Accelerating Population-Based Search Heuristics by Adaptive Resource Allocation

Joachim Lepping
INRIA Rhône-Alpes
51 Av. Jean Kuntzmann
38330 Montbonnot-St-Martin, France
joachim.lepping@udo.edu

Panayotis Mertikopoulos
French National Center for Scientific Research (CNRS), Laboratoire d'Informatique de Grenoble (LIG), France
panayotis.mertikopoulos@imag.fr

Denis Trystram
LIG, Grenoble University, IUF
51 Av. Jean Kuntzmann
38330 Montbonnot-St-Martin, France
denis.trystram@imag.fr

ABSTRACT

We investigate a dynamic, adaptive resource allocation scheme with the aim of accelerating the convergence of multi-start population-based search heuristics (PSHs) running on multiple parallel processors. Given that each initialization of a PSH performs differently over time, we develop an exponential learning scheme which allocates computational resources (processors) to each variant in an online manner, based on the performance level attained by each initialization. For the well-known example of $(\mu + \lambda)$–evolution strategies, we show that the time required to reach the target quality level of a given optimization problem is significantly reduced and that the utilization of the parallel system is likewise optimized. Our learning approach is easily implementable with currently available batch management systems and provides notable performance improvements without modifying the employed PSH, so it is very well-suited to improve the performance of PSHs in large-scale parallel computing environments.

Keywords

exponential learning, online algorithms, cluster- and grid-computing

1. INTRODUCTION

For massively complex industrial problems (such as large parameter optimization or design optimization), exact solution methods are rarely applicable: the search space is often too large to be fully explorable and/or a mathematical model of the optimization problem is hard to obtain. In such cases, population-based randomized search heuristics (PSH) like evolutionary algorithms or particle swarms have become very popular because they do not require a specific mathematical model of the optimization problem. For instance, when using computational fluid dynamics in order to optimize the aerodynamic properties of a vehicle's shape, it is impossible to define a formal fitness expression and one uses simulations in order to determine the fitness function of a candidate solution.

In this context, a widely used class of population-based randomized search strategies are the so-called evolution strategies (ES): at each iteration, an ES (see also Section 3.2) provides certain solution candidates (which are commonly denoted as *individuals*), and selects the best ones based on selection-recombination mechanisms inspired by evolutionary biology. In this way, the next generation of individuals will be biased towards regions of the search space where better solutions have already been found, so the quality of the solution is improved at each iteration of the ES until a termination criterion is fulfilled.

Of course, this iterative process requires the fitness evaluation of every newly generated individual, a calculation which involves very complex simulations in an industrial setting: for instance, since the 1990s, aerodynamic optimization of real-world design objects requires extremely demanding computational fluid dynamics simulations that are all but impossible to execute in individual machines [1]. On the other hand, recent developments in high performance computing and clusters allow the evaluation of increasingly complex models on comparatively cheap hardware by virtue of parallelization. As a result, the performance of a PSH depends not only on its internal structure and convergence properties, but also on the efficient scheduling of the parallel machines – a factor which becomes even more important if the evaluations are time-consuming and the simulation times vary. More importantly, if one employs several different initializations of a PSH in order to reduce the risk of getting stuck in a local optimum (the so-called *multi-start* approach), allocating computational resources between better- and worse-performing initializations becomes a crucial issue.

In this paper, we seek to reduce the time required to reach a certain fitness value for a given optimization problem using multi-start PSHs by optimizing the allocation of computational resources to different variants of the PSH in an online, adaptive way. In contrast to cooperative problem-solving strategies [2] which require a significant level of interaction and coordination between different heuristics (or multiple initializations of the same heuristic), we consider here a fully decentralized setting where no such central coordination is possible; in a similar vein, any modifications of the PSH that would have to be performed by a central coordinator (such as model-assisted evaluations [3]) are likewise not considered. Instead, we only adjust the amount of resources assigned to each PSH over time based on its performance record: inspired by the process of natural selection, variants with fitter individuals obtain more resources, allowing them to spread faster and explore the more profitable regions of the problem's search space. More precisely, we compare the best fitness value achieved by each heuristic at arbitrary timestamps, and depending on the fitness progress of each PSH so far (normalized by the number of already evaluated individuals), we reassign resources by means of an exponential stimulus-response scheme which is intimately related to the replicator dynamics of evolutionary game theory [4, 5].

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

In particular, the dynamic resource allocation scheme that we propose has the following pleasant properties:

- It is easy to implement in existing batch scheduling systems (such as Torque/PBS, LSF, etc.).
- It applies to every PSH that could benefit from a multi-start approach.
- No modifications in the internal algorithmic structure are required, so the original search heuristics remain untouched.
- There are no cooperation and/or central control requirements beyond the resource allocation policy of the batch scheduler.

In this framework, our evaluation with simulations of well-known test functions (see Section 6) shows that we achieve a better utilization of the parallel system and that we significantly speed up the convergence to the predefined quality level.

In the next section, we detail our resource allocation problem in the light of well-studied learning problems and we motivate the choice of our methodology. In Section 3, we present the precise formulation of our problem and we point out certain parallelization and convergence issues that appear in PSHs. Our proposed learning scheme is then presented in detail in Section 4, and its performance is examined and assessed in Sections 5 and 6.

2. BACKGROUND

The parallelization of PSHs has been extensively addressed in the literature – for instance, see Alba and Tomassini [6] for a good survey on existing approaches and technical solutions. Most of these works however modify the internal structure of the PSHs to take advantage of parallel resources and parallel communication models; in particular, the usage of parallel agents is often realized as collaborative or competitive search [7, 8].

