Chapter 2

Computational Wall: Langford Problem

2.1 Introduction

Our goal is to determine the behavior of accelerators compared to classical processor in case of irregular-computationally heavy problems. For this purpose we choose the Langford problem which is an academic problem of combinatorial counting.

We present the problem and expose the two possible methods to solve it:

- The tree traversal, call Miller's method, providing us benchmark targeting irregular memory and computation.
- The algebraic version, calls Godfrey's method, showing the performances of many-core vs multi-core architectures in respect to computationally heavy and memory irregular behaviors.

We show the optimizations made to the regular processor algorithm to efficiently implement this application on GPU. We compare the two approaches with classical processor and GPU implementation. The results are then presented to show the acceleration using the whole ROMEO supercomputer.

2.1.1 The Langford problem

C. Dudley Langford gave his name to a classic permutation problem [Gar56, Sim83]. While observing his son manipulating blocks of different colors, he noticed that it was possible to arrange three pairs of different colored blocks (yellow, red and blue) in such a way that only one block separates the red pair - noted as pair 1 - , two blocks separate the blue pair - noted as pair 2 - and finally three blocks separate the yellow one - noted as pair 3 - , see figure 2.1.

This problem has been generalized to any number n of colors and any number s of blocks having the same color. $L(s, n)$ consists in searching for the number of solutions to the Langford problem, up to a symmetry. 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 he stated that

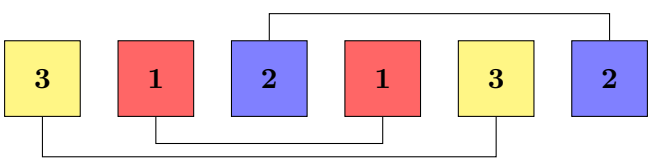


Figure 2.1: L(2,3) arrangement

Instance	Solutions	Method	Computation time
L(2,3)	1	Miller algorithm	-
L(2,4)	1		-
...	...		...
L(2,16)	326,721,800		120 hours
L(2,19)	256,814,891,280		2.5 years (1999) DEC Alpha
L(2,20)	2,636,337,861,200	Godfrey algorithm	1 week
L(2,23)	3,799,455,942,515,488		4 days with CONFIIT
L(2,24)	46,845,158,056,515,936		3 months with CONFIIT
L(2,27)	111,683,611,098,764,903,232		2 days on ROMEO
L(2,28)	1,607,383,260,609,382,393,152		23 days on ROMEO

Table 2.1: Solutions and time for Langford problem using different methods

$L(2,n)$ has solutions for all n such that:

$$\text{solutions for: } \begin{cases} n = 4k \\ n = 4k-1 \end{cases} \quad k \in \mathbb{N}^+ \tag{2.1}$$

The central resolution method consists in placing the pairs of cubes, one after the other, on the free places and backtracking if no place is available (see figure 2.3 for detailed algorithm).

The Langford problem has been approached in different ways: discrete mathematics results, specific algorithms, specific encoding, constraint satisfaction problem (CSP), inclusion-exclusion ... [Mil99, Wal01, Smi00, Lar09]. In 2004, the last solved instance, $L(2,24)$, was computed by our team [JK04] using a specific algorithm. Table 2.1 presents the latest results and number of solutions. $L(2,27)$ and $L(2,28)$ have just been computed but no details were given.

The most efficient known algorithms are: the Miller backtrack method, the Godfrey algebraic method and the Larsen inclusion-exclusion method. The Miller one is based on backtracking and can be modeled as a CSP; it allowed us to move the limit of explicits solutions building up to $L(2,21)$ but combinatorial explosion did not allow us to go further. Then, we use the Godfrey method to achieve $L(2,24)$ more quickly and then recompute $L(2,27)$ and $L(2,28)$, presently known as the last instances. The Larsen method is based on inclusion-exclusion [Lar09]; although this method is effective, practically the Godfrey one is better. The latest known work on the Langford Problem is a GPU implementation proposed in [ABL15] in 2015. Unfortunately this study does not provide any performance considerations but just gives the number of solution of $L(2,27)$ and $L(2,28)$.

2.2 Miller algorithm

The Miller's method is based on a tree traversal to be able to check all the cubes positions. This method finds its limits because even using branch cutting algorithm to traverse the tree, the number of branches to explore stay very high. We implemented a multi-core and many-core version of it for the purpose of our benchmark. Indeed, this huge tree traversal requires irregular computation and memory accesses: two elements fitting the wall we want to confront to our architectures.

In this section we present our multi-GPU cluster implementation of the Miller's algorithm. First, we introduce the backtrack method itself and the elements allowing us to consider it as a good candidate for our metric. Then we present our implementation in order to fit the GPUs architecture. The last section presents our results.

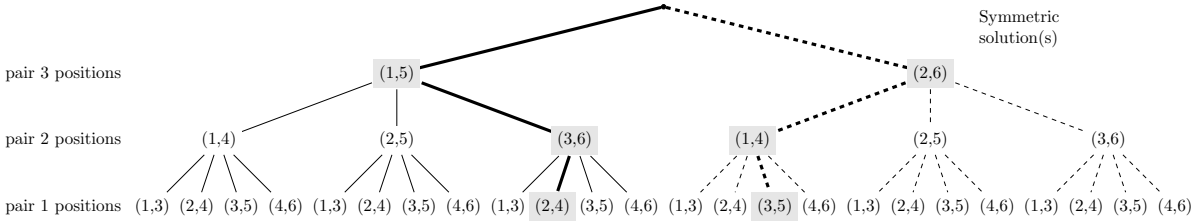


Figure 2.2: Search tree for $L(2,3)$

2.2.1 CSP

Combinatorial problems are NP-complete [GJ79] and can be described as satisfiability problems (SAT) using a polynomial transformation. They can be transformed into CSP formalism. A *Constraint Satisfaction Problem* (CSP), first introduced by Montanari [Mon74], is defined as a triple $< X, D, C >$ where:

$$\begin{cases} X = \{X_1, ..., X_n\}\text{: a finite set of variables} \\ D = \{D_1, ..., D_n\}\text{: their finite domains of values} \\ C = \{C_1, ..., C_p\}\text{: a finite set of constraints} \end{cases} \tag{2.2}$$

The goal in this formalism is to assign values in D to n-uple X respecting all the C p-uple constraints. This approach is a large field of research. [AC14] developed *local search* and compares GPU to CPU. This first work brings to light that GPU is a real contributor to the global computation speed. [CDPD$^+$14] proposes a solver using *propagator* on a GPU architecture to solve CSP problems. [JAO$^+$11] cares about GPU weak points, loading bandwidth and global memory latency.

