Prediction of Parallel Speed-ups for Las Vegas Algorithms

Charlotte Truchet
LINA, UMR 6241 / University of Nantes
Charlotte.Truchet@univ-nantes.fr

Florian Richoux
LINA, UMR 6241 / University of Nantes
Florian.Richoux@univ-nantes.fr

Philippe Codognet
JFLI - CNRS / UPMC / University of Tokyo
codognet@is.s.u-tokyo.ac.jp

***Abstract*—We propose a probabilistic model for the parallel execution of *Las Vegas algorithms*, i.e. randomized algorithms whose runtime might vary from one execution to another, even with the same input. This model aims at predicting the parallel performances (i.e. speedups) by analyzing the runtime distribution of the sequential runs of the algorithm. Then, we study in practice the case of a particular Las Vegas algorithm for combinatorial optimization on three classical problems, and compare the model with an actual parallel implementation up to 256 cores. We show that the prediction can be accurate, matching the actual speedups very well up to 100 parallel cores and then with a deviation of about 20% up to 256 cores.**

I. Introduction

We consider in this paper *Las Vegas algorithms*, a generalization of Monte-Carlo algorithms introduced a few decades ago by [1], *i.e.,* randomized algorithms whose runtime might vary from one execution to another, even with the same input. An important class of Las Vegas algorithms is the family of *Local Search methods* and *metaheuristics* such as Simulated Annealing, Genetic Algorithms, Tabu Search, Swarm Optimization, Ant-Colony optimization, which have been applied to different sets of problems such as resource allocation, scheduling, packing, layout design, frequency allocation, etc. They have been used in Combinatorial Optimization for finding optimal or near-optimal solutions for several decades [2], stemming from the pioneering work of Lin on the Traveling Salesman Problem [3]. These methods are now widely used in combinatorial optimization to solve real-life problems when the search space is too large to be explored by complete search algorithm, such as Mixed Integer Programming or Constraint Solving, c.f. [4].

In the last years, several proposals for implementing local search algorithms on parallel computers have been proposed, the most popular being to run several competing instances of the algorithm on different cores with different initial conditions or parameters, and let the fastest process win over others. We thus have an algorithm with the minimal execution time among the launched processes. This leads to so-called independent multi-walk algorithms in the local search community [5] and portfolio algorithms in the SAT community (satisfiability of Boolean formula) [6]. This parallelization scheme can of course be generalized to any Las Vegas algorithm.

The goal of this paper is to study the parallel performance of Las Vegas algorithms under this independent multi-walk scheme, and to predict the performance of parallel execution thanks to a probabilistic model based on the sequential runtime distribution of the algorithm. It is indeed important to know how a given Las Vegas algorithm (or, more precisely, a couple formed by the algorithm and the problem instance) would scale on massively parallel hardware. This would make it possible to estimate the maximum number of cores until which parallelization is efficient and thus the actual parallel computing power needed to solve a problem. As supercomputers or systems such as Google Cloud and Amazon EC2 can be rented by core-hour with a limit on the maximum number of cores to be used, this is an important information to have before actually deciding to use a given platform. Moreover, We confront the predictions computed by our probabilistic model with actual speedups obtained for a parallel implementation of a local search algorithm and show that the prediction can be accurate, matching the actual speedup very well up to 100 parallel cores and then with a deviation limited to about 20% up to 256 cores.

The paper is organized as follows. Section 2 is devoted to present the definition of Las Vegas algorithms, their parallel multi-walk execution scheme, and the main idea for predicting the parallel speedups. Section III defines the probabilistic model of Las Vegas algorithms and their parallel execution scheme. Section IV presents the example of local search algorithms for combinatorial optimization, while Section V introduced the benchmark problems and their sequential and parallel performances. Section VI then applies the general probabilistic model to the benchmark problems in order to predict their parallel speed-ups, which are compared to experimental parallel speed-ups in Section VII. A short conclusion and future work will end the paper.

II. Multi-Walk Las Vegas Algorithms

We borrow the following definition from [7], Chapter 4.

Definition 1 (Las Vegas Algorithm). *An algorithm A for a problem class Π is a (generalized) Las Vegas algorithm if and only if it has the following properties:*

1) *If for a given problem instance $\pi \in \Pi$, algorithm A terminates returning a solution s, s is guaranteed to be a correct solution of π.*

2) *For any given instance $\pi \in \Pi$, the runtime of A applied to π is a random variable.*

This definition includes algorithms which are not guaranteed to return a solution. However in practice, we will only consider terminating Las Vegas algorithms, such as local search algorithms which always terminate if run for an unbounded time.

A. Multi-walk Parallel Extension

Parallel implementation of local search metaheuristics [4], [8] has been studied since the early 1990s, when parallel machines started to become widely available [9], [5]. With the increasing availability of PC clusters in the early 2000s, this domain became active again [10], [11]. Apart from domain-decomposition methods and population-based method (such as genetic algorithms), [5] distinguishes between single-walk and multi-walk methods for Local Search. Single-walk methods consist in using parallelism inside a single search process, *e.g.,* for parallelizing the exploration of the neighborhood (see for instance [12] for such a method making use of GPUs for the parallel phase). Multi-walk methods (parallel execution of multi-start methods) consist in developing concurrent explorations of the search space, either independently or cooperatively with some communication between concurrent processes. Sophisticated cooperative strategies for multi-walk methods can be devised by using solution pools [13], but require shared-memory or emulation of central memory in distributed clusters, thus impacting on performance. A key point is that a multi-walk scheme is easier to implement on parallel computers without shared memory and can lead, in theory at least, to linear speedups [5]. However this is only true under certain assumptions and we will see that we need to develop a more realistic model in order to cope with the performance actually observed in parallel executions.

Let us now formally define a parallel multi-walk Las Vegas algorithm.

Definition 2 (Multi-walk Las Vegas Algorithm)**.** *An algorithm A' for a problem class Π is a (parallel) multi-walk Las Vegas algorithm if and only if it has the following properties:*

1) *It consists of n instances of a sequential Las Vegas algorithm A for Π, say $A_1, ..., A_n$.*
2) *If, for a given problem instance $\pi \in \Pi$, there exists at least one $i \in [1, n]$ such that A_i terminates, then let $A_m, m \in [1, n]$, be the instance of A terminating with the minimal runtime and let s be the solution returned by A_m. Then algorithm A' terminates in the same time as A_m and returns solution s.*
3) *If, for a given problem instance $\pi \in \Pi$, all $A_i, i \in [1, n]$, do not terminate then A' does not terminate.*

B. How to Estimate Parallel Speedup ?

The multi-walk parallel scheme is rather simple, yet it provides an interesting test-case to study how Las Vegas algorithms can scale-up in parallel. Indeed runtime will vary among the processes launched in parallel and the overall runtime will be that of the instance with minimal execution time (*i.e.,* "long" runs are killed by "shorter" ones). The question is thus to quantify the relative notion of short and long runs and their probability distribution. This might gives us a key to quantify the expected parallel speed-up. Obviously, this can be deduced from the sequential behavior of the algorithm, and more precisely from the proportion of long and short runs in the sequential runtime distribution.

