Journal of Heuristics manuscript No.
(will be inserted by the editor)

Estimating Parallel Runtimes for Randomized Algorithms in Constraint Solving

Charlotte Truchet · Alejandro Arbelaez · Florian Richoux · Philippe Codognet

the date of receipt and acceptance should be inserted later

Abstract This paper presents a detailed analysis of the scalability and parallelization of Local Search algorithms for constraint-based and SAT (Boolean satisfiability) solvers. We propose a framework to estimate the parallel performance of a given algorithm by analyzing the runtime behavior of its sequential version. Indeed, by approximating the runtime distribution of the sequential process with statistical methods, the runtime behavior of the parallel process can be predicted by a model based on order statistics. We apply this approach to study the parallel performance of a Constraint-Based Local Search solver (Adaptive Search), two SAT Local Search solvers (namely Sparrow and CCASAT), and a propagation-based constraint solver (Gecode, with a random labeling heuristic). We compare the performance predicted by our model to actual parallel implementations of those methods using up to 384 processes. We show that the model is accurate and predicts performance close to the empirical data. Moreover, as we study different types of problems, we observe that the experimented solvers exhibit different behaviors and that their runtime distributions can be approximated by two types of distributions: exponential (shifted and non-shifted) and lognormal. Our results show that the proposed framework estimates the runtime of the parallel algorithm with an average discrepancy of 21% w.r.t. the empirical data across all the experiments with the maximum allowed number of processors for each technique.

Charlotte Truchet, Florian Richoux
LINA, UMR 6241/ University of Nantes
E-mail: {charlotte.truchet,florian.richoux}@univ-nantes.fr

Alejandro Arbelaez
INSIGHT Centre for Data Analytics / University College Cork
E-mail: alejandro.arbelaez@insight-centre.org

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

1 Introduction

In the last years, parallel algorithms for solving hard combinatorial problems, such as Constraint Satisfaction Problems (CSP), have been of increasing interest in the scientific community. The combinatorial nature of the problem makes it difficult to parallelize existing solvers, without costly communication schemes. Several parallel schemes have been proposed for incomplete or complete solvers, one of the most popular (as observed in the latest SAT competitions www.satcompetition.org) being to run several competing instances of the algorithm on different processes with different initial conditions or parameters, and let the fastest process win over others. The resulting parallel algorithm thus terminates with the minimal runtime among the launched processes. The framework of independent multi-walk parallelism, seems to be a promising way to deal with large-scale parallelism. Cooperative algorithms might perform well on shared-memory machines with a few tens of processors, but are difficult to extend efficiently on distributed hardware. This leads to so-called independent multi-walk algorithms in the CSP community [55] and portfolio algorithms in the SAT community (satisfiability of Boolean formulae) [28].

However, although it is easy to obtain good Speed-up on a small-scale parallel machine (*viz.* with a few tens of processes), it is not easy to know how a parallel variant of a given algorithm would perform on a massively parallel machine (*viz.* with thousands of processes). Parallel performance models are thus particularly important for parallel constraint solvers, and any indication on how a given algorithm (or, more precisely, a pair formed by the algorithm and the problem instance) would scale on massively parallel hardware is valuable. If it becomes possible to estimate the maximum number of processes until which parallelization is efficient, then the actual parallel computing power needed to solve a problem could be deduced. This piece of information might be quite relevant, since supercomputers or systems such as Google Cloud and Amazon EC2 can be rented by processor-hour with a limit on the maximum number of processors to be used. In this context, modelling tools for the behavior of parallel algorithms are expected to be very valuable in the future.

The goal of this paper is to study the parallel performance of randomized constraint solving algorithms under the independent multi-walk scheme, and to model the performance of the parallel execution from the runtime distribution of sequential runs of a given algorithm. Randomized constraint solvers considered in this paper include Local Search algorithms for Constraint Satisfaction Problems, Local Search techniques for SAT, and complete algorithms with random components *e.g.,* a propagation-based backtrack search with random heuristics. An important application of this work relates to the increasing computational power being available in cloud systems (e.g., Amazon Cloud EC2, Google Cloud and Microsoft Azure), a good estimate on how the algorithm scales might allow users to rent just the right number of cores. Most papers on the performance of stochastic Local Search algorithms focus on the average runtime in order to measure the performance of both sequential and parallel executions. However, a more detailed analysis of the runtime behavior could

be done by looking at the runtime of the algorithm (*e.g.,* CPU-time or number of iterations) as a random variable and performing a statistical analysis of its probability distribution. More precisely, we first approximate the empirical sequential runtime distribution by a well-known statistical distribution (*e.g.,* exponential or lognormal) and then derive the runtime distribution of the parallel version of the solver. Our model is related to *order statistics*, a rather recent domain of statistics [23], which is the statistics of sorted random draws. Our method encompasses any algorithm whose solving time is random and makes it possible to formally determine the average parallel runtime of such algorithms for any number of processors.

For Local Search, we will consider algorithms in the framework of *Las Vegas algorithms* [10], a class of algorithms related to Monte-Carlo algorithms introduced a few decades ago, whose runtime may vary from one execution to another, even on the same input. The classical parallelization scheme of multi-walks for Local Search methods can easily be generalized to any Las Vegas algorithm. We will study two different sets of algorithms and problems: first, a Constraint-Based Local Search solver on CSP instances, and, secondly, two SAT Local Search solvers on random and crafted instances. Interestingly, this general framework encompasses other types of Las Vegas algorithms, and we will also apply it to a propagation-based constraint solver with a random labeling procedure on CSP instances.

We will confront the performance predicted by the statistical model with actual speed-ups obtained for parallel implementations of the above-mentioned algorithms and show that the prediction can be quite accurate, matching the actual speed-up up to a large number of processors. More interestingly, we can also model both the initial and the asymptotic behavior of the parallel algorithm.

This paper extends [51] and [9] by giving a detailed presentation of the runtime estimation model, based on order statistics, and by validating the model on randomized propagation-based constraint solvers, extensive experimental results for stochastic local search algorithms on well-known CSP instances from CSPLib and SAT instances obtained from the international SAT competition. Additionally, we provide a more detailed theoretical analysis of the reference distributions used for predicting the parallel performance.

The paper is organized as follows. Section 2 presents the existing approaches in parallel constraint solving, and formulates the question we address in the following. Section 3 details our probabilistic model for the class of parallel algorithms we tackle in this article, based on Las Vegas algorithms. Several such algorithms can be used in constraint solving. Each of the three last sections is dedicated to a specific family of Las Vegas algorithm: Constraint-Based Local Search in Section 4, SAT Local Search in Section 5 and propagation-based methods with randomization in Section 6. For each method, we apply the model to compute the parallel speed-ups, run the parallel algorithm in practice and compare and analyze the results. Section 7 presents a comparison of our probabilistic method against a method that does not involve statistical tools. A conclusion, in Section 8, ends the paper.

2 Parallel Constraint Solving

This section presents existing approaches for parallel constraint solving. The base methods are Constraint-Based Local Search, Local Search for SAT, and complete solvers based on propagation techniques and backtrack search. We then present the state of the art on estimating parallel speed-up for randomized solvers.

2.1 Parallel Local Search

Parallel implementation of Local Search metaheuristics [30, 35] has been studied since the early 1990s, when parallel machines started to become widely available [44, 55]. With the increasing availability of PC clusters in the early 2000s, this domain became active again [4, 21]. [42] recently proposed a scalable parallel algorithm for backtracking algorithms based on load-balancing to distribute the work across multiple processes with low communication overhead. Apart from domain-decomposition methods and population-based method (such as genetic algorithms), [55] 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 [54] for such a method making use of GPUs for the parallel phase). Multi-walk methods, *i.e.,* 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 [22], 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 speed-ups [55]. 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.

2.2 Parallel Local Search for SAT

Like for CSP, it is now currently admitted that an easy and effective manner to parallelize Local Search solvers consists in executing in parallel multiple copies of the solver with or without cooperation. We remark that nearly all parallel solvers in the parallel SAT competitions are based on the multi-walk framework[1]. The non-cooperative approach has been used in the past to solve SAT and MaxSAT instances. gNovelty+ [47] executes multiple copies of gNovelty without cooperation until a solution is obtained or a given timeout is reached;

[1] The SAT community usually refer to the multi-walk framework as portfolio algorithms.

and [45] executes multiple copies of GRASP until an assignment which satisfies a given number of clauses is obtained. Strategies to exploit cooperation between parallel SAT Local Search solvers have been studied in [8] in the context of multi-core architectures with shared memory and in [6] in massively parallel systems with distributed memory.

2.3 Parallel Complete Solvers

Early experiments on the parallelization of propagation-based complete solvers date back to the beginning of Constraint Logic Programming, cf. [52], and used the search parallelism of the host logic language [36]. Most of the proposed implementations have been based on some kind of OR-parallelism, splitting the search space between different processors and relying on the shared-memory multicore architecture as the different processors work on shared data-structures representing a global environment in which the subcomputations take place. However only very few implementations of efficient constraint solvers on such machines have ever been reported, for instance [46] or [18] for a shared-memory architecture with 8 CPU processors. The Comet system [53] has been parallelized for small clusters of PCs, both for its Local Search solver [39] and its propagation-based constraint solver [40]. More recent experiments have been done up to 12 processors [41].

Search-space splitting techniques such as domain decomposition have also been implemented in the domain of Constraint Programming, but initial experiments [13] show that the speed-up goes to flatten after a few tens of processors, thus away from linear speed-up. A recent approach based on a smaller granularity domain decomposition [48] shows better performance. The results for all-solution search on classical CSPLib benchmarks are quite encouraging and show an average speed-up of 14 to 20 up with 40 processors w.r.t. the base sequential solver.

In the domain of combinatorial optimization, the most popular complete method that has been parallelized at a large scale is the classical branch and bound method [26], because it does not require much information to be communicated between parallel processes: basically only the current upper (or lower) bound of the solution. It has thus been a method of choice for experimenting the solving of optimization problems using grid computing, see for instance [1] and also [17], which use several hundreds of nodes of the Grid'5000 platform. Good speed-ups are achieved up to a few hundreds of processors but interestingly, their conclusion is that runtimes tend to stabilize afterward. A simple solving method for project scheduling problems has also been implemented on an IBM Bluegene/P supercomputer [57] up to 1,024 processors, but with mixed results since they reach linear speed-ups until 512 processors only, and then no improvements beyond this limit.

2.4 How to Estimate Parallel Speed-Up ?

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 equal to 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. 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 [23]. 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 sharp and opens the way for defining more accurate models and applying them to larger classes of algorithms.