Considering a basic approach, combinatorial problems formed into CSP can be represented as a tree search. Each level corresponds to a given variable, with values in its domain. Leaves of the tree correspond to a complete assignment (all variables are set). If it meets all the constraints this assignment is called an acceptor state. Depending on the constraints set, the satisfiability evaluation can be made either on complete or partial assignment.

2.2.2 Backtrack resolution

As presented above the Langford problem is known to be a highly irregular combinatorial problem. We first present here the general tree representation and the ways we regularize the computation for GPUs. Then we show how to parallelize the resolution over a multi-GPU cluster.

Langford's problem tree representation

As explained, CSP formalized problems can be transformed into tree evaluations. In order to solve $L(2,n)$, we consider a tree of height n: see example of $L(2,3)$ in figure 2.2.

- Every level of the tree corresponds to a cube color.

- Each node of the tree corresponds to the placement of a pair of cubes without worrying about the other colors. Color p is represented at depth $n-p+1$, where the first node corresponds to the first possible placement (positions 1 and $p+2$) and i^{th} node corresponds to the placement of the first cube of color p in position i, $i \in [1,\ 2n-1-p]$.

- Solutions are leaves generated without any placement conflict.

```
while not done do
 test pair            <- test
 if successful then
   if max depth then
     count solution
     higher pair
   else
     lower pair       <- remove
 else
   higher pair        <- add
```

Figure 2.3: Backtrack algorithm

```
for pair 1 positions
  assignment                  <- add
  for pair 2 positions
    assignment                <- add
    for ...
      for pair n positions
        assignment            <- add
        if final test ok then
          count solution
```

Figure 2.4: Regularized algorithm

	pair 1	pair 2	pair 3
1	0 0 0 1 0 1	0 0 1 0 0 1	0 1 0 0 0 1
2	0 0 1 0 1 0	0 1 0 0 1 0	1 0 0 0 1 0
3	0 1 0 1 0 0	1 0 0 1 0 0	
4	1 0 1 0 0 0		

Figure 2.5: Bitwise representation of pairs positions in $L(2,3)$

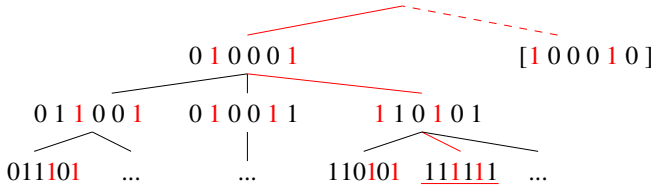


Figure 2.6: Bitwise representation of the Langford $L(2,3)$ placement tree

- As we consider the solution up to a symmetry, the left part is represented dashed and is in fact not traversed.

There are many ways to browse the tree and find the solutions: *backtracking*, *forward-checking*, *backjumping*, etc [Pro93]. We limit our study to the naive *backtrack* resolution and choose to evaluate the variables and their values in a static order; in a depth-first manner, the solutions are built incrementally and if a partial assignment can be aborted, the branch is cut. A solution is found each time a leaf is reached.

The recommendation for performance on GPU accelerators is to use non test-based programs. Due to its irregularity, the basic *backtracking* algorithm, presented on figure 2.3, is not supposed to suit the GPU architecture. Thus a vectorized version is given when evaluating the assignments at the leaves' level, with one of the two following ways: assignments can be prepared on each tree node or totally set on final leaves before testing the satisfiability of the built solution (figure 2.4).

Data representation

In order to count every Langford problem solution, we first identify all possible combinations for one color without worrying about the other ones. Each possible combination is coded within an integer, a bit to 1 corresponding to a cube presence, a 0 to its absence. This is what we called a *mask*. This way figure 2.5 presents the possible combinations to place the one, two and three weight cubes for the $L(2,3)$ Langford instance.

Furthermore the masks can be used to evaluate the partial placements of a chosen set of colors: all the 1 correspond to occupied positions; the assignment is consistent *iff* there are as many 1 as the number of cubes set for the assignment.

With the aim to find solutions, we just have to go all over the tree and *sum* one combination of each of the colors: a solution is found *iff* all the bits of the sum are set to 1.

Each route on the tree can be evaluated individually and independently; then it can be evaluated as a thread on the GPU. This way the problem is massively parallel and can be, indeed, computed on GPU. figure 2.6 represents the tree masks' representation.

