Constructing Optimal Golomb Rulers 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, a construction of the Golomb optimal rulers is studied with a tree search approach. Improvements to the basic algorithm are given and it is parallelized using a shared memory. The application associated to this approach is written in C using the standard OpenMP and MPI libraries.**

The algorithm takes advantage of a collaboration mechanism between the processors. Different load balancing strategies are studied.

The application has proved efficient up to 32 processors. This solution opens up some new perspectives such as solving the already resolved instances of the problem more quickly, and solving next open instances in the future.

***Index Terms*— Golomb ruler, parallel algorithm, load balancing, shared memory, OpenMP, distributed memory, MPI**

I. Introduction

Golomb rulers are numerical sequences. The Optimal Golomb Rulers (*OGR*) construction is a hard combinatorial problem that consumes a high computation time. 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 construct optimal Golomb rulers.

The paper is organized as follows: the first section is dedicated to the Golomb rulers and introduces a classical approach.

The next section proposes a parallel shared memory solution for the construction. Then 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.

Section IV gives experimental results. It illustrates the tasks allocation and the effect of collaboration between the processors. The results of the different load balancing strategies are compared and the impact of granularity is studied.

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

II. Golomb rulers

A Golomb ruler is an ordered sequence of integers, such that all differences between any two of them are different: considering that these *marks* refer positions on a linear scale, the *distances* on the *ruler* have all to be different [2] (see figure 1).

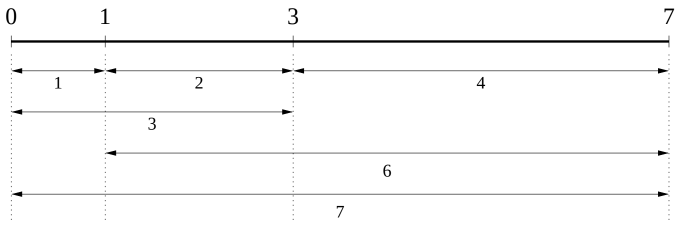

Fig. 1. A *non optimal* 4 marks Golomb ruler

It is easy to construct a ruler for any number n of marks, but the interesting cases deal with minimal length sequences [3] (*ig* (0,1,4,6) is optimal for 4 marks).

These sequences are a mathematical curiosity, but have also many applications in a wide variety of fields, including x-ray crystallography [4], self-orthogonal codes (for error detection and correction in coding theory) and communication (radio frequences, radio astronomy and PPM[1]).

[1]pulse phase modulation

This construction problem is a combinatorial one, and has been approached in different ways [5], [6].

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”.

A. A tree search algorithm

The construction of optimal Golomb rulers can be modelized as a tree search problem, with the view to minimize the length of the constructed sequence:

- as the length of an optimal n marks ruler does not exceed $1 + 2 + 4 + \cdots + 2^{n-2} = 2^{n-1} - 1$, 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 2);
- every leaf of the tree symbolizes a sequence which is a solution if the values are ordered and if all the differences are different.

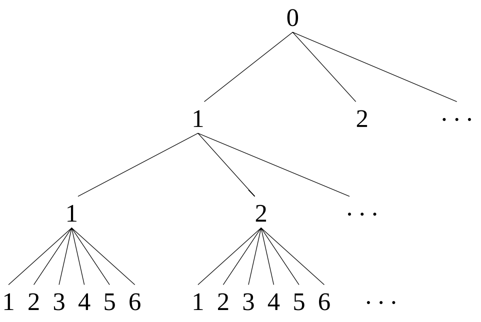

Fig. 2. Search tree for 4 marks Golomb rulers

It is now sufficient to propose a walk through the search tree, in depth first, keeping the best observed length, to get a simple sequential algorithm finding an optimal Golomb ruler.

This constructive algorithm belongs to the backtrack algorithms class; to be efficient, any recursive tree traversal must be avoided and the use of arrays as elementary data structure is strongly recommended.