Previous works [55] studied the case of a particular distribution for the sequential algorithm: the exponential distribution. This case is ideal and it yields a linear speed-up. Our model enables us 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 goes toward infinity. We will see that these distributions fit experimental data for some problems.

The literature provides other probabilistic models for parallel Las Vegas algorithms. For parallelization schemes based on restart strategies, Luby et al. [38] proposed an optimal universal strategy, which achieves the best speed-ups amongst the restart-based universal strategies. Although our work, which is not based on restarts, does not fit within this framework, it is related to it since it uses similar probabilistic ideas. In our case, a probabilistic model is used to model the efficiency of a parallel scheme, without modifying the algorithm. In their case, probabilistic tools are used to find an optimal restart scheme for a base algorithm. With the same idea of restart-based parallelization of Las Vegas algorithms, [50] goes a step further and provides a more detailed model. It clarifies in particular the cases of super-linear speed-ups, which are due, in this framework, to inefficient (or not well-tuned) sequential algorithms. Interestingly, this article also experiments with log-normally distributed sequential algorithms, which confirms our hypothesis that not only exponential distributions have to be investigated (as assumed by [3, 2]).

3 Probabilistic Model

Randomized algorithms are stochastic processes. Their behavior (output, running time...) is non-deterministic and varies according to a probability distribution, which may or may not be known. For instance, Local Search algorithms include several random components: choice of an initial configuration, choice of a move among several candidates, plateau mechanism, random restart, etc. A complete algorithm may also have a stochastic behavior when they include random components, such as random heuristics, restarts with randomization, etc.

In the following, we first define the class of algorithms that our model encompasses. We then present our probabilistic model, considering the *computation time* of an algorithm (whatever it is) as a random variable, and using elements of probability theory to study its multi-walk parallel version.

3.1 Parallel Las Vegas Algorithms

The notion of Las Vegas algorithm encompasses a wide range of combinatorial solvers. We borrow the following definition from [32], 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.

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.

3.2 Min Distribution

Consider the problem of solving a given problem instance using a Las vegas algorithm, say, tabu search on 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 us 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. $\mathcal{F}_Y(x) = \mathbb{P}[Y \leq x]$, that is, the function which, for a x in the scope of the random variables, gives the probability that a random draw is smaller than x. Another possible tool to model a random variable is its distribution, which is by definition the derivative of $\mathcal{F}_Y$: $f_Y = \mathcal{F}'_Y$. 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.

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 $\mathbb{R}$. This step is merely technical, and the probability distribution in $\mathbb{N}$ can be retrieved from the distribution in $\mathbb{R}$.

The expectation of the computation is then defined by the standard formula for real-valued distributions: $\mathbb{E}[Y] = \int_0^\infty t f_Y(t) dt$. This formula is the extension to $\mathbb{R}$ of the classical expectation formula in the case of integer distributions ($\sum_0^\infty t f_Y(t)$).

Assume that the base algorithm is concurrently run in parallel on n processors. In other words, over each processor the running process is a copy of the algorithm with different initial random seed. 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:

$$
\begin{aligned}
\mathcal{F}_{Z^{(n)}} &= \mathbb{P}[Z^{(n)} \leq x] && \text{by definition} \\
&= \mathbb{P}[\exists i \in \{1...n\}, X_i \leq x] && \text{because of the multiwalk rule} \\
&= 1 - \mathbb{P}[\forall i \in \{1...n\}, X_i > x] && \text{probability formula for the negation} \\
&= 1 - \textstyle\prod_{i=1}^n \mathbb{P}[X_i > x] && \text{because the random variables are i.i.d.} \\
&= 1 - (1 - \mathcal{F}_Y(x))^n && \text{by definition}
\end{aligned}
$$

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, Fig. 1 shows this phenomenon when the base algorithm admits a Gaussian distribution.

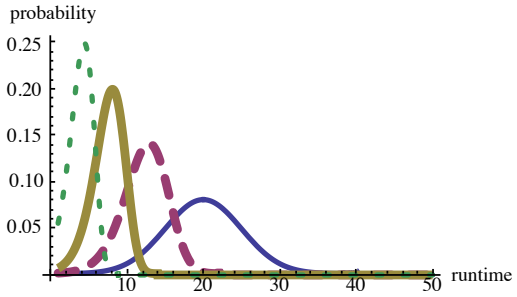


Fig. 1 Distribution of $Z^{(n)}$, in the case where Y admits a Gaussian distribution (cut on $\mathbb{R}^-$ and renormalized). The blue curve is Y. The distributions of $Z^{(n)}$ are in pink for $n = 10$, in yellow for $n = 100$ and in green for $n = 1000$.

3.3 Expectation and Speed-up

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

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

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

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

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

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 [23] 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 processors, 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 [43] give explicit formulas for this quantity for several classical probability distributions. Except in the case of the exponential distribution, detailed below, the formulas given for the minimum order statistics are rather complicated. In some cases, a symbolic computation may not even succeed in a reasonable amount of time. When this happens, we will perform a first step of symbolic computation, then another step of numeric integration to obtain the numerical value for the speed-up.

3.4 Case of an Exponential Distribution

Assume that Y has a shifted exponential distribution, as it has been suggested by [3, 2].

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

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

$$\mathbb{E}[Y] = x_0 + 1/\lambda$$

Then the formula of Section 3.2 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 3.2 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 Fig. 2.

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

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

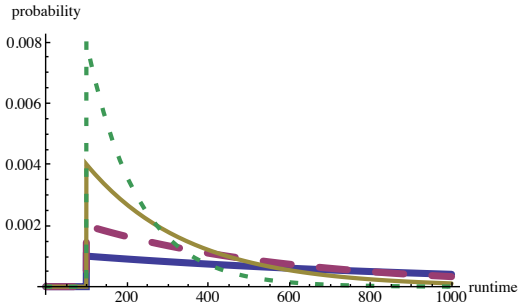


Fig. 2 For an exponential distribution, here in blue with $x_0 = 100$ and $\lambda = 1/1000$, simulations of the distribution of $Z^{(n)}$for $n = 2$ (pink), $n = 4$ (yellow) and $n = 8$ (green).

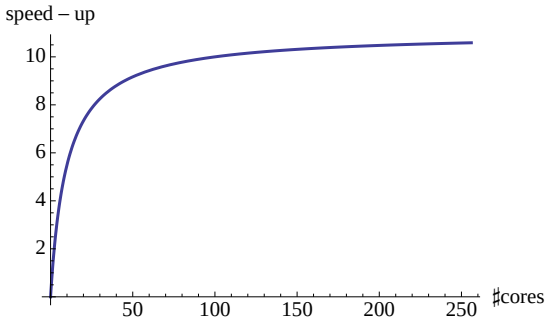


Fig. 3 Predicted speed-up in case of an exponential distribution, with $x_0 = 100$ and $\lambda = 1/1000$, w.r.t. the number of processors.

Fig. 3 shows the evolution of the speed-up when the number of processors increases. With such a rather simple formula for the speed-up, it is worth studying what happens when the number of processors n goes to infinity. Depending on the chosen algorithm, If $x_0 = 0$, then the expectation goes to 0 and the speed-up is equal to n. This case has already been studied by [55]. 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. From the previous formula we observe that the closer x_0 is to zero, the better is the speedup. Another interesting value is the coefficient of the tangent at the origin, which approximates the speed-up for a small number of processors. 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.

3.5 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. The lognormal distribution appears in several of our experiments in Section 4 and 5. The lognormal distribution has two parameters, the mean μ and the standard deviation σ. In the same way as the shifted exponential distribution, we shift the distribution so that it starts at a given parameter x_0. Formally, a (shifted) lognormal distribution is defined as:

$$f_Y(t) = \begin{cases} 0 & \text{if } t < x_0 \\ \frac{\Phi(log(t-x_0))}{t-x_0} & \text{if } t > x_0 \end{cases}$$

where $\Phi(t) = \frac{1}{\sqrt{2*\pi}} e^{-t^2/2}$.

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.

Fig. 4 depicts lognormal distributions of $Z^{(n)}$, for several n. The computations for the distribution of $Z^{(n)}$and the theoretical speed-up are the same as given in Section 3.3. The computation of the expectation for the moments of order statistics of a lognormal distribution can be found in [43], which gives an explicit formula with only a numerical integration step. We only recall from [43] this formula for the first moment (expectation) of the first order statistics (minimum distribution), shifted by x_0, which we are interested in.

We will need Lauricella functions defined by:

$$F_A^{(n)}(a; b_1, ..., b_n; c_1, ..., c_n; x_1, ..., x_n)) = \sum_{m_1=0}^{\infty} ... \sum_{m_n=0}^{\infty} \frac{(a)_{m_1+...+m_n}(b_1)_{m_1}...(b_n)_{m_n}}{(c_1)_{m_1}...(c_n)_{m_n}} \frac{x_1^{m_1}...x_n^{m_n}}{m_1!...m_n!}$$

Then one has:

$$\mathbb{E}\left[Z^{(n)}\right] = x_0 + ne^{1/2} \sum_{l=0}^{n-1} \binom{n-1}{l} \left(-\frac{1}{2}\right)^l \sum_{p=0}^{r} \binom{l}{p} \left(-\frac{2}{\sqrt{\pi}}\right)^p * \mathbb{E}\left[\left(\frac{\mathcal{N}\sqrt{2}-1}{2\sqrt{2}}\right)^p F_A^{(p)}(\frac{1}{2}, ..., \frac{1}{2}; \frac{3}{2}, ..., \frac{3}{2}; -\frac{(\mathcal{N}\sqrt{2}-1)^2}{8}, ..., -\frac{(\mathcal{N}\sqrt{2}-1)^2}{8})\right]$$

where $\mathcal{N}$ is a standard normal random variable. In addition, [43] provides pointers to routines in Mathematica to compute the terms of this formula with only one step of numerical integration. In practice, we will get our numerical results with an earlier step of numerical integration in Mathematica, with a good accuracy.

This allows us to draw the general shape of the speed-up, an example being given on Fig. 5. Due to the numerical integration step, which requires numerical values for the number of processors n, we restrict the computation to integer values of n.

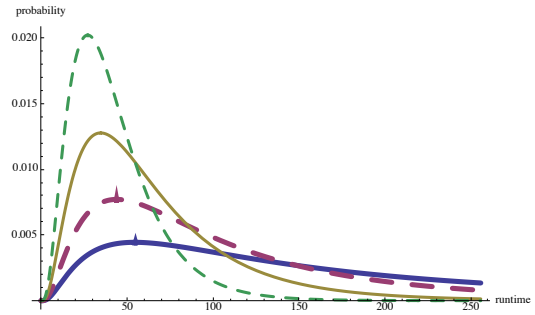


Fig. 4 For a lognormal distribution (in blue), with $x_0 = 0$, $\mu = 5$ and $\sigma = 1$, simulations of the distribution of $Z^{(n)}$for $n = 2$ (pink), $n = 4$ (yellow) and $n = 8$ (green).

