Solving the Langford problem in parallel

Christophe Jaillet
Université de Reims Champagne-Ardenne
Département de Mathématiques et Informatique
Moulin de la Housse - BP 1039
F-51687 Reims Cedex 2
Email : christophe.jaillet@univ-reims.fr

Michaël Krajecki
Université de Reims Champagne-Ardenne
Département de Mathématiques et Informatique
Moulin de la Housse - BP 1039
F-51687 Reims Cedex 2
Email : michael.krajecki@univ-reims.fr

***Abstract*— In this paper, the parallel resolution of the Langford problem is studied. Two different approaches are developed.**

First, an explicit construction of all the solutions is done using a shared memory. The application associated to this approach is written in C using the standard OpenMP library.

Second, a parallelization of the algebraic method introduced by Godfrey is proposed. The application is taking advantage of MPI and has revealed efficient up to 128 processors. This solution opens up some new perspectives such as solving the already resolved instances of the problem more quickly and solving the next two open instances of the problem in a near future.

***Index Terms*— Langford problem, OpenMP, MPI, parallel algorithm**

I. Introduction

The Langford problem is a typicaly hard combinatorial problem. The last open instance to be solved took one week CPU time on a pool of 3 PCs with quite a specific search algorithm in 2002. It represents a real *challenging* problem especially for parallel combinatorial search.

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 parallel resolution of CSP (Constraint Satisfaction Problems) with a shared memory [1]. The conclusions of that work provided a general approach to solve combinatorial problems in parallel and conducted the first choices made to solve the Langford problem.

The first part of this paper presents a general framework for parallel resolution of combinatorial problems. A first step consists in a simple decomposition strategy of the Tree Search. 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 second part is dedicated to the parallelization of the algebraic method introduced by M. Godfrey. A simple client/server scheme has been developed in C/MPI to solve large Langford problems using up to 128 processors. The experiments show that $L(2, 20)$ can now be solved in less than 40 minutes.

The paper is organized as follows: the first section is dedicated to the Langford problem and introduces the two classical approaches which are the enumerative and algebraic ones.

The next section proposes a parallel shared memory enumerative solution to the Langford problem. An implementation using OpenMP is introduced. The use of the schedule clause of the parallel *for* loop and of the OpenMP parallel region is discussed in detail. Some experiments on $L(2, 16)$ conclude this part of the work.

Section IV details the MPI parallel version of the algebraic approach. The impact of the granularity is discussed as the number of processors increases. The experiments provided clearly show that the next open instances ($L(2, 23)$ and $L(2, 24)$) will be solved in a near future.

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

II. The Langford problem

C. Dudley Langford gave his name to a classic problem of permutation [2], [3]. While observing his son manipulating blocks of different colors, he noticed that it was possible to arrange three pairs of blocks of different colors (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 1).

Proceedings of the ISPDC/HeteroPar'04
0-7695-2210-6/04 $20.00 © 2004 **IEEE**

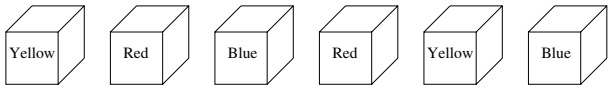

Fig. 1. L(2,3): arrangement for 6 blocks of 3 colors: yellow, red and blue

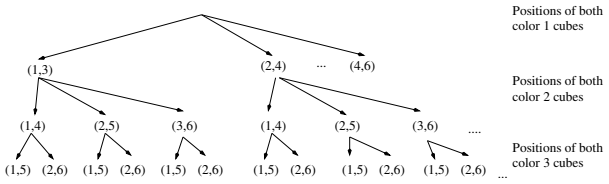

Fig. 2. Search tree for $L(2,3)$

The problem has been generalized to any number of colors n and any number of blocks having the same color 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 colors) 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 $k \in \mathbb{N} \setminus \{0\}$.

At the moment, the instances solved in practice, in a merely combinatorial manner, limit themselves to a small number of colors. 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.

Recently, Toby Walsh and Barbara Smith formulated this problem as a Constraint Satisfaction Problem [4], [5].

The Langford Problem has been approached in different ways (discrete mathematics results, specific algorithms, specific encoding, ...) [6].