On the other hand, from a learning point of view, our resource allocation problem bears close ties to the class of discrete choice problems known as "learning with expert advice" [9]. In this class of problems, a decision-maker (agent) is asked to choose between the advice of K "experts", and his reward is based on how good each expert's advice is. Of course, the performance of each expert is not constant over time, so the agent seeks to maximize his long-term payoff by selecting with higher probability the advice of experts who seem to be performing well and assigning lower probability to the rest. Accordingly, in our context, we will seek to devise a learning scheme for the allocation of computational resources so that the PSH variant which seems to be closer to reaching the target quality level gets more resources than others that lag consistently behind.

To that end, an approach which has met with particular success is that of Boltzmann-driven learning [10] where one keeps and updates a score variable for each expert by aggregating the expert's performance over time, and then choosing an expert at each instance with probability that is exponentially proportional to these scores – for a detailed account of this "exponential learning" process and its ties to evolutionary biology, see, e.g. [11, 12, 13, 9] and references therein. That said, a tacit assumption that underlies this "expert advice" literature is that the decision of the agent does *not* affect the performance of each expert: experts are not swayed in their predictions by what the agent does. In our computational setting however, this assumption fails spectacularly and for an unavoidable reason: by allocating more resources to a heuristic, this heuristic *will* run faster (and hence perform better), so we can no longer assume that the agent's decision-making does not interfere with the expert's advice.

Accordingly, the main challenge that needs to be overcome is how to estimate the speedup in the performance of a PSH variant that has been afforded extra resources, and how to compare the performance of different variants on an even keel. To overcome this challenge of spurious speedups, we introduce in Section 4.1 an unbiased estimator which compensates the performance profile of a heuristic by normalizing the fraction of resources allocated to it, and in so doing, we manage to compare the various PSH variants on an even keel.

3. PROBLEM FORMULATION

In this section, we describe the basic system model for a general multi-start PSH; for the sake of concreteness however, we will mostly focus on the widely used class of $(\mu + \lambda)$–Evolution Strategies (see Section 3.2).

3.1 System Model

We consider a set $\mathcal{K} = \{1, \ldots, K\}$ of K realizations of a population-based randomized search heuristic. In each generation $i = 1, 2 \ldots$, a heuristic must evaluate λ individuals in order to determine the objective values of its population – themselves representing the values of a global objective function f that we seek to minimize. Obviously, a new generation (i.e. the submission of λ new individuals to be simulated) can only start if all individuals of the incumbent generation have been completed, so each heuristic generates a set of λ jobs in each generation i. The simulation performed to determine an individual's objective will be regarded as a job and indexed by j, and we will also assume that the internal creation of the next generation (i.e., the process of selection, mutation, and recombination) requires negligible time compared to the actual job execution time.

We further consider a set of M identical machines (e.g., processors of a cluster) where evaluations can be performed in parallel.[1] Since we consider an online problem where the execution time of a job is only known after its completion, the scheduling on uniform or related parallel machines may only be done heuristically [14], and is beyond the scope of this paper. Here, the execution time of a job only depends on its processing time p_j (and thus not on a particular machine allocation), and the internal machine scheduling is realized (as in most batch scheduling systems) in a First-Come-First-Served fashion.

For a heuristic $k \in \mathcal{K}$, the best found value of the objective function f in generation i will be denoted by $f_k(i)$. As mentioned before, we assume the minimization of f throughout this paper, and we also assume that each generation improves on the previous one, i.e. $f_k(i-1) \geq f_k(i)$ (an assumption which is actually the evolutionary drive behind $(\mu+\lambda)$–evolutionary strategies). Additionally, we consider the best found objective value at a timestamp t and we again guarantee that $f_k(t') \geq f_k(t)$ whenever $t' \leq t$; on the other hand, since we are not considering cooperation and communication between heuristics we might well have $f_k(t') > f_\ell(t')$ and $f_k(t) < f_\ell(t)$ for two heuristics $k, \ell \in \mathcal{K}$, and $t' < t$.

Since all evaluations of a population within a generation are independent of each other, they can be executed in parallel. Thus, PSHs are well-suited to be executed on multi-core systems, clusters, or cloud-computing systems. The generation-based nature of these heuristics creates barriers in the parallel execution schedule. The problem occurs if the processing time for simulating an individual's fitness vary. Since new individuals are created by selection and variation depending on the fitness properties of the previous

[1] Our resource adaption approach can be realized in arbitrary machine environments, so the restriction to parallel machines is hardly restrictive. We restrict ourselves to this model because it does not require the explicit modeling of machine-dependent job execution times.

generation, the parallel evaluation of an generation i can only start if all λ individuals of the previous generation $i-1$ have been evaluated.

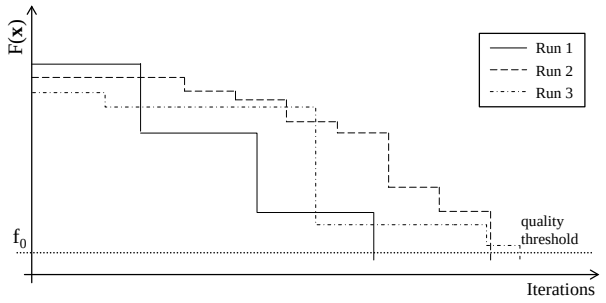

Figure 1: Typical fitness progress for a multiple runs of a PSH.

The convergence behavior of randomized search heuristics is only hardly predictable but it can be influenced by several algorithm parameters such as selection pressure or mutation rates. Once however the algorithm configuration is fixed, the heuristic's convergence behavior is determined by the random operators of the algorithms. Thus, the number of generations required to reach a solution below a predefined fitness values may well vary between two runs of otherwise identically configured PSH (cf. Figure 1).

