}$.

If $G(X,n) = X_1 \dots X_{2n}F(X,n)$ then it has been shown that:

$$\sum_{(x_1,\dots,x_{2n})\in\{-1,1\}^{2n}} G(X,n)_{(x_1,\dots x_{2n})} = 2^{2n+1}L(2,n)$$

So:

$$\sum_{(x_1,\dots,x_{2n})\in\{-1,1\}^{2n}} (\prod_{i=1}^{2n} x_i) \prod_{i=1}^{n} \sum_{k=1}^{2n-i-1} x_k x_{k+i+1} = 2^{2n+1}L(2,n)$$

The computation of $L(2,n)$ is in $O(4^n \times n^2)$ and an efficient long integer arithmetic is needed. This principle can be optimized by taking into account the symmetry of the problem and using the Gray code[RS90].

By using this approach, M. Godfrey has solved $L(2,20)$ in one week on three PCs in 2002.

2.3.1. Godfrey's algorithm in parallel

Section 2.3 introduced the evaluation of $L(2,n)$ by

$$\sum_{(x_1,\dots,x_{2n})\in\{-1,1\}^{2n}} (\prod_{i=1}^{2n} x_i) \prod_{i=1}^{n} \sum_{k=1}^{2n-i-1} x_k x_{k+i+1}$$

It is quite obvious that a parallel version can be derived from this formula, tasks being generated by choosing a value in $\{-1,1\}$ for one or more of the x_i in $\sum_{(x_1,\dots,x_{2n})\in\{-1,1\}^{2n}}$. At depthlevel k, as the values of x_1, $x_2, \dots, x_k$ are fixed (either to 1 or -1), a set of 2^k independent tasks is generated.

Optimization using the Gray code inside the tasks

$\sum_{k=1}^{2n-i-1} x_k x_{k+i+1}$ has to be computed for each value of the $2n$-uple $(x_1, \ldots, x_{2n})$ in $\{-1, 1\}^{2n}$. But the computation time for this sum might be very important. So it is interesting to do that in a quick way, by changing the value of the x_i only one by one (which allows to get one sum from the previous one). The ordering of these changes is made using the Gray code sequence, and it is interesting to pre-calculate it.

The sequence cannot be stored in an array because it would be too large (it would contain 2^{2n} byte values). This is the reason why only one part of the Gray code sequence is stored in memory and the values are calculated from this array.

The size of the stored part of the Gray code sequence is chosen as large as possible to be contained in the processor's cache memory: so the accesses are fastened and the computation of the Gray code is optimized.

For an efficient use of the SGI R14000 processors, which dispose of 8 MB of level-2 cache memory, the Gray code sequence is developed recursively up to depth 22, so that it uses 4 MB (the rest of the memory being used for the computation itself).

3. Constructing optimal Golomb rulers in parallel

Whereas the Langford problem consists in a combinatorial search, the optimal Golomb ruler construction is a combinatorial optimization problem, with the inherent specificities (see 1.2.3).

A Golomb ruler is an ordered sequence of non negative integers, such that all the differences between any two of them are different: considering that these *marks* refer to positions on a linear scale, the *distances* on the *ruler* have all to be different [Gar72] (see figure 3.4).

It is easy to construct a ruler for any number n of marks, but the interesting cases deal with minimal length sequences [Ran93]. For example, (0,1,4,6) is optimal for 4 marks.

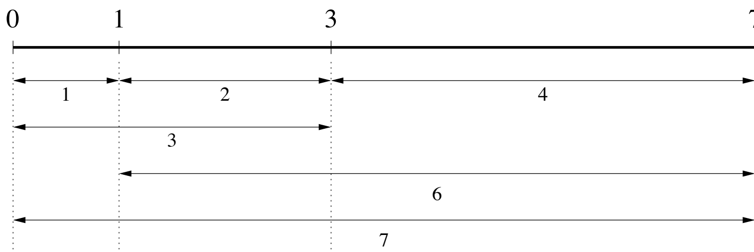

Figure 3.4: A *non optimal* 4 marks Golomb ruler.

These sequences are a mathematical curiosity, but have also many applications in a wide variety of fields [BG77], including x-ray crystallography, self-orthogonal codes (for error detection and correction in coding theory) and communication (radio frequencies, radio astronomy and PPM[7]).

This construction problem is a combinatorial optimization one, and has been approached in different ways [SSW99, SHL95], including CSPs.

For the rest of this document, let $G(n)$ represent the minimal length of an n marks Golomb ruler. Our goal is to compute that value (and give the constructed ruler). Once a ruler of a given length is found, the whole effort consists in finding a solution with a better length. This is the reason why we formulate the problem as "finding the optimal length under a given limit for the Golomb ruler".

3.1. Tree search algorithm

The construction of optimal Golomb rulers (*OGR*) can be modelized as a tree search problem, with the view to minimize the length of the constructed sequence ([DRM98]):

- as the length of an optimal n marks ruler does not exceed $2^{n-1} - 1$ ($1 + 2 + 4 + \cdots + 2^{n-2} = 2^{n-1} - 1$), it is possible to consider that the root value is 0 and that the values of the nodes are between 1 and $2^{n-1} - 2$ (see figure 3.5);
- as it may be far from the final best one (especially when the number of marks to be placed increases), the previous bound is practically not so good

[7] pulse phase modulation

because it imposes to search in a too wide tree: the user can specify a value closer to the foreseen optimal one;

- every leaf of the tree symbolizes a sequence which is a solution if the values are ordered and if all the differences are different.

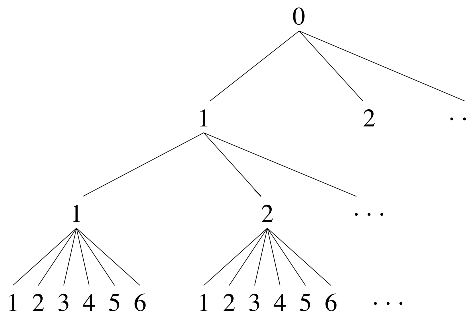

Figure 3.5: Search tree for 4 marks Golomb rulers.