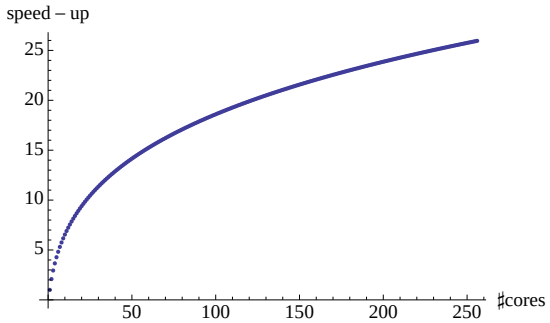


Fig. 5 Predicted speed-up in case of a lognormal distribution, with $x_0 = 0$, $\mu = 5$ and $\sigma = 1$, w.r.t. the number of processors.

3.6 Methodology

In this section we detail how to perform the prediction of the parallel runtime and speed-up with respect to sequential execution from the analysis of the sequential runtime distribution. 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 threshold (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 using the formulas of Section 3.3.

In the following, all the analyses are done on the number of iterations, because they are more likely to be unbiased, and all the mathematical computations are done with Mathematica [56].

4 Application to Constraint-based Local Search

Recently, the application of Local Search to solve combinatorial problems has attracted the attention of researcher [19, 53], as it can tackle Constraint Satisfaction Problems (CSP) 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. The main 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.

We used for our experiments the reference implementation of Adaptive Search (AS) which has been developed as a framework library in C is and available as a freeware at the URL:
`http://cri-dist.univ-paris1.fr/diaz/adaptive/`

4.1 Experimental Protocol and Benchmarks

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[2]: ALL-INTERVAL Series (prob007 in CSPLib), and the MAGIC-SQUARE problem (prob019 in CSPLib). We refer the reader to [51] for a complete description of the benchmarks and the local search framework.

We run our benchmarks in a sequential manner a certain number of times to evaluate the quality of the proposed model. Sequential experiments, as well as parallel experiments, have been done on the *Griffon* cluster of the Grid'5000 platform [12], the French national grid for research, which contains 8,596 processors deployed on 11 sites distributed over France. In our experiments, we used the *Griffon* cluster at Nancy, composed of 184 Intel Xeon L5420 (Quadcore, 2.5GHz, 12MB of L2-cache, bus frequency at 1333MHz), thus with a total of 736 processes available giving a peak performance of 7.36TFlops. The following table (Table 1) shows the minimum, mean, median, maximum and standard deviation among the runtimes (in seconds) and the number of iterations for our three benchmarks.

	MS 200		AI 700		Costas 21	
	time	iterations	time	iterations	time	iterations
Min	5.5	6,210	23.3	1,217	6.6	321,361
Mean	382.0	443,970	1,354.0	110,393	3,744.4	183,428,617
Median	126.3	164,042	945.4	76,242	2,457.4	119,667,588
Max	7,441.6	7,895,872	10,243.4	826,871	19,972.0	977,709,115
Std Dev	873.0	933,766	1,363.4	111,352	3,655.5	179,049,696

Table 1 Sequential executions in seconds and number of iterations

One can see from Table 1 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.

Table 2 presents the speed-up for the runtime and the number of iterations up to 256 processors 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. The speed-up for a given parallel execution is calculated against the mean performance of its sequential version as follows: $Speed\text{-}Up = \frac{Mean(Solver\ on\ 1\ processor)}{Mean(Solver\ on\ N\ processors)}$

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

[2] `http://www.csplib.org`

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 [16].

Problem instance		on 1 proc.	runtime on k processors				
			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 2 Parallel Speed-ups time and number of iterations.

For Magic-Square and All-Interval one can observe the stabilization point is not yet obtained for 256 processor, even if speed-ups do not increase as fast as the number of processors, *i.e.,* are getting further away from linear speed-up. For the Costas Array Problem, our algorithm reaches linear or even super-linear speed-ups for up to 256 processors. Actually, it scales linearly far beyond this point, *i.e.,* at least up to 8,192 processors, as reported in [24]. 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.

4.1.1 The All-Interval *Series Problem*

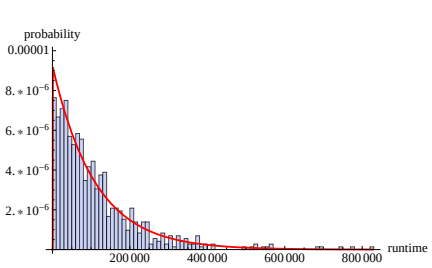
The analysis is done on 720 runs of the Adaptive Search algorithm on the instance of All-Interval series for 700 notes. The sequence of observations is written AI 700 in the following.

We test the hypothesis that the observations admit a shifted exponential distribution as introduced in Section 3.4. 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.[3] 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 $\lambda = 9.15956 \cdot 10^{-6}$.

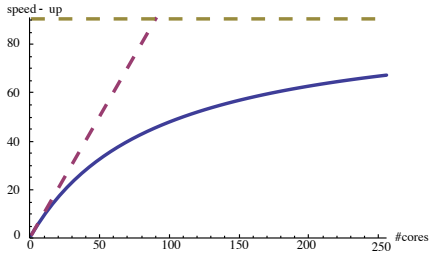
We then run the Kolmogorov-Smirnov test on the shifted exponential distribution with these values of x_0 and λ, which answers positively (computed p-value: 0.77435). We thus admit the hypothesis that AI 700 fits this shifted exponential distribution. As an illustration, Fig. 6(a) shows the normalized 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 multi-walk parallel scheme. We use the formulas of Section 3.4 with the estimated parameters and obtain a theoretical expression for the

[3] The Notation is the same as in Section 3.



(a) Histogram of sequential behaviour



(b) Predicted speedup with its limit yellow (dashed), and the ideal speedup in pink (dashed)

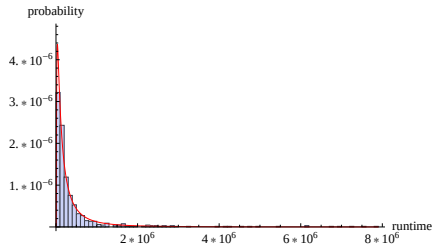
Fig. 6 Analysis of AI 700

speed-up. This allows us to calculate its value for different numbers of processors.

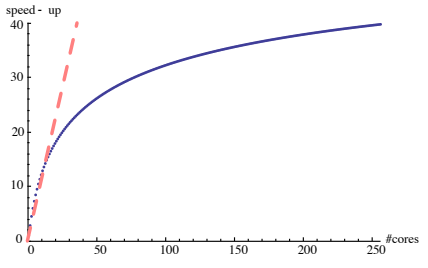
The results are given in Fig. 6(b). With this approximated distribution, the limit of the speed-up when the number of processors goes to infinity is 90.7087. One can see that, with 256 processors, the curve has not reached its limit, but comes close. Thus, the speed-up for this instance of All-Interval appears significantly less than linear (*i.e.,* less than the number of processors).

4.1.2 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, Fig. 7(a) shows the observations and the theoretical estimated distribution.



(a) Histogram of sequential behaviour



(b) Predicted speedup with the ideal speedup in pink (dashed)

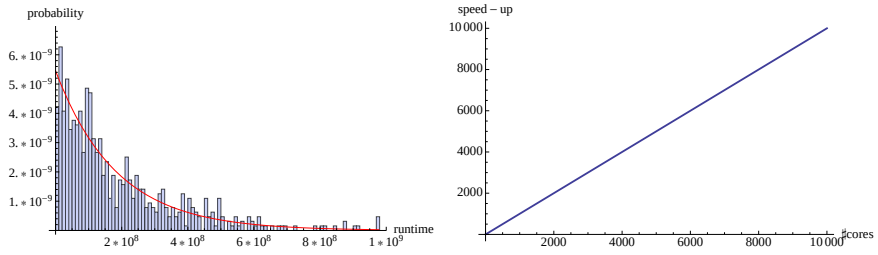
Fig. 7 Analysis of MS 200

The speed-up can be computed by integrating the minimum distribution with numerical integration techniques. The results are presented in Fig. 7(b). 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 processors onwards, and the limit of the speed-up when the number of processors goes to infinity is about 71.5.

4.1.3 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, $3.2 \cdot 10^5$ is negligible compared to its mean ($1.8 \cdot 10^8$). Thus, we estimate $x_0 = 0$ and perform a Kolmogorov-Smirnov test for a (non-shifted) exponential distribution, with $\lambda = 1/\text{mean}(Costas\ 21) = 5.4 \cdot 10^{-9}$. The test is positive for this exponential distribution, with a p-value of 0.751915. Fig. 8(a) shows the estimated distribution compared to the observations.



(a) Histogram of sequential behaviour (b) Predicted speedup

Fig. 8 Analysis of Costas 21

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 3.4. The results are given on Fig. 8(b). This explains that one may observe linear speed-up when parallelizing Costas Array.

4.2 Analysis and Discussion

Table 3 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 processors and then the divergence is limited even for 256 parallel processors.

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

Instance		speed-up on k processors				
		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
Costas21	experimental	15.8	26.4	60.0	159.2	290.5
	predicted	16.0	32.0	64.0	128.0	256.0

Table 3 Comparison: experimental and predicted speed-ups

It is worth noting 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 speed-up, up to 10,000 processors and beyond, and the experimental data gathered for this paper confirms this linear speed-up up to 256 processors. Would it scale up with a larger number of processors? Indeed we did such an experiment up to 8,192 processors on the JUGENE IBM Bluegene/P at the Jülich Supercomputing Center in Germany (with a total 294,912 processors), and reported it in [24], the speed-up is linear up to 8,192 processors, thus showing an excellent fit between the prediction model and real data

5 Application to SAT Local Search

Let us now look at a different problem domain (SAT - the Satisfiability Problem for Boolean formulas) and different local search solvers.

In this section we focus our attention on two well-known problem families: random and crafted instances[4]. Moreover, we consider the two best Local Search solvers from the 2012 SAT competition[5]: CCASAT [15] and Sparrow [11]. Both solvers were used with their default parameters and with a timeout of 3 hours for each experiment. All the experiments were performed on the Grid'5000 platform, the French national grid for research. We used a 44-node cluster with 24 cores (2 AMD Opteron 6164 HE processors at 1.7 Ghz) and 44 GB of RAM per node.