3.2 $(\mu + \lambda)$–Evolution Strategies

As we mentioned before, our approach applies to any PSH that is amenable to multiple initializations. However, for the sake concreteness, we will focus in what follows on the popular class of PSHs known as $(\mu + \lambda)$–evolution strategies (ES) – for a detailed description and examples of other popular and practically relevant PSHs, see [15]. As all PSHs, an ES searches within a "population" of solutions wherein each individual represents a certain candidate solution of the optimization problem. The quality of this solution is called the "fitness" of an individual, and it is determined by evaluating the individual on the optimization problem. Following Schwefel and Rudolph [15], μ denotes here the number of *parents* while λ denotes the number of all offspring created by these parents within one synchronized generation. In the $(\mu + \lambda)$-ES the offspring and their λ parents are merged before the μ fittest individuals are selected from the $\mu + \lambda$ population (in particular, fitness never decreases), so each generation calls for the evaluation of λ individuals, and all these evaluations are independent.

4. ONLINE RESOURCE ALLOCATION

Before we start our adaptive resource allocation scheme, all heuristics must be initialized since the first parent population of μ individuals must be evaluated. We do not perform any update step during this phase and assume that at the initial timestamp $t = 0$ the μ progenitors have completed their evaluation. Once the K heuristics are initialized, we execute them in parallel assigning initially an equal portion of $m_k(0) = M/K$ resources to each heuristic. The jobs are assigned in a LIST-scheduling fashion to the machines, and as soon as a machine idles, it is assigned to a new job from any of the heuristics. Besides the already described dependencies between iterations and the resulting availability of jobs, we ensure that at no time more than $m_k(t)$ jobs are simultaneously executed for every heuristic k.

In Figure 2, we illustrate a typical scheduling and resource assignment concept assuming for simplicity identical processing times

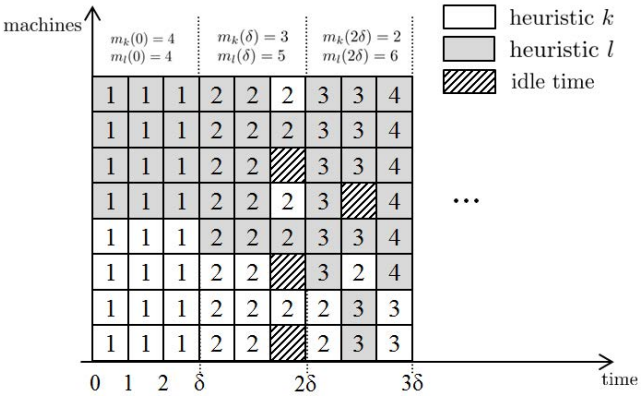

Figure 2: Scheduling and resource allocation concept for two PSHs and identical processing times in each evaluation. Each PSH has to evaluate $\lambda = 12$ individuals per generation and the cluster consists of $M = 8$ machines. The numbers denote the generation the evaluation belongs to.

for all evaluations. From timestamp δ and onwards, PSH k gets three of the eight resource assigned while PSH ℓ is allowed to use five resources at a time. At timestamp $t = 5$ heuristic ℓ has to finish generation 2 before it can start the next generation. The assigned resources cannot be entirely utilized by heuristic ℓ and the machines become idle. It would be possible to use idle resource to perform evaluations of other heuristics. However, we want to ensure that the updated resource allocations can be fulfilled during the next interval. This might become impossible if machines get occupied by other heuristics in a work-stealing like fashion.

4.1 Updating the allocation of resources

To optimize the allocation of resources per heuristic, we will score each heuristic's performance over time and then assign to each heuristic an amount of resources which is (roughly) exponentially proportional to this score. Specifically, at time t (counted every δ units of time), we measure the fitness values $f_k(t)$ of each heuristic $k \in \mathcal{K}$ and we iteratively define the heuristic's score as

$$U_k(t+\delta) = \eta_k(t)\left(U_k(t) - \frac{f_k(t)}{\max_{\ell\in K}\{f_l(t)\}}\right) \tag{1}$$

with $U_k(0) = 0$ for all heuristics. In the above, $\eta_k(t)$ is a normalization factor defined as

$$\eta_k(t) = \frac{n_k(t)}{\max_{\ell\in K}\{n_\ell(t)\}} \tag{2}$$

where $n_k(t)$ represents the overall number of evaluations that heuristic k has executed by timestamp t. This factor weighs the performance of a heuristic based on the number of evaluations it has already performed, so η_k compensates for the fact that the more evaluations a heuristic has executed, the better its objective value will be. As such, this aggregation method can be seen essentially as a moving average between the (normalized) performance history of heuristic k and its current fitness.

Indeed, by a slight rearrangement of terms, we obtain:

$$U_k(t+\delta) - U_k(t) = (\eta_k(t) - 1)U_k(t) - \frac{\eta_k(t) f_k(t)}{\max_{l\in K}\{f_l(t)\}}, \tag{3}$$

or, descending to continuous time for simplicity:

$$\dot{U}_k(t) = (\eta_k(t) - 1)U_k(t) - \overline{f}_k(t), \tag{4}$$

where $\overline{f}_k$ denotes the rightmost term of Equation (3), i.e. the relative performance of a heuristic normalized with regards to the fraction of resources that were allocated to it. Thus, if we set $\tau(t) = \int_0^t (1 - \eta_k(s))ds$ and then multiply both sides of this expression with the integrating factor $e^{\tau(s)}$, a simple integration yields:

$$U_k(t) = -\int_0^t e^{-(\tau(t)-\tau(s))}\overline{f}(s)ds. \tag{5}$$

This last expression (which can also be verified by differentiating under the integral sign) shows that the scores $U_k(t)$ can be interpreted as exponential moving averages of the heuristic's normalized performances, taken with respect to a non-uniform time-scale that accounts for the fact that the amount of resources allocated to a heuristic changes over time (and, thus, the time that it takes a heuristic to reach a given performance level should be scaled commensurately). Importantly, this is a different kind of averaging than the one induced by η in (1): η represents a normalization average over resources whereas the exponential factor in the integral of (3) represents a discounting of past information in order to construt an exponentially moving average over time.