B. Improvements

The search space can be reduced by applying some simple construction remarks (see figure 3):

- the best possible length $best\_length$ is set to $initial\_limit = 2^n - 1$ at the beginning (or it can be fixed at the execution time);
- when placing the k-th mark at position $pos(k)$, there are still $r = n - k$ remaining marks to be placed, which can’t have a length $remaining\_length(r)$ less than $1 + 2 + \cdots + r = r(r + 1)/2$ or than $G(r + 1)$: it induces the following new constraint:

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

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

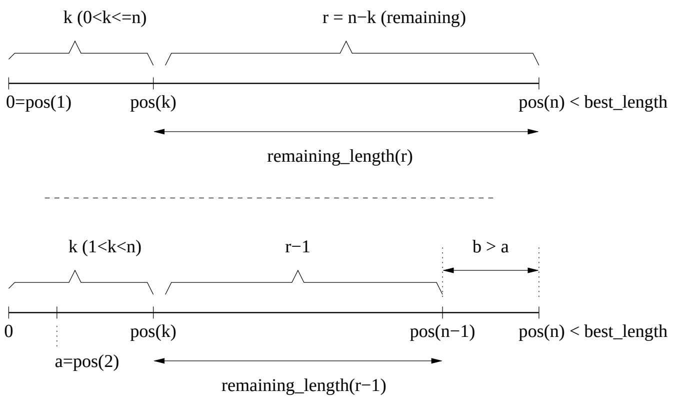

Fig. 3. New constraints based on construction remarks

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 CSP).

III. Golomb rulers parallel algorithm

In [7], 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. Tasks generation

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 $(initial\_limit - 1)^k$ tasks can be defined.

Assume that this construction only keeps the *possible* tasks (those whose values are well ordered and satisfy the *all-diff differences* constraint, and which respect the search reduction constraints) that are accessible using a unique identifier.

The algorithm can be summarized in c-like mode by:

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

where `nbTasks` is the number of initially useful tasks, deduced by the development of all subtrees to the depth `k` by the function `generateTasks` for `n` marks with `initial_limit` as initial best length. The function `solveTask` is in charge of traversing the subtree associated with the task numbered `task`. At the end, the variable `best_length` contains the optimal Golomb ruler length for the given value of `n`.

It can be noticed that, when introducing a backtracking scheme on the inconsistent branches (as soon as a first constraint is violated) the computations associated to these remaining tasks are particulary irregular.

B. Collaboration

The tasks may take advantage of the results of the previous ones by upgrading their value of $best\_length$, if such a collaboration scheme is used, because of the search space reduction.

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 three different levels of information: *global information*, *processor information*, and *task information*.

The goal of our application is to compute, for a given number of marks, the (global) minimum length of a Golomb ruler. It can be reached in three different manners:

- by keeping the best length 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 collaboration);
- by sharing the best length between all the tasks of each processor (the processors gather their last values before the program concludes): this enables to cut branches in the search tree, and even to directly evict some of the tasks (even in a sequental execution);
- by sharing the best length between all the tasks and updating it after each task: this solution provides an efficient way to decrease the best length used by each task; it may reduce the search space more quickly, but may impose an exchange overhead.

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 of collecting the $Global\_best$ value as a shared information.

C. Parallel execution with OpenMP

The OpenMP environment has evolved to a standard for shared memory parallelism [8], [9]. It is a complete API for programming shared memory multiprocessors systems and makes it possible to obtain a parallel code easily (almost close to the sequential one).

1) Tasks allocation in a parallel loop: We will study the three following schedules: the *Static* repartition, the *Modulo Nbproc* static repartition, and the *Dynamic* one.

Thanks to OpenMP the parallel code is easy to define, adding a compilation directive before the choosen *for* loop instruction (here for a dynamic schedule with a full collaboration):

```
nbTasks = generateTasks(n,max,k);
Global_best = initial_limit;
#pragma omp parallel for
                  schedule(dynamic)
                  shared(Global_best)
for(task=0 ; task<nbTasks ; task++)
  if ( useful(task,Global_best) ) {
    best=solveTask(task,Global_best);
    if ( best < Global_best )
#pragma omp critical
        if ( best < Global_best )
          Global_best = best;
  }
```

The reader may notice that the only sensible part of the code consists in the critical zone, necessary to update the `Global_best` variable.

2) Tasks allocation in a parallel region: A parallel region is executed by all the processors. The user is explicitly in charge of distributing the load among the processors.