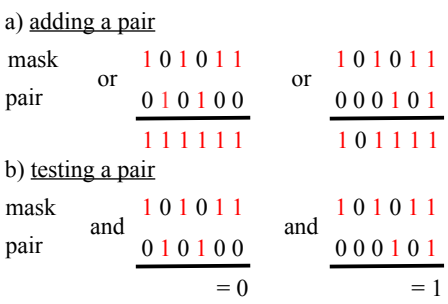


Figure 2.7: Testing and adding position

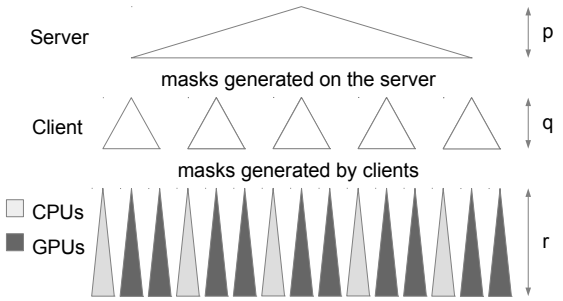


Figure 2.8: Server client distribution

Specific operations and algorithms

Three main operations are required in order to perform the tree search. The first one, used for both backtrack and regularized methods, aims to add a pair to a given assignment. The second one, allowing to check if a pair can be added to a given partial assignment, is only necessary for the original backtrack scheme. The last one is used for testing if a global assignment is an available solution: it is involved in the regularized version of the Miller algorithm.

Add a pair: Top of figure 2.7 presents the way to add a pair to a given assignment. With a *binary OR*, the new mask contains the combination of the original mask and of the added pair. This operation can be performed even if the position is not available for the pair (however the resulting mask is inconsistent).

Test a pair position: On the bottom part of the same figure, we test the positioning of a pair on a given mask. For this, it is necessary to perform a *binary AND* between the mask and the pair.

$= 0$: *success*, the pair can be placed here

$\neq 0$: *error*, try another position

Final validity test: The last operation is for *a posteriori* checking. For example the mask 101111, corresponding to a leaf of the tree, is inconsistent and should not be counted among the solutions. The final placement mask corresponds to a solution *iff* all the places are occupied, which can be tested as $\neg mask = 0$.

Using this data representation, we implemented both *backtrack* and *regularized* versions of the Miller algorithm, as presented in figure 2.3 and 2.4.
The next section presents the way we hybridize these two schemes in order to get an efficient parallel implementation of the Miller algorithm.

2.2.3 Hybrid parallel implementation

Section 2.2.2 presents the Miller algorithm and the tools needed for the resolution. The irregularity is present in the tree traversal and specific data representation has been chosen for efficient use of the memory. This section presents our methodology to implement Miller's method on a multi-GPU cluster.

Tasks generation: In order to parallelize the resolution we have to generate tasks. Considering the tree representation, we construct tasks by fixing the different values of a first set of variables [pairs] up to a given level. Choosing the development level allows to generate as many tasks as necessary. This leads to a *Finite number of Irregular and Independent Tasks* (*FIIT* applications [Kra99]).

Cluster parallelization: The generated tasks are independent and we spread them in a client-server manner: a server generates them and makes them available for clients. As we consider the cluster as a set of CPU-GPU(s) machines, the clients are these machines. At the machines level, the role of the CPU is, first, to generate work for the GPU(s): it has to generate sub-tasks, by continuing the tree development as if it were a second-level server, and the GPU(s) can be considered as second-level client(s).
The sub-tasks generation, at the CPU level, can be made in parallel by the CPU cores. Depending on the GPUs number and their computation power the sub-tasks generation rhythm may be adapted, to maintain a regular workload both for the CPU cores and GPU threads: some CPU cores, not involved in the sub-tasks generation, could be made available for sub-tasks computing.
This leads to the 3-level parallelism scheme presented in figure 2.8, where p, q and r respectively correspond to: (p) the server-level tasks generation depth, (q) the client-level sub-tasks generation one, (r) the remaining depth in the tree evaluation, *i.e.* the number of remaining variables to be set before reaching the leaves.

***Backtrack* and *regularized* methods hybridization:** The Backtrack version of the Miller algorithm suits CPU execution and allows to cut branches during the tree evaluation, reducing the search space and limiting the combinatorial explosion effects. A regularized version had to be developed, since GPUs execution requires synchronous execution of the threads, with as few branching divergence as possible; however this method imposes to browse the entire search space and is too time-consuming.
We propose to hybridize the two methods in order to take advantage of both of them for the multiGPU parallel execution: for tasks and sub-tasks generated at sever and client levels, the tree development by the CPU cores is made using the backtrack method, cutting branches as soon as possible [and generating only possible tasks]; when computing the sub-tasks generated at client-level, the CPU cores involved in the sub-tasks resolution use the backtrack method and the GPU threads the regularized one.

2.2.4 Experiments tuning

In order to take advantage of all the computing power of the GPU we have to refine the way we use them: this section presents the experimental study required to choose optimal settings. This tuning allowed us to prove our proposal on significant instances of the Langford problem.

Registers, blocks and grid: In order to use all GPUs capabilities, the first way was to fill the blocks and grid. To maximize occupancy (ratio between active warps and the total number of warps) NVIDIA suggests to use 1024 threads per block to improve GPU performances and proposes a CUDA occupancy calculator[1]. But, confirmed by the Volkov's results[Vol10], we experimented that better performances may be obtained using lower occupancy. Indeed, another critical criterion is the inner GPU registers occupation. The optimal number of registers (57 registers) is obtained by setting 9 pairs placed on the client for $L(2, 15)$, thus 6 pairs are remaining for GPU computation.

In order to tune the blocks and grid sizes, we performed tests on the ROMEO architecture. Figure 2.9 represents the time in relation with the number of blocks per grid and the number of threads per block. The most relevant result, observed as a local minimum on the 3D surface, is obtained near 64 or 96 threads per block; for the grid size, the limitation is relative to the GPU global memory size. It can be noted that we do not need shared memory because their are no data exchanges between threads. This allows us to use the total available memory for the L1 cache for each thread.