A. 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):

- every node of the tree corresponds to the place in the sequence of the cubes of a determined color;
- to the depth p, the first node corresponds to the place of the first cube of color p in first position and it ith node corresponds to the investment of the first cube of color p in position i, $i \in [1,\ 2n-1-p]$;
- every leaf of the tree symbolizes the positions of all cubes;
- a leaf is a solution if it respects the color constraint defined by the Langford problem.

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.

To be efficient, this algorithm should avoid the recursive tree traversal. Moreover, the use of arrays as elementary data structure is strongly recommended. A Sequential Array-Based, Non-recursive Algorithm (written in C) is accessible on page `http://www.lclark.edu/~miller/langford/langford-algorithm.html`.

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

B. Godfrey's algorithm

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 color 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 $X_1X_2X_3X_4X_5...X_{2n}$.

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 :

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

$$2^{2n+1} L(2,n)$$

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[7].

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

III. Solving the Langford problem using OpenMP

In [8], we proposed to formalize the Langford problem as a CSP (*Constraint Satisfaction Problem*) and showed that an efficient parallel resolution is possible. In this section, we propose to develop a specific algorithm taking into account the conclusions of our previous works (management of the memory and load balance).

A. How to solve in parallel the Langford problem

The tree traversal induced by the explicit construction of all solutions can be made in parallel while introducing the following definition for the notion of task: it is associated to the traverse of a particular subtree. While choosing to develop all subtrees to a depth k, at most $(2n-2)^k$ independent tasks can be defined and are accessible using a unique identifier (numbering of the nodes in $2n-2$ basis).

It is noticed that, when introducing a backtracking scheme on the inconsistent branches (for which the first placed cubes do not already respect the color constraint), we observe that the computations associated to these tasks are especially irregular. Finally, it is easy to verify that these tasks are independent and can be solved in any order.

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

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

Where `nbTasks` is the number of tasks deduced by the development of all subtrees to the depth `k` for n colors by the function `generateTasks`. The function `solveTask` is in charge of traversing the subtree associated with the task numbered `task`. At the end, the variable `nbSolutions` contains the number of solutions for $L(2,n)$.

B. Parallel execution with OpenMP

The OpenMP environment has evolved to a standard for shared memory parallelism [9], [10]. It is a complete API for programming shared memory multiprocessors systems. It enables to obtain a parallel code easily since it is 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 parallelism with a shared memory. It conforms to SPMD programming language style. One key advantage of OpenMP, comparing with a Message Passing implementation, is that the development cost of an MPI version would be much more important. It would potentially induce numerous 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 as follows: it is a Coarse-Grained parallelization which uses only one level of parallel *for loop* or only one *parallel section* consisting in the resolution of one subproblem.

C. 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 *# pragma omp for schedule()* in the parallel *for loop*. We will study the three following options:

1) The *static* repartition : each processor is in charge of $\frac{Nbtasks}{Nbproc}$ consecutive tasks. This solution seems to be inadequate in our case because of the irregularity of the Langford tasks.
2) The *Modulo Nbproc* static repartition: the tasks allocation to the processors is computed once at the compiling time. Each processor receives $\frac{Nbtasks}{Nbproc}$ different subproblems following a repartition modulo the number of available processors $Nbproc$. Its main advantage is to distribute the so-called search tree irregularity among the processors.
3) The *Dynamic* repartition: the different tasks are dynamically allocated to processors by the system at the execution time ; there is no guarantee on which thread the tasks are executed.

An experimental evaluation of the parallel resolution of the Langford problem with the Search Tree decomposition strategy was lead using a Silicon Graphics Origin'3800 with 512 R14K 500 MHz processors. This study was limited to 64 processors.

To have a parallel version of the program, an OpenMP directive is added before the loop (here for a dynamic schedule) :

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

The schedule clause takes three different values which are `static`, `(static,1)` and `dynamic`. The reader may notice that, thanks to OpenMP, the parallel version is easy to define.

First, a comparison between the three repartition policies for the Search Tree decomposition strategy is provided. Table I shows the execution time for $L(2,14)$ where k (the depth of subdivision) is equal to 5. As expected, the static repartition is not very efficient when the number of processors grows up. It is interesting to notice that the modulo repartition is not so far from the dynamic repartition which is the best in our experiment.