In the following, we propose a probabilistic model to quantify the expected speed-up of multi-walk Las Vegas algorithms. This makes it possible to give a general formula for the speed-up, depending on the sequential behavior of the algorithm. Our model is related to *order statistics*, which is the statistics of sorted random draws, a rather new domain of statistics [14]. Indeed, explicit formulas have been given for several well-known distributions. Relying on an approximation of the sequential distribution, we compute the average speed-up for the multi-walk extension. Experiments show that the prediction is quite good and opens the way for defining more accurate models and apply them to larger classes of algorithms.

Previous works [5] studied the case of a particular distribution for the sequential algorithm: the exponential distribution. This case is ideal and the best possible, as it yields a linear speed-up. Our model makes it possible to approximate Las Vegas algorithms by other types of distribution, such as a shifted exponential distribution or a lognormal distribution. In the last two cases the speed-up is no longer linear, but admits a finite limit when the number of processors tends toward infinity. We will see that it fits experimental data for some problems.

III. Probabilistic Model

Local Search algorithms are stochastic processes. They include several random components: choice of an initial configuration, choice of a move among several candidates, plateau mechanism, random restart, etc. In the following, we will consider the *computation time* of an algorithm (whatever it is) as a random variable, and use elements of probability theory to study its multi-walk parallel version. Notice that the computation time is not necessarily the cpu-time; it can also be the number of iterations performed during the execution of the algorithm.

A. Min Distribution

Consider a given algorithm on a given problem of a given size, say, the Magic-Square 10×10. Depending on the result of some random components inside the algorithm, it may find a solution after 0 iterations, 10 iterations, or 10^6

iterations. The number of iterations of the algorithm is thus a discrete random variable, let's call it Y, with values in $\mathbb{N}$. Y can be studied through its cumulative distribution, which is by definition the function $\mathcal{F}_Y$ s.t. $\mathcal{F}_Y(x) = \mathbb{P}[Y \leq x]$, or by its distribution, which is by definition the derivative of $\mathcal{F}_Y$: $f_Y = \mathcal{F}'_Y$.

It is often more convenient to consider distributions with values in $\mathbb{R}$ because it makes calculations easier. For the same reason, although f_Y is defined in $\mathbb{N}$, we will use its natural extension to $\mathbb{R}$. The expectation of the computation is then defined as $\mathbb{E}[Y] = \int_0^\infty t f_Y(t) dt$

Assume that the base algorithm is concurrently run in parallel on n cores. In other words, over each core the running process is a fork of the algorithm. The first process that finds a solution then kills all others and the algorithm terminates. The i-th process corresponds to a draw of a random variable X_i, following distribution f_Y. The variables X_i are thus independently and identically distributed (i.i.d.). The computation time of the whole parallel process is also a random variable, let's call it $Z^{(n)}$, with a distribution $f_{Z^{(n)}}$ that depends on both n and f_Y. Since all the X_i are i.i.d., the cumulative distribution $\mathcal{F}_{Z^{(n)}}$ can be computed as follows:

$$\begin{aligned}
\mathcal{F}_{Z^{(n)}} &= \mathbb{P}[Z^{(n)} \leq x] \\
&= \mathbb{P}[\exists i \in \{1...n\}, X_i \leq x] \\
&= 1 - \mathbb{P}[\forall i \in \{1...n\}, X_i > x] \\
&= 1 - \prod_{i=1}^{n} \mathbb{P}[X_i > x] \\
&= 1 - (1 - \mathcal{F}_Y(x))^n
\end{aligned}$$

which leads to:

$$\begin{aligned}
f_{Z^{(n)}} &= (1 - (1 - \mathcal{F}_Y)^n)' \\
&= n f_Y (1 - \mathcal{F}_Y)^{n-1}
\end{aligned}$$

Thus, knowing the distribution for the base algorithm Y, one can calculate the distribution for $Z^{(n)}$. In the general case, the formula shows that the parallel algorithm favors short runs, by killing the slower processes. Thus, we can expect that the distribution of $Z^{(n)}$ moves toward the origin, and is more peaked. As an example, Figure 1 shows this phenomenon when the base algorithm admits a Gaussian distribution.

B. *Expectation and Speed-up*

The model described above gives the probability distribution of a parallelized version of any random algorithm. We can now calculate the expectation for the parallel process with the following relation:

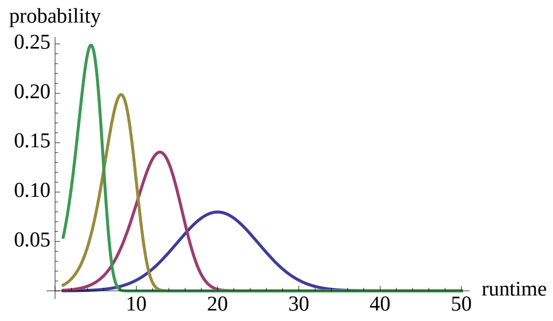

Figure 1. Distribution of $Z^{(n)}$, in the case where Y admits a Gaussian distribution (cut on $\mathbb{R}^-$ and renormalized). The blue curve is Y. The distributions of $Z^{(n)}$ are in pink for $n = 10$, in yellow for $n = 100$ and in green for $n = 1000$.

$$\begin{aligned}
\mathbb{E}[Z^{(n)}] &= \int_0^\infty t f_{Z^{(n)}}(t) dt \\
&= n \int_0^\infty t f_Y(t) (1 - \mathcal{F}_Y(t))^{n-1} dt
\end{aligned}$$

Unfortunately, this does not lead to a general formula for $\mathbb{E}[Z^{(n)}]$. In the following, we will study it for different specific distributions.

To measure the gain obtained by parallelizing the algorithm on n core, we will study the speed-up $\mathcal{G}_n$ defined as:

$$\mathcal{G}_n = \mathbb{E}[Y] / \mathbb{E}[Z^{(n)}]$$

Again, no general formula can be computed and the expression of the speed-up depends on the distribution of Y.