We performed experiments with two well-known problem families of instances coming from the SAT'11 competition: random and crafted. In particular, we used 10 random instances (6 around the phase transition and 4 outside the phase transition) and 10 crafted instances. Complete details of the selected instances for the evaluation is available at [9], hereafter we denote random instances as Rand-[1 to 10] and crafted instances as crafted-[1 to 9].

In order to obtain the empirical data for the theoretical distribution (predicted by our model from the sequential runtime distribution), we performed 500 runs of the sequential algorithm. The Mathematica software [56], version 8.0, was used to estimate the parameters of the theoretical distributions and to integrate numerically the formulas of the lognormal distribution. In order to evaluate the accuracy of the learned statistical model, we performed 50 runs of the multi-walk parallel algorithms.

5.1 Analysis of the Parallel Speed-ups

Let us now present the empirical and predicted results for random and crafted instances. We will not perform here again a detailed analysis as in Section 3.6, but rather report directly the computed p-value when comparing the empirical runtime distribution to either a shifted exponential distribution and a lognormal distribution.

5.1.1 Random instances

We start our analysis with Table 4, which presents initial statistics for the sequential version of Sparrow and CCASAT. We present the minimum, maximum, and mean runtime values, as well as the outcome of the Kolmogorov-Smirnov (KS) test for two types of distributions: shifted exponential and lognormal. In the following tables, bold numbers indicate the distribution chosen to predict the performance of a given solver. As it can be observed for a few instances, the best distribution does not pass the KS test, but we still use the distribution with the best KS value for predicting the runtime of the parallel solver. .

[4] We also experimented with industrial instances but the tested LS solvers performed poorly and a very limited number of instances could be solved with a reasonable time limit.

[5] http://baldur.iti.kit.edu/SAT-Challenge-2012/results.html

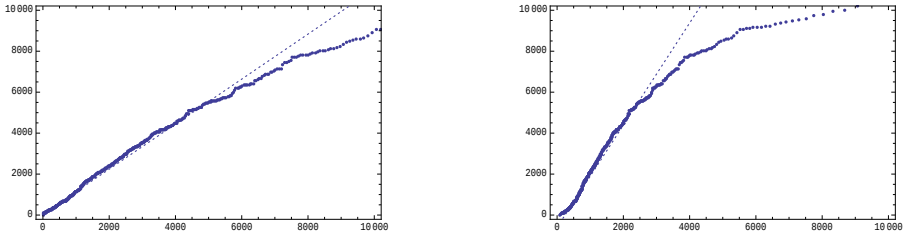
Instance	Alg	Min	Max	Mean	p-value	
					Shifted exp. dist.	Lognormal dist.
Rand-1	Sparrow	98.0	4860.0	793.9	$5.6{\cdot}10^{-19}$	**0.76**
	CCASAT	103.6	1340.1	458.5	$8.4{\cdot}10^{-55}$	**0.96**
Rand-2	Sparrow	91.8	5447.0	1007.5	$1.8{\cdot}10^{-20}$	**0.91**
	CCASAT	108.8	1652.9	497.9	$5.1{\cdot}10^{-58}$	**0.71**
Rand-3	Sparrow	104.8	3693.7	797.0	$2.9{\cdot}10^{-26}$	**0.64**
	CCASAT	126.48	1125.4	359.6	$2.3{\cdot}10^{-108}$	**0.90**
Rand-4	Sparrow	162.4	3037.8	781.5	$1.6{\cdot}10^{-38}$	**0.03**
	CCASAT	132.5	980.9	382.0	$1.5{\cdot}10^{-117}$	**0.92**
Rand-5	Sparrow	164.0	7946.3	952.4	$1.8{\cdot}10^{-31}$	**0.16**
	CCASAT	158.6	1177.9	403.1	$5.1{\cdot}10^{-134}$	**0.20**
Rand-6	Sparrow	142.0	4955.8	763.5	$1.6{\cdot}10^{-31}$	**0.64**
	CCASAT	142.9	890.9	354.1	$9.6{\cdot}10^{-137}$	**0.46**
Rand-7	Sparrow	35.5	10637.4	3464.2	**0.01**	$1.5{\cdot}10^{-4}$
	CCASAT	61.6	6419.1	1801.0	$1.0{\cdot}10^{-5}$	**0.13**
Rand-8	Sparrow	23.2	10738.0	3412.9	**0.05**	$4.7{\cdot}10^{-4}$
	CCASAT	35.9	10443.7	2007.6	**0.50**	0.03
Rand-9	Sparrow	6.8	5935.8	1028.2	**0.23**	$3.7{\cdot}10^{-3}$
	CCASAT	18.1	2830.4	476.8	$7.0{\cdot}10^{-3}$	**0.03**
Rand-10	Sparrow	19.0	10800.0	1726.3	**0.65**	0.15
	CCASAT	19.8	4854.5	758.4	$7.6{\cdot}10^{-10}$	**0.18**

Table 4 Performance of sequential algorithms on random instances

The results presented in this table are consistent with the results of the 2012 SAT competition (random category) where CCASAT greatly outperformed Sparrow. For this set of instances, we choose the shifted exponential distribution in lieu of the exponential distribution as the Min runtime value for the reference solvers is not negligible compared to its mean value across 500 executions (about 100 times smaller in the best case).

As can be seen from the table, both solvers report a tendency which indicates that the empirical data for instances around the phase transition are better approximated by a lognormal distribution; all these instances pass the KS test with a confidence level (p-value) above 0.05, except for Sparrow on rand-4. Alternatively when the KS test is not enough to decide the most appropriate distribution, we use the Q-Q plot to decide whether a given distribution fits the empirical data. The closer the empirical data is to the reference line (theoretical distribution), the better the fit. Fig. 9 shows the Q-Q plot for Sparrow to solve rand-7, unlike Fig. 9(b) where a clear deviation between the empirical data and the lognormal distribution is observed, Fig. 9(a) shows that the empirical distribution is very close to the shifted exponential distribution. We have performed the same analysis for the remaining random instances and observed a similar behaviour between the empirical data and the reference distributions

It is well recognized in the SAT community that k-SAT instances around the phase transition (*i.e.,* No of clauses/No of Variables = 4.2) are known to be difficult [27]. Taking this into consideration, for instances outside the



(a) Seq. vs. Exp. dist. (b) Seq. vs. Log-normal dist.

Fig. 9 Q-Q Plot Analysis of Sparrow on rand-7

phase transition, Sparrow reports enough statistical evidence to infer that the shifted exponential distribution fits better the empirical data. For CCASAT, 3 out of 4 instances outside the phase transition are better characterized with a lognormal distribution and the remaining instance pass the KS test for the shifted exponential distribution.

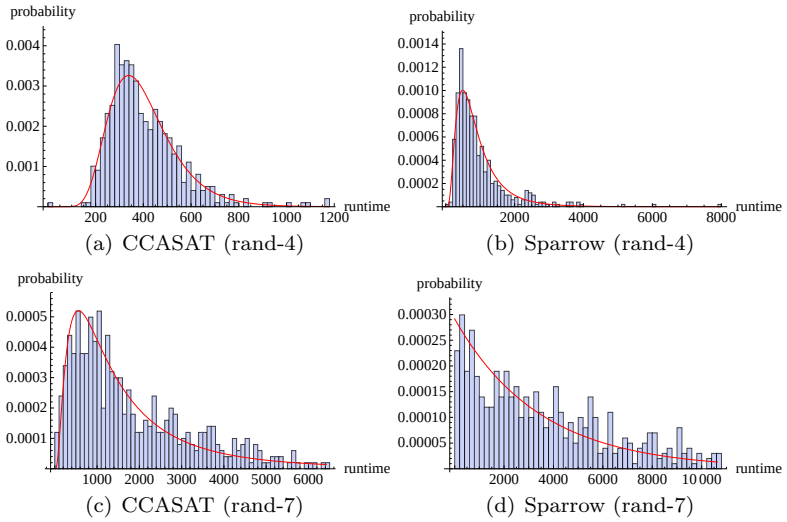
Instance		Sparrow - Runtime on k proc.				CCASAT - Runtime on k proc.			
		48	96	192	384	48	96	192	384
Rand-1	Actual	163.8	140.4	125.2	113.7	160.0	143.0	122.8	112.0
	Predicted	133.8	110.5	92.7	78.8	137.7	120.6	106.7	95.3
Rand-2	Actual	213.2	191.4	166.2	142.5	186.8	169.3	159.3	142.8
	Predicted	183.5	152.8	129.2	110.6	153.4	134.7	119.6	107.1
Rand-3	Actual	175.9	151.2	135.8	123.5	166.7	155.6	143.5	132.2
	Predicted	183.5	152.8	129.2	110.6	153.4	134.7	119.6	107.1
Rand-4	Actual	202.3	179.2	159.5	141.8	193.1	176.0	169.4	158.7
	Predicted	175.7	149.5	128.9	112.4	170.6	155.9	143.5	132.8
Rand-5	Actual	219.6	201.0	182.5	161.9	212.2	191.3	176.8	165.8
	Predicted	185.0	155.3	132.3	114.0	179.8	164.3	151.2	140.0
Rand-6	Actual	185.5	167.1	150.3	137.5	190.9	179.3	168.4	153.4
	Predicted	158.3	133.6	114.4	99.1	160.6	147.0	135.4	125.6
Rand-7	Actual	151.2	102.7	63.8	51.1	22.9	33.7	54.3	67.8
	Predicted	195.8	143.0	107.3	82.3	182.8	142.6	113.7	92.2
Rand-8	Actual	126.6	81.9	51.1	30.9	131.8	83.9	64.8	39.7
	Predicted	93.8	58.5	40.8	32.0	76.9	56.4	46.1	41.0
Rand-9	Actual	33.9	18.4	13.1	9.0	45.0	31.0	22.7	16.3
	Predicted	28.1	17.4	12.1	9.4	38.5	29.4	23.0	18.3
Rand-10	Actual	63.4	48.9	40.7	30.9	113.8	94.7	72.9	54.2
	Predicted	54.6	36.8	27.9	23.4	105.6	85.3	70.2	58.6

Table 5 Runtimes for random instances up to 384 proc. (processors)

We now look at the parallel performance of the solvers. Table 5 shows the empirical and predicted runtime for both Sparrow and CCASAT on all instances using 48, 96, 192, and 384 processors. Detailed tables with the speed-up for Sparrow and CCASAT are available in [9]. Summing up both solver report the same tendency as the empirical data. Furthermore, the speed-up

factor of the references far from linear (ideal), a phenomenon well described by the predicted model.