Needless to say, better-scoring heuristics should be getting more resources; hence, inspired by its widespread use in the theory of learning [9], we will assign computational resources based on the Boltzmann–Gibbs stimulus-response scheme:

$$X_k(t) = \frac{\exp\{\alpha U_k(t)\}}{\sum_{\ell=1}^{K}\exp\{\alpha U_\ell(t)\}}, \tag{6}$$

where $X_k(t)$ is the fraction of resources appointed to heuristic k at time t. In this equation, the factor $\alpha < 1$ represents the inverse temperature of the Boltzmann distribution and will control the exploration/exploitation tradeoff of the dynamic policy (6). Since a small α induces a high temperature the learning speed is decreased as the behavior of (6) becomes less greedy. A pre-experimental study revealed that $\alpha = 5$ is a reasonable choice to ensure both fast learning and more-than-adequate adaptivity over a wide range of experimental parameters.

Now, following (6) the actual number of allocated resources will be determined using the expression:

$$m_k(t+1) = \lfloor X_k(t) \cdot (M - K)\rfloor + 1, \tag{7}$$

which guarantees that at least one resource is assigned to every heuristic at all times. To further guarantee that all resources are used for every instance we assign the remaining

$$\bar{M}(t+1) = M - \sum_{k=1}^{K} m_k(t+1) \tag{8}$$

resources to the $\bar{M}(t+1)$ currently best performing heuristics – and to avoid initialization issues, we set $\bar{M}(0) = 0$. Finally, to ensure the numerical stability of the discrete updating scheme in Equation (1) during the following steps, we shift the scores to zero level for all heuristics and actually use the updating rule:

$$U_k(t+\delta) \leftarrow U_k(t+\delta) - \max_{\ell\in K}\{U_\ell(t+\delta)\}. \tag{9}$$

Thanks to the exponential updating rule (6), this transformation does not affect the actual allocation profile because only differences in scores matter for the updating step.

Remark. It should be noted that this resource allocation scheme does *not* boil down to a "winner-takes-all" policy, whereby the "best" heuristic gets all computational resources. Thanks to the normalization of execution times, underperforming heuristics still get a share of the resources, so if they start performing well, the online learning scheme will re-allocate resources accordingly.

5. ASSESSING EVALUATION RESULTS

Our implementation of the $(\mu+\lambda)$-ES follows closely the detailed description in [16]. We thus consider different population sizes (a parameter which has a strong influence on the convergence behavior and also influences the resource adaptation process), and based on Schwefel's recommendation [17], we took a ratio of $\mu/\lambda \approx 1/7$ for all simulations. The parameter settings above may well be improved on for specific cases, but this parameter range leads to a better overall convergence rate on average [17]. More to the point, since we will show that it is possible to accelerate the algorithm performance by improved resource assignments the actual convergence behavior is only of minor importance – it only serves as an example which can be substituted by any other population-based randomized search heuristic.

5.1 Definition of Performance Criteria

As noted before, we are interested in minimizing the overall time it takes to reach a predefined level of our objective function. Thus, in order to evaluate our results, we consider the following performance metrics (for a single run of all experiments with the same configuration):

T_l : Amount of time that the online learning approach required to reach the predefined quality threshold f_0.

T_u : Amount of time that the fastest heuristic would have taken to reach the predefined quality threshold f_0 assuming uniform resource allocation: $m_k(t) = M/K$.

T_b : Amount of time that the fastest heuristic (i.e., the heuristic that requires fewest iterations to reach f_0) would have taken if all M resource had been exclusively assigned to it.

T_w : Amount of time that the slowest heuristic (i.e., the heuristic that requires most iterations to reach f_0) would have taken if all M resource had been exclusively assigned to it.

To measure the improvements of the online learning approach over a static uniform allocation, we define the average gain G over uniform resource allocation in Equation (10).

$$G = \frac{T_u - T_l}{T_u} \tag{10}$$

Furthermore, we define the average efficiency E_u of the uniform allocation, see Equation (11),

$$E_u = \frac{T_w - T_u}{T_w - T_b} \tag{11}$$

and the average efficiency E_l of the online learned allocations, see Equation (12).

$$E_l = \frac{T_w - T_l}{T_w - T_b} \tag{12}$$

In all cases, we compare the required time with the discrepancy between the worst and best heuristics: if we observe a negative efficiency, then simply choosing a heuristic at random and assigning all resources to it would have been the better choice (naïve as this might seem). In such cases, neither uniform allocation nor online learning is reasonable, so a sophisticated approach seems meaningless (the uncertainty is just too high).

Importantly, a standard evaluation criterion that is used in the learning literature to assess the performance of a dynamic learning scheme is its *regret series*:

$$R_k(t) = \frac{1}{t}\left[\sum_{n=1}^{t}\left(f_k^*(n) - f(n)\right)\right]_+ \tag{13}$$

where $f(n)$ is the value achieved at step n of the process, $f_k^*(n)$ is the value that would have been achieved by heuristic k at stage n if all resources had been allocated to heuristic k for all $n \leq t$ (see, e.g., Cesa-Bianchi and Lugosi [9] and references therein). In other words, the regret series R_k simply measures how much has been lost in terms of performance by not using heuristic k all the time: if k is consistently worse than all possible resource allocation to other heuristics then R_k will be identically equal to 0, whereas if k consistently outperforms all other heuristics, R_k will vanish if and only if all available resources are allocated to k. As a result, the standard objective in learning is to devise a learning scheme which is asymptotically *consistent* in the sense that $\lim_{t\to 0} R_k(t) = 0$ for all k (for an extended discussion of this issue, see [9, 18, 19] and references therein).