[1] `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

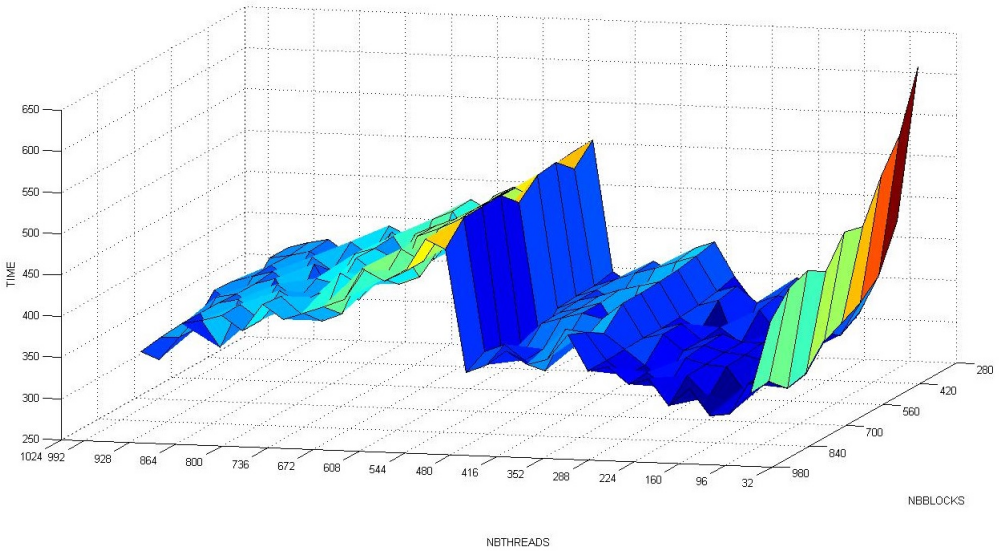


Figure 2.9: Time depending on grid and block size on $n = 15$

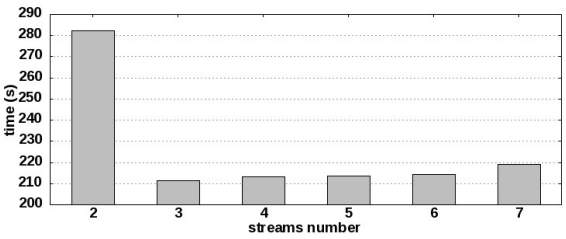


Figure 2.10: Computing time depending on streams number

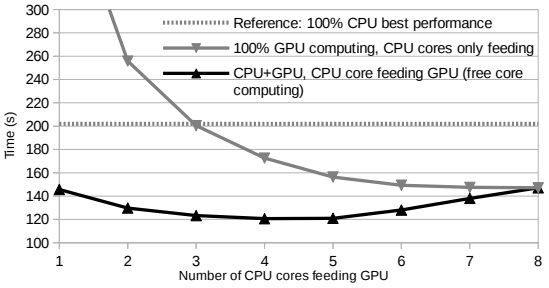


Figure 2.11: CPU cores optimal distribution for GPU feeding

Streams: A client has to prepare work for GPU. There are four main steps: generate the tasks, load them into the device memory, process the task on the GPU and then get the results.

CPU-GPU memory transfers cause huge time penalties (about 400 cycles latency for transfers between CPU memory and GPU *device memory*). At first, we had no overlapping between memory transfer and kernel computation because the tasks generation on CPU was too long compared to the kernel computation. To reduce the tasks generation time we used OpenMP in order to use the eight available CPU cores. Thus CPU computation was totally hidden by memory transfers and GPU kernel computation. We tried using up to 7 streams; as shown by figure 2.10, using only two simultaneous streams did not improve efficiency because the four steps did not overlap completely; the best performances were obtained with three streams; the slow increase in the next values is caused by synchronization overhead and CUDA streams management.

Setting up the server, client and GPU depths: We now have to set the depths of each actor, server (p), client (q) and GPU (r) (see figure 2.8).

First we set the $r = 5$ for large instances because of the GPU limitation in terms of registers by threads, exacerbated by the use of numerous $64bits$ integers. For $r \geq 6$, we get too many registers (64) and for $r \leq 4$ the GPU computation is too fast compared to the memory load overhead.

Clients are the buffers between the server and the GPUs: $q = n - p - r$. So we have conducted tests by varying the server depth, p. The best result is obtained for $p = 3$ and

n	CPU (8c)	GPU (4c) + CPU (4c)
15	2.5	1.5
16	21.2	14.3
17	200.3	120.5
18	1971.0	1178.2
19	22594.2	13960.8

(a) Regularized method (seconds)

n	CPU (8c)	GPU (4c) + CPU (4c)
17	29.8	7.3
18	290.0	73.6
19	3197.5	803.5
20	–	9436.9
21	–	118512.4

(b) Backtrack (seconds)

Table 2.2: Comparison between multi-core processors and GPUs for regularized and backtrack method

performance decreases quickly for higher values. This can be explained since more levels on the server generates smaller tasks; thus GPU use is not long enough to overlap memory exchanges.

CPU: Feed the GPUs and compute The first work of CPU cores is to prepare tasks for GPU so that we can generate overlapping between memory load and kernel computation. In this configuration using eight cores to generate GPU tasks under-uses CPU computation power. It is the reason why we propose to use some of the CPU cores to take part of the sub-problems treatment. Figure 2.11 represents computation time in relation with different task distributions between CPU and GPU. We experimentally demonstrated that only 4 or 5 CPU cores are enough to feed GPU, the other ones can be used to perform backtrack resolution in competition with GPUs.

2.2.5 Results

Regularized method results We now show the results obtained for our massively parallel scheme using the previous optimizations, comparing the computation times of successive instances of the Langford problem. These tests were performed on 20 nodes of the ROMEO supercomputer, hence 40 CPU/GPU machines.

The previous limit with Miller's algorithm was $L(2, 19)$, obtained in 1999 after 2.5 years of sequential effort and at the same time after 2 months with a distributed approach[Mil99]. Our computation scheme allowed us to obtain it in less than 4 hours (Table 2.2a), this being not only due to Moore law progress.
Note that the computation is 1.6 faster with CPU+GPU together than using 8 CPU cores. In addition, the GPUs compute $200,000\times$ more nodes of the search tree than the CPUs, with a faster time.