3.2. Improvements

The search space can be reduced by applying some simple construction remarks (see figure 3.6):

- the best possible length $best_length$ is initially set to $initial_limit$, which is fixed as a parameter with default value $2^{n-1} - 1$;
- a preprocessing step can (if asked) construct a first Golomb ruler under the initial limit, not necessary close to the optimal value but certainly much better than the original one;
- when placing the k-th mark at position $pos(k)$, there still are $r = n - k$ remaining marks to be placed. With the current one, these have to constitute a $(r+1)$-ruler: the corresponding length, $remaining_length(r)$, cannot be less than $1+2+\cdots+r = r(r+1)/2$ or than $G(r+1)$ *if known*: it induces the following additional constraint:

$$pos(k) + max(\frac{r(r+1)}{2}, G(r+1)) < best_length$$

- mirror solutions can be avoided and thus the search space reduced, by considering only sequences whose last distance is greater than the first one: if

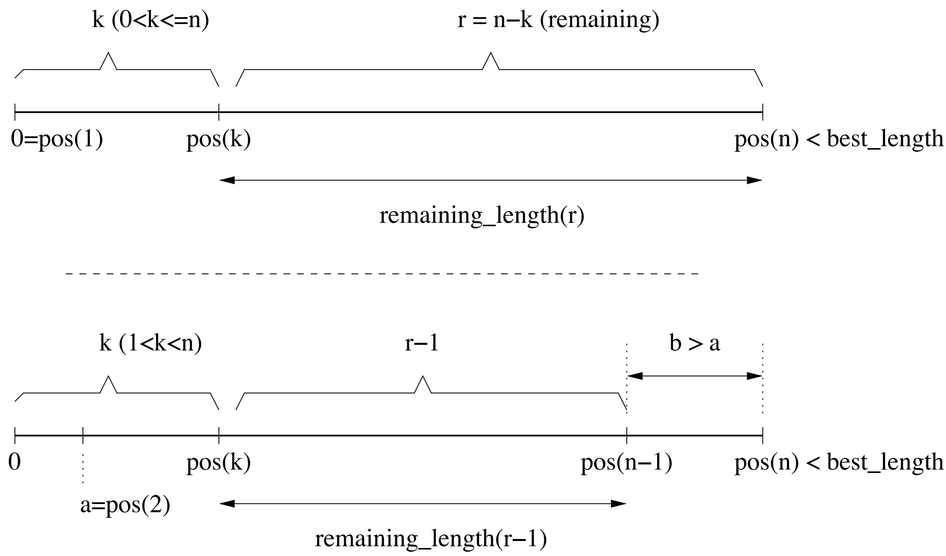

Figure 3.6: New constraints based on construction remarks.

the second mark (first except 0) has already been set with the a value, and in order for the last distance b to satisfy $b \geq a+1$, another constraint has to be introduced:

$$pos(k) + max(\frac{(r-1)r}{2}, G(r)) + a + 1 < best_length$$

- as the best observed length decreases, more and more inconsistent values can be ignored.

According to the above global constraint, an efficient filtering algorithm is defined which eliminates remaining values in the the position variables domains in order to cut branches of the search tree. Note that it becomes a *Forward Checking* algorithm (*FC* in the field of CSPs).

3.3. Tasks generation

At most $(initial_limit - 1)^k$ tasks can be defined while choosing to develop all subtrees to a depth k. Assume that this construction only keeps the *possible* tasks

The sequential algorithm can be summarized in C-like mode by:

```
nbTasks = generateTasks(n,initial_limit,k);
```

```
best_length = initial_limit;
for(task=0 ; task<nbTasks ; task++)
 if ( useful(task,best_length) )
  best_length = solveTask(task,best_length);
```

The reader may also remember that this problem is a combinatorial optimization one, which has two main consequences (see the experimental results in next section):

- if such a collaboration scheme is used, the tasks may take advantage of the results of the previous ones because of the search space reduction when the value of $best_length$ is improved;
- although only the initially possible tasks were generated, some of them may become useless when the $best_length$ value decreases: an adapted *dynamic tasks jumping* load balancing may improve the efficiency of the application.

3.4. Dynamic tasks jumping

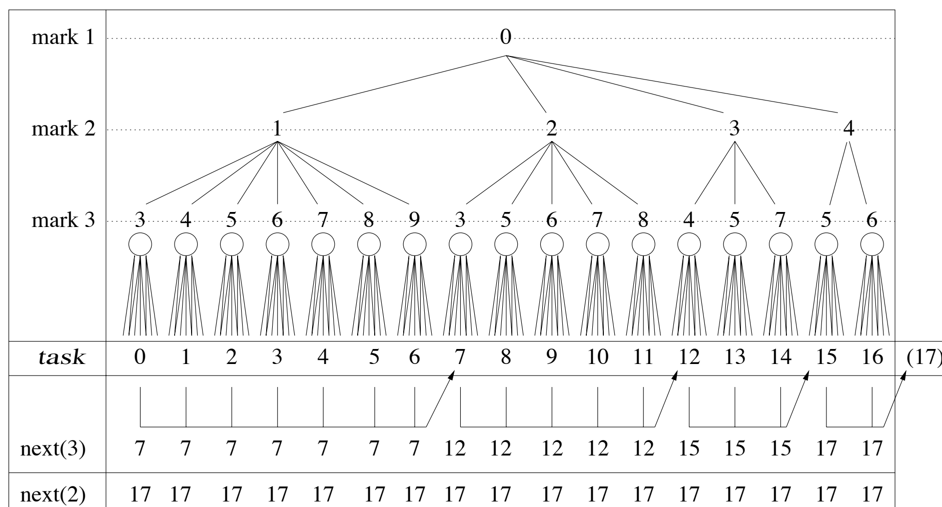

Figure 3.7: Dynamic jumping for skipping the useless tasks.

As introduced in 1.3.2, it is possible to develop a particular load balancing scheme that uses the precedence graph of the tasks which can be computed as a preprocessing statement: if a given task is observed useless, it is possible to

determine at which depthlevel it is inconsistent and then what are the first next tasks that can be useful (see figure 3.7).

In order to make it more efficient, another preprocessing treatment can be added that computes, for each task and depthlevel, the first value of the length that makes the task fail.