We showed in [10] how to emulate the main parallel *for* loop schedule with parallel regions, and proposed other adapted load balancing strategies to solve the Langford problem in parallel. The good results obtained for this study encouraged us to carry on in this direction.

For the specific problem of constructing optimal Golomb rulers, we first noted that most of the initially useful tasks have actually became useless when they have to be treated (see the experimental results in next section). This phenomenon induces that, when a processor becomes idle, it has to take several tasks before having an *interesting* one. This creates a high concurrency for the tasks allocation which may considerably reduce computation efficiency.

Thus, two different load balancing strategies using parallel regions are proposed :

- A servor initiated one first divides the tasks into tasks queues in the static modulo manner. Every processor begins with its queue. When a processor becomes idle it chooses a non empty queue and takes tasks in it, one by one, in a *while* manner, or by directly taking half of this queue.
- The client-servor load balancing strategy shares the index of the next available task between the processors. In order to improve the "one by one" dynamic schedule, processors take as many tasks as necessary to find a useful one: a lock is set on the $next$ variable and tasks are consulted in a *while* manner; then the lock can be unset. The advantage of this strategy is to reduce the number of times a lock is set, but the locks are set for a longer period.

D. 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 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. The skeleton of the MPI program can now be introduced. Note that all the processors generate the possible tasks and then that the servor distributes them.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nbp);
MPI_Comm_rank(MPI_COMM_WORLD, &p);
if ( p == 0 ) { /* server */
  nbTasks = generateTasks(n, max,k);
  global_best = initial_limit;
  initialSends();
  while ( (nextTask < nbTasks)) {
    /*  get the result */
    MPI_Recv(&noClient, 1,
    MPI_INT, MPI_ANY_SOURCE,
    1, MPI_COMM_WORLD, &ierr);
    MPI_Recv(&best, 1, MPI_INT,
    noClient, 2, MPI_COMM_WORLD, &ierr);
    if (best < global_best)
      global_best = best ;

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

} /* end of server */
else { /* client */
  nbTasks = generateTasks(n, max,k);
  best = initial_limit;
  MPI_Recv(&task, 1, MPI_INT, 0, 0,
  MPI_COMM_WORLD, &ierr);
  while ( task < nbTasks ) {
    best = solveTask(task,best);

    /* send the result */
    MPI_Send(&p, 1, MPI_INT, 0,
    1, MPI_COMM_WORLD);
    MPI_Send(&best, 1, MPI_INT, 0,
    2, MPI_COMM_WORLD);