The computation time between two different consecutive instances being multiplied by 10 approximately, this could allow us to obtain $L(2, 20)$ in a reasonable time.

Backtracking on GPUs It appears at first sight that using backtracking on GPUs without any regularization is a bad idea due to threads synchronization issues. But in order to compare CPU and GPU computation power in the same conditions we decided to implement the original backtrack method on GPU (see figure 2.3) with only minor modifications. In these conditions we observe very efficient work of the NVIDIA scheduler, which perfectly handles threads desynchronization. Thus we use the same server-client distribution as in 2.2.3, each client generates masks for both CPU and GPU cores. The workload is then statically distributed on GPU and CPU cores. Executing the backtrack algorithm on a randomly chosen set of sub-problems allowed us to set the GPU/CPU distribution ratio experimentally to 80/20%.

The experiments were performed on 129 nodes of the ROMEO supercomputer, hence 258 CPU/GPU machines and one node for the server. Table 2.2b shows the results with this configuration. This method first allowed us to perform the computation of $L(2, 19)$ in less than 15 minutes, $15\times$ faster than with the regularized method; then, we pushed the limitations of the

Miller algorithm up to $L(2, 20)$ in less than 3 hours and even $L(2, 21)$ in about 33 hours[2].

This first benchmark application provides a perfect example for the behavior of many-core accelerators confronted to irregular application. Even if the application does not seemed to fit with the direct tree traversal, the results prove that if used in the right way performance can be achieve in this case.

2.3 Godfrey's algebraic method

The previous part presents the Miller algorithm for the Langford problem, this method cannot achieve bigger instances than the $L(2, 21)$. It allows us the target specific walls with memory and computation irregularities.
An algebraic representation of the Langford problem has been proposed by M. Godfrey in 2002. This example is perfect in our study to target computationally heavy context with irregular memory behavior. In this part we describe this algorithm and optimizations, and then our implementation on multiGPU clusters. As a side note, this implementation also allowed us to break a new world record on the computation time needed to solve the last instances, $L(2, 27)$ and $L(2, 28)$.

2.3.1 Method description

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:

$$\begin{aligned} F(X,3) =& (X_1X_3 + X_2X_4 + X_3X_5 + X_4X_6)\times \\ & (X_1X_4 + X_2X_5 + X_3X_6)\times \\ & (X_1X_5 + X_2X_6) \end{aligned} \tag{2.3}$$

In this approach each term represents a position of both cubes of a given color and a solution to the problem corresponds to a term developed as $(X_1X_2X_3X_4X_5X_6)$; thus the number of solutions is equal to the coefficient of this monomial in the development. More generally, the solutions to $L(2, n)$ can be deduced from $(X_1X_2X_3X_4X_5...X_{2n})$ terms in the development of $F(X, n)$.

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

$$\sum_{(x_1,...,x_{2n})\in\{-1,1\}^{2n}} G(X,n)_{(x_1,...,x_{2n})} = 2^{2n+1}L(2,n) \tag{2.4}$$

So

$$\sum_{(x_1,...,x_{2n})\in\{-1,1\}^{2n}} (\prod_{i=1}^{2n} x_i) \prod_{i=1}^{n} \sum_{k=1}^{2n-i-1} x_k x_{k+i+1} = 2^{2n+1}L(2,n) \tag{2.5}$$

That allows to get $L(2, n)$ from polynomial evaluations. The computational complexity of $L(2, n)$ is of $O(4^n \times n^2)$ and an efficient big integer arithmetic is necessary. This principle can be optimized by taking into account the symmetries of the problem and using the Gray code: these optimizations are described below.

2.3.2 Method optimizations

Some works focused on finding optimizations for this arithmetic method[Jai05]. Here we explain the symmetric and computation optimizations used in our algorithm.

[2]Even if this instance has no interest since it is known to have no solution

Evaluation parity:

As $[F(-X,n) = F(X,n)]$, G is not affected by a global sign change. In the same way the global sign does not change if we change the sign of each pair or impair variable.

Using these optimizations we can set the value of two variables and accordingly divide the computation time and result size by four.

Symmetry summing:

In this problem we have to count each solution up to a symmetry; thus for the first pair of cubes we can stop the computation at half of the available positions considering
$S'_1(x) = \sum_{k=1}^{n-1} x_k x_{k+2}$ instead of $S_1(x) = \sum_{k=1}^{2n-2} x_k x_{k+2}$. The result is divided by 2.

Sums order:

Each evaluation of $S_i(x) = \sum_{k=1}^{2n-i-1} x_k x_{k+i+1}$, before multiplying might be very important regarding to the computation time for this sum. Changing only one value of x_i at a time, we can recompute the sum using the previous one without global re-computation. Indeed, we order the evaluations of the outer sum using Gray code sequence. Then the computation time is considerably reduced.

Based on all these improvements and optimizations we can use the Godfrey method in order to solve huge instances of the Langford problem. The next section develops the main issues of our multiGPU architecture implementation.

2.3.3 Implementation details

In this part we present the specific adaptations required to implement the Godfrey method on a multiGPU architecture.

Optimized big integer arithmetic:

In each step of computation, the value of each S_i can reach $2n - i - 1$ in absolute value, and their product can reach $\frac{(2n-2)!}{(n-2)!}$. As we have to sum the S_i product on 2^{2n} values, in the worst case we have to store a value up to $2^{2n}\frac{(2n-2)!}{(n-2)!}$, which corresponds to 10^{61} for $n = 28$, with about 200 bits.

