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@s.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 analysis 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 with an actual parallel implementation up to 256 cores. We show that the prediction can be quite accurate, matching the actual speedups very well up to 100 parallel cores and then with a deviation of about 20% up to 256 cores.

Categories and Subject Descriptors D [1.3]: Parallel programming; F [1.2]: Parallelism and concurrency; G [1.6]: Stochastic programming; G [3]: Probabilistic algorithms (including Monte Carlo); G [2.1]: Combinatorial algorithms

General Terms Theory, Algorithms, Performance.

Keywords Las Vegas algorithms, Prediction, Parallel Speed-ups, Local Search, Statistical Modeling, Runtime Distributions.

1. Introduction

We propose in the present paper Las Vegas algorithms, introduced a few decades ago by [5], 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 Stochastic Local Search methods [29]. They have been used in Combinatorial Optimization for finding optimal or near-optimal solutions for several decades [1], stemming from the pioneering work of Lin on the Traveling Salesman Problem [31]. Theses 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 algorithms, such as Mixed Integer Programming or Constraint Solving, c.f. [26].

In the last years, several proposal for implementing local search algorithms on parallel computer have been proposed, the most popular being to run several competing instances of the algorithms 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 lead to so-called independent multi-walk algorithms.

2. Preliminaries

2.1 Las Vegas Algorithms

We borrow the following definition from [29], Chapter 4.

Definition 1 (Las Vegas Algorithm). An algorithm A for a problem class II 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 \( \pi \).
2. For any given instance \( \pi \in \Pi \), the runtime of A applied to \( \pi \) is a random variable.

This is a slight generalization of the classical definition, as it includes algorithms which are not guaranteed to return a solution.

A large class of Las Vegas algorithms is the so-called family of metaheuristics, such as Simulated Annealing, Genetic Algorithms, Tabu Search, Swarm Optimization, Ant-Colony optimization, etc, which have been applied to different sets of problems ranging from resource allocation, scheduling, packing, layout design, frequency allocation, etc.

2.2 Multi-walk Parallel Extension

Parallel implementation of local search metaheuristics [26, 30] has been studied since the early 1990s, when parallel machines started...
to become widely available [34] [39]. With the increasing availability of PC clusters in the early 2000s, this domain became active again [4] [13]. Apart from domain-decomposition methods and population-based method (such as genetic algorithms), [39] 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 [38] 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 [14], 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 [39]. 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 $\Pi$ 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 $\Pi$, say $A_1, \ldots, A_n$.
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.

2.3 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 of $A$ terminating with the minimal runtime (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 give us a key to quantify the expected parallel speed-up. Obviously, this can be observed 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, a rather new domain of statistics [13], which is the statistics of sorted random draws. 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 [39] 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.

3. 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.

3.1 Min Distribution

Consider a given algorithm on a given problem of a given size, say, the MAGIC-SQUARE $10 \times 10$. Depending on the result of some random components inside the algorithm, it may find a solution after 0 iterations, 10 iterations, or $10^8$ 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 $F_Y$ s.t. $F_Y(x) = \mathbb{P}[Y \leq x]$, or by its distribution, which is by definition the derivative of $F_Y$: $f_Y = F_Y'$.

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 $\mathbb{E}[Y] = \int_0^\infty f_Y(t) dt$.

Assume that the base algorithm is concurrently run in parallel on $n$ cores. In other words, each core is running its own process. The first process which 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 $Z^{(n)}$, with a distribution $f_{Z^{(n)}}$ that depends both on $n$ and on $f_Y$. Since all the $X_i$ are i.i.d., the cumulative distribution $F_{Z^{(n)}}$ can be computed as follows:

$$F_{Z^{(n)}} = \mathbb{P}[Z^{(n)} \leq x] = \mathbb{P}[\exists i \in \{1 \ldots n\}, X_i \leq x] = 1 - \mathbb{P}[\forall i \in \{1 \ldots n\}, X_i > x] = 1 - \prod_{i=1}^n \mathbb{P}[X_i > x] = 1 - (1 - F_Y(x))^n$$

which leads to:

$$f_{Z^{(n)}} = (1 - (1 - F_Y)^n)' = nf_Y(1 - F_Y)^{n-1}$$

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,
3.3 Case of an Exponential Distribution