Nonetheless, the regret series as defined above suffers from two very important drawbacks in our setting: First of all, it is a *fictitious* criterion in the sense that R_k cannot be calculated unless we know the value $f_k^*(n)$ that the k-th heuristic *would* have achieved at each $n \leq t$ *if* all resources had been assigned to it. Even more importantly, consistency is an *unbalanced* criterion in the sense that R_k could stay unboundedly positive for perfectly reasonable resource allocation policies. To illustrate this last point, assume the following example which, despite its simplicity, captures the essence of the problem: we are given two heuristics, A and B with A two times slower than B, so that $f_A^*(n) = 2f_B^*(n) = 2n$ for all n; assume further for simplicity that the performance of each heuristic depends linearly on the fraction of alloted resources. Then, if we allocate resources with a ratio $2:1$ in favor of the *slow* heuristic A, we will have $f(n) = \max\{2/3 \times n, 1/3 \times 2n\} = 2n/3$. In turn, this yields a *positive* and bounded away from zero regret series R_A for the slower heuristic, indicating (quite falsely!) a regret for not giving all resources to the slow heuristic A.

In view of the above, we see that regret-based consistency is not a suitable performance evaluation criterion for our resource allocation scheme, so, in all our experiments, our analysis will be focused on the runtime evaluation criteria defined at the beginning of this section.

6. SIMULATION SETUP AND RESULTS

To investigate the performance of our approach in challenging optimization problems with wildly varying convergence behaviors, we focused our experiments on two well-known test problems for evolutionary algorithms, the Ackley and Rastrigin functions [20, 21]. Of course, real-world problems might exhibit different behavior than these artificially constructed functions; the reason that we chose to focus on these highly challenging problems is that they represent the worst-case scenarios that could arise in real-world conditions, and they are also precisely the type of problems that require a massively parallel computing environment to boot.

Specifically, the first test problem that we are considering is the *Ackley function* [20] in its standard variant. The problem is finding a string $\vec{x} = \{x_1, x_2, \ldots, x_N\}$, with $x_i \in (-32.768, 32.768)$, that minimizes Equation (14).

$$F_1(\vec{x}) \quad = 20\left(1 - e^{-0.2\sqrt{\frac{1}{N}\sum_i x_i^2}}\right) + e - e^{\frac{1}{N}\sum_i \cos(2\pi x_i)} \tag{14}$$

Next, we also consider the *Rastrigin function* first proposed in [21] as a typical example of a (2-dimensional) non-linear multimodal function:

$$F_2(\vec{x}) = 20 + \sum_{i=1}^{2}\left(x_i^2 - 10\cos(2\pi x_i)\right), \quad x_i \in (-5.12, 5.12). \tag{15}$$

6.1 Experimental setup

μ	λ	K	M	δ	p_j
3	21	4/6/8	40/60/80	7/5	$\sim \mathcal{U}(1,5)/\mathcal{E}(2)$
4	30	4/6/8	40/60/80	9/6	$\sim \mathcal{U}(1,5)/\mathcal{E}(2)$
5	35	4/6/8	40/60/80	11/7	$\sim \mathcal{U}(1,5)/\mathcal{E}(2)$
6	40	4/6/8	40/60/80	12/8	$\sim \mathcal{U}(1,5)/\mathcal{E}(2)$
15	100	4/6/8	40/60/80	30/20	$\sim \mathcal{U}(1,5)/\mathcal{E}(2)$

Table 1: Detailed configuration of the discussed experiments. Note that δ is independent of the number of heuristics since we always chose $M = 10K$. It depends only on the processing time estimate and the population size. Here we use $\bar{P}(\mathcal{U}(1,5)) = 3$ and $\bar{P}(\mathcal{E}(2)) = 2$.

In our experiments, we run each ES until it falls below an objective quality threshold of $f_0 = 10^{-4}$ and we simulate a compute cluster with $M = 10K$ computational resources (reflecting a reasonable load/resource scaling). The details of our experimental setup are shown in Table 1; in what follows, we will simply give a short account of some important experimental issues.

To begin with, in real batch-systems like PBS/Torque, the scheduler is only triggered at discrete timestamps. Due to performance issues the number of scheduling decisions is limited (although it might slightly reduce the utilization), so we submit new jobs only at integer time units ($t = 1, 2, \ldots$) while the online resource allocation is updated every δ units of time with

$$\delta = \left\lceil \bar{P} M^{-1} \lambda K \right\rceil, \tag{16}$$

where $\bar{P}$ involves the estimated mean execution time of an evaluation. Of course, in an online scheduling context the processing times of jobs are not known n advance, so we determine this value by a rough estimation based on pre-experimental reasoning; however, experiments showed that even a very coarse estimate is already sufficient, so we will not elaborate further on this issue.

Another important aspect of our simulations is the distribution of the jobs' execution times. Unfortunately, there exists no standard model for the processing times of sequential jobs submitted to clusters. The closest results that we could find was the analysis of [22] where the authors show that the runtimes in common workload traces follows a hyper-gamma distribution – however, the model of [22] refers to parallel workloads *only*. In light of the above, we chose instead to sample processing times $p_j \sim \mathcal{U}(a, b)$ for each evaluation form a uniform distribution over the interval $[a, b]$, and from an exponential distribution $p_j \sim \mathcal{E}(\rho)$ with mean ρ (in the displayed experiments, we chose $\rho = 2$). Our simulations showed that our approach is fairly indifferent to the processing time distributions, see Figures 3 and 4, so knowledge of the exact distribution of processing times does not seem to be particularly important.