TABLE I
EXECUTION TIMES FOR $L(2,14)$ IN SECONDS

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

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

D. Tasks allocation in a parallel region

OpenMP also provides another way to produce parallel application. The parallel region is executed by all the processors. The user is explicitly in charge of distributing the load among the processors. In the new experiment, we redefine the three different load balancing strategies provided by the parallel loop.

The static repartition is defined by the following statements :

```
nbTasks = generateTasks(n,k);
nbSolutions=0;
#pragma omp parallel
            reduction(+:nbSolutions) {
  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++) {
    nbSolutions+=solveTask(task);}}
```

The variables `start` and `end` are introduced in order to explicitly balance the load to the processors. Because they are defined in the parallel region, each processor has its own copy of these variables. The reduction clause is necessary to ensure the correctness of the computation.

The definition of the modulo repartition is simpler than the static distribution. In this solution, processor P_i begins with task T_i, continues with T_{i+nbp}, and so on :

```
nbTasks = generateTasks(n,k);
nbSolutions=0;
#pragma omp parallel
            reduction(+:nbSolutions) {
  int nbp, p, task;
  nbp = omp_get_num_threads();
  p = omp_get_thread_num();
  for(task=p;task<end;task+=nbp) {
    nbSolutions+=solveTask(task);
  }
}
```

The dynamic repartition is implemented using a shared variable `nextTask` which indicates the 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). The *for* loop is also replaced by a *while* loop to take into account the load balance factor.

```
nbTasks = generateTasks(n,k);
nbSolutions=0;
int nextTask;
#pragma omp parallel
            reduction(+:nbSolutions) {
  int nbp, p, task;
  nbp = omp_get_num_threads();
  p = omp_get_thread_num();
  #pragma omp single
  nextTask = nbp;
  task=p;
  while(task<nbTasks) {
    nbSolutions+=solveTask(task);
    #pragma omp critical (load)
    task = nextTask++;
  }
}
```

Figure 3 gives an overview of the different experiments. The efficiencies observed are very good with a dynamic load balancing scheme. The reader shall take into account that the execution time for 64 processors

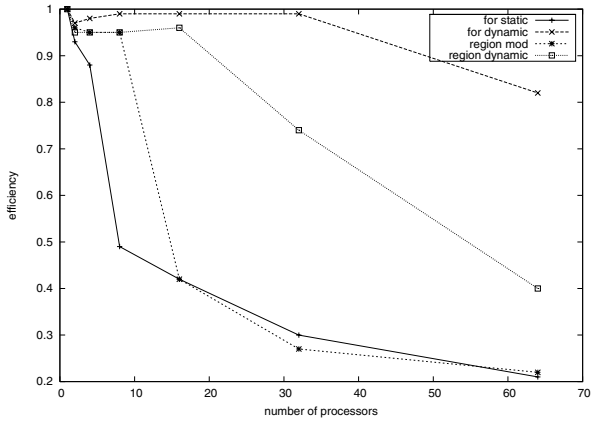

Fig. 3. Efficiency for $L(2, 14)$

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. The dynamic region, using a shared variable accessed in a critical way, is not very effective with more than 16 processors. To observe 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.

To conclude, some remarks have to be made. First, the Origin'3800 was not dedicated to these experiments, other users compute at the same time on this architecture. It is probably the reason why some bad results were observed. Each computation has been repeated 50 times. For example, $L(2, 14)$ 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 observed more often. Thus, $L(2, 14)$ is 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 closed to 60-65 seconds for 21 tries. An explanation can be proposed. As the architecture is shared by different users, the Langford application has no guaranty about the memory allocation : one job started before the Langford one can consume a large part of the whole memory and in particular the local memory of the processors in charge of the Langford job. In this case, the memory access time is very bad and this aspect is very critical for the Langford application.

IV. Godfrey's algorithm in parallel

In section II-B, the evaluation of $L(2, n)$ by $\sum_{(x_1,\ldots,x_{2n})\in\{-1,1\}^{2n}} (\prod_{i=1}^{2n} x_i) \prod_{i=1}^{n} \sum_{k=1}^{2n-i-1} x_k x_{k+i+1}$ has been introduced.

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

A. Optimization using the Gray code

$\sum_{k=1}^{2n-i-1} x_k x_{k+i+1}$ has to be calculated 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 only one of the x_i for each time (which allows to get one sum from the previous one). The ordering of these changes is made using the Gray code sequence, and it would be 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 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, though it uses 4 Mb ; the rest of the memory is used for the computation itself.