So we need few big integer arithmetic functions. After testing existing libraries like GMP for CPU or CUMP for GPU, we came to the conclusion that they implement a huge number of functionalities and are not really optimized for our specific problem implementation: product of "small" values and sum of "huge" values.

Finally, we developed a light CPU and GPU library adapted to our needs. In the summation, for example, as maintaining carries has an important time penalty, we have chosen to delay the spread of carries by using buffers: carries are accumulated and spread only when useful (for example when the buffer is full). Figure 2.12 represents this big integer handling.

This big integer library imposes many constraint on the accelerator memory. We conduct tests using simple and double precision for the basic unit. The carries propagation is also triggered when necessary only. This strategy implies random memory accesses to the words and irregular behavior between the threads.

Gray sequence in memory:

The Gray 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 missing terms are directly computed from the known ones using arithmetic

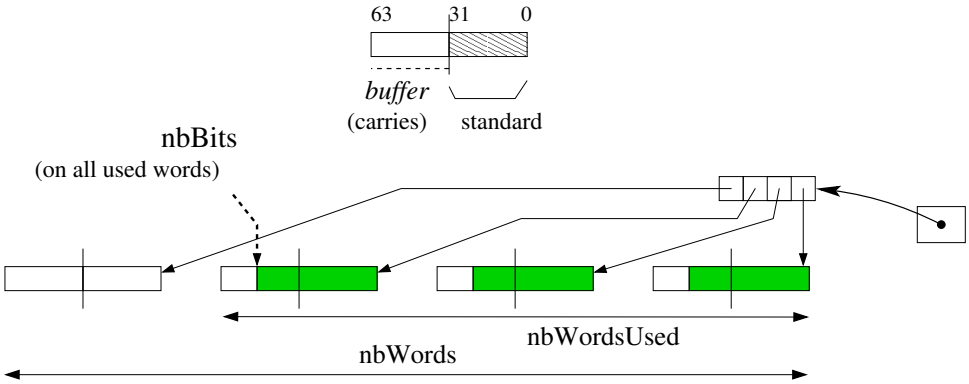


Figure 2.12: Big integer representation, 64 bits words

considerations. The size of the stored part of the Gray code sequence is chosen to be as large as possible to be contained in the processor's cache memory, the L1 cache for the GPUs threads: so the accesses are fastened and the computation of the Gray code is optimized. For an efficient use of the E5-2650 v2 ROMEO's CPUs, which disposes of 20 MB of level-3 cache, the CPU Gray code sequence is developed recursively up to depth 25. For the K20Xm ROMEO's GPUs, which dispose of 8 KB of constant memory, the sequence is developed up to depth 15. The rest of the memory is used for the computation itself.

Tasks generation and computation:

In order to perform the computation of the polynomial, two variables can be set among the $2n$ available. For the tasks generation we choose a number p of variables to generate 2^p tasks by developing the evaluation tree to depth p.

The tasks are spread over the cluster, either synchronously or asynchronously.

Synchronous computation: A first experiment was carried out with an MPI distribution of the tasks of the previous model. Each MPI process finds its tasks list based on its process id; then converting each task number into binary gives the task's initialization. The processes work independently; finally the root process ($id = 0$) gathers all the computed numbers of solutions and sums them.

Asynchronous computation: In this case the tasks can be computed independently. As with the synchronous computation, the tasks' initializations are retrieved from their number. Each machine can get a task, compute it, and then store its result; then when all the tasks have been computed, the partial sums are added together and the total result is provided.

2.3.4 Experimental settings

This part presents the experimental context and methodology, and the way the experiments were carried out. This study has similar goals as for the Miller's resolution experiments.

Methodology:

We present here the way the experimental settings were chosen. Firstly we define the tasks distribution, secondly we set the number of threads per GPU block; finally, we set the CPU/GPU distribution.

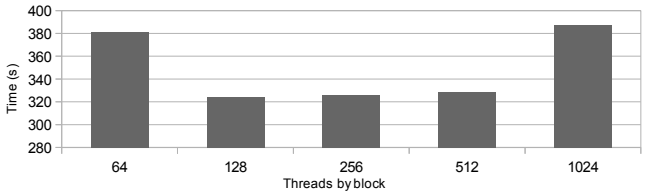
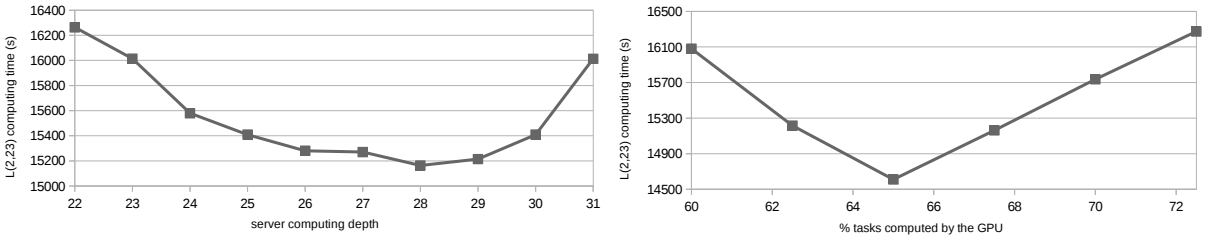


Figure 2.13: $L(2, 20)$, number of threads per block



(a) Influence on server generation depth

(b) Influence of tasks repartition

Figure 2.14: Influences of repartitions of depths and CPU-GPU tasks

Tasks distribution depth: This value being set it is important to get a high number of blocks to maintain sufficient GPU load. Thus we have to determine the best number of tasks for the distribution. As presented in part 2.3.3 the number p of bits determines 2^p tasks. On the one hand, too many tasks are a limitation for the GPU that cannot store all the tasks in its 6GB memory. On the other hand, not enough tasks means longer tasks and too few blocks to fill the GPU grid. figure 2.14a shows that for the $L(2, 23)$ instance the best task number is with generation depth 28.