We used this strategy with a client server strategy for the optimal Golomb ruler construction and had good efficiencies (see the experimental results in 4.4.4).

4. Experimental results

4.1. Context

We had experiments with various parallel machines based on different architectures. In order to keep the results comparable one from another, we only present those obtained on an SGI Origin'3800[8], but all the phenomenons demonstrated in this paper were observed likewise on the other machines.

We chose OpenMP to implement shared memory, and MPI for programming with message passing.

4.1.1. Shared memory with OpenMP

The OpenMP environment has evolved to a standard for shared memory parallelism [Ope05, CDK$^+$00] [9]. It is a complete API for programming shared memory multiprocessor systems and enables to obtain a parallel code easily since it is kept quite close to the sequential one. In addition, it makes it possible to test different work distribution policies. OpenMP derives from the ANSI X3115 efforts and is a set of compiler directives and runtime library routines that extend a sequential programming language (C or FORTRAN) to express with a shared memory. It conforms to the SPMD programming language style. One key advantage of OpenMP, compared to MPI, is that the development cost of a message passing version would be much more important. It would potentially induce numerous

[8] 768 R14000 processors at 500 MHz, 500 MB of memory per processor

[9] see also pThreads or DSM (*Distributed Shared Memory*)

additional programming efforts to deal with the crucial load balancing problem and especially for *Irregular Applications*, which is the case here.

The main feature of our proposal is Coarse-Grained parallelization which uses only one level of parallel *for loop* or only one *parallel section*, consisting in the resolution of subproblems after distributing them over the processors.

Tasks allocation in a parallel loop

Within the OpenMP environment the tasks allocation to the processors (*threads*) can be done very easily by one compilation directive added to the *for* loop. This allows to balance the tasks with static schedules (*static* or *static modulo Nbproc* repartitions) or dynamically.

For example, to have a *dynamic for* schedule, an OpenMP directive is added before the loop:

```
nbTasks = generateTasks(n,k);
#pragma omp parallel for schedule(dynamic)
for(task=0 ; task<nbTasks ; task++)
 solveTask(task);
```

Tasks allocation in a parallel region

OpenMP also provides another way to produce parallel applications. A parallel region is executed by all the processors. The user is explicitly in charge of distributing the load among the processors.

- For example if we want to redefine the static repartition with an OpenMP parallel region, the following statements can be used:

```
nbTasks = generateTasks(n,k);
#pragma omp parallel
{
 int nbp, p, start, end, task;
 nbp = omp_get_num_threads();
 p = omp_get_thread_num();
 start = p * nbTasks / nbp;
 end = (p+1) * nbTasks / nbp;
```

```
 for(task=start ; task<end ; task++)
  solveTask(task);
}
```

The variables `start` and `end` are introduced in order to explicitly balance the load to the processors. Because these are defined in the parallel region, each processor has its own copy of these variables.

- The dynamic *client server* repartitions are implemented using a shared variable `nextTask` which indicates the index-number of the next task to be computed. When a processor needs a new task, it accesses this variable in a critical way (only one processor at the same time) by using *critical sections* or *locks*, and eventually skips the useless tasks (dynamic jumping schedule for combinatorial optimization problems: see 1.3.2).
- *Server initiated* strategies can also be developed with OpenMP parallel regions. The set of tasks is globally shared and a queue is characterized by two variables $begin$ and end. Each processor has its own tasks queue and they have to share the lists informations; so we use the shared arrays $Begin$ and End: $Begin[i]$ and $End[i]$ correspond to the processor i and are changed in critical sections determined by locks.

Computational variability using OpenMP

A remark has to be made on the computation time irregularity observed for the OpenMP versions of our programs: the computation times had a 0% to 320% irregularity[10] with large instances of our problems, especially with an increasing number of processors (see 4.2.3).

Note that such an irregularity was observed with OpenMP on all the different parallel machines we used (an SGI Origin'3800, an IBM SP Power 3 NH2 and a Sunfire 6800, which are SMP[11] or CC-NUMA[12] machines), but never when using POSIX threads or MPI.

[10]relative standard deviation

[11]SMP: *Symmetric MultiProcessor*

[12]CC-NUMA: *Cache-Coherent Non Uniform Memory Access*

We proved in [Jai05] that this anomaly is due to the OpemMP compiler allocation strategy which induces cache misses and hence this variability. The effects are increased by the fact that our algorithm uses a lot of memory and is based on a very large number of memory accesses, in private memory as well as in shared one. This phenomenon can be observed on any application with critical and intensive memory use, which is the case for any combinatorial search or combinatorial optimization problem.

Because of the irregularity observed, the computation had to be repeated 20 to 50 times for each experiment and we only retained the best value (which corresponds to the ”good” memory allocation case).

We developed a solution to correct this phenomenon which evicts any computation time variability with no computation or memory overhead [Jai05].

4.1.2. Parallel version using a Message Passing Interface

Message passing is a programming paradigm used widely on parallel computers, especially with distributed memory[13]. The Message Passing Interface (MPI) is a standard approach for message passing programming [Pac96]. This standard defines the functioning and user interface for a large number of message-passing capabilities and with a large degree of portability.

As it represents the simplest solution for the user, the tasks allocation is done by a client/server scheme. This allows different strategies based on different ways to choose the next task to be distributed to a given processor. We decided to limit our first developments to a simple client-server strategy (the tasks being distributed one by one on demand in their natural order) and to the dynamic tasks jumping provided for combinatorial optimization problems (which is limited to the OGR construction in this experimental study).

[13] see also PVM (*Parallel Virtual Machine*) for example

4.2. Miller's algorithm for the Langford problem in parallel using shared memory

The experiments we led are based on OpenMP and enabled us to measure the efficiencies provided by the OpenMP for loop schedules, and to compare them to the corresponding hand-made parallel region schedules.

This study was limited to 64 processors.

4.2.1. OpenMP parallel for loop schedules

First, a comparison between the three repartition policies for the Search Tree decomposition strategy is provided.