6.2 Experimental results

Figure 3(a) shows the improvement of the online learning approach over a static uniform allocation for several population size configurations. Since the median gain is positive for all settings, the exponential learning scheme is superior to a static uniform resource assignment. For smaller populations, improvements lie between 7.5 % ($3+21$ and $k = 8$) and 15.23 % ($3+21$ and $k = 4$), but for larger populations, our adaptive resource allocation scheme is particularly advantageous: we observe a gain in convergence time between 30.47 % ($15 + 100$ and $k = 8$) and 36.6 % ($15 + 100$ and $k = 8$) over the simpler uniform allocation scheme.

However, the larger population configurations also highlight the only drawback that we were able to identify in our adaptive re-

source allocation scheme. Figure 3(b) shows the efficiency of a uniform allocation scheme and it reveals that resource sharing and a multi-start variant is not advantageous in those cases: apparently, the evolution strategy converges faster to the desired quality level if all resources are assigned to any arbitrary heuristic. One possible explanation of this phenomenon is that the explorative power of a (15 + 100)-population is so large that any multi-start variant will converge quickly to the optimal solution. In our approach, we always have to share the resource among the multi-start variant. for large populations, giving all resources to a single start of the PSH would have been obviously advantageous. Even the online adaption of the resource can not ameliorate this tendency entirely although results are slightly improved, see Figure 3(c).

In this way, a natural question that emerges is the following: do multi-start variants with online resource sharing lead to advantages in optimization time? To answer this question, we present the averaged absolute optimization durations (in seconds) in Table 3. Results are given for the Rastrigin function ($F_2(\vec{x})$) over 40 runs and for both uniformly and exponentially sampled processing times (we only present the results of $F_2(\vec{x})$ due to space limitations). Qualitatively, the results for $F_2(\vec{x})$ which would correspond to Figure 3, are similar. First, using multi-start variants on the parallel resources pays off in any case, since T_u becomes always smaller with increasing values of K. The learning (T_l) improves these values in every case over static shares T_u. This tendency is even stronger if the processing times are exponentially distributed (cf. Table 3).

7. CONCLUSION

We presented an approach to accelerate the convergence of any kind of population-based randomized search heuristic (PSH) that is executed on parallel computing installations. Taking a multi-start approach, we executed multiple initializations of the same PSH to solve the same optimization problem and used an exponential learning technique in order to adapt the fraction of computational resources assigned to each heuristic based on its performance envelope. The novelty of the approach lies in that the overall optimization process revolves exclusively around resource assignment: in particular, the actual PSH does not need to be modified and any kind of optimizer that is able to deliver intermediate results (to compare the current progress) may be improved with the approach presented in this paper. Our resource adaption scheme is particularly suited for evolutionary algorithms: evaluations with two common test functions and randomly generated processing times show that our online learning scheme yielded significant performance improvements and was robust to the processing time distribution.

An open (but orthogonal) question for further research is whether it really pays off to have a multi-start variant with online resource adaptation instead of a significantly larger population size – a question which we did not address due to space limitations. Indeed, this investigation requires a thorough look into the quality/speed tradeoff between exploring greater regions of the state space (multi-start) and potential premature convergence in local optima (single run of PSH with large explorative power). In this context, we also have to investigate self-adaptation of the mutation step-size for the large population ES and alternative PSHs like particle swarms or population-based ant colony optimization algorithms.

8. REFERENCES

[1] W. D. Gropp and E. B. Smith, "Computational fluid dynamics on parallel processors," *Computers & Fluids*, vol. 18, no. 3, pp. 289–304, 1990.

[2] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Autonomous Agents and Multi-Agent Systems*, vol. 11, pp. 387–434, 2005.

[3] M. Emmerich, A. Giotis, M. Özdemir, T. Bäck, and K. Giannakoglou, "Metamodel-assisted evolution strategies," in *In Parallel Problem Solving from Nature VII*. Springer, 2002, pp. 361–370.

[4] P. D. Taylor and L. B. Jonker, "Evolutionary stable strategies and game dynamics," *Mathematical Biosciences*, vol. 40, no. 1-2, pp. 145–156, 1978.

[5] W. H. Sandholm, *Population Games and Evolutionary Dynamics*, ser. Economic learning and social evolution. Cambridge, MA: MIT Press, 2011.

[6] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, 2002.

[7] H. Mühlenbein, *Computational Intelligence*, ser. Intelligent Systems Reference Library. Springer, 2009, vol. 1, ch. Evolutionary Computation: Centralized, Parallel or Collaborative, pp. 561–595.

[8] C. Grimme, J. Lepping, and A. Papaspyrou, "Parallel predator-prey interaction for evolutionary multi-objective optimization," *Natural Computing*, vol. 11, no. 3, pp. 519–533, 2011.

[9] N. Cesa-Bianchi and G. Lugosi, *Prediction, Learning, and Games*. Cambridge University Press, 2006.

[10] D. Fudenberg and D. K. Levine, *The Theory of Learning in Games*, ser. Economic learning and social evolution. Cambridge, MA: The MIT Press, 1998, vol. 2.

[11] A. Rustichini, "Optimal properties of stimulus-response learning models," *Games and Economic Behavior*, vol. 29, pp. 230–244, 1999.

[12] R. Cominetti, E. Melo, and S. Sorin, "A payoff-based learning procedure and its application to traffic games," *Games and Economic Behavior*, vol. 70, pp. 71–83, 2010.

[13] P. Mertikopoulos and A. L. Moustakas, "The emergence of rational behavior in the presence of stochastic perturbations," *The Annals of Applied Probability*, vol. 20, no. 4, pp. 1359–1388, 2010.