B. A parallel version using a Message Passing Interface

Message passing is a programming paradigm used widely on parallel computers, especially with distributed memory. the Message Passing Interface (MPI) is a standard approach for message passing programming [11]. This standard defines the user interface and functionality for a large number of message-passing capabilities. The users expect form MPI a degree of portability comparable to that given by programming languages such as C or Fortran. They want the same message-passing source code to be executed on a variety of architectures as long as the MPI library is available.

The skeleton of the MPI program can now be introduced. As in the parallelization of the first approach, the simplest solution for the user has been chosen. The task allocation is done during the execution by a client/server scheme.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nbp);
MPI_Comm_rank(MPI_COMM_WORLD, &p);
if ( p == 0 ) { /* server */
  nbTasks = localPow(2, k);
  while ( (nextTask < nbTasks)) {
    /*  get the result */
    MPI_Recv(&noClient, 1,
    MPI_INT, MPI_ANY_SOURCE,
    1, MPI_COMM_WORLD, &ierr);
    MPI_Recv(&currentTask, 1, MPI_INT,
    noClient, 2, MPI_COMM_WORLD, &ierr);
    MPI_Recv( &(sum.nbMots), 1, MPI_INT,
    noClient, 4, MPI_COMM_WORLD, &ierr);
    MPI_Recv( sum.sequence, NB_MAX_MOTS,
    MPI_LONG, noClient,
    5, MPI_COMM_WORLD, &ierr);

    /* send the next task */
    nextTask++;
    MPI_Send(&nextTask, 1, MPI_INT,
    noClient, 0, MPI_COMM_WORLD);
  } /* end while */
  MPI_Finalize();

} /* end of server */
else { /* client */
  MPI_Recv(&task, 1, MPI_INT, 0, 0,
  MPI_COMM_WORLD, &ierr);
  while ( task < nbTasks ) {
    sum=solveTask(task);

    /* send the result */
    MPI_Send(&p, 1, MPI_INT, 0,
    1, MPI_COMM_WORLD);
    MPI_Send(&task, 1, MPI_INT, 0,
    2, MPI_COMM_WORLD);
    MPI_Send(&(sum), 1, MPI_INT, 0,
    4, MPI_COMM_WORLD);
    MPI_Send(sum, NB_MAX_MOTS, MPI_LONG, 0,
    5, MPI_COMM_WORLD);

    /* receive next task */
    MPI_Recv(&task, 1, MPI_INT, 0,
    0, MPI_COMM_WORLD, &ierr);
  } /* fin du while */
  MPI_Finalize();