Assume that $Y$ has a shifted exponential distribution, as it has been suggested by [21,3].

\[
\begin{align*}
    f_Y(t) &= \begin{cases} 
    0 & \text{if } t < x_0 \\
    \lambda e^{-\lambda(t-x_0)} & \text{if } t > x_0
    \end{cases} \\
    F_Y(t) &= \begin{cases} 
    0 & \text{if } t < x_0 \\
    1 - e^{-\lambda(t-x_0)} & \text{if } t > x_0
    \end{cases} \\
    \mathbb{E}[Y] &= x_0 + 1/\lambda
\end{align*}
\]

Then the above formula can be symbolically computed by hand:

\[
\begin{align*}
    f_{Z(n)}(t) &= \begin{cases} 
    0 & \text{if } t < x_0 \\
    n\lambda e^{-n\lambda(t-x_0)} & \text{if } t > x_0
    \end{cases} \\
    F_{Z(n)}(t) &= \begin{cases} 
    0 & \text{if } t < x_0 \\
    1 - e^{-n\lambda(t-x_0)} & \text{if } t > x_0
    \end{cases}
\end{align*}
\]

The intuitive observation of section 3.1 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.

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

\[
\begin{align*}
    \mathbb{E}[Z(n)] &= n\lambda \int_{x_0}^{\infty} te^{-n\lambda(t-x_0)} dt \\
    &= x_0 + \frac{1}{n\lambda} \\
    G_n &= \frac{x_0 + \frac{1}{n\lambda}}{x_0 + 1/\lambda}
\end{align*}
\]

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 [19]. For $x_0 > 0$, the speed-up admits a finite limit which is $\frac{x_0 + 1}{x_0 + 1/\lambda} = 1 + \frac{1}{x_0 \lambda}$. Yet, this limit may be reached slowly, but depends on the value of $x_0$ and $\lambda$; 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 $\lambda$, the bigger is the speed-up at the beginning. In the following, we will see that, depending on the combinations of $x_0$ and $\lambda$, different behaviors can be observed.

3.4 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 6.2 that it fits one experiment. It has two parameters, the mean $\mu$ and the standard deviation $\sigma$. 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:

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

$$F_Y(t) = \begin{cases} 0 & \text{if } t < x_0 \\ \frac{1}{2}\text{erfc}\left(\frac{\log(t-x_0)}{\sqrt{2}\sigma}\right) & \text{if } t > x_0 \end{cases}$$

where $\text{erfc}$ is the complementary error function defined by $\text{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2} dt$.

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 quite complicated formulas. But [33] 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 $n$. This is a reasonable limitation as the number of cores is indeed an integer.

4. Application to Local Search

Since about a decade, the interest for the family of Local Search methods and Metaheuristics for solving large 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 [26,28,30].

4.1 Local Search for Constraint Solving

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 been attracting some interest since about a decade [10,21,28], as it can tackle CSPs instances far beyond the reach of classical propagation-based constraint solvers [7,27]. A generic, domain-independent constraint-based local search method, named Adaptive Search, has been proposed by [10,11]. 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.

4.2 Adaptive Search

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/](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 [56]. 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. For each constraint, an error function needs to be defined; it gives, for each tuple of variable values, an indication of how much the constraint is violated. This idea has also been proposed independently by [21], where it is called “penalty functions”, and then reused by the Comet system [28], where it is called “vi-
Adaptive Search relies on iterative repair, based on variable and constraint error information, seeking to reduce the error on the worst variable so far. The basic idea is to compute the error function for each constraint, then combine for each variable the errors of all constraints in which it appears, thereby projecting constraint errors onto the relevant variables. This combination of errors is problem-dependent, see [10] for details and examples, but it is usually a simple sum or a sum of absolute values, although it might also be a weighted sum if constraints are given different priorities. Finally, the variable with the highest error is designated as the “culprit” and its value is modified. In this second step, the well known min-conflict heuristic [32] is used to select the value in the variable domain which is the most promising, that is, the value for which the total error in the next configuration is minimal.