[14] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 2nd ed. Prentice-Hall, 2002.

[15] H.-P. Schwefel and G. Rudolph, "Contemporary evolution strategies," in *Proceedings of the 3rd European Conference on Artificial Life*, F. Morán, A. Moreno, J. J. Merelo, and P. Chacón, Eds. Springer, 1995, pp. 893–907.

[16] T. Bäck, *Evolutionary algorithms in theory and practice*. New York: Oxford University Press, 1996.

[17] H.-P. Schwefel, *Evolution and Optimum Seeking*. John Wiley & Sons, 1995.

[18] S. Hart and A. Mas-Colell, "A simple adaptive procedure leading to correlated equilibrium," *Econometrica*, vol. 68, no. 5, pp. 1127–1150, September 2000.

[19] S. Sorin, "Exponential weight algorithm in continuous time," *Mathematical Programming*, vol. 116, no. 1, pp. 513–528, 2009.

[20] D. H. Ackley, *A connectionist machine for genetic hillclimbing*. Boston: Kluwer, 1987.

[21] A. Törn and A. Zilinskas, *Global Optimization*, ser. Lecture Notes in Computer Science. Springer, 1989, vol. 350.

[22] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: Modeling the characteristics of rigid jobs." *Journal of Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1105–1122, 2003.

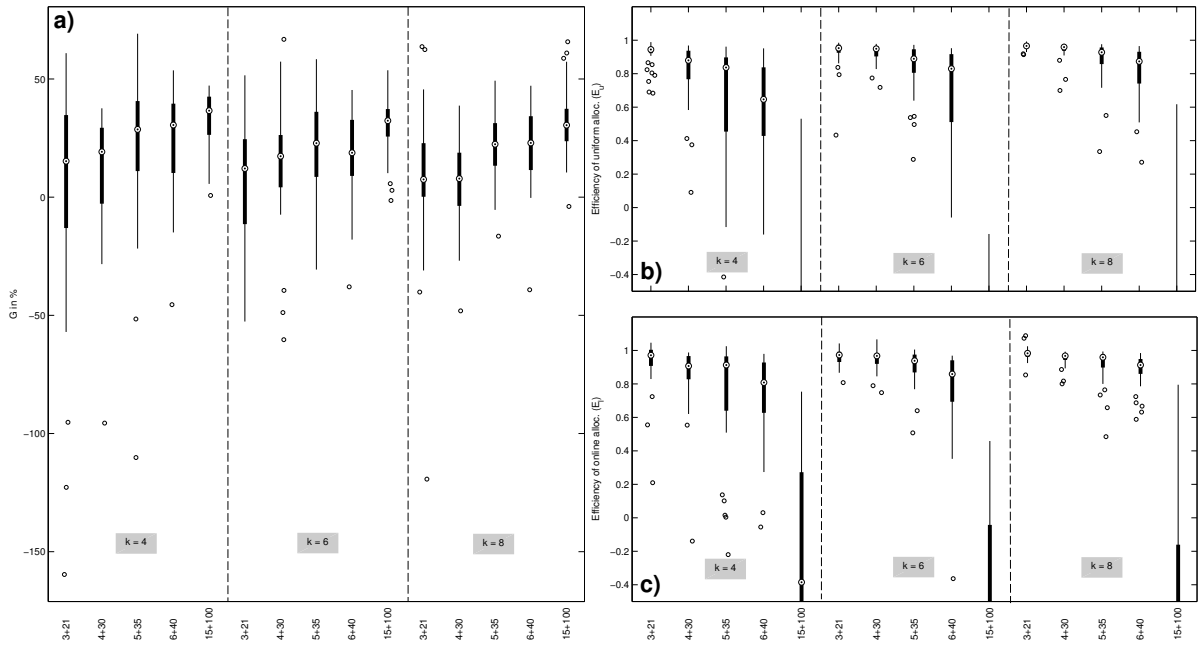

Figure 3: Boxplots of population-based search heuristic configurations over 40 runs on problem $F_1(\vec{x})$ with uniformly sampled processing times ($\mathcal{U}(1,5)$). The configurations comprise various population sizes ($\mu + \lambda$) and number of heuristics ($k = 4, 6, 8$). Figure (a) shows the gain of the online learning over uniform allocation in %. Figure (b) shows the efficiency of uniform allocation and Figure (c) the efficiency of the online learning approach.