} /* end of client */
```

C. Solving $L(2,16)$

Table II 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.

TABLE II
EXECUTION TIMES FOR $L(2,16)$ IN SECONDS

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

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 SGI'3800.

The reader may remember that only $p-1$ processors on p effectively resolve the problem because of the server defined 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 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.

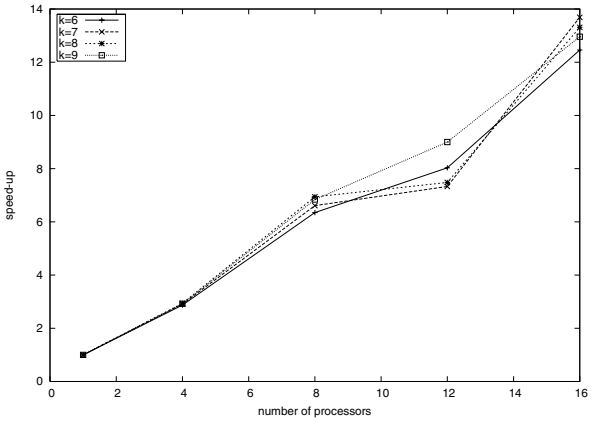

Fig. 4. Speed-up for $L(2,16)$

D. 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 $L(2,20)$.

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

1) The impact of the depthlevel on $L(2,19)$: Table III 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.

TABLE III
EXECUTION TIMES FOR $L(2,19)$ IN SECONDS

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

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.

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.

2) $L(2,20)$ 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 $L(2,24)$.

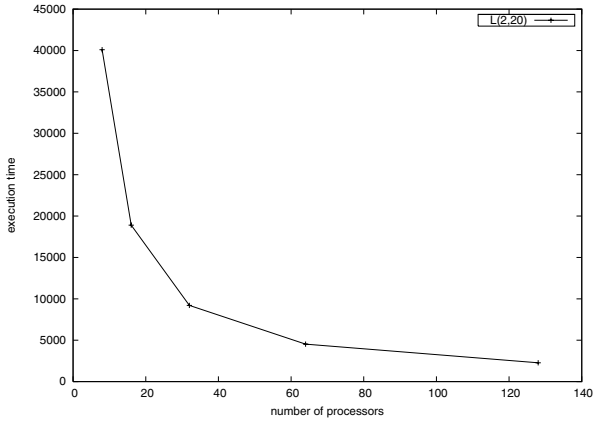

Fig. 5. Execution times for $L(2,20)$ in seconds

V. CONCLUSION AND PERSPECTIVES

In this paper, the parallel resolution of the Langford problem has been studied.

The first part of this work consisted in a parallel shared memory enumeration of all the solutions. The application has been written in C using OpenMP. The key advantage of OpenMP is the ease of its use to parallelize the sequential algorithm. With minimum changes to this algorithm, it has been possible to design an efficient parallel version. The provided experiments on SGI'3800 show that the parallel *for* loop with the dynamic schedule provided by OpenMP is an effective solution. The use of parallel region is also possible, but the programmer has to make some efforts to design an efficient parallel application. The main drawback highlighted by the experiments is that the execution times for the same problem can be very different from one execution to another. It is probably due to the memory management in a multi-user environment.

The second part of this study proposed an MPI parallel application to solve the Langford problem using the algebraic method. The granularity of the parallel algorithm can be easily adjusted by a depthlevel factor. A very simple client/server application written in C with MPI has been developed. It has been shown that the granularity of the application has a significant impact on the execution time when the number of processors increases. The experiments provided are conclusive : this solution has reduced the resolution time of $L(2,20)$ from one week to nearly 40 minutes using 128 processors.

The main perspective to this work is to write a hybrid solution based and OpenMP and MPI to solve $L(2,23)$

which is the next open problem. Based on experimental observations, the execution time of $L(2,23)$ should be 120 to 200 times much more time consuming than $L(2,20)$. This is the reason why the execution on a large cluster of SMP (Symmetric Multi-Processor) should be considered as an interesting alternative to solve $L(2,23)$ in an acceptable range of time (one week or less). Inside an SMP node, the parallelism should be managed in a shared memory by OpenMP. Outside the nodes, a message passing solution will allow the management of the load between the nodes.

The Langford problem can be considered as a prototype for the permutation problems class, as suggested by T. Walsh and B. Smith. Therefore, this work can be extended to other problems of this interesting class.

ACKNOWLEDGMENTS

This work was partly supported by "Romeo"[1], the high performance computing center of the University of Reims Champagne-Ardenne and the "Centre Informatique National de l'Enseignement Superieur"[2] (CINES), France.

REFERENCES

[1] Z. Habbas, M. Krajecki, and D. Singer, "Parallel resolution of csp with openmp," in *Proceedings of the second European Workshop on OpenMP*, Edinburgh, Scotland, 2000, pp. 1–8.

[2] M. Gardner, *Mathematics, Magic and Mystery*, 1956.

[3] J. E. Simpson, "Langford sequences: perfect and hooked," *Discrete Math*, vol. 44, no. 1, pp. 97–104, 1983.

[4] T. Walsh, "Permutation problems and channelling constraints," APES Research Group, Tech. Rep. APES-26-2001, January 2001. [Online]. Available: http://www.dcs.st-and.ac.uk/~apes/reports/apes-26-2001.ps.gz

[5] B. Smith, "Modelling a Permutation Problem," in *Proceedings of ECAI'2000, Workshop on Modelling and Solving Problems with Constraints, RR 2000.18*, Berlin, 2000. [Online]. Available: http://www.dcs.st-and.ac.uk/~apes/2000.html

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

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

[8] 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.

[9] *OpenMP C and C++ Application Program Interface*, OpenMP Architecture Review Board, Oct. 1997, http://www.openmp.org.

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

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

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

[2] http://www.cines.fr