As we are counting the number of solutions of the Langford problem, the algorithm can be summarized in C-like mode by:

```
nbTasks = generateTasks(n,k);
nbSolutions = 0;
#pragma omp parallel for schedule(dynamic) \
             reduction(+:nbSolutions)
for(task=0 ; task<nbTasks ; task++)
 nbSolutions += solveTask(task);
```

At the end, the variable `nbSolutions` contains the number of solutions for $L(2, n)$. Note that the reduction clause is necessary to ensure the correctness of the computation.

Table 1 shows the execution times for $L(2, 14)$ where k (the subdivision depth) is equal to 5. As expected, the static repartition is not very efficient when the number of processors increases. It is interesting to notice that the modulo repartition is not so far from the dynamic repartition which is the best in our experiment.

Finally, the efficiency observed is very good. With 64 processors, the dynamic repartition obtains an efficiency superior to 85%.

4.2.2. OpenMP parallel region schedules

We redefined the 3 above schedules using parallel regions.

Procs	Static	Modulo	Dynamic
1	223.13	224.26	234.7
2	119.5	116.4	114.72
4	63.56	58.58	56.67
8	57.24	29.33	28.19
16	33.49	14.72	14.05
32	22.81	7.42	7.03
64	16.46	17.45	4.24

Table 1: Execution times for $L(2, 14)$ in seconds (depthlevel 5).

Figure 4.8 gives an overview of the different experiments. The static schedules had comparable results, even for those implemented with parallel regions.

The efficiencies observed are very good with a *dynamic for loop* load balancing scheme. The dynamic schedule with a parallel region, using a shared variable accessed in a critical way, is not very efficient with more than 16 processors. The reader shall take into account that the execution time for 64 processors is less than 5 seconds.

For the Langford problem, these experiments show that the best solution is to take advantage of the parallel *for* loop provided by OpenMP with the dynamic schedule clause. To reach good efficiency when the number of processors is large, the programmer should be ready to define a more accurate solution to manage the load avoiding the bottleneck introduced by this critical variable.

4.2.3. Computation time irregularity with OpenMP

As we mentioned in 4.1.1, the computation times were really irregular. Thus each computation has been repeated 50 times.

As an illustration of this irregularity, $L(2, 14)$ with the dynamic *for* loop and 2 processors is solved in 114-115 seconds 41 times and in 224-250 seconds in 9 different experiments. When the number of processors increases, this instability is

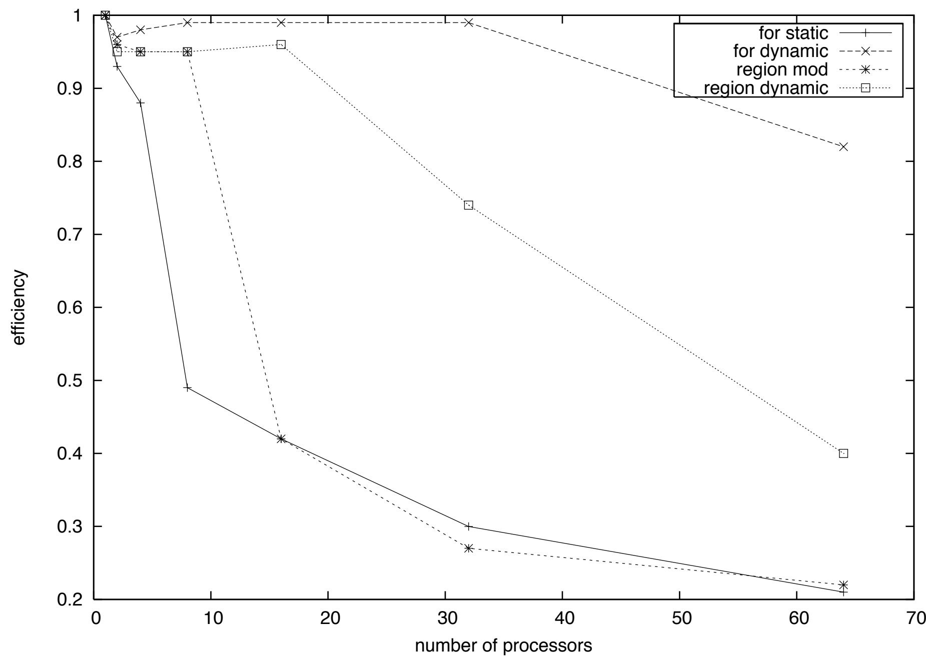

Figure 4.8: Efficiency for $L(2, 14)$.

observed more often: $L(2, 14)$ was solved on 32 processors using the modulo *for* loop in 7.42 seconds only 1 time in 5 tries on average: on 50 tries, the execution time was close to 60-65 seconds for 21 tries. See [Jai05] for more detailed results.

Because of this computational variability, we had to consider only the best computation times observed, which correspond to the ”good” memory allocation cases.

4.3. A parallel version of Godfrey's algorithm using a Message Passing Interface

Since all the tasks are available, their allocation is performed dynamically: the server sends the tasks indexes to the clients and then, when a client receives such a number, it dynamically constructs the task by deducing from the task index the values to be affected to the first position variables.

4.3.1. Solving $L(2,16)$

Table 2 sums up the results obtained for $L(2,16)$ with up to 16 processors. The depthlevel successively equals 6, 7, 8 and 9. The number of tasks is respectively 64, 128, 256 and 512.

Procs	k=6	k=7	k=8	k=9
1	972	991	972	993
4	339	334	330	333
8	153	147	140	142
12	121	132	130	108
16	78	71	73	75

Table 2: Execution times for $L(2,16)$ in seconds.

These experiments show that the parallelization of the Langford problem is also effective using Godfrey's approach. By using 16 processors, L(2,16) can be solved in less than 80 seconds (on a SGI'3800).

The reader may remember that only $p-1$ processors on p effectively solve the problem because of the server used to distribute dynamically the set of tasks. By defining a static distribution or a fully distributed dynamic distribution, it should be possible to use effectively the p processors to solve the problem.

Figure 4.9 shows the speed-ups. The server is taken into account in the evaluation. This is the reason why they are not so good with 4 processors, but when the number of processors increases, the penalty induced by the server is less important. Using 16 processors, the speed-up is near 14 and the efficiency is equal to 85% while the optimal efficiency is equal to 94% taking the server into account.