Fig. 10 depicts observations and the theoretical estimated distribution, and Fig. 11 shows a performance summary of the reference solvers to tackle an instance on the phase transition (rand-4) and another instance outside the phase transition (rand-7). The *y-axis* gives the probability ($\mathbb{P}[Y \leq x]$) of finding a solution in a time less or equal to x and the *x-axis* gives the runtime in seconds. From now on, in all figures 'Emp' stands for Empirical distribution, 'LN' stands for lognormal distribution, and 'SExp' stands for shifted exponential distribution. As expected CCASAT dominates the performance on one processor. For example to solve rand-4, CCASAT reports $\mathbb{P}[Y \leq \textit{16-mins}] \approx 1.0$, while Sparrow reports $\mathbb{P}[Y \leq \textit{16-mins}] \approx 0.75$. Interestingly, we observe that for CCASAT increasing the number of processors does not significantly improve the solving time. Consequently, Sparrow becomes more effective for a large number of processors. Therefore, Fig. 11(b) and 11(d) show that Sparrow is better than CCASAT when using 384 processors. Interestingly, the same pattern is observed for other random instances (see Table 5).

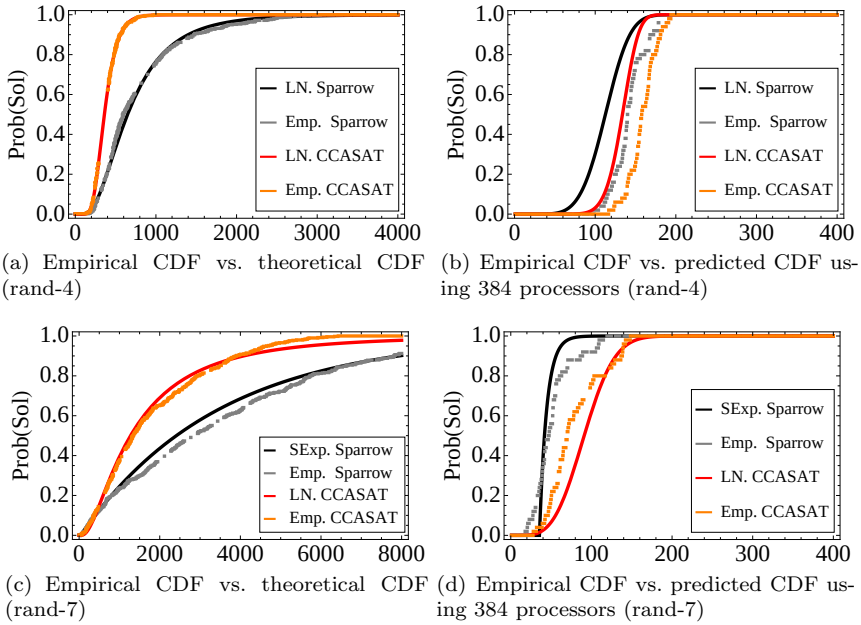


(a) CCASAT (rand-4) (b) Sparrow (rand-4)

(c) CCASAT (rand-7) (d) Sparrow (rand-7)

Fig. 10 Histogram for the observed runtimes for CCASAT and Sparrow to solve rand-4 and rand-7

Finally, it can also be observed that random instances around the phase transition exhibit a lower speed-up factor than the remaining random instances. For instance, the best empirical speed-up factor obtained for instances in the phase transition is 7.0 for Sparrow and 3.4 for CCASAT; and the best speed-up factor obtained for instances outside the phase transition is 114.2 for Sparrow and 50.5 for CCASAT.



(a) Empirical CDF vs. theoretical CDF (rand-4)

(b) Empirical CDF vs. predicted CDF using 384 processors (rand-4)

(c) Empirical CDF vs. theoretical CDF (rand-7)

(d) Empirical CDF vs. predicted CDF using 384 processors (rand-7)

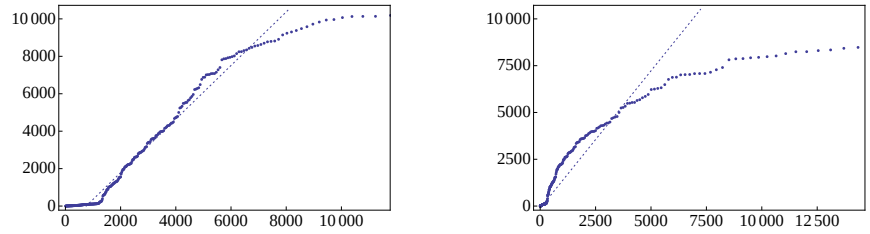
Fig. 11 Performance summary to solve rand-4 and rand-7

5.1.2 Crafted instances

Let us switch our attention now to crafted instances, for which we have to treat CCASAT and Sparrow differently. For CCASAT, we were unable to find a theoretical distribution which fits the empirical data. It should be also noticed that CCASAT has mainly been designed and tuned to handle random instances. Let us look for instance at Fig. 13(a), which depicts the cumulative runtime distribution of CCASAT to solve Crafted-1 using the two reference distributions detailed in this paper (lognormal and exponential) and two extra distributions (Weibull and beta-prime). None of the theoretical distributions seems to be a good approximation of the empirical data. More precisely, the KS test reported a p-value of $2.7{\cdot}10^{-7}$ (lognormal); $7.0{\cdot}10^{-24}$ (exponential); $2.4{\cdot}10^{-6}$ (Weibull); and $6.9{\cdot}10^{-15}$ (beta-prime). Therefore, none of the theoretical distributions pass the KS test with a high-enough p-value. We also experimented with other instances and observed a similar behavior.

Fig. 12(b) and 12(b) display the Q-Q plots between the empirical distribution and two theoretical distribution: exponential and log-normal. Both figures show that the data is neither exponentially nor log-normally distributed.

For Sparrow on all crafted instances, the KS test shows a much better p-value for the exponential distribution than for the lognormal one, see Table 6. The confidence level is quite high for the instances Crafted-2,-3,-4,-5,-8,-9, with p-value up to 0.97, while the p-value is between 0.01 and 0.02 for Crafted-1,-6,-7. Also, as the minimum runtime is much smaller than the mean (at least



(a) Seq. vs. Exp. dist. (b) Seq. vs. Log-normal dist.

Fig. 12 Q-Q Plot Analysis of CCASAT on crafted-1

Instance	Alg	Min	Max	Mean	p-value	
					Exp. dist.	Lognormal dist.
Crafted-1	Sparrow	9.9	10800.0	3440.3	**0.02**	$1.2{\cdot}10^{-4}$
Crafted-2	Sparrow	1.0	10800.0	2711.2	**0.57**	$1.4{\cdot}10^{-4}$
Crafted-3	Sparrow	8.7	10800.0	3432.7	**0.14**	$1.1{\cdot}10^{-3}$
Crafted-4	Sparrow	2.2	10800.0	2701.6	**0.11**	$9.6{\cdot}10^{-3}$
Crafted-5	Sparrow	4.1	10800.0	1564.1	**0.95**	$9.2{\cdot}10^{-4}$
Crafted-6	Sparrow	2.9	10800.0	3599.6	**0.01**	$1.0{\cdot}10^{-5}$
Crafted-7	Sparrow	4.4	10800.0	3598.7	**0.01**	$7.8{\cdot}10^{-6}$
Crafted-8	Sparrow	3.5	5456.0	972.046	**0.67**	0.17
Crafted-9	Sparrow	1.9	7876.5	1298.24	**0.97**	$7.0{\cdot}10^{-3}$

Table 6 Sequential performance of Sparrow on crafted instances

300 times smaller), we can approximate the empirical data by a non-shifted exponential distribution [51].

As can be seen in Table 7 the multi-walk parallel approach scales well for Sparrow on crafted instances as the number of processors increases. Indeed a nearly linear speed-up is obtained for nearly all the instances. As expected, the speed-up predicted by our model is optimal, and this result is consistent with those obtained in [33].

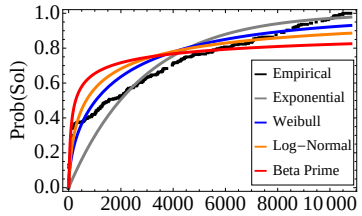
5.2 Analysis and Discussion

Several works have been devoted to the experimental study of parallel multi-walk extensions of Local Search algorithms [6, 7, 32], but we presented here the first approach (to our knowledge) which applies order statistics in order to predict the parallel performance of Local Search algorithms for SAT. As in Section 4.2, the lognormal or shifted-exponential distributions need to be considered for the runtime distribution, and not only the exponential distribution as suggested by most of the literature.

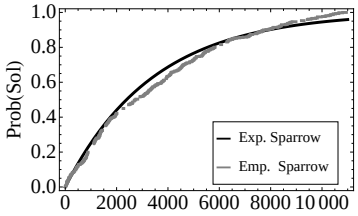
Interestingly, the phase transition point of SAT instances also seems to have important consequences in the parallel performance of Local Search al-

Instance		Runtime on k proc.				Speed-Up ok k proc.			
		48	96	192	384	48	96	192	384
Crafted-1	Actual	97.7	43.7	19.1	9.8	35.1	78.6	179.6	349.8
	Predicted	71.6	35.8	17.9	8.9	48.0	96.0	192.0	384.0
Crafted-2	Actual	67.8	36.4	17.5	7.2	39.9	74.4	154.7	375.2
	Predicted	56.4	28.2	14.1	7.0	48.0	96.0	192.0	384.0
Crafted-3	Actual	94.8	49.3	23.2	11.9	36.1	69.6	147.6	286.1
	Predicted	71.5	35.7	17.8	8.9	48.0	96.0	192.0	384.0
Crafted-4	Actual	87.5	42.0	17.3	9.7	30.8	64.2	155.4	277.8
	Predicted	56.2	28.1	14.0	7.0	48.0	96.0	192.0	384.0
Crafted-5	Actual	33.7	15.1	7.6	4.2	46.3	103.2	204.1	371.6
	Predicted	32.5	16.2	8.1	4.0	48.0	96.0	192.0	384.0
Crafted-6	Actual	130.0	69.8	25.6	12.8	27.6	51.5	140.5	279.5
	Predicted	74.9	37.4	18.7	9.3	48.0	96.0	192.0	384.0
Crafted-7	Actual	95.0	51.3	28.4	11.6	37.8	70.0	126.3	308.0
	Predicted	74.9	37.4	18.7	9.3	48.0	96.0	192.0	384.0
Crafted-8	Actual	17.2	10.8	5.3	2.6	56.4	89.6	181.1	363.6
	Predicted	20.2	10.1	5.0	2.5	48.0	96.0	192.0	384.0
Crafted-9	Actual	27.2	12.1	5.9	3.6	47.5	106.6	217.3	358.0
	Predicted	27.0	13.5	6.7	3.3	48.0	96.0	192.0	384.0

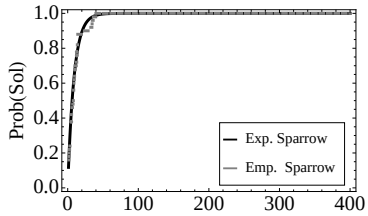
Table 7 Parallel performance of Sparrow on crafted instances (proc. stands for processors)



(a) Empirical CDF vs. theoretical CDDs (CCASAT)



(b) Empirical CDF vs. theoretical CDF (Sparrow)



(c) Empirical CDF vs. predicted CDF using 384 processors (Sparrow)

Fig. 13 Performance summary on crafted-1

gorithms. For Sparrow at least, which is the solver with an overall best speed-up factor, instances around the phase transition region are lognormally distributed, while instances outside the phase transition are shifted-exponentially distributed. As for Constraint-Based Local Search, the probability of returning a solution in *no* iterations is not zero in theory for SAT Local Search. How-

ever, in practice a minimum number of steps is in general required to reach a solution cf. [33, 49] for the sequential case. Therefore, experimental data may be better approximated by a shifted distribution with $x_0 > 0$: this is the case for the random instances. This leads to a non-linear speed-up with a finite limit, even in the case of an exponential distribution. Indeed, the experimental speed-up for both CCASAT and Sparrow on random instances is far from linear. On the contrary, Sparrow on crafted instances has a linear speed-up which could be explained by the fact that the minimal runtime is negligible w.r.t. the mean time (*i.e.,* $1/\lambda$ for an exponential distribution). Therefore, the statistical test succeeds for $x_0 \simeq 0$. This suggests that, in general, the comparison between the minimal time and the mean time is a key element for the study of the parallel behavior.