In order to prevent being trapped in local minima, the Adaptive Search method also includes a short-term memory mechanism to store configurations to avoid (variables can be marked Tabu and “frozen” for a number of iterations). It also integrates reset transitions to escape stagnation around local minima. A reset consists in assigning fresh random values to some variables (also randomly chosen). A reset is guided by the number of variables being marked Tabu. It is also possible to restart from scratch when the number of iterations becomes too large (this can be viewed as a reset of all variables but it is guided by the number of iterations). The core ideas of adaptive search can be summarized as follow:

• 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.

5. Benchmark Problems and Experimental Results

We have chosen to test this method on two problems from the CSPLib benchmark library [23], and on a hard combinatorial problem abstracted from radar and sonar applications. After briefly introducing the classical benchmarks, we detail the latter problem, called Costas Array. Then we show the performance and the speed-ups obtained with both sequential and a multi-walk Adaptive Search algorithm on these problems.

We use two classical benchmarks from CSPLib consisting of:

• The ALL-INTERVAL Series problem (prob007 in CSPLib),
• The MAGIC-SQUARE problem (prob019 in CSPLib).

Although these benchmarks are academic, they are abstractions of real-world problems and 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 10166713 configurations. Classical propagation-based constraint solvers cannot solve this problem for instances higher than 20x20. 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.

5.1 The ALL-INTERVAL Series Problem

This problem is described as prob007 in the CSPLib. This benchmark is in fact a well-known exercise in music [37] where the goal is to compose a sequence of N notes such that all are different and tonal intervals between consecutive notes are also distinct. This problem is equivalent to finding a permutation of the N first integers such that the absolute difference between two consecutive pairs of numbers are all different. This amounts to finding a permutation (X1, ..., XN) of {0, ..., N − 1} such that the list (abs(X1 − X2), abs(X2 − X3), ..., abs(XN−1 − XN)) is a permutation of 1, ..., N − 1. A possible solution for N = 8 is 3, 6, 0, 7, 2, 4, 5, 1 since all consecutive distances are different:

\[ 3 \rightarrow 6 \rightarrow 0 \rightarrow 7 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 1 \]

5.2 The MAGIC-SQUARE Problem

The MAGIC-SQUARE problem is catalogued as prob019 in CSPLib and consists in placing the numbers \{1, 2, ..., N^2\} on an N × N square such that each row, column and main diagonal equal the same sum (the constant N(N^2 + 1)/2).

For instance, this figure shows a well-known solution for N = 4 (depicted by Albrecht Dürer in his engraving Melancholia I, 1514).

5.3 The Costas Array Problem

A Costas array is an N × 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. We give here an example of Costas array of size 5. It is convenient to see the Costas Array Problem (CAP) as a permutation problem by considering an array of N variables (V1, ..., VN) which forms a permutation of {1, 2, ..., N}. The above Costas array can thus be represented by the array [3, 4, 2, 1, 5].

Historically these arrays have been developed in the 1960’s to compute a set of sonar and radar frequencies avoiding noise [12]. A very complete survey on Costas arrays can be found in [17].

The problem of finding a Costas array of size N is very complex, since the required time grows exponentially with N. In the 1980’s, several algorithms have been proposed to build a Costas array given N (methods to produce Costas arrays of order 24 to 29 can be found in [6, 18, 19, 35]), such as the Welch construction [23] and the Golomb construction [24], but these methods cannot build Costas arrays of size 32 and some higher non-prime sizes. Nowadays, after many decades of research, it remains unknown if there exist any Costas arrays of size 32 or 33. Another difficult problem is to enumerate all Costas arrays for a given size. Using the Golomb and Welch constructions, Drakakis et al present in [19] all Costas arrays for N = 29. They show that among the 29! permutations, there are only 164 Costas arrays, and 23 unique Costas arrays up to rotation and reflection.