Number of threads per block: In order to take advantage of the GPU computation power, we have to determine the threads/block distribution. Inspired by our experiments with Miller's algorithm we know that the best value may appear at lower occupancy. We perform tests on a given tasks set varying the threads/block number and grid size associated. Figure 2.13 presents the tests performed on the $n = 20$ problem: the best distribution is around 128 threads per block.

CPU vs GPU distribution:

The GPU and CPU computation algorithm will approximately be the same. In order to take advantage of all the computational power of both components we have to balance tasks between CPU and GPU. We performed tests by changing the CPU/GPU distribution based on simulations on a chosen set of tasks. Figure 2.14b shows that the best distribution is obtained when the GPU handles 65% of the tasks. This optimal load repartition directly results from the intrinsics computational power of each component; this repartition should be adapted if using a more powerful GPU like Tesla K40 or K80.

Computing context:

As presented in part **??**, we used the ROMEO supercomputer to perform our tests and computations. On this supercomputer SLURM[JG03] is used as a reservation and job queue manager. This software allows two reservation modes: a static one-job limited reservation or the opportunity to dynamically submit several jobs in a Best-Effort manner.

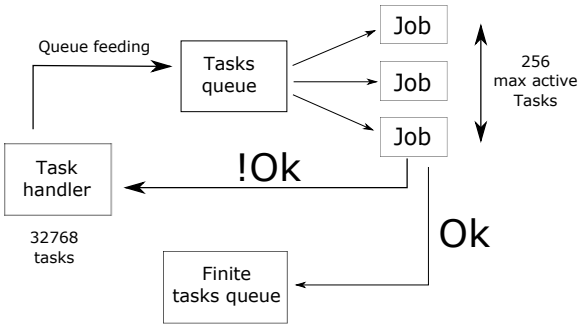


Figure 2.15: Best-effort distribution

Static distribution: In this case we used the synchronous distribution presented in 2.3.3. We submited a reservation with the number of MPI processes and the number of cores per process. This method is useful to get the results quickly if we can get at once a large amount of computation resources. It was used to perform the computation of small problems, and even for $L(2, 23)$ and $L(2, 24)$.
As an issue, it has to be noted that it is difficult to quickly obtain a very large reservation on such a shared cluster, since many projects are currently running.

Best effort: SLURM allows to submit tasks in the specific Best-Effort queue, which does not count in the user *fair-share*. In this queue, if a node is free and nobody is using it, the reservation is set for a job in the best effort queue for a minimum time reservation. If another user asks for a reservation and requests this node, the best effort job is killed (with, for example, a SIGTERM signal). This method, based on asynchronous computation, enables a maximal use of the computational resources without blocking for a long time the entire cluster.

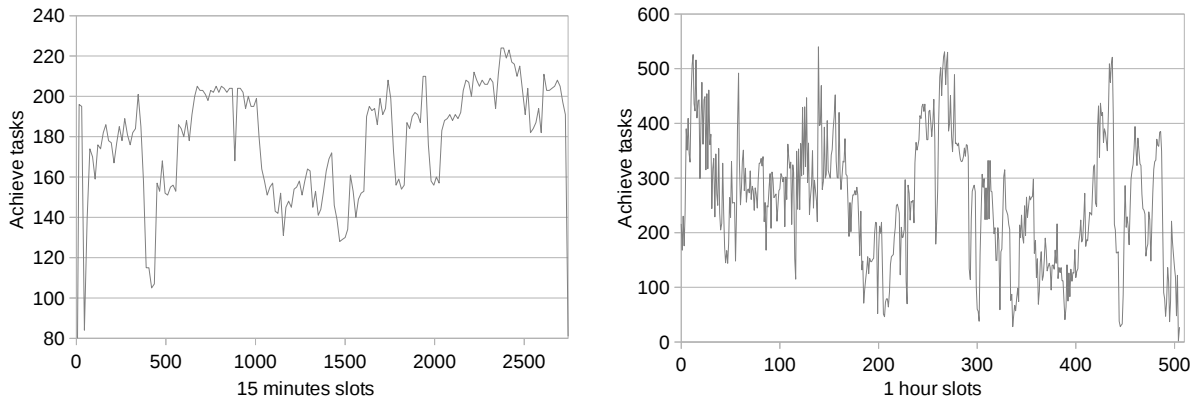
For $L(2, 27)$ and even more for $L(2, 28)$ the total time required is too important to use the whole machine off a challenge period, thus we chose to compute in a Best-Effort manner. In order to fit with this submission method we chose a reasonable time-per-task, sufficient to optimize the treatments with low loading overhead, but not too long so that killed tasks are not too penalizing for the global computation time. We empirically chose to run 15-20 minute tasks and thus we considered $p = 15$ for $n = 27$ and $p = 17$ for $n = 28$.

The best effort based algorithm is presented on figure 2.15. The task handler maintains a maximum of 256 tasks in the queue; in addition the entire process is designed to be fault-tolerant since killed tasks have to be launched again. When finished, the tasks generate an output containing the number of solutions and computation time, that is stored as a file or database entry. At the end the outputs of the different tasks are merged and the global result can be provided.

2.3.5 Results

After these optimizations and implementation tuning steps, we conducted tests on the ROMEO supercomputer using best-effort queue to solve $L(2, 27)$ and $L(2, 28)$. We started the experiment after an update of the supercomputer, that implied a cluster shutdown. Then the machine was restarted and was about 50% idle for the duration of our challenge. The computation lasted less than 2 days for $L(2, 27)$ and 23 days for $L(2, 28)$. The following describes performances considerations.

Computing effort - For $L(2, 27)$, the effective computation time of the 32,768 tasks was about 30 million seconds (345.4 days), and 165,000" elapsed time (1.9 days); the average time of the tasks was 911", with a standard deviation of 20%. For the $L(2, 28)$ 131,072 tasks the total computation time was about 1365 days (117 million seconds), as 23 day elapsed time; the tasks lasted 1321" on average with a 12% standard deviation.