We do not discard that other parameters for the reference solvers would lead to other theoretical distributions (*e.g.,* exponential distribution for random instances). In [37] the authors showed that a well-tuned version of WalkSAT is exponentially distributed for instances in the phase transition region. However, we experimented by increasing the *ps* (smoothing probability) parameter of Sparrow and still obtained the same theoretical distribution. In addition, when *ps* is too high the solver was unable to solve the instances within the 3 hour time limit. Unfortunately, CCASAT is only available in binary form, and it is not possible to experiment with other parameters for the solver.

We expect this work to have significant implications in the area of automatic parameter tuning to devise scalable Local Search algorithms. Currently, most parameter tuning tools (*e.g.,* [34, 5]) are designed to improve the expected mean (or median) runtime, however as observed in this paper, unless the algorithms exhibit a non-shifted exponential distribution, their parallel performance is far from linear and varies from algorithm to algorithm.

6 Application to Propagation-based Constraint Solving

In the previous two sections we have been studying the parallel performance of incomplete algorithms based on Local Search. In this section, we shift our focus to complete algorithms to tackle CSPs. Broadly speaking, complete algorithms employ a combination of domain reduction and tree-based search method, in which at the root node all variables are associated with a given domain. At each step of the search a value is assigned to some variable. Each assignment is preceded by a look-ahead process called constraint propagation which reduces the domains of the variables. The search process is stopped when a solution has been obtained or when the complete tree has been explored. In particular, we consider *wdeg* [14] and *min-dom* [31] state-of-the-art variable selection heuristics that focus on difficult parts of the search.

6.1 Experimental Protocol and Benchmarks

In order to perform the following experimentation we equipped Gecode[6] with the multi-walk framework. We also used a randomized version of the solver. At each node, we use *wdeg* and *min-dom* to select the most appropriate variable. The former selects the variable with the largest *weighted degree*[7] value is selected (breaking ties at random), the latter selects the variables with the smallest valid domain at a given state of the search. The value for the chosen variable is selected uniformly at random. It is highly recognized that the runtime distribution for tree-based search methods for a large number of problems is heavy-tailed [29] and adding restarts (*i.e.,* stopping the search after a given number of backtracks, and re-initializing it) helps to alleviate the heavy-tailed phenomenon. In this paper, we use a fixed restart strategy where the restart cutoff is 250 backtracks. For each problem instance we compute the sequential performance using *wdeg* and *min-dom*, and use the heuristic with better performance (in sequential settings) to evaluate the model described above to estimate the performance of the solver in parallel.

In these experiments, we use the same computer settings as for the previous section and considered the same baseline problems as for the adaptive search framework in Section 4, that is, Magic-Square, All-Interval, and Costas Array. For Magic-Square and All-Interval we used the model provided in the Gecode distribution (version 4.2.1), and the Gecode version of Costas Array is available at `http://www.hakank.org/gecode/costas_array.cpp`. It is worth noting that the All-Interval model for Gecode allows the trivial solution (0, n-1, n-2, n-3, ...), and such a solution is obtained without backtracking. Therefore, we removed the trivial solution in the model by only allowing positive numbers for the first variable in the list, *i.e.,* adding the constraint $X_0 > 0$.

6.2 Analysis of the Parallel Speed-ups

Table 8 shows the runtime of the sequential version. We present the minimum, maximum, mean, and the p-value for the reference line distributions of the sequential runs for Magic-Square 15, All-Interval 18, Costas Array 17. Highlighted numbers indicate the best heuristic w.r.t. Mean value, this heuristic will be used later on for the parallel evaluation of the solver. The Kolmogorov-Smirnov test suggest that the sequential execution for Magic-Square can be characterized using a shifted exponential distribution with a p-value of 0.21 (Magic-Square 15 with *wdeg*), All-Interval can be characterized using an exponential distribution with a p-value of 0.75 (All-Interval 18 with *min-dom*)[8], and Costas Array can be characterized using a log-

[6] www.gecode.org

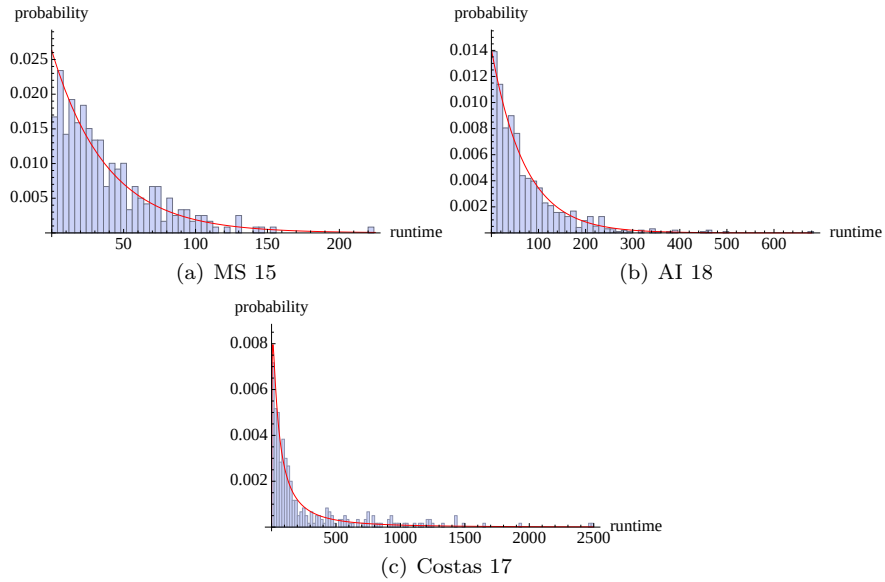
[7] We use AFC, the Gecode implementation of *wdeg*

[8] We use the exponential distribution in lieu of the shifted exponential because the min value is negligible w.r.t. to the mean 0.008 vs. 58.08.

normal distribution with a p-value 0.04 (COSTAS ARRAY 17 with *min-dom*). It is worth pointing out that neither MAGIC-SQUARE nor ALL-INTERVAL passed the KS test for the log-normal distribution, and COSTAS ARRAY did not pass KS for the (shifted/and not shifted) exponential distribution. Once again, we use the formulas of Section 3.3 with λ=1/mean and x_0 is set to the minimal value of the empirical data, and used Mathematica to estimate the parameters of the log-normal distribution for COSTAS ARRAY. Hereafter we use the indicated distributions for each algorithm to build the model and estimate the parallel performance in solver.

Instance	Alg	Min	Max	Mean	p-value	
					shifted exp. dist.	Lognormal dist.
MS-15	*wdeg*	0.200	222.86	**38.20**	0.21	$8.6{\cdot}10^{-3}$
	min-dom	0.891	3600	488.55	0.78	0.02
AI-18	*wdeg*	0.027	677.20	76.09	0.24	$7.3{\cdot}10^{-4}$
	*min-dom**	0.008	264.64	**58.08**	0.75	$4.1{\cdot}10^{-3}$
Costas-17	*wdeg*	0.177	2481.45	272.86	$2.1{\cdot}10^{-10}$	0.68
	min-dom	0.262	332.58	**40.39**	0.04	$1.4{\cdot}10^{-3}$

Table 8 Sequential executions in seconds for complete search. *Indicates that for AI-18 with *min-dom* we use non-shifted distributions because the min value is considerably smaller than the mean



(a) MS 15 (b) AI 18

(c) Costas 17

Fig. 14 Histogram for the observed runtimes for complete search

Tables 9 reports the estimated runtime (top) and speed-up (bottom) of the multi-walk version of the propagation-based constraint solving algorithm. Interestingly, this simple parallel scheme exhibits very good speedup factors for the three benchmarks: super-linear for Magic-Square with 48, 96, cores, linear for nearly all experiments for All-Interval, and a factor 122 (w.r.t. the sequential solver) for Costas Array with 384 cores.

6.3 Analysis and Discussion

We recall that the goal of this section is not to compare incomplete methods based on Local Search and complete methods based on tree search. Instead, we intend to show the application of the proposed framework to a large variety of randomized techniques to solve combinatorial problems. Interestingly, even though the search strategy used in these experiments largely differs from the experiments presented in the two previous sections, we have observed that the empirical data can be characterised according to a theoretical distribution, ie shifted exponential (Magic-Square and All-Interval) and log-normal (Costas Array) distributions, and accurately estimate the runtime of the parallel algorithm.

The runtime prediction is very close to the empirical data for the three problems with gap between the prediction and the actual execution time (Table 8) of up 13.7% for Magic-Square with 384 cores, 14% for All-Interval with 92 cores, and 8% for Costas Array with 384 cores. Furthermore, the predictions of the speedup of the solvers are very encouraging, for the Magic-Square problem it can be observed that the gap between the actual and estimated speedup is up to 13% with 24 cores, for the All-Interval problem is up to 12% with 96 cores, and for Costas Array is up to 4% with 192 cores. Finally, we would like to point out that this methodology requires independently and identically distributed executions of the sequential algorithm, so that, parallel algorithms based on search-splitting techniques [13, 18, 42] and cooperative techniques [6] do not fit within the proposed framework.