However, it is worth noting that our computation of the speed-up is related to order statistics, see [14] for a detailed presentation. Order statistics are the statistics of sorted random draws. For instance, the first order statistics of a distribution is its minimal value, and the k^{th} order statistic is its k^{th}-smallest value. For predicting the speed-up of a multi-walk Las Vegas algorithm on n cores, we are indeed interested in computing the expectation of the distribution of the minimum among n draws. As the above formula suggests, this may lead to heavy calculations, but recent studies such as [15] give explicit formulas for this quantity for several classical probability distributions.

C. *Case of an Exponential Distribution*

Assume that Y has a shifted exponential distribution, as it has been suggested by [16], [17].

$$\begin{aligned}
f_Y(t) &= \begin{cases} 0 & \text{if } t \leq x_0 \\ \lambda e^{-\lambda(t-x_0)} & \text{if } t > x_0 \end{cases} \\
\mathcal{F}_Y(t) &= \begin{cases} 0 & \text{if } t \leq x_0 \\ 1 - e^{-\lambda(t-x_0)} & \text{if } t > x_0 \end{cases} \\
\mathbb{E}[Y] &= x_0 + 1/\lambda
\end{aligned}$$

Then the above formula can be symbolically computed by hand:

$$f_{Z^{(n)}}(t) = \begin{cases} 0 & \text{if } t \leq x_0 \\ n\lambda e^{-n\lambda(t-x_0)} & \text{if } t > x_0 \end{cases}$$

$$\mathcal{F}_{Z^{(n)}}(t) = \begin{cases} 0 & \text{if } t \leq x_0 \\ 1 - e^{-n\lambda(t-x_0)} & \text{if } t > x_0 \end{cases}$$

The intuitive observation of section III-A is easily seen on the expression of the parallel distribution, which has an initial value multiplied by n but an exponential factor decreasing n-times faster, as shown on the curves of Figure 2.

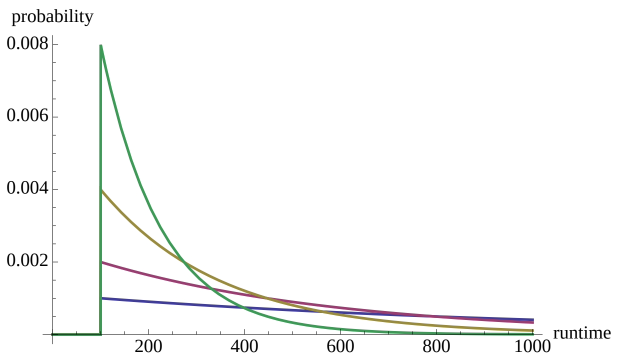

Figure 2. For an exponential distribution, here in blue with $x_0 = 100$ and $\lambda = 1/1000$, simulations of the distribution of $Z^{(n)}$for $n = 2$ (pink), $n = 4$ (yellow) and $n = 8$ (green).

And in this case, one can symbolically compute both the expectation and speed-up for $Z^{(n)}$:

$$\begin{aligned} \mathbb{E}[Z^{(n)}] &= n\lambda \int_{x_0}^{\infty} te^{-n\lambda(t-x_0)} dt \\ &= x_0 + \frac{1}{n\lambda} \\ \mathcal{G}_n &= \frac{x_0 + \frac{1}{\lambda}}{x_0 + \frac{1}{n\lambda}} \end{aligned}$$

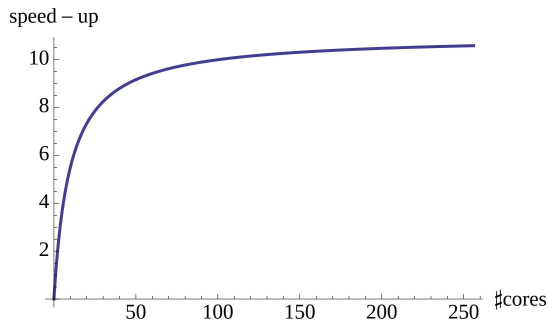

Figure 3. Predicted speed-up in case of an exponential distribution, with $x_0 = 100$ and $\lambda = 1/1000$, w.r.t. the number of cores.

Figure 3 shows the evolution of the speed-up when the number of cores increases. With such a rather simple formula for the speed-up, it is worth studying what happens when the number of cores n tends to infinity. Depending on the chosen algorithm, x_0 may be null or not. If $x_0 = 0$, then the expectation tends to 0 and the speed-up is equal to n. This case has already been studied by [5]. For $x_0 > 0$, the speed-up admits a finite limit which is $\frac{x_0+\frac{1}{\lambda}}{x_0} = 1+\frac{1}{x_0\lambda}$. Yet, this limit may be reached slowly, and depends on the values of x_0 and λ: obviously, the closest x_0 is to zero and the higher it will be. Another interesting value is the coefficient of the tangent at the origin, which approximates the speed-up for a small number of cores. In case of an exponential, it is $(x_0 * \lambda + 1)$. The higher x_0 and λ, the bigger is the speed-up at the beginning. In the following, we will see that, depending on the combinations of x_0 and λ, different behaviors can be observed.

D. *Case of a Lognormal Distribution*

Other distributions can be considered, depending on the behavior of the base algorithm. We will study the case of a lognormal distribution, which is the log of a Gaussian distribution, because it will be shown in Section VI-B that it fits the data of one experiment. It has two parameters, the mean μ and the standard deviation σ. In the same way as the shifted exponential, we shift the distribution so that it starts at a given parameter x_0. Formally, a (shifted) lognormal distribution is defined as:

$$f_Y(t) = \begin{cases} 0 & \text{if } t < x_0 \\ \frac{e^{-\frac{(-\mu+log(t-x_0))^2}{2\sigma^2}}}{\sqrt{2\pi}(t-x_0)\sigma} & \text{if } t > x_0 \end{cases}$$

$$\mathcal{F}_Y(t) = \begin{cases} 0 & \text{if } t < x_0 \\ \frac{1}{2}\text{erfc}(\frac{\mu-log(t-x_0)}{\sqrt{2}\sigma}) & \text{if } t > x_0 \end{cases}$$

where erfc is the complementary error function defined by $\text{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^{\infty} e^{-t^2} dt$. The mean, variance and median are known and equal to $e^{\mu+\frac{\sigma^2}{2}}$, $e^{2\mu+\sigma^2}(e^{\sigma^2} - 1)$ and e^{μ} respectively.