4.3.2. Some interesting results on $L(2,19)$ and $L(2,20)$

Considering the good results provided by the parallelization of Godfrey's method on $L(2,16)$, some experiments have been conducted on $L(2,19)$ and $L(2,20)$.

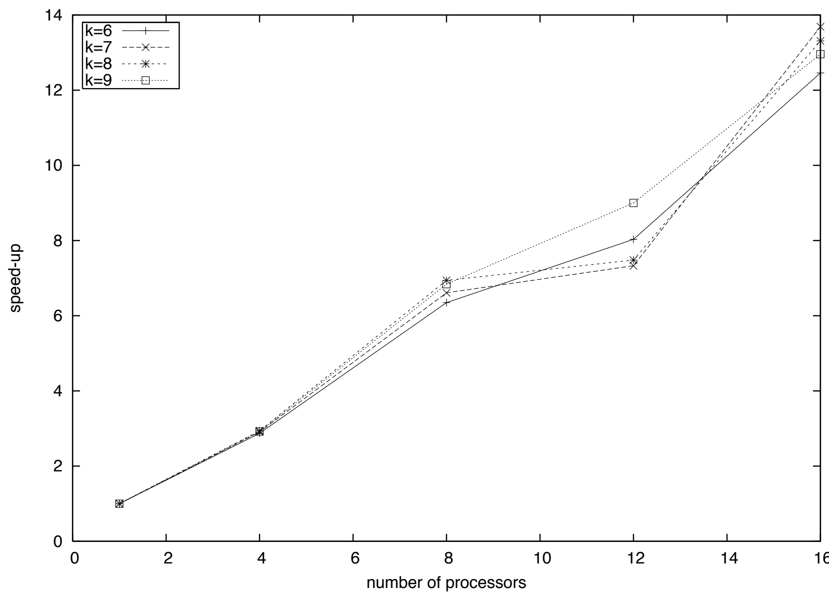

Figure 4.9: Speed-up for $L(2, 16)$.

Due to the long execution time for both problems, no result can be exhibited with less than 8 processors on the SGI'3800.

The impact of the depthlevel on $L(2, 19)$

Table 3 contains the execution times in seconds for 8, 12, 16, 32 and 64 processors. The depthlevel k moves from 5 to 10. It is noticeable that for $k = 5$, only 32 tasks are generated which is not enough to farm 64 processors.

The depthlevel is quite important when the number of processors increases. Using 64 processors, the execution time is reduced by 2 when the depthlevel moves from 6 to 10.

This fact can be explained by two factors. First, the set of tasks must be larger than the number of processors to be able to correct the load imbalance. Second, by fixing more values for the x_i (which is the case when the depthlevel increases), the memory needed to solve the task (and especially to construct the Gray code) is less important. Then a cache memory factor impacts the results in a significant way.

Procs	k=5	k=6	k=7	k=8	k=9	k=10
8	9720	9735	9307	8975	8987	8935
12	5950	5917	5941	5790	5697	5670
16	5790	4842	4392	4290	4251	4214
32	3859	2891	2433	2175	2083	2060
64	2031	1940	1473	1209	1094	1031

Table 3: Execution times for $L(2, 19)$ in seconds.

The conclusion of these experiments is that $L(2, 19)$ can be solved in less than 20 minutes, to be compared to the first results published by Miller on his web page.

$L(2, 20)$ can be solved in 1 hour and even less

To conclude the experiments, $L(2, 20)$ has been solved using 8, 16, 32, 64 and 128 processors. The depthlevel is equal to 12, so 4096 tasks are generated and distributed among the processors.

The average execution time of a task is close to 69 seconds. The minimum and maximum times are respectively 67 and 80 seconds.

The execution time on 32 processors is 9208 seconds and is reduced to 4530 with 64 processors. It is interesting to note that the execution time is reduced by half when the number of processors is doubled. Finally, $L(2, 20)$ is solved in 2274 seconds using 128 processors.

The use of parallelism and the algorithm's improvements have reduced the resolution time of $L(2, 20)$ from one week to 38 minutes! This result opens the perspective of $L(2, 23)$ and $L(2, 24)$, but with lots of processors during a very long period.

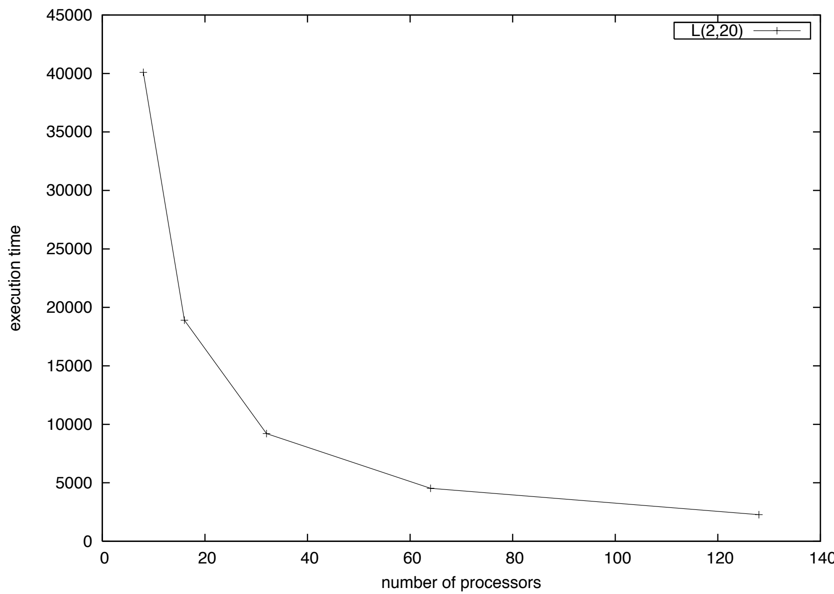

Figure 4.10: Execution times for $L(2, 20)$ in seconds.

4.3.3. $L(2, 23)$ and $L(2, 24)$

We ported our application to CONFIIT, a fully distributed peer-to-peer middleware developed by our team [KFM04].

This allowed us to solve $L(2, 23) = 3,799,455,942,515,488$ in April 2004 after 4 days of computation over about 85 processors: Sunblade 100, PCs (Intel PIV and Xeon, AMD Athlon XP) and a Sunfire'6800 with 24 processors.