Finally, we would like to point out that we also experimented with the losing technique in sequential settings (see Table 8) and also found out very accurate predictions of up to 5% for Magic-Square with *min-dom*, up to 4% for All-Interval with *wdeg*, and up to 2% for Costas Array with *wdeg*.

7 Empirical Estimation

In this section we study a method that only uses the empirical sequential distribution to estimate the parallel runtime distribution. This method is much simpler than our model, as it does not involve many statistical tools. Our goal is to compare this empirical method to our model.

Let $\{k_1, ..., k_m\} \in K$ be the set of m sequential executions of a given randomised algorithm to solve a given instance. We assume K to be sorted,

Problem	1 proc.		performance (time/speed-up) on k proc. 24	48	96	192	384
MS 15	38.20	Actual	2.06	0.66	0.47	0.31	0.25
			18.546	57.87	81.27	123.22	152.8
		Predicted	1.78	0.99	0.59	0.39	0.29
			21.46	38.58	64.74	97.94	131.72
AI 18	58.08	Actual	3.54	1.56	0.69	0.42	0.28
			16.41	37.23	84.17	138.29	207.43
		Predicted	2.42	1.21	0.60	0.30	0.15
			24.00	48.00	96.00	192.00	384.00
Costas 17	40.39	Actual	3.41	1.64	0.93	0.49	0.33
			11.83	24.62	43.43	82.42	122.39
		Predicted	1.93	1.09	0.68	0.47	0.36
			20.92	37.05	59.39	85.93	112.191

Table 9 Parallel runtimes in seconds (proc. stands for processors). Each cell in the performance indicates the runtime (top) and speedup (bottom) for MS with *wdeg*, AI and Costas with *min-dom*

thus k_i denotes the i^{th} sequential runtime. The associated empirical sequential distribution is the probability distribution defined by assigning a probability $1/m$ to each sequential execution. Therefore the probability of each observation can be defined as $\mathbb{P}[Y \leq t_i] = i/m$. Considering the min distribution rule of Section 3.2 we compute the empirical parallel runtime distribution $\mathcal{F}_{Z^n}$ as follows:

$$\begin{aligned} \mathbb{P}[Z^{(n)} \leq t_i] &= 1 - (1 - \mathcal{F}_Y(t_i))^n \text{ Min Distribution, Section 3.2} \\ &= 1 - \left(1 - \tfrac{i}{m}\right)^n \quad \text{by definition} \end{aligned}$$

Figure 15 shows the assigned probabilities of each element in the sorted set of sequential observations, the y-axis shows the probability of selecting each $k_i \in K$ (x-axis) for the sequential algorithm with 1 processor,and the derived probabilities for the empirical parallel runtime distribution with 2, 4, 8, 16, and 32 processors. As expected, increasing the number of processors also increases the chances of selecting short runs.

Hereafter, we refer to prediction method for our model based on a theoretical distribution (e.g., lognormal distribution) as shown in Sections 4, 5, and 6, and we use the term *estimation method* to refer to the simpler method described here. Certainly, a key factor for the accuracy of both methods is the number of sequential observations. We are interested in the robustness of both methods w.r.t. the number of observations (m) and the number of cores (n). We thus show experiments where we vary the number of sequential observations to build the probability distribution model. Table 10 shows the average discrepancy across all instances with 48, 96, 192, and 384 cores[9]. We use a baseline of 500 instances for Sparrow (random and crafted), CCASAT and the Adaptive Search, and 300 instances for experiments with the Gecode solver and gradually decrease the number of observations m by considering 50%, 25%, and 12% of the sequential data.

[9] For experiments with the Adaptive Search we use report values with 32, 64, 128, and 256 cores

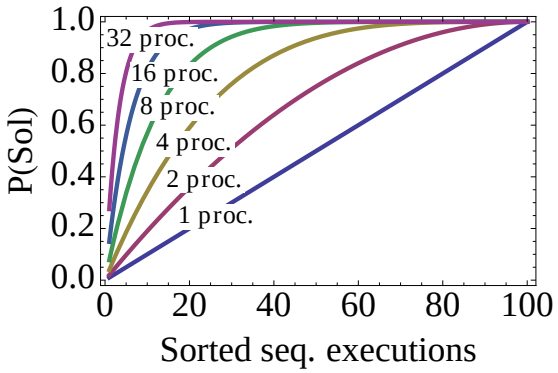
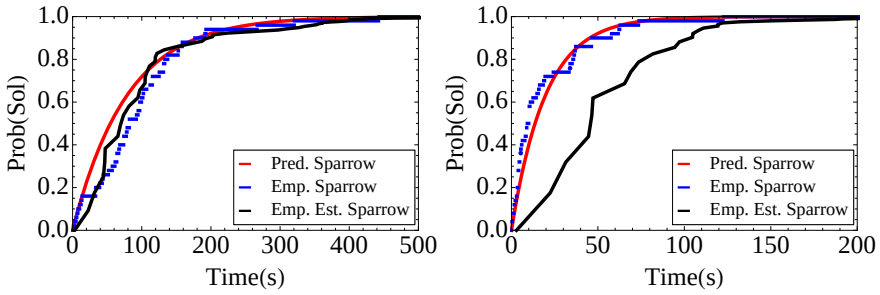


Fig. 15 Probability of selecting each element $k_i \in K$, for $|K|$=100

Cores	Method	Discrepancy			
		100%	50%	25%	12%
48	EE	**17.92**	**19.04**	33.71	48.25
	PE	25.66	21.80	**23.67**	**27.79**
96	EE	**18.36**	24.40	47.58	51.62
	PE	25.45	**20.11**	**25.17**	**31.50**
192	EE	**22.64**	33.33	50.91	65.04
	PE	23.90	**22.76**	**27.06**	**31.89**
384	EE	28.22	40.03	52.89	67.05
	PE	**21.89**	**24.86**	**28.25**	**29.29**
ALL	EE	**21.79**	29.20	46.27	57.99
	PE	24.23	**22.38**	**26.04**	**30.12**

Table 10 Average Discrepancy across all instances w.r.t. the actual solution for the empirical evaluation (EE) and the predicted evaluation (PE) varying the sequential executions and number of cores

As it can be observed in the table, the empirical evaluation is a robust alternative when a large enough number of observations is available. In particular, we observe that the empirical method slightly outperforms our statistical model in 3 out of 4 scenarios when considering 100% of the sequential data. On the other hand, our statistical model reports better results in 11 out of the 12 scenarios when considering 50%, 25% and 12% of the data. Interestingly, the difference between the two prediction models is more significant as we increase the number of cores and reduce the number of observations. For instance, when using 12% of the data and 384 cores we observe that the discrepancy w.r.t. the actual data is 67.05% for empirical estimations and only 29.29% for the prediction model. In addition, summing up all the experiments (last row in Table 10), we observe that the quality of the prediction model is more stable and better than the estimation method.



(a) 48 cores (b) 192 cores

Fig. 16 Predicted, Empirical, and Empirical Estimated probability distributions of Sparrow to solve crafted-1

Fig. 16 shows a comparison using 100% of the sequential execution for the prediction, empirical[10], and empirical estimation methods for Sparrow to solve crafted-1 with 48 and 192 processors[11]. As it can be observed in this figure the empirical estimation greatly matches the performance with 48 processors (Fig. 16(a)). However, the quality considerably degrades when using 192 processors (Fig. 16(b)), for instance, the empirical estimation suggests that the probability of finding a solution in 50 seconds for Sparrow with 192 processors is $\mathbb{P}[t \leq 50] \approx 60\%$ vs. $\mathbb{P}[t \leq 50] \approx 90\%$ for the actual performance in parallel settings, that is, about 30% discrepancy.

Finally, it can be observed that our prediction model, based in order statistics, closely matches the performance for an important number of processors. It is worth noting that the performance deterioration of the empirical estimation method is even higher when using a small set of sequential observations, as observed in Table 10. Furthermore, the prediction method also allows us to predict the limit of the speedup when the number of processors tends to infinity as shown in Section 3.4 for the exponential distribution. An example of the maximal speedup is depicted for the Magic-Square problem in Section 4.1.1.

The prediciton method thus ensures a more robust accuracy. The intuitive explanation is the following: as explained above, compared to the sequential distribution, the minimum distribution is concentrated close to the origin (or x_0). Thus, when it is approached with the empirical method, only the samples next to the origin count, no matter what the rest of the observations are: only a few samples really count in the model. On the contrary, our method finds a curve which best approximates the whole sequential distribution, leveraging the information at hand. Hence, it is more robust when the number of observations decreases. Notice also that the empirical method does not allow us to compute the theoretical limit for the speed-up.

[10] For empirical we mean the empirical runtime distribution computed with the actual empirical results of the experiments with 48 and 192 processors.

[11] We have performed the same analysis with other instances and observed a similar tendency, i.e., increasing the number of processors degrades the performance of the empirical estimated probability distribution

8 Conclusion and Future Work

We have proposed a theoretical model for predicting and analyzing the speed-ups of Las Vegas algorithms and applied it for Local Search methods in two different domains, Constraint-Based Local Search and SAT, and also for propagation-based constraint solving with random labeling heuristics. Interestingly, we have observed that, for the different algorithms and the variety of instances considered in this study, the runtime distribution can be characterized using two types of distributions: exponential (shifted and non-shifted) and lognormal.

It is worth noting that our model mimics the behaviors of the experimental results very closely, as shown by the predicted speed-ups matching closely the real ones. We showed that the parallel speed-ups predicted by our statistical model are accurate, matching the actual speed-ups very well up to several hundreds of processors.

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 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. Our observations suggest that the sequential runtime of both LS and complete search are well approximated by a small number of distributions. This could be intensively tested on a wider range of problems. In particular, it is important to know wether the sequential distribution of different instances of a given problem belong to the same family. Then we can devise a method for estimating the sequential distribution based on a limited number of observations, possibly on small instances, and then estimate the parallel speed-up for larger instances. This would allow us to predict if a simple multi-walk parallelization scheme, which does not imply to modify the algorithms, is likely to be efficient, or not. In the same spirit, further research includes investigating machine learning techniques to infer the distribution from a set of experiments as small as possible.

Aknowledgments

We acknowledge that some results in this paper have been achieved using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several universities as well as other funding bodies. We acknowledge that some other results in this paper have been achieved using the PRACE Research Infrastructure resource JUGENE based in Germany at Jülich Supercomputing Centre. The

authors would like to thank the anonymous reviewers for their comments and suggestions that helped to improve the paper. We would like to thank the anonymous reviewers for suggesting to compare our method vs. the empirical method depicted in Section 7.