Figure 4 depicts lognormal distributions of $Z^{(n)}$, for several n. The computations for the distribution of $Z^{(n)}$, its expectation and the theoretical speed-up are given by quite complicated formulas. But [15] gives an explicit formula for all the moments of lognormal order statistics with only a numerical integration step, from which we can derive a computation of the speed-up (since the expectation of $Z^{(n)}$is the first order moment for the first order statistics). This allows us to draw the general shape of the speed-up, an example being given on Figure 5. Due to the numerical integration step, which requires numerical values for the number of cores n, we restrict the computation to integer values of n.

IV. Application to Local Search

Since about a decade, the interest for the family of Local Search methods and Metaheuristics for solving large

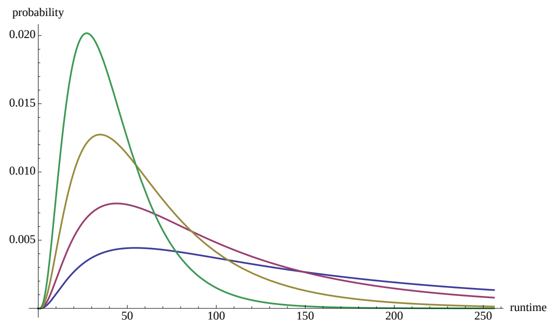

Figure 4. For a lognormal distribution (in blue), with $x_0 = 0$, $\mu = 5$ and $\sigma = 1$, simulations of the distribution of $Z^{(n)}$for $n = 2$ (pink), $n = 4$ (yellow) and $n = 8$ (green).

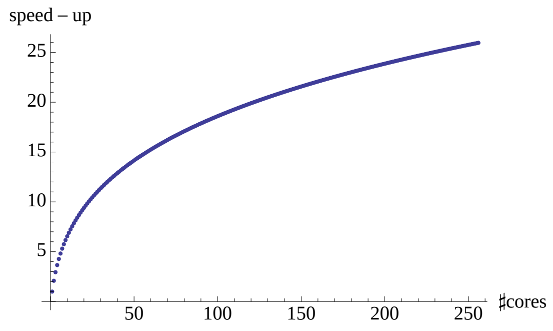

Figure 5. Predicted speed-up in case of a lognormal distribution, with $x_0 = 0$, $\mu = 5$ and $\sigma = 1$, w.r.t. the number of cores.

combinatorial problems has been growing and has attracted much attention from both the Operations Research and the Artificial Intelligence communities for solving real-life problems [8], [4]. Efficient general-purpose systems for Local Search now exist, see for instance [18].

Local Search starts from a random configuration and tries to improve this configuration, little by little, through small changes in the values of the problem variables. Hence the term "local search" as, at each time step, only new configurations that are "neighbors" of the current configuration are explored. The definition of what constitutes a neighborhood will of course be problem-dependent, but basically it consists in changing the value of a few variables only (usually one or two). The advantage of Local Search methods is that they will usually quickly converge towards a solution (if the optimality criterion and the notion of neighborhood are defined correctly...) and not exhaustively explore the entire search space.

Applying Local Search to Constraint Satisfaction Problems (CSP) has also been attracting some interest since about a decade [19], [18], as it can tackle CSPs instances far beyond the reach of classical propagation-based constraint solvers. A generic, domain-independent constraint-based local search method, named Adaptive Search, has been proposed by [19], [20]. This meta-heuristic takes advantage of the structure of the problem in terms of constraints and variables and can guide the search more precisely than a single global cost function to optimize, such as for instance the number of violated constraints. The algorithm also uses a short-term adaptive memory in the spirit of Tabu Search in order to prevent stagnation in local minima and loops.

An implementation of Adaptive Search (AS) has been developed in C language as a framework library and is available as a freeware at the URL:
http://cri-dist.univ-paris1.fr/diaz/adaptive/

We used this reference implementation for our experiments. The Adaptive Search method can be applied to a large class of constraints (*e.g.,* linear and non-linear arithmetic constraints, symbolic constraints, etc) and naturally copes with over-constrained problems. The input of the method is a Constraint Satisfaction Problem (CSP for short), which is defined as a triple (X;D;C), where X is a set of variables, D is a set of domains, *i.e.,* finite sets of possible values (one domain for each variable), and C a set of constraints restricting the values that the variables can simultaneously take.

Also note that we are tackling constraint *satisfaction* problems as optimization problems, that is, we want to minimize the global error (representing the violation of constraints) to value zero, therefore finding a solution means that we actually reach the bound (zero) of the objective function to minimize.

The core ideas of adaptive search can be summarized as follows:

- to consider for each constraint a heuristic function that is able to compute an approximated degree of satisfaction of the goals (the current *error* on the constraint);
- to aggregate constraints on each variable and project the error on variables thus trying to repair the *worst* variable with the most promising value;
- to keep a short-term memory of bad configurations to avoid looping (*i.e.,* some sort of *tabu list*) together with a reset mechanism.

V. Benchmark Problems and Experimental Results

We detail here the performance and speed-ups obtained with both sequential and parallel multi-walk Adaptive Search implementations. We have chosen to test this method on a hard combinatorial problem abstracted from radar and sonar applications (Costas Array) and two problems from the CSPLib benchmark library[1]:

- All-Interval Series (prob007 in CSPLib)
- The Magic-Square problem (prob019 in CSPLib).

A Costas array is an $N \times N$ grid containing N marks such that there is exactly one mark per row and per column and the $N(N-1)/2$ vectors joining the marks are all different. It is convenient to see the Costas Array Problem (CAP) as a permutation problem by considering an array of N variables

[1]http://www.csplib.org

$(V_1, \ldots, V_N)$ which forms a permutation of $\{1, 2, \ldots, N\}$. We refer the reader to [21] for a complete survey on the COSTAS ARRAY Problem and to [22] for the modeling and solving by local search.

These benchmarks could involve very large combinatorial search spaces, *e.g.,* the 200×200 MAGIC-SQUARE problem requires 40,000 variables whose domains range over 40,000 values. Indeed the search space in the Adaptive Search model (using permutations) is $40,000!$, *i.e.,* more than 10^{166713} configurations. Classical propagation-based constraint solvers cannot solve this problem for instances higher than 20x20.