Parallel Las Vegas Algorithms

5

2012/12/19
### 5.4 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. The following Tables 1 and 2 shows the minimum, mean, median and maximum respectively among the runtimes and the number of iterations of our benchmarks.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Min</th>
<th>Mean</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS 200</td>
<td>5.51</td>
<td>382.0</td>
<td>126.3</td>
<td>7,441.6</td>
</tr>
<tr>
<td>AI 700</td>
<td>23.25</td>
<td>1,354.0</td>
<td>945.4</td>
<td>10,243.4</td>
</tr>
<tr>
<td>Costas 21</td>
<td>6.55</td>
<td>3,744.4</td>
<td>2,457.4</td>
<td>19,972.0</td>
</tr>
</tbody>
</table>

Table 1. Sequential execution times (in seconds)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Min</th>
<th>Mean</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS 200</td>
<td>6,210</td>
<td>443,969</td>
<td>164,042</td>
<td>7,895,872</td>
</tr>
<tr>
<td>AI 700</td>
<td>1,217</td>
<td>110,393</td>
<td>76,242</td>
<td>826,871</td>
</tr>
<tr>
<td>Costas 21</td>
<td>321,361</td>
<td>183,428,617</td>
<td>119,667,588</td>
<td>977,709,115</td>
</tr>
</tbody>
</table>

Table 2. Sequential number of iterations

One can see that runtimes and the number of iterations, respectively from Tables 1 and 2, are spread over a large interval, illustrating the stochasticity of the algorithm. Depending on the benchmark, there is a ratio of a few thousands between the minimum and the maximum runtimes.

### 5.5 Parallel Results

We have conduct parallel experiments on the Grid5000 platform [8], the French national grid for research, which contains 8,596 cores deployed on 11 sites distributed in France. 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.

Tables 3 and 4 present the execution times and the number of iterations, respectively, with speed-ups for executions of large benchmarks MAGIC-SQUARE 200×200, ALL-INTERVAL with \( n = 700 \) and COSTAS ARRAY with \( n = 21 \), up to 256 cores. The same code has been ported and executed, timings are given in seconds and are the average of 50 runs. One can notice there are no significant differences between speed-ups of these two tables, therefore we will prefer as a time measurement the number of iterations, which has the good property of not being machine-dependent. Anyway, similar speed-ups have been achieved on other parallel machines [9].

<table>
<thead>
<tr>
<th>Problem</th>
<th>time on 1 core (seconds)</th>
<th>speed-up on k cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>MS 200</td>
<td>382.0</td>
<td>18.3</td>
</tr>
<tr>
<td>AI 700</td>
<td>1,354.0</td>
<td>12.9</td>
</tr>
<tr>
<td>Costas 21</td>
<td>3,744.4</td>
<td>15.7</td>
</tr>
</tbody>
</table>

Table 3. Speed-ups with respect to sequential time

<table>
<thead>
<tr>
<th>Problem</th>
<th># of iterations on 1 core</th>
<th>speed-up on k cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>MS 200</td>
<td>443,969</td>
<td>16.6</td>
</tr>
<tr>
<td>AI 700</td>
<td>110,393</td>
<td>12.8</td>
</tr>
<tr>
<td>Costas 21</td>
<td>183,428,617</td>
<td>15.8</td>
</tr>
</tbody>
</table>

Table 4. Speed-ups with respect to sequential 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. This is visually depicted on Figure 6. For the COSTAS ARRAY Problem, the AS algorithm reaches linear or even supra-linear speed-ups up to 256 cores, as depicted in Figure 7. Actually, it scales linearly far beyond this point, i.e., at least up to 8,192 cores, as reported in [16].

### 6. 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 \( H_0 \) that the collected observations correspond to a theoretical distribution. Assuming \( H_0 \), 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 threshold (usually 0.05). If it is smaller, one rejects \( H_0 \). 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.

The distributions tested for local search algorithms are the exponential distribution, as suggested by [20], and the lognormal distribution, because it appears to fit the MAGIC-SQUARE problem. 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 will 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 3.2. All the mathematical calculations are done with Mathematica, a commercial software for symbolic computation [40].

In the following, all the analyses are done on the number of iterations, because they are more likely to be unbiased.