We obtained $L(2, 24)$ in April 2005 after 3 months' computation with 12 to 15 processors: there are $46,845,158,056,515,936 \sim 4.7\ 10^{16}$ solutions to this problem (when evicting the symmetric ones). These results are more fully developed in [KFJ$^+$05].

4.4. Constructing optimal Golomb rulers in parallel

The Langford problem is a combinatorial one. We are going to present you the experimental results obtained for the OGR (optimal Golomb ruler) construction, which is a combinatorial optimization one. These illustrate the study presented in section 3.

4.4.1. Collaboration

Our first experiments with OpenMP concerning the collaboration immediately proved that collaboration has to be used at least between the tasks, and that hardly any exchange overhead is induced by the full collaboration: the algorithm takes a real advantage from collecting the $Global_best$ value as a shared information.

4.4.2. Granularity

Considering a given problem with a given value of $initial_limit$, the choice of the depthlevel may influence the computational times, as shown by the following example, where G(13) was computed with $initial_limit = 200$ on 32 processors (here with OpenMP with a dynamic for schedule; see figure 4.11).

- The depthlevel must not be too small because a good load balancing is impossible if the average number of tasks per processor is not important enough. For example, with depthlevel 1, only 45 tasks were generated: this created a high irregularity (one of the processors finishing in 16.3 seconds and another one in 696.2)
- If the number of tasks becomes too important, the processors may spend some more time in racing to have new tasks instead of computing the tasks themselves. For instance, we had still no answer after 40,000 seconds with depthlevel 5 (for G(13) with 32 processors and $initial_limit = 200$).

4.4.3. Useless tasks

The results we obtained show that, depending on the initial limit given for the optimization, most of the initially useful tasks may actually have became useless when they have to be treated, because of the bound improvement (see table 4).

This phenomenon is especially observed if the initial limit is far from the optimal one. It induces that, when a processor becomes idle, it has to take several tasks before getting an *interesting* one. This creates a high concurrency for the tasks allocation which may considerably reduce computation efficiency.
Thus, the naive ”one by one” schedules cannot have very good efficiencies.

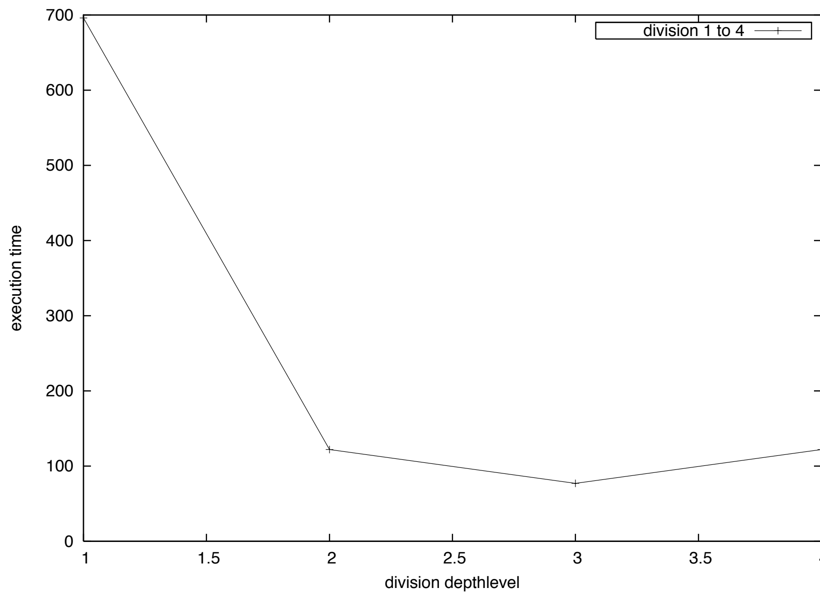

Figure 4.11: G(13), $initial_limit$=200, influence of the granularity.

initial_limit		200	122	118	90	85
TASKS	possible	39601	14641	13689	7921	7056
	generated	5808	1349	1202	399	280
	useful	286	286	286	285	280
useful / generated		4.9 %	21.2 %	23.8 %	71.4 %	100 %

Table 4: $G(12) = 85$ - Tasks: possible / generated / useful.

For the same reason, we added a preprocessing treatment that consists (if used) in generating a first consistent sequence under the limit: it immediately improves the bound, especially if the initial limit is far from the optimal one. It allows to decrease considerably the number of generated tasks and therefore to improve the ratio of useful tasks, with quite no computational effort for our problem. The effect of this preprocessing treatment can be observed in table 4: with an initial value of 200, the pretreatment gives an initial limit of 122, which changes the ratio of useful tasks from 4.9% to 21.2%.

4.4.4. Load balance

Because of the tasks irregularity, the OpenMP static and static modulo schedules can't have good results: for example for the computation of G(13) using 32 processors with a limit[14] of 147 and dividing the search tree with depthlevel 3, one finishes in about 60 seconds and another one in about 140).

Table 5 presents the results of the computation of $G(14)$ using 4 to 32 processors with a limit of 181 and dividing the search tree with depthlevel 3, comparing the OpenMP dynamic schedule on a parallel loop (*Dyn*) with the OpenMP server initiated, client server and dynamic jumping client server parallel region schedules (*SI*, *CS* and *DJCS*), and the MPI client server schemes (*CS* and *DJCS*). Note that the sequential execution takes 31450 seconds.

	OpenMP				MPI	
Procs	**Dyn**	**SI**	**CS**	**DJCS**	**CS**	**DJCS**
4	7643.7	7501.1	10951.9	7642.8	10258.1	10298.2
8	3852.2	3814.6	3971.5	3873.5	4582.7	4474.6
16	1972.0	1997.9	2008.3	2184.5	2216.5	2102.3
32	1147.7	1038.1	1064.1	1125.0	1115.0	1036.5

Table 5: Execution times in seconds for $G(14)$, $k = 3$, $limit = 181$.

Using a parallel *for* loop with a dynamic schedule gives very good results (with an efficiency of about 93%); adapted OpenMP strategies are really efficient too, but if they don't have to use too much memory, which is the case of the dynamic client server schedule.