A. *Sequential Results*

We run our benchmarks in a sequential manner in order to have about 650 runtimes for each. Sequential experiments, as well as parallel experiments, have been done on the *Griffon* cluster of the Grid'5000 platform [23], the French national grid for research, which contains 8,596 cores deployed on 11 sites distributed in France. The following Table I shows the minimum, mean, median, maximum and standard deviation among the runtimes (in seconds) and the number of iterations for our three benchmarks.

Seconds	MS 200	AI 700	Costas 21
Min	5.5	23.3	6.6
Mean	382.0	1,354.0	3,744.4
Median	126.3	945.4	2,457.4
Max	7,441.6	10,243.4	19,972.0
Std Dev	873.0	1,363.4	3,655.5

#iterations	MS 200	AI 700	Costas 21
Min	6,210	1,217	321,361
Mean	443,970	110,393	183,428,617
Median	164,042	76,242	119,667,588
Max	7,895,872	826,871	977,709,115
Std Dev	933,766	111,352	179,049,696

Table I
SEQUENTIAL EXECUTIONS IN SECONDS AND ITERATIONS

One can see from Table I that runtimes and numbers of iterations are spread over a large interval for each benchmark, illustrating the stochasticity of the algorithm. Depending on the benchmark, there is a ratio of a few thousands times between the minimum and the maximum runtimes.

B. *Parallel Results*

We have conduct parallel experiments on the Grid5000 platform.For our experiments, we used the *Griffon* cluster at Nancy, composed of 184 Intel Xeon L5420 (Quad-core, 2.5GHz, 12MB of L2-cache, bus frequency at 1333MHz), thus with a total of 736 cores available giving a peak performances of 7.36TFlops.

Table II presents the speedup for the execution time and the number of iteration up to 256 cores for the executions of large benchmarks: MAGIC-SQUARE (instance of size 200×200), ALL-INTERVAL (instance of size 700) and COSTAS ARRAY (instance of size 21). The same code has been ported and executed, timings are given in seconds and are the average of 50 runs. One can notice there is no significant difference between speed-ups in cpu-time and in number of iterations, therefore we will prefer as a time measure the number of iterations, which has the good property of not being machine-dependent. Similar speed-ups have been achieved on other parallel machines [24].

Problem		on 1 core	speed-up on k cores				
			16	32	64	128	256
MS200	time	382.0	18.3	24.5	32.3	37.0	47.8
	#iter.	443,970	16.6	22.2	29.9	34.3	45.0
AI700	time	1,354.0	12.9	19.3	30.6	39.2	45.5
	#iter.	110,393	12.8	20.2	29.3	37.3	48.0
Costas21	time	3,744.4	15.7	26.4	59.8	154.5	274.8
	#iter.	183428617	15.8	26.4	60.0	159.2	290.5

Table II
PARALLEL SPEED-UPS (IN TIME AND NUMBER OF ITERATIONS)

For the two CSPLib benchmarks, one can observe the stabilization point is not yet obtained for 256 cores, even if speed-ups do not increase as fast as the number of cores, *i.e.,* are getting further away from linear speed-up. For the COSTAS ARRAY Problem, our algorithm reaches linear or even supra-linear speed-ups up to 256 cores. Actually, it scales linearly far beyond this point, *i.e.,* at least up to 8,192 cores, as reported in [22]. These speed-ups are visually depicted on Figure 6, up to 64 cores only to improve readability. Speed-ups of the average runtime for MAGIC-SQUARE and ALL-INTERVAL look similar, but their actual runtime behaviors are different, as will be seen in the next section.

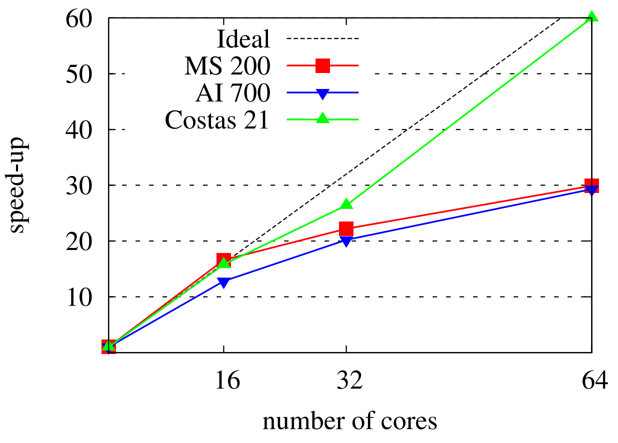

Figure 6. Speed-ups for the three benchmarks problems

VI. PREDICTION OF PARALLEL SPEED-UPS

On each problem, the sequential benchmark gives observations of the distribution of the algorithm runtime f_Y.

Yet, the exact distribution is still unknown. It can be any real distribution, not even a classical one. In the following, we will rely on the assumption that Y is distributed with a known parametric distribution. We perform a statistical test, called Kolmogorov-Smirnov test, on the hypothesis $\mathcal{H}_0$ that the collected observations correspond to a theoretical distribution. Assuming $\mathcal{H}_0$, the test first computes the probability that the distance between the collected data and the theoretical distribution does not significantly differ from its theoretical value. This probability is called the p-value. Then, the p-value is compared to a fixed threshhold (usually 0.05). If it is smaller, one rejects $\mathcal{H}_0$. For us, it means that the observations do not correspond to the theoretical distribution. If the p-value is high, we will consider that the distribution of Y is approximated by the theoretical one. Note that the Kolmogorov-Smirnov test is a statistical test, which in no way proves that Y follows the distribution. However, it measures how well the observations fit a theoretical curve and, as it will be seen in the following, it is accurate enough for our purpose.

Our benchmarks appear to fit with two distributions: the exponential distribution, as suggested by [25], and the lognormal distribution. We have also performed the Kolmogorov-Smirnov test on other distributions (*e.g.,* Gaussian and Lévy), but obtained negative results w.r.t. the experimental benchmarks, thus we do not include them in the sequel. For each problem, we need to estimate the value of the parameters of the distribution, which is done on a case by case basis. Once we have an estimated distribution for the runtimes of Y, it becomes possible to compute the expectation of the parallel runtimes and the speed-up thanks to formulas of Section III-B.

In the following, all the analyses are done on the number of iterations, and all the mathematical computations are done with Mathematica [26].

A. *The* ALL-INTERVAL *Series Problem*

The analysis is done on 720 runs of the Adaptive Search algorithm on the instance of ALL-INTERVAL series for 700 notes. The sequence of observations is written AI 700 in the following.