References

1. Aida K, Osumi T (2005) A Case Study in Running a Parallel Branch and Bound Application on the Grid. In: SAINT '05: Proceedings of the The 2005 Symposium on Applications and the Internet, IEEE Computer Society, Washington, DC, USA, pp 164–173
2. Aiex R, Resende M, Ribeiro C (2007) TTT Plots: A Perl Program to Create Time-To-Target Plots. Optimization Letters 1:355–366
3. Aiex RM, Resende MGC, Ribeiro CC (2002) Probability distribution of solution time in grasp: An experimental investigation. Journal of Heuristics 8(3):343–373
4. Alba E (2004) Special Issue on New Advances on Parallel Meta-Heuristics for Complex Problems. Journal of Heuristics 10(3):239–380
5. Ansótegui C, Sellmann M, Tierney K (2009) A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In: Gent IP (ed) 15th International Conference on Principles and Practice of Constraint Programming, Springer, Lisbon, Portugal, LNCS, vol 5732, pp 142–157
6. Arbelaez A, Codognet P (2012) Massivelly Parallel Local Search for SAT. In: ICTAI'12, IEEE Computer Society, Athens, Greece, pp 57–64
7. Arbelaez A, Codognet P (2013) From Sequential to Parallel Local Search for SAT. In: 13th European Conference on Evolutionary Computation in Combinatorial Optimisation (EvoCOP'13), to appear
8. Arbelaez A, Hamadi Y (2011) Improving Parallel Local Search for SAT. In: Coello CAC (ed) Learning and Intelligent Optimization, 5th International Conference, LION'11, Springer, LNCS, vol 6683, pp 46–60
9. Arbelaez A, Truchet C, Codognet P (2013) Using Sequential Runtime Distributions for the Parallel Speedup Prediction of SAT Local Search. Journal Theory and Practice of Logic Programming (TPLP) Special issue, proceedings of ICLP13, to Appear
10. Babai L (1979) Monte-Carlo Algorithms in Graph Isomorphism Testing. Research Report D.M.S. No. 79-10, Université de Montréal
11. Balint A, Fröhlich A (2010) Improving Stochastic Local Search for SAT with a New Probability Distribution. In: Strichman O, Szeider S (eds) SAT'10, Springer, Edinburgh, UK, LNCS, vol 6175, pp 10–15
12. Bolze R, al (2006) Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. Int J High Perform Comput Appl 20(4):481–494
13. Bordeaux L, Hamadi Y, Samulowitz H (2009) Experiments with Massively Parallel Constraint Solving. In: Boutilier C (ed) Proceedings of IJCAI

2009,21st International Joint Conference on Artificial Intelligence, pp 443–448
14. Boussemart F, Hemery F, Lecoutre C, Sais L (2004) Boosting Systematic Search by Weighting Constraints. In: ECAI'2004, pp 146–150
15. Cai S, Luo C, Su K (2012) CCASAT: Solver Description. In: SAT Challenge 2012: Solver and Benchmark Descriptions, University of Helsinki, vol B-2012-2 of Department of Computer Science Series of Publications B, pp 13–14
16. Caniou Y, Diaz D, Richoux F, Codognet P, Abreu S (2012) Performance Analysis of Parallel Constraint-Based Local Search. In: PPoPP 2012, 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press, New Orleans, LA, USA, poster paper
17. Caromel D, di Costanzo A, Baduel L, Matsuoka S (2007) Grid'BnB: A Parallel Branch and Bound Framework for Grids. In: proceedings of HiPC'07, 14th international conference on High performance computing, Springer Verlag, pp 566–579
18. Chu G, Schulte C, Stuckey PJ (2009) Confidence-Based Work Stealing in Parallel Constraint Programming. In: Gent IP (ed) CP 2009, 15th International Conference on Principles and Practice of Constraint Programming, Springer, Lecture Notes in Computer Science, vol 5732, pp 226–241
19. Codognet P, Diaz D (2001) Yet Another Local Search Method for Constraint Solving. In: proceedings of SAGA'01, Springer Verlag, pp 73–90
20. Codognet P, Diaz D (2003) An Efficient Library for Solving CSP with Local Search. In: Ibaraki T (ed) MIC'03, 5th International Conference on Metaheuristics
21. Crainic T, Toulouse M (2002) Special Issue on Parallel Meta-Heuristics. Journal of Heuristics 8(3):247–388
22. Crainic TG, Gendreau M, Hansen P, Mladenovic N (2004) Cooperative Parallel Variable Neighborhood Search for the -Median. Journal of Heuristics 10(3):293–314
23. David H, Nagaraja H (2003) Order Statistics. Wiley series in probability and mathematical statistics. Probability and mathematical statistics, John Wiley
24. Diaz D, Richoux F, Caniou Y, Codognet P, Abreu S (2012) 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, IEEE Press, Shanghai, China
25. Eadie W (1971) Statistical Methods in Experimental Physics. North-Holland Pub. Co.
26. Gendron B, Crainic T (1994) Parallel Branch-And-Bound Algorithms: Survey and Synthesis. Operations Research 42(6):1042–1066
27. Gent IP, Walsh T (1994) The SAT Phase Transition. In: ECAI'94, pp 105–109
28. Gomes CP, Selman B (2001) Algorithm Portfolios. Artificial Intelligence 126(1-2):43–62

29. Gomes CP, Selman B, Crato N, Kautz HA (2000) Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. J Autom Reasoning 24(1/2):67–100
30. Gonzalez T (ed) (2007) Handbook of Approximation Algorithms and Metaheuristics. Chapman and Hall / CRC
31. Haralick RM, Elliott GL (1979) Increasing Tree Search Efficiency for Constraint Satisfaction Problems. In: IJCAI, pp 356–364
32. Hoos H, Stütze T (2005) Stochastic Local Search: Foundations and Applications. Morgan Kaufmann
33. Hoos HH, Stützle T (1999) Towards a Characterisation of the Behaviour of Stochastic Local Search Algorithms for SAT. Artif Intell 112(1-2):213–232
34. Hutter F, Hoos HH, Leyton-Brown K, Stützle T (2009) ParamILS: An Automatic Algorithm Configuration Framework. Journal of Artificial Intelligence Research 36:267–306
35. Ibaraki T, Nonobe K, Yagiura M (eds) (2005) Metaheuristics: Progress as Real Problem Solvers. Springer Verlag
36. de Kergommeaux JC, Codognet P (1994) Parallel Logic Programming Systems. ACM Computing Surveys 26(3):295–336
37. Kroc L, Sabharwal A, Selman B (2010) An Empirical Study of Optimal Noise and Runtime Distributions in Local Search. In: Strichman O, Szeider S (eds) SAT'10, Springer, Edinburgh, UK, LNCS, vol 6175, pp 346–351
38. Luby M, Sinclair A, Zuckerman D (1993) Optimal Speedup of Las Vegas Algorithms. Information Processing Letters 47:173–180
39. Michel L, See A, Van Hentenryck P (2006) Distributed Constraint-Based Local Search. In: Benhamou F (ed) CP'06, 12th Int. Conf. on Principles and Practice of Constraint Programming, Springer Verlag, Lecture Notes in Computer Science, pp 344–358
40. Michel L, See A, Van Hentenryck P (2007) Parallelizing Constraint Programs Transparently. In: Bessiere C (ed) CP'07, 13th Int. Conf. on Principles and Practice of Constraint Programming, Springer Verlag, Lecture Notes in Computer Science, pp 514–528
41. Michel L, See A, Van Hentenryck P (2009) Parallel and Distributed Local Search in Comet. Computers and Operations Research 36:2357–2375
42. Moisan T, Gaudreault J, Quimper C (2013) Parallel Discrepancy-Based Search. In: Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings, pp 30–46, DOI 10.1007/978-3-642-40627-0_6, URL http://dx.doi.org/10.1007/978-3-642-40627-0_6
43. Nadarajah S (2008) Explicit Expressions for Moments of Order Statistics. Statistics & Probability Letters 78(2):196–205
44. Pardalos PM, Pitsoulis LS, Mavridou TD, Resende MGC (1995) Parallel Search for Combinatorial Optimization: Genetic Algorithms, Simulated Annealing, Tabu Search and GRASP. In: Parallel Algorithms for Irregularly Structured Problems (IRREGULAR), pp 317–331
45. Pardalos PM, Pitsoulis LS, Resende MGC (1996) A Parallel Grasp for MAX-SAT Problems. In: Wasniewski J, Dongarra J, Madsen K, Olesen D

(eds) 3rd International Workshop on Applied Parallel Computing, Industrial Computation and Optimization, Springer, Lyngby, Denmark, LNCS
46. Perron L (1999) Search Procedures and Parallelism in Constraint Programming. In: CP'99, 5th Int. Conf. on Principles and Practice of Constraint Programming, Springer Verlag, Lecture Notes in Computer Science, pp 346–360
47. Pham DN, Gretton C (2007) gNovelty+. In: Solver description, SAT competition 2007
48. Régin JC, Rezgui M, Malapert A (2013) Embarrassingly Parallel Search. In: Schulte C (ed) Proceedings of CP'2013, 19th International Conference on Principles and Practice of Constraint Programming, Springer Verlag, Lecture Notes in Computer Science, vol 8124, pp 596–610
49. Ribeiro C, Rosseti I, Vallejos R (published online 2011/08/17) Exploiting Run Time Distributions to Compare Sequential and Parallel Stochastic Local Search Algorithms. Journal of Global Optimization pp 1–25
50. Shylo OV, Middelkoop T, Pardalos PM (2011) Restart Strategies in Optimization: Parallel and Serial Cases. Parallel Computing 37(1):60–68, DOI 10.1016/j.parco.2010.08.004
51. Truchet C, Richoux F, Codognet P (2013) Prediction of Parallel Speed-ups for Las Vegas Algorithms. In: Dongarra J, Robert Y (eds) Proceedings of ICPP-2013, 42nd International Conference on Parallel Processing, IEEE Press
52. Van Hentenryck P (1989) Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In: ICLP'89, International Conference on Logic Programming, MIT Press, pp 165–180
53. Van Hentenryck P, Michel L (2005) Constraint-Based Local Search. The MIT Press
54. Van Luong T, Melab N, Talbi EG (2010) Local Search Algorithms on Graphics Processing units. In: Evolutionary Computation in Combinatorial Optimization, LNCS 6022, Springer Verlag, pp 264–275
55. Verhoeven M, Aarts E (1995) Parallel Local Search. Journal of Heuristics 1(1):43–65
56. Wolfram S (2003) The Mathematica Book, 5th Edition. Wolfram Media, URL `http://reference.wolfram.com`
57. Xie F, Davenport AJ (2010) Massively Parallel Constraint Programming for Supercomputers: Challenges and Initial Results. In: CPAIOR'10, 7th International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Springer, Lecture Notes in Computer Science, vol 6140, pp 334–338