The MPI codes do not suffer of the OpenMP codes irregularity, with an efficiency of 94.5% for the dynamic client server schedule (88% for the simple client server) while the optimal efficiency is equal to 96.9% taking the server into account. The penalty induced by the server is less important when the number of

[14]The limit is set to 200 and a preliminary treatment constructs a first solution of length 147 instantly.

processors increases, and may become tiny with a very large number of processors.

4.4.5. Further results

Using a great number of processors makes it possible to construct optimal Golomb rulers for larger instances. For example, 32 processors gave the result for G(15) (with $initial_limit$=200 and depthlevel=3) in 4h40, and even in 1h25 with 128 processors.

As shown by our current experiments, the adapted data structures and methods used by the McCracken *Shift Algorithm* [DRM98] may allow to reach really good results.

5. Conclusion and perspectives

The heart of this paper is to give a generic approach for the parallel resolution of combinatorial problems and combinatorial optimization ones: these are modelized with a tree search strategy and then the algorithm is parallelized. For this a set of tasks is generated by dividing the search space. The tasks are subproblems which are independent one from another and can be treated with the same algorithm as for the global problem; the most interesting factor of the parallel algorithm is to balance the tasks between the processors.

We applied this theoretical approach to two particular problems and showed that the efficiencies are good in practice. The applications have been written in C and parallelized using OpenMP or MPI.

The key advantage of OpenMP is to offer a user-friendly tool to parallelize the sequential algorithm: it enables the programmer to develop a parallel application from de sequential one with minimum changes. The experiments provided are those led on a 256 processors SGI'3800. These show that the OpenMP parallel *for* loop with a dynamic schedule is an effective solution. The use of OpenMP parallel region or MPI are also possible, but the programmer has to make some efforts and change some of the code structure to design an efficient parallel application.

A major drawback of OpenMP highlighted by our experiments is that the execution times for the same problem can be very different from an execution to another. It is due to the memory allocation management and we brought a low cost solution to this computation time irregularity.

We proposed a specific dynamic load balancing algorithm (DJCS), based on a client-sever one, taking advantage of the combinatorial optimization specificities. Both with shared memory or message passing, this strategy obtains very good efficiencies with the OGR construction problem.

The main perspective of this work is to write an hybrid solution based on shared memory and message passing to solve the next instances more quickly and to compute open ones. As the computation effort grows geometrically with the size of the problem, the execution on a large cluster of SMP (Symmetric Multi-Processor) should be considered as an interesting alternative to solve one of the first open instances in an acceptable range of time: inside a SMP node, the parallelism should be managed with a shared memory model; outside, a message passing solution would enable the optimization collaboration and the load management between the differents nodes.

Another interesting perspective of this work would be to develop a performance evaluation model for the parallel resolution of combinatorial problems.

Acknowledgments

This work was partly supported by "Romeo"[15], the high performance computing center of the University of Reims Champagne-Ardenne and the "Centre Informatique National de l'Enseignement Supérieur"[16] (CINES, France).

References

[BG77] W. S. Bloom and S. W. Golomb. Applications of Numbered Unidirected Graphs. In *Proceedings of the IEEE, second European Workshop on OpenMP*, pages 562–570, Edinburgh, Scotland, 1977.

[15] `http://www.univ-reims.fr/Calculateur`

[16] `http://www.cines.fr`

[BvR95] F. Bacchus and P. van Run. Dynamic Variable Ordering in CSPs. In *Proceedings of the first International Conference on Principles and Practice of Constraint Programming*, pages 258–274, Cassis, France, 1995.

[CDK+00] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[Chm96] A. Chmeiss. *Réseaux de contraintes : algorithmes de propagation et de décomposition*. PhD thesis, Université des Sciences et Techniques du Languedoc, Aix-Marseille I, France, 1996.

[Dec90] R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Journal of Artificial Intelligence Research (AI)*, 41(3):273–312, 1990.

[DP89] R. Dechter and J. Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.

[DRM98] A. Dollas, W. Rankin, and D. McCracken. A new Algorithm for Golomb Ruler Derivation and Proof of the 19 Marker Ruler. *IEEE Trans. Inform. Theory*, 44:379–382, 1998.

[FD95] D. Frost and R. Dechter. Look-Ahead Value Ordering for Constraint Satisfaction Problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95), Montreal, Quebec, August 20-25, 1995*, pages 600–606, 1995.

[FH95] E. C. Freuder and P. D. Hubbe. Extracting Constraint Satisfaction Subproblems. In *Proceedings of the fourteenth International Joint Conference on Artificial Intelligence*, pages 548–555, Montreal, Quebec, 1995.

[Gar56] Martin Gardner. *Mathematics, Magic and Mystery*. 1956.

[Gar72] M. Gardner. *Mathematical Games*. Scientific American, 1972.

[GLS99] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. In *Proceedings of IJCAI'99*, pages 394–399, 1999.

[HE80] R. M. Haralick and G. L. Elliot. Increasing the Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[HHMS97] Z. Habbas, F. Herrmann, P.-P. Mérel, and D. Singer. Load Balancing strategies for Parallel Forward Search Algorithm with Conflict Based Backjumping. . In *Proceedings of the International Conference on Parallel and Distributed Systems*, Séoul, Korea, 1997.

[HKS00] Z. Habbas, M. Krajecki, and D. Singer. Parallel Resolution of CSP with OpenMP. In *Proceedings of the 2nd European Workshop on OpenMP (EWOMP 2000)*, pages 1–8, Edinburgh, Scotland, 2000.

[HKS01] Z. Habbas, M. Krajecki, and D. Singer. The Langford Problem: A Challenge for Parallel Resolution of CSP. In *Fourth International Conference on Parallel Processing*

and Applied Mathematics (PPAM'2001), volume 2328 of *Lecture Notes in Computer Science*, pages 789–797. Springer-Verlag, Naleczow, Pologne, September 2001.

[HKS02] Z. Habbas, M. Krajecki, and D. Singer. Parallelizing Combinatorial Search in Shared Memory. In *Proceedings of the fourth European Workshop on OpenMP*, Roma, Italy, 2002.