We test the hypothesis that the observations admit a shifted exponential distribution as introduced in Section III-C. The first step consists in estimating the parameters of the distribution, which for a shifted exponential are the value of the shift x_0 and λ[2]. We take for x_0 the minimum observed value, $x_0 = 1217$. The exponential parameter is estimated thanks to the following relation: for a non-shifted exponential distribution, the expectation is $1/\lambda$. Thus we take $\lambda = 1/(\text{mean(AI 700)} - x_0)$, which gives $\lambda = 9.15956 * 10^{-6}$.

We then run the Kolmogorov-Smirnov test on the shifted exponential distribution with these values of x_0 and λ, which answers positively (computed p-value: 0.77435). We thus admit the hypothesis that AI 700 fits this shifted exponential distribution. As an illustration, Figure 7 shows the normalized histogram of the observed runtimes and the theoretical distribution.

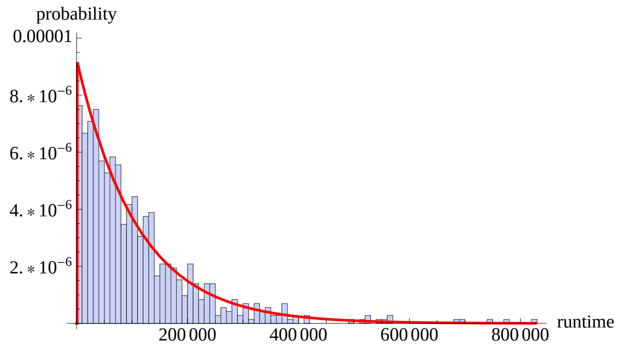

Figure 7. Histogram of the observed number of iterations for 720 runs on the ALL-INTERVAL series problem with $N = 700$, in blue. In red, the corresponding shifted exponential distribution, statistically estimated.

It is then possible to symbolically compute the speed-up that can be expected with the parallel scheme described in Section II-A. We use the formulas of Section III-C with the estimated parameters and obtain a theoretical expression for the speed-up. This allows us to calculate its value for different number of cores.

The results are given on Figure 8. With this approximated distribution, the limit of the speed-up when the number of cores tends to infinity is 90.7087. One can see that, with a 256 cores, the curve has not reached its limit, but comes close. Thus, the speed-up for this instance of ALL-INTERVAL appears significantly less than linear (*i.e.,* less than the number of cores).

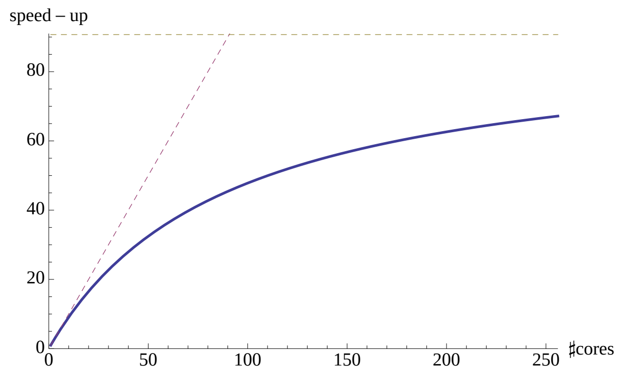

Figure 8. Predicted speed-up for AI 700 (plain blue), with its limit (dashed yellow) and the ideal linear speed-up (dashed pink).

B. *The* MAGIC-SQUARE *Series Problem*

For the MAGIC-SQUARE problem with $N = 200$, the observations are the number of iterations on 662 runs, with a

[2]All the notations are the same as in section III.

minimum of $x_0 = 6210$. The Kolmogorov-Smirnov test on a shifted exponential distribution fails, but we obtain a positive result with a lognormal distribution, with $\mu = 12.0275$ and $\sigma = 1.3398$, shifted to x_0. These parameters have been estimated with the use of the Mathematica software. As an illustration, Figure 9 shows the observations and the theoretical estimated distribution.

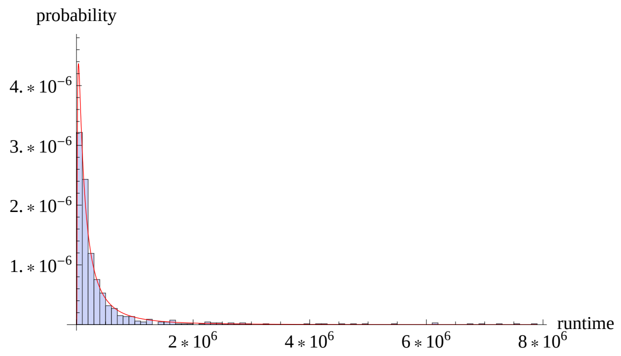

Figure 9. Histogram of the observed number of iterations for 662 runs on the MAGIC-SQUARE problem with $N = 200$, in blue. In red, the corresponding shifted lognormal distribution, statistically estimated.

The speed-up can be computed by integrating the minimum distribution with numerical integration techniques. The results are presented on Figure 10. We can observe that the speed-up grows very fast at the origin, which can be explained by the high peak of the lognormal distribution with these parameters. Again, the speed-up is computed with a numerical integration step, and we only draw the curve for integer values of n. In this case again, the speed-up is significantly less than linear from 50 cores onwards, and the limit of the speed-up when the number of cores tends to infinity is about 71.5.

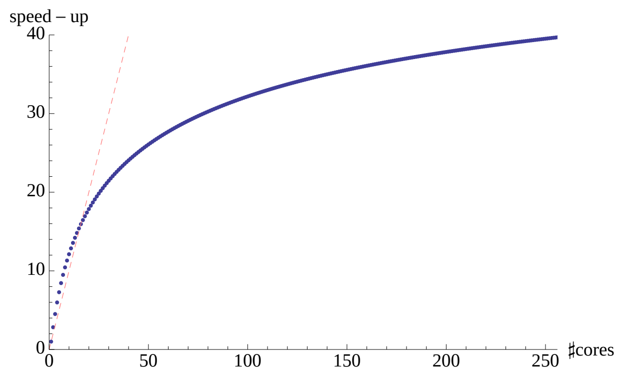

Figure 10. Predicted speed-up for MS 200 (in blue), with the ideal linear speed-up (dashed pink).

C. *The* COSTAS ARRAY *Problem*

The same analysis is done for the runs of the AS algorithm on the COSTAS ARRAY problem with $N = 21$. The observations are taken from the benchmark with 638 runs. The sequence of observations is written Costas 21.