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

} /* end of client */
```

IV. Experimental results

As evoked in the previous sections, the tested algorithm takes advantage of a full cooperation between the tasks, through the processors that share the information of the best known length.

The experimental evaluation we present here was led using a Silicon Graphics Origin'3800 with 512 R14K 500 MHz processors. This study was limited to 32 processors to experiment the effects of granularity and of the different load balancing strategies, and to 128 for constructing further rulers.

A. Granularity

Considering a given problem with a given value of $initial\_limit$, the choice of the depthlevel may influence computational time, as shown by the following example, where G(13) was computed with $initial\_limit = 200$ on 32 processors (see figure 4).

- 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.

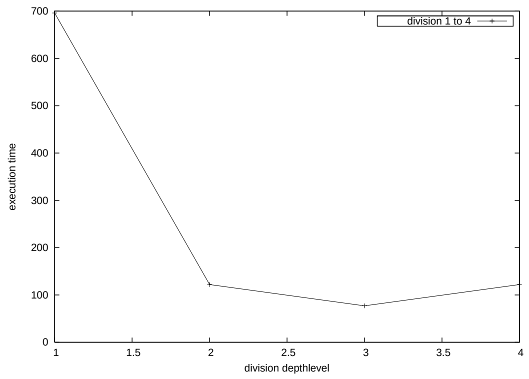

Fig. 4. G(13), $initial\_limit$=200, influence of the granularity

B. 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[2] of 147 and dividing the search tree with depthlevel 3, one finishes in about 60 seconds and another one in about 140).

Table I 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 servor initiated and client servor parallel region schedules (*SI* and *CS*), and the MPI client servor. Note that the sequential execution takes 31450 seconds.

TABLE I

Execution times for $G(14)$ in seconds

	OpenMP			MPI
Procs	**Dyn**	**SI**	**CS**	**CS**
4	7643.7	7501.1	10951.9	10258.1
8	3852.2	3814.6	3971.5	4582.7
12	2714.1	2679.9	2831.3	2881.5
16	1972.0	1997.9	2008.3	2216.5
24	1466.0	1441.8	1521.4	1535.9
32	1147.7	1038.1	1064.1	1115.0

Using a parallel *for* loop with a dynamic schedule gives very good results (with an efficiency of about 93%), however adapted the other OpenMP strategies are.

The MPI code gives good results too, with an efficiency of about 88% while the optimal efficiency is equal to 91% taking the server into account. The penalty induced by the server is less important when the number of processors increases, and may become tiny with a very large number of processors.

C. 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.

D. How the OpenMP experiments were led

To conclude, a remark has to be made on the computational irregularity observed for the OpenMP versions of the program.

Each computation has been repeated 20 times. For example, $G(13)$ with the dynamic *for* loop and 2 processors was computed in 991-995 seconds 17 times and in 1150-1540 seconds 3 times. When the number of processors

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

increases, this instability is observed more often. Thus, $G(13)$ was computed on 16 processors using the modulo *for* loop in about 180 seconds only 1 time in 20 tries, and the relative standard deviation was about 135% (the values went from 180 to 1110 seconds and even two computations didn't finish in the 30 minutes allowed).

Such an irregularity was observed on all the parallel machines we used (an IBM SP Power 3 NH2 and a Sunfire 6800)

As the architecture is shared by different users, the application has no guaranty about the memory allocation. On the other hand their is no way to know where a processor allocates its own memory; moreover shared memory, allocated in a particular place, is *near* some of the processors and *far* from other ones, whose memory access times may become very bad. The effects are increased by the fact that our algorithm uses much memory and is based on a very large number of memory accesses, in private memory as well as in shared one. This is the reason why we had to do many computations for each experiment, and we only retained the best value (which corresponds to the ”good” memory allocation case).

V. Conclusion and perspectives

In this paper, the parallel construction of the optimal Golomb rulers has been studied using a tree search algorithm. The application has been written in C using OpenMP or MPI. The key advantage of OpenMP is to offer a user-friendly tool to parallelize the sequential algorithm: with minimum changes to this algorithm, it has been possible to design an efficient parallel version. The experiments provided on SGI'3800 show that the parallel *for* loop with the dynamic schedule provided by OpenMP 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 highlighted by the experiments is that with OpenMP 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 main perspective of this work is to write a hybrid solution based on OpenMP and MPI to solve the next instances more quickly and compute open ones. As the computation effort grows geometrically with the number of marks, 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 in a shared memory by OpenMP. Outside the nodes, a message passing solution would enable the management of the load between the nodes.

Acknowledgments

This work was partly supported by "Romeo"[3], the high performance computing center of the University of Reims Champagne-Ardenne and the "Centre Informatique National de l'Enseignement Superieur"[4] (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, *Mathematical Games*. Scientific American, 1972.

[3] W. Rankin, “Optimal golomb rulers: An exhaustive parallel search implementation,” Master's thesis, Duke University, 1993.

[4] W. Bloom and S. Golomb, “Applications of numbered unidirected graphs,” in *Proceedings of the IEEE, 65second European Workshop on OpenMP*, Edinburgh, Scotland, 1977, pp. 562–570.

[5] B. Smith, K. Stergiou, and T. Walsh, “Modelling the golomb ruler problem,” 1999.

[6] S. Soliday, A. Homaifar, and G. Lebby, “Genetic algorithm approach to the search for golomb rulers,” in *Proceedings of the Sixth International Conference on Genetic Algorithms*, L. Eshelman, Ed. San Francisco, CA: Morgan Kaufmann, 1995, pp. 528–535.

[7] 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.

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

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

[10] C. Jaillet and M. Krajecki, “Solving the langford problem in parallel,” in *Proceedings of ISPDC'04, International Symposium on Distributed and Parallel Computing*, Cork (Ireland), 2004.

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

[3]`http://www.univ-reims.fr/Calculateur`
[4]`http://www.cines.fr`