[HKS04] Z. Habbas, M. Krajecki, and D. Singer. Decomposition Techniques for Parallel Resolution of Constraint Satisfaction Problems in Shared Memory: a Comparative Study. *International Jounral of Computational Science and Engeniering (IJCSE)*, to appear, 2004.

[Jai05] Ch. Jaillet. *Résolution parallèlle de problèmes combinatoires en mémoire partagée*. PhD thesis, Université de Reims Champagne-Ardenne, France, 2005.

[JK04a] Ch. Jaillet and M. Krajecki. Constructing Optimal Golomb Ruler in Parallel. In *Proceedings of the 6th European Workshop on OpenMP (EWOMP 2004)*, pages 29–34, Stockholm, Sweden, October 2004.

[JK04b] Ch. Jaillet and M. Krajecki. Solving the Langford Problem in Parallel. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004)*, pages 83–90, Cork, Ireland, July 5-7 2004. IEEE Computer Society.

[Kas89] S. Kasif. Parallel Solutions to Constraint Satisfaction Problems. In *Proceedings of the first International Conference on Principles of Knowledge Representation and Reasoning*, pages 180–188, Toronto, Ontario, 1989.

[KFJ+05] M. Krajecki, O. Flauzac, Ch. Jaillet, P.-P. Mérel, and R. Tremblay. Solving an open Instance of the Langford Problem using CONFIIT: a Middleware for Peer-to-Peer Computing. *Parallel Processing Letters*, to appear, 2005.

[KFM04] M. Krajecki, O. Flauzac, and P.-P. Mérel. Focus on the Communication Scheme in the Middleware CONFIIT Using XML-RPC. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, USA, April 26-30 2004. IEEE Computer Society.

[Kra99] M. Krajecki. An Object Oriented Environment to Manage the Parallelism of the FIIT Applications. In V. Malyshkin, editor, *Parallel Computing Technologies, 5th International Conference, PaCT-99*, volume 1662 of *Lecture Notes in Computer Science*, pages 229–234. Springer-Verlag, St. Petersburg, Russia, September 1999.

[LHB92] Q. Y. Luo, P. G. Hendry, and J. T. Buchanan. A New Algorithm for Dynamic Distributed Constraint Satisfaction Problems. In *Proceedings of the fifth Florida Artificial Intelligence Research Symposium*, pages 52–56, 1992.

[Mac77] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mér98] P.-P. Mérel. *Les problèmes de satisfaction de contraintes : recherche n-aire et parallélisme – Application au placement en CAO*. PhD thesis, Université de Metz, France, 1998.

[Mil99] J.E. Miller. *Langford's Problem. Online*, 1999. http://www.lclark.edu/˜miller/langford.html.

[MM88] R. Mohr and G. Masini. Good Old Discrete Relaxation. In *Proceedings of ECAI'88*, pages 651–656, 1988.

[Mon74] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Pictures Processing. *Information Sciences*, 7:95–132, 1974.

[Ope05] OpenMP Architecture Review Board. *OpenMP Application Program Interface, version 2.5*, May 2005. available at http://www.openmp.org.

[Pac96] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[Pro93] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9:268–299, 1993.

[RAASR99] A. Ruiz-Andino, L. Araujo, F. Sàenz, and J. Ruz. Parallel Implementation of Constraint Solving. In *Proceedings of the 5th. Intenational Conference PaCT-99, LNCS-1662*, pages 466–471, St.Petersbourg, Russia, 1999.

[Ran93] W. Rankin. Optimal Golomb Rulers: An Exhaustive Parallel Search Implementation, Master's thesis, Duke University, 1993.

[RK87] V. N. Rao and V. Kumar. Parallel Depth First Search. Part I. Implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.

[RS90] Sanjay Ranka and Sartaj Sahni. *Hypercube Algorithms with Applications to Image Processing and Pattern Recognation*. Springer-Verlag, 1990.

[SF94] D. Sabin and E. C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the eleventh European Conference on Artificial Intelligence*, pages 125–129, Amsterdam, Netherlands, 1994.

[SHL95] S. Soliday, A. Homaifar, and G. Lebby. Genetic Algorithm Approach to the Search for Golomb Rulers. In L. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 528–535, San Francisco, CA, USA, 1995. Morgan Kaufmann.

[Sim83] J. E. Simpson. Langford Sequences: perfect and hooked. *Discrete Math*, 44(1):97–104, 1983.

[Smi00] B. Smith. Modelling a Permutation Problem. In *Proceedings of ECAI'2000, Workshop on Modelling and Solving Problems with Constraints, RR 2000.18*, Berlin, Germany, 2000.

[Smi01] B. Smith. Dual Models of Permutation Problems. In Toby Walsh, editor, *Proceedings of CP 2001, 7th International Conference on Principles and Practice of Constraint*

Programming, volume 2239 of *Lecture Notes in Computer Science*, pages 615–619, Paphos, Cyprus, November 26 - December 1, 2001. Springer.

[SSW99] B. Smith, K. Stergiou, and T. Walsh. Modelling the Golomb Ruler Problem. In *Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99), Workshop on Non Binary Constraints.*, Stockholm, Sweden, August 2, 1999.

[Ter01] C. Terrioux. Cooperative Search and Nogood Recording. In *Proceedings of IJCAI'01, International Joint Conference on Artificial Intelligence*, pages 260–265, Seattle, USA, 2001.

[Wal93] R. J. Wallace. Why AC-3 is Almost Always Aetter than AC-4 for Establishing Arc-consistency in CSPs. In *Proceedings of the thirteenth International Joint Conference on Artificial Intelligence*, pages 239–245, Chambéry, France, 1993.

[Wal01] T. Walsh. Permutation Problems and Channelling Constraints. Technical Report APES-26-2001, APES Research Group, January 2001. available at http://www.dcs.st-and.ac.uk/˜apes/apesreports.html.

<!– Local IspellDict: english –> <!– Local IspellPersDict: /emacs/.ispell-english –>

Authors addresses:

CReSTIC, LICA

Département de Mathématiques et Informatique

UFR Sciences Exactes et Naturelles

Université de Reims Champagne-Ardenne

Campus du Moulin de la Housse

BP 1039 - 51687 REIMS Cedex 2 { michael.krajecki, christophe.jaillet, alain.bui} @univ-reims.fr