This benchmark has an interesting property: the observed minimum, $3.2 * 10^5$ is neglictible compared to its mean $(1.8 * 10^8)$. Thus, we estimate $x_0 = 0$ and perform a Kolmogorov-Smirnov test for a (non-shifted) exponential distribution, with $\lambda = 1/\text{mean}(Costas\ 21) = 5.4 * 10^{-9}$. The test is positive for this exponential distribution, with a p-value of 0.751915. Figure 11 shows the estimated distribution compared to the observations.

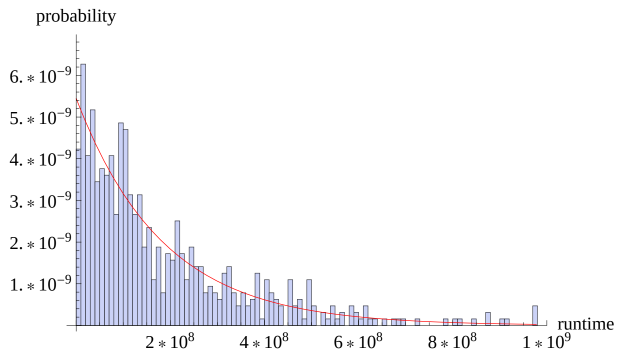

Figure 11. Histogram of the observed number of iterations for 638 runs on the COSTAS ARRAY problem with $N = 21$, in blue. In red, the corresponding exponential distribution, statistically estimated.

The computation of the theoretical speed-up is then done in the same way as for AI 700. Yet, in this case, the observed minimum for x_0 is so small that we can approximate the observations with a non-shifted distribution, thus the predicted speed-up is strictly linear, as shown in Section III-C. The results are given on Figure 12. This explains that one may observe linear speed-up when parallelizing COSTAS ARRAY.

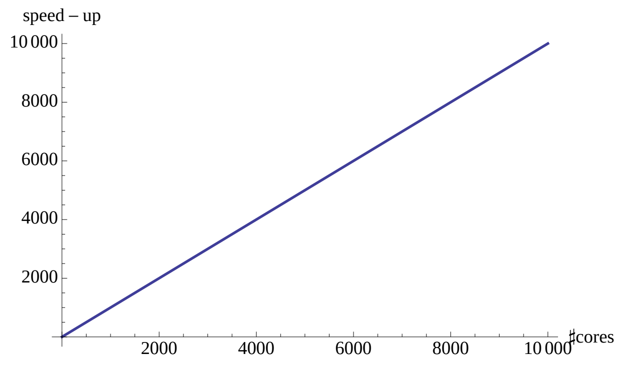

Figure 12. Predicted speed-up for Costas 21.

VII. ANALYSIS

Table III presents the comparison between the predicted and the experimental speed-ups. We can see that the accuracy of the prediction is very good up to 64 parallel cores and then the divergence is limited even for 256 parallel cores. Figure 13 illustrates this comparison graphically.

For the MS 200 problem, the experimental speed-up and the predicted one are almost identical up to 128 cores and diverging by 10% for 256 cores. For the AI 700 problem, the experimental speed-up is below the predicted one by a maximum of 30% for 128 and 256 cores. For the Costas 21 problem, the experimental speed-up is above the predicted one by 15% for 128 and 256 cores.

Problem	speed-up on k cores				
	16	32	64	128	256
MS200					
experimental	16.6	22.2	29.9	34.3	45.0
predicted	15.94	22.04	28.28	34.26	39.7
AI700					
experimental	12.8	20.2	29.3	37.3	48.0
predicted	13.7	23.8	37.8	53.3	67.2
Costas 21					
experimental	15.8	26.4	60.0	159.2	290.5
predicted	16.0	32.0	64.0	128.0	256.0

Table III
COMPARISON: EXPERIMENTAL AND PREDICTED SPEEDUPS

It is worth noticing that our model approximates the behaviors of experimental results very closely, as shown by the predicted speed-ups matching closely the real ones. Moreover we can see that on the three benchmark programs, we needed to use three different types of distribution (exponential, shifted exponential and lognormal), in order to approximate the experimental data most closely. This shows that our model is quite general and can accommodate different types of parallel behaviors.

A quite interesting behavior is exhibited by the Costas 21 problem. Our model predicts a linear speedup, up to 10,000 cores and beyond, and the experimental data gathered for this paper confirms this linear speed-up up to 256 cores. Would it scale up with a larger number of cores? Indeed we did such an experiment up to 8,192 cores on the JUGENE IBM Bluegene/P at the Jülich Supercomputing Center in Germany (with a total 294,912 cores), and reported it in [22]. The speed-up is linear up to 8,192 cores, thus showing the an excellent fit between the prediction model and real data.

Finally, let us note that our method exhibits an interesting phenomenon. For the three problems considered, the probability of returning a solution in *no* iterations is non-null: since they start by a uniform random draw on the search space, there is a very small, but not null, probability that this random initialization directly returns the solution. Hence, in theory, $x_0 = 0$ and the speed-up should admit an infinite limit when the number of cores tends to infinity. Intuitively, if the number of cores tends to infinity, at some point it will be large compared to the size of the search space (for AI 700,

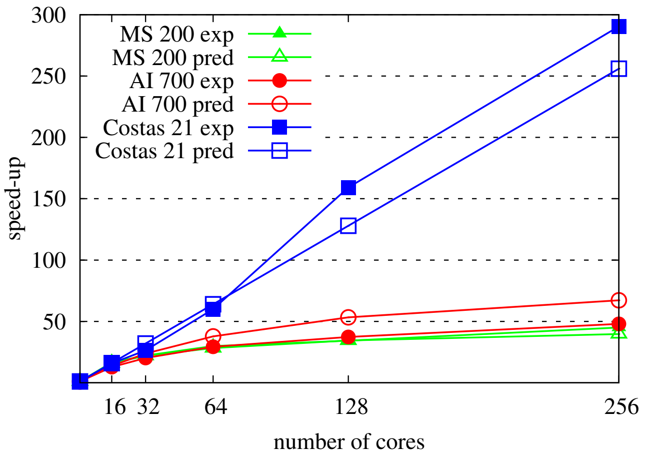

Figure 13. Predicted versus experimental speed-ups

a number with 1690 digits !), and one of the cores is likely to find the solution immediately. Yet, in practice, observations show that the experimental data may be better approximated by a shifted exponential with $x_0 > 0$, as it is the case for AI 700. Indeed, the experimental speed-up for AI 700 is far from linear and has a finite limit. However, Costas 21 has a linear speed-up due to its $x_0 << 1/\lambda$, which makes the statistical test succeed for $x_0 \simeq 0$. Firstly, this suggests that the comparison between x_0 and $1/\lambda$ on a number of observations is a key element for the parallel behavior. It also means that the number of observations needed to properly approximate the sequential distribution probably depends on the problem.