(a) $L(2, 27)$ tasks grouped by 15" slots

(b) $L(2, 28)$ tasks grouped by 1 hour slots

Figure 2.16: Task repartition for $L(2, 27)$ and $L(2, 28)$

Best-effort overhead - With $L(2, 27)$ we used a specific database to maintain information concerning the tasks: 617 tasks were aborted [by regular user jobs] before finishing (1.9%), with an average computing time of 766" (43% of the maximum requested time for a task). This consumed 472873", which overhead represents 1.6% of the effective computing effort.

Cluster occupancy - Figure 2.16a presents the tasks resolution over the two computation days for $L(2, 27)$. The experiment elapse time was 164700" (1.9 days). Compared to the effective computation time, we used an average of 181.2 machines (CPU-GPU couples): this represents 69.7% of the entire cluster.

Figure 2.16b presents the tasks resolution flow during the 23 days computation for $L(2, 28)$. We used about 99 machines, which represents 38% of the 230 available nodes.

For $L(2, 27)$, these results confirm that the computation took great advantage of the low occupancy of the cluster during the experiment. This allowed us to obtain a weak best-effort overhead, and an important cluster occupancy. Unfortunately for $L(2, 28)$ on such a long period we got a lower part of the supercomputer dedicated to our computational project. Thus we are confident in good perspectives for the $L(2, 31)$ instance if computed on an even larger cluster or several distributed clusters.

2.4 Conclusion

This first benchmark presents two methods to solve the Langford pairing problem on multi-GPU clusters. Those methods are presented as the Miller's and Godfrey's algorithms.

2.4.1 Miller's method: irregular memory + computation

In this first implementation we showed that the accelerator can target irregularity in memory and computation. The tree traversal, even with branch cutting, gives us excellent results on many-core architecture. They GPUs handle 80% of the computation effort. Also as any combinatorial problem can be represented as a CSP, the Miller algorithm can be seen as general resolution scheme based on the backtrack tree browsing. A three-level tasks generation allows to fit the multiGPU architecture. MPI or Best-Effort are used to spread tasks over the cluster, OpenMP for the CPU cores distribution and then CUDA to take advantage of the GPU computation power. We were able to compute $L(2, 20)$ with this regularized method and to get an even better time with the basic backtrack. This proves the proposed approach and also exhibits that the GPU scheduler is very efficient at managing highly divergent threads.

2.4.2 Godfrey's method: irregular memory + heavy computation

We also presented the Godfrey's method which target irregular memory accesses but also heavy computational behavior. The results shows that the many-core version completely exceed the classical processor best performances. The GPU handle 65% of the computation effort. In addition we beat the Langford limits using multi-GPUs. In order to use the supercomputer ROMEO, which is shared by a large scientific community, we have implemented a distribution that does not affect the machine load, using a best-effort queue. The computation is fault-tolerant and totally asynchronous. The utilization of this technology also show the reliability of nowadays accelerators technologies. Indeed, they can be use at demand on specific tasks without specific warm-up.

Langford problem results: This work also enabled us to compute $L(2, 27)$ and $L(2, 28)$ in respectively less than 2 days and 23 days on the University of Reims ROMEO supercomputer. This is the fastest time on those problems nowadays. The total number of solutions is:

L(2,27) = 111,683,611,098,764,903,232
L(2,28) = 1,607,383,260,609,382,393,152

This first benchmark implementation is clearly in favor of hybrid architecture. The many-core architecture shows a perfect handling of irregularity and heavy computations.

Bibliography

[ABL15] Ali Assarpour, Amotz Barnoy, and Ou Liu. Counting the number of langford skolem pairings. 2015.

[AC14] Alejandro Arbelaez and Philippe Codognet. A gpu implementation of parallel constraint-based local search. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 648–655. IEEE, 2014.

[CDPD+14] Federico Campeotto, Alessandro Dal Palu, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. Exploring the use of gpus in constraint solving. In *Practical Aspects of Declarative Languages*, pages 152–167. Springer, 2014.

[Gar56] Martin Gardner. *Mathematics, magic and mystery*. Dover publication, 1956.

[GJ79] M. R. Garey and D. S. Johnson. *Computer and Intractability*. Freeman, San Francisco, CA, USA, 1979.

[Jai05] Christophe Jaillet. *In french: Résolution parallèle des problèmes combinatoires*. Phd, Université de Reims Champagne-Ardenne, France, December 2005.

[JAO+11] John Jenkins, Isha Arkatkar, John D Owens, Alok Choudhary, and Nagiza F Samatova. Lessons learned from exploring the backtracking paradigm on the gpu. In *Euro-Par 2011 Parallel Processing*, pages 425–437. Springer, 2011.

[JG03] Morris Jette and Mark Grondona. *SLURM : Simple Linux Utility for Resource Management*. U.S. Departement of Energy, June 23, 2003.

[JK04] Christophe Jaillet and Michaël Krajecki. Solving the langford problem in parallel. In *International Symposium on Parallel and Distributed Computing*, pages 83–90, Cork, Ireland, July 2004. IEEE Computer Society.

[Kra99] Michaël Krajecki. An object oriented environment to manage the parallelism of the fiit applications. In *Parallel Computing Technologies*, pages 229–235. Springer, 1999.

[Lar09] J Larsen. Counting the number of skolem sequences using inclusion exclusion. 2009.

[Mil99] J.E. Miller. Langford's problem: http://dialectrix.com/langford.html, 1999.

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

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[Sim83] James 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, 2000.

[Vol10] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[Wal01] T. Walsh. Permutation problems and channelling constraints. Technical Report APES-26-2001, APES Research Group, January 2001.