### 6.1 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 3.3. The first step consists in estimating the parameter of the distribution, which for a shifted exponential is the value of the shift $x_0$ and $\lambda$. 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 $\lambda = 1/(\text{mean}(\text{AI 700}) - x_0)$, which gives $\lambda = 9.15956 \times 10^{-6}$.

We then run the Kolmogorov-Smirnov test on the shifted exponential distribution with these values of $x_0$ and $\lambda$, 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 8 shows the normalized histogram of the observed runtimes and the theoretical distribution.

It is then possible to symbolically compute the speed-up that can be expected with the parallel scheme described in Section 2.2. We use the formulas of Section 3.3 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 9. 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 (linear meaning: equal to the number of cores).

### 6.2 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 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 10 shows the observations and the theoretical estimated distribution.

The speed-up can be computed by integrating the minimum runtime 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.

6.3 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, \( 3.2 \times 10^5 \), is negligible compared to its mean (\( 1.8 \times 10^7 \)). Thus, we estimate \( x_0 = 0 \) and perform a Kolmogorov-Smirnov test for a (non-shifted) exponential distribution, with \( \lambda = 1/\text{mean(Costas 21)} = 5.4 \times 10^{-9} \). The test is positive for this exponential distribution, with a p-value of 0.751915. Figure 12 shows the estimated distribution compared to the observations.

The computation of the theoretical speed-up is also 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 5.3. The results are given on Figure 13. This explains that one may observe linear speed-up when parallelizing COSTAS ARRAY.

### Table 5: Comparison: experimental and predicted speedups

<table>
<thead>
<tr>
<th>Problem</th>
<th>speed-up on ( k ) cores</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS 200</td>
<td>experimental</td>
<td>16.6</td>
<td>22.2</td>
<td>29.9</td>
<td>34.3</td>
<td>45.0</td>
</tr>
<tr>
<td></td>
<td>predicted</td>
<td>15.94</td>
<td>22.04</td>
<td>28.28</td>
<td>34.26</td>
<td>39.7</td>
</tr>
<tr>
<td>AI 700</td>
<td>experimental</td>
<td>12.8</td>
<td>20.2</td>
<td>29.3</td>
<td>37.3</td>
<td>48.0</td>
</tr>
<tr>
<td></td>
<td>predicted</td>
<td>13.7</td>
<td>23.8</td>
<td>37.8</td>
<td>53.3</td>
<td>67.2</td>
</tr>
<tr>
<td>Costas 21</td>
<td>experimental</td>
<td>15.8</td>
<td>26.4</td>
<td>60.0</td>
<td>159.2</td>
<td>290.5</td>
</tr>
<tr>
<td></td>
<td>predicted</td>
<td>16.0</td>
<td>32.0</td>
<td>64.0</td>
<td>128.0</td>
<td>256.0</td>
</tr>
</tbody>
</table>

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 such an experiment has been done up to 8,192 cores on the JU(GNE) IBM Bluegene/P at the Jülich Supercomputing Center in Germany (with a total 294,912 cores), and reported in [16], of which Figure 7 is adapted. We can see that the speed-up is indeed linear up to 8,192 cores, thus showing the adequation of the prediction model with the 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 be linear, with 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 and one of the cores is likely to immediately find the solution.

Yet, in practice, observations show that the experimental curves may be better approximated by a shifted exponential with \( x_0 > 0 \), as it is the case for AI 700. With an exponential distribution, this leads to non-linear speed-up with a finite limit. Indeed, the experimental speed-up for AI 700 is far from linear. On the contrary, Costas 21 has a linear speed-up due to its small distribution probably depends on the problem.

### 8. 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 speedups 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, one limitation of our approach is that, in practice, we need to be able to compute the expectation of the minimum distribution. Nevertheless, apart from the exponential distribution for which this computation is easy, recent results in the field of order statistics gives explicit formulas for a number of useful distributions: gaussian, lognormal, gamma, beta. This provides a wide range of tools to analyze different behaviors. 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 SAT solvers and other randomized algorithms (e.g. quick sort).

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/algorithm, then we can develop a method for predicting the speed-up for large instances by learning the distribution shape on small instances (which are easier to solve), and then estimating the parallel speed-up for larger instances with our model.

**References**