VIII. CONCLUSION

We have proposed a theoretical model for predicting and analyzing the speed-ups of Las Vegas algorithms. It is worth noticing that our model mimics the behaviors of the experimental results very closely, as shown by the predicted speed-ups matching closely the real ones. Our practical experiments consisted in testing the accuracy of the model with respect to three instances of a local search algorithm for combinatorial optimization problems. We showed that the parallel speed-ups predicted by our statistical model are accurate, matching the actual speed-ups very well up to 64 parallel cores and then with a deviation of about 10%, 15% or 30% (depending on the benchmark problem) up to 256 cores.

However, a limitation of our approach is that, in practice, we need to be able to approximate the sequential distribution. In addition, this distribution must be one of the distributions for which the first order statistics is known, symbolically (as the exponential) or numerically (as the lognormal). Nevertheless, recent results in the field of order statistics give explicit formulas for a number of useful distributions: Gaussian, lognormal, gamma, beta. This provides a wide range of tools to analyze different behaviors.

Another question is the quality of the Kolmogorov-Smirnov test, which we use to estimate the sequential distribution. For the considered benchmark, it proved to be accurate enough. However, we plan to investigate other statistical tests, such as the Cramér-von Mises test which could take into account multiple occurrences in the distribution. In this paper we validated our approach on classical combinatorial optimization and CSP benchmarks, but further research will consider a larger class of problems and algorithms, such as other randomized algorithms.

Another interesting extension of this work would be to devise a method for predicting the speed-up from scratch, that is, without any knowledge on the algorithm distribution. Preliminary observation suggests that, given a problem and an algorithm, the general shape of the distribution is the same when the size of the instances varies. For example, the different instances of ALL-INTERVAL that we tested all admit a shifted exponential distribution. If this property is valid on a wide range of problems/algorithms, then we can develop a method for predicting the speed-up for large instances by learning the distribution shape on small instances (which are easy to solve), and estimating the parallel speed-up for larger instances with our model.

REFERENCES

[1] L. Babai, "Monte-carlo algorithms in graph isomorphism testing," Université de Montréal, Research Report D.M.S. No. 79-10, 1979.

[2] E. Aarts and J. K. Lenstra, Eds., *Local Search in Combinatorial Optimization*. Chichester, UK: John Wiley and Sons, 1997.

[3] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, pp. 2245–2269, 1965.

[4] T. Gonzalez, Ed., *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall / CRC, 2007.

[5] M. Verhoeven and E. Aarts, "Parallel local search," *Journal of Heuristics*, vol. 1, no. 1, pp. 43–65, 1995.

[6] C. P. Gomes and B. Selman, "Algorithm portfolios," *Artificial Intelligence*, vol. 126, no. 1-2, pp. 43–62, 2001.

[7] H. Hoos and T. Stütze, *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, 2005.

[8] T. Ibaraki, K. Nonobe, and M. Yagiura, Eds., *Metaheuristics: Progress as Real Problem Solvers*. Springer Verlag, 2005.

[9] P. M. Pardalos, L. S. Pitsoulis, T. D. Mavridou, and M. G. C. Resende, "Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing, tabu search and GRASP," in *proceedings of IRREGULAR*, 1995, pp. 317–331.

[10] E. Alba, "Special issue on new advances on parallel metaheuristics for complex problems," *Journal of Heuristics*, vol. 10, no. 3, pp. 239–380, 2004.

[11] T. Crainic and M. Toulouse, "Special issue on parallel metaheuristics," *Journal of Heuristics*, vol. 8, no. 3, pp. 247–388, 2002.

[12] T. Van Luong, N. Melab, and E.-G. Talbi, "Local search algorithms on graphics processing units," in *Evolutionary Computation in Combinatorial Optimization*. LNCS 6022, Springer Verlag, 2010, pp. 264–275.

[13] T. G. Crainic, M. Gendreau, P. Hansen, and N. Mladenovic, "Cooperative parallel variable neighborhood search for the -median," *Journal of Heuristics*, vol. 10, no. 3, pp. 293–314, 2004.

[14] H. David and H. Nagaraja, *Order Statistics*, ser. Wiley series in probability and mathematical statistics. Probability and mathematical statistics. John Wiley, 2003.

[15] S. Nadarajah, "Explicit expressions for moments of order statistics," *Statistics & Probability Letters*, vol. 78, no. 2, pp. 196–205, Feb. 2008.

[16] R. M. Aiex, M. G. C. Resende, and C. C. Ribeiro, "Probability distribution of solution time in grasp: An experimental investigation," *Journal of Heuristics*, vol. 8, no. 3, pp. 343–373, 2002.

[17] R. Aiex, M. Resende, and C. Ribeiro, "TTT plots: a perl program to create time-to-target plots," *Optimization Letters*, vol. 1, pp. 355–366, 2007.

[18] P. V. Hentenryck and L. Michel, *Constraint-Based Local Search*. The MIT Press, 2005.

[19] P. Codognet and D. Diaz, "Yet another local search method for constraint solving," in *proceedings of SAGA'01*. Springer Verlag, 2001, pp. 73–90.

[20] ——, "An efficient library for solving CSP with local search," in *MIC'03, 5th International Conference on Metaheuristics*, T. Ibaraki, Ed., 2003.

[21] K. Drakakis, "A review of costas arrays," *Journal of Applied Mathematics*, vol. 2006, pp. 1–32, 2006.

[22] D. Diaz, F. Richoux, Y. Caniou, P. Codognet, and S. Abreu, "Parallel local search for the costas array problem," in *IEEE Workshop on new trends in Parallel Computing and Optimization (PC012), in conjunction with IPDPS 2012*. Shanghai, China: IEEE Press, May 2012.

[23] R. Bolze and al., "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 4, pp. 481–494, 2006.

[24] Y. Caniou, D. Diaz, F. Richoux, P. Codognet, and S. Abreu, "Performance analysis of parallel constraint-based local search," in *PPoPP 2012, 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New Orleans, LA, USA: ACM Press, 2012, poster paper.

[25] W. Eadie, *Statistical methods in experimental physics*. North-Holland Pub. Co., 1971.

[26] S. Wolfram, *The Mathematica Book, 5th edition*. Wolfram Media, 2003. [Online]. Available: http://reference.wolfram.com