A Natural Evolution Strategy with Asynchronous Strategy Updates

Tobias Glasmachers
Institut für Neuroinformatik
Ruhr-Universität Bochum, Germany
tobias.glasmachers@ini.rub.de

ABSTRACT

We propose a generic method for turning a modern, non-elitist evolution strategy with fully adaptive covariance matrix into an asynchronous algorithm. This algorithm can process the result of an evaluation of the fitness function anytime and update its search strategy, without the need to synchronize with the rest of the population. The asynchronous update builds on the recent developments of natural evolution strategies and information geometric optimization.

Our algorithm improves on the usual generational scheme in two respects. Remarkably, the possibility to process fitness values immediately results in a speed-up of the sequential algorithm. Furthermore, our algorithm is much better suited for parallel processing. It allows to use more processors than offspring individuals in a meaningful way.

Categories and Subject Descriptors

[**Evolution Strategies and Evolutionary Programming**]

General Terms

Algorithms

Keywords

Evolution strategies, Speedup technique, Parallelization

1. INTRODUCTION

In pure form, most evolutionary algorithms (EAs) operate in generation cycles. For many specific variants it is easy to weaken this assumption and to allow for anytime or asynchronous application of certain operators, such as selection of parents, generation of offspring (application of variation operators), as well as the evaluation of offspring, an operation that is typically considered expensive. Other operators basically require synchronous operation, such as survivor selection. Among these, the evaluation of individuals by means of the fitness function is of primary interest, since it is a standard assumption that the lion's share of the overall computation time is spent on this step. The possibility to perform individual steps asynchronously enables more efficient parallelism, since synchronization points form potential bottlenecks.

Evolution Strategies (ESs) are special within the wider field of evolutionary computation in that they rely heavily on active "self"-adaptation of their search or mutation distribution. While this technique is a blessing in many respects (it enables linear convergence into twice continuously differentiable optima), it is also a curse, since it can interfere with typical difficulties of evolutionary optimization, such as fitness noise and constraint handling. The same holds for asynchronous processing. In the present study we show how to make fitness evaluation in a modern non-elitist evolution strategy an asynchronous operation, while preserving meaningful strategy updates of the full covariance structure.

Parallelization of evolutionary algorithms (EAs) has been subject to a large number of studies since EAs, being population-based algorithms, are well suited for parallelization. Different types of parallelism have been identified, see e.g. [1]. Relatively simple master-slave architectures can distribute fitness evaluations within an offspring population to multiple processors. More complex systems are based on distributed or otherwise structured populations, like for example island models [2] with synchronous or asynchronous message passing.

Recently there have been impressive demonstrations of massively parallel evolutionary algorithms using huge population sizes of tens of thousands [7]. Such implementation rely heavily of the general purpose computing capabilities of modern graphics processing units (GPUs). Such massively parallel implementations are most commonly found for genetic algorithms (GAs) and genetic programming (GP) systems, where the search distribution is defined by static operators (without self-adaptation) applied to the current population.

In general, parallelism and synchronity of different steps on an algorithm are distinct properties. However, asynchronicity of an operation is useless in a strictly sequential program, while it can avoid synchronization overheads in a parallel computation. Asynchronicity is not a prerequisite for parallelism, but it can increase its efficiency.

In this paper we do not aim for the massive parallelism mentioned above. Speed-ups in the order of hundreds are not to be expected for (todays) ESs, simply for their comparatively small population sizes. Therefore, instead of GPU hardware, we consider a setting that is better fitted for the

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

standard ES: assume the fitness evaluation involves a computation intensive simulation that needs a full blown CPU-based architecture to run efficiently. We have a few dedicated compute servers available for the task, with a total of c independent processors, with c in the order of maybe dozens. Then we can choose any multiple of c for the offspring population size n to obtain a basically linear speed-up, compared to the naive sequential implementation. This can be achieved with a simple master-slave architecture. It is assumed in particular that the lion's share of the computation time is spent on fitness evaluations, not on communication overhead, strategy updates, sampling of random numbers, and relatively cheap bookkeeping tasks that are performed after synchronization by the central master process. Such work has been conducted, e.g., for the highly efficient CMA-ES algorithm [5].

Although we have already made quite a few assumptions at this point, our performance calculation can still turn out to be very far off. For example, it may well happen that the ES would actually run at its highest efficiency with a smaller population size. While this effect is often small (although considerable for huge populations, see e.g. [5]), another one may turn out to be crucial: the runtime of the simulation may vary unpredictably (with the search point, with the compute node, with factors deeply hidden in industrial simulation software, or with other factors outside our control). Then most processors will have to wait for the synchronization with the slowest-to-evaluate individual in the population.

A principled solution to this problem is to break up the generation cycle and to turn to asynchronous algorithms. The synchronous generation cycle of many EAs is obviously an over-simplification of natural evolution, and there has been considerable work to overcome this restriction algorithmically [2].

The standard solution to the asynchronous update requirement in EAs is steady state selection. The elitist character of selection schemes directly suitable for asynchronicity is problematic in the presence of multi-modality and noise in fitness values, which are both common characteristics of fitness functions based on simulations.

Furthermore, most existing schemes are designed for genetic algorithms or genetic programming systems and therefore ignore issues of search strategy updates. It is a priori unclear how step size (or full covariance) adaptation can be achieved in a fully asynchronous evolution strategy with a non-trivial offspring population.

Recent research on evolution strategies in particular [11] and on randomized optimization in general [3] emphasizes the connection between the updates of mean and covariance of the search distribution. This discourages asynchronous updates of the mean (e.g., the elitist) together with a separate, possibly synchronous, and thus different type of update for the covariance.

This paper tries to fill this gap with a straightforward approach to asynchronous search strategy updates within the framework of so-called *natural evolution strategies (NES)*. This framework has recently been put into an even broader context by the work on information geometric optimization [3]. This approach interprets certain ESs (e.g., CMA-ES [6] and xNES [4]) as a time discrete Monte Carlo approximated continuous flow in the space of search distributions. Since the flow is given by a gradient, strategy updates essentially become stochastic gradient steps in parameter space, which are augmented by learning rates.

Our approach modifies a generational ES so that it becomes a fully asynchronous algorithm. The asynchronous scheme creates an offspring x_i as soon as a compute node becomes idle. This compute node is then busy for some time evaluating the offspring's fitness $f(x_i)$. In general, fitness values may be returned in an arbitrary order by the compute nodes, so that evaluated offspring $(x_i, f(x_i))