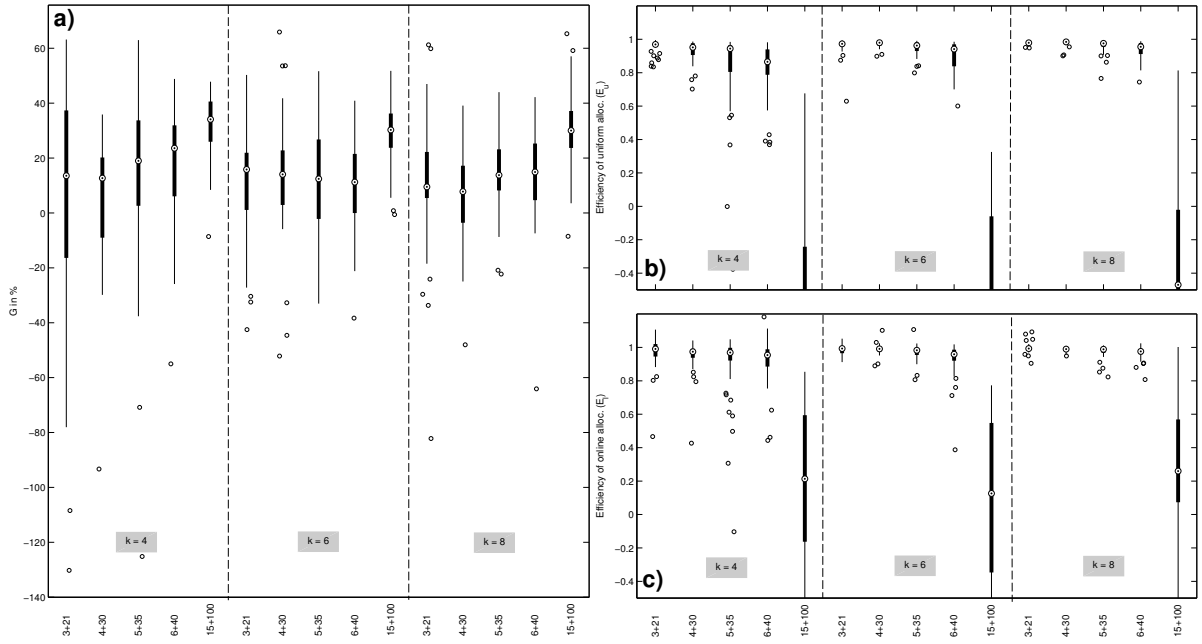

Figure 4: Boxplots of population-based search heuristic configurations over 40 runs on the $F_1(\vec{x})$ with exponentially sampled processing times ($\mathcal{E}(2)$). The configurations comprise various population sizes ($\mu + \lambda$) and number of heuristics ($k = 4, 6, 8$). Figure (a) shows the gain of the online learning over uniform allocation in %. Figure (b) shows the efficiency of uniform allocation and Figure (c) the efficiency of the online learning approach.

Processing Time ~ $\mathcal{U}(1,5)$					
$\mu + \lambda$	3+21	4+30	5+35	6+40	15+100
Average T_b					
K=4	1270.33	643.58	535.76	373.73	459.85
K=6	909.24	513.11	415.12	329.23	340.49
K=8	683.02	449.49	359.86	329.66	279.06
Average T_w					
K=4	9447.47	6070.25	5193.26	3362.32	922.88
K=6	8846.67	8394.89	6237.05	4213.93	744.49
K=8	9512.00	8716.71	7045.93	5217.18	787.73
Average T_u					
K=4	1872.82	1298.05	1318.86	1016.13	1429.19
K=6	1332.73	988.12	996.75	882.23	1350.82
K=8	993.22	856.79	848.25	862.34	1345.67
Average T_l					
K=4	1809.38	1122.08	1083.63	784.58	947.58
K=6	1223.73	831.80	775.60	738.83	928.75
K=8	872.40	800.28	671.08	691.35	917.18

Processing Time ~ $\mathcal{E}(2)$					
$\mu + \lambda$	3+21	4+30	5+35	5+35	6+40
Average T_b					
K=4	2185.64	1189.02	1013.96	728.44	643.79
K=6	1563.15	952.45	788.95	643.63	534.36
K=8	1185.85	828.04	683.09	642.75	504.45
Average T_w					
K=4	16239.86	11229.63	9876.43	6550.66	1288.59
K=6	15244.93	15499.63	11856.80	8223.91	1168.56
K=8	16330.41	16094.16	13428.35	10170.93	1421.76
Average T_u					
K=4	2752.20	1652.56	1533.94	1185.03	1556.82
K=6	1987.23	1286.34	1164.10	1016.80	1472.83
K=8	1500.02	1109.70	996.65	995.02	1465.85
Average T_l					
K=4	2552.45	1524.03	1439.48	984.83	1062.55
K=6	1769.90	1104.13	989.05	922.90	1044.15
K=8	1250.70	1035.00	855.78	867.30	1016.35

Table 2: Ackley Function over 40 runs.

Processing Time ~ $\mathcal{U}(1,5)$					
$\mu + \lambda$	3+21	4+30	5+35	6+40	15+100
Average T_b					
K=4	726.87	561.31	388.96	339.26	333.73
K=6	685.71	345.14	308.59	250.56	244.30
K=8	511.64	286.01	262.56	217.19	190.32
Average T_w					
K=4	8269.21	6326.30	6066.77	4951.90	748.49
K=6	9235.50	8458.56	7228.31	5699.95	1516.46
K=8	9593.47	9431.75	7707.12	7182.11	1366.91
Average T_u					
K=4	1074.58	1130.26	956.03	922.98	1035.93
K=6	1004.99	670.11	742.15	672.31	970.96
K=8	747.36	547.97	618.11	570.55	916.25
Average T_l					
K=4	1097.50	1002.13	737.48	775.18	727.40
K=6	964.88	566.60	577.28	543.45	620.78
K=8	689.13	458.08	504.40	445.58	618.83

Processing Time ~ $\mathcal{E}(2)$					
$\mu + \lambda$	3+21	4+30	5+35	6+40	15+100
Average T_b					
K=4	1106.01	920.56	667.02	593.53	376.75
K=6	1033.57	561.38	523.75	437.34	331.11
K=8	772.53	470.61	445.73	380.79	307.20
Average T_w					
K=4	12493.72	10370.81	10291.12	8658.72	860.73
K=6	13966.17	13873.15	12269.04	9941.66	2053.86
K=8	14507.25	15462.75	13084.37	12552.53	2204.08
Average T_u					
K=4	1352.06	1189.07	882.53	833.81	801.79
K=6	1309.81	723.13	691.49	606.01	757.23
K=8	995.53	605.05	590.84	521.74	714.76
Average T_l					
K=4	1377.85	1160.00	814.63	796.00	607.13
K=6	1214.33	658.45	650.30	559.43	522.28
K=8	854.48	544.48	546.45	479.33	516.58

Table 3: Absolute results values (time in seconds) for the Rastrigin function ($F_2(\vec{x})$) over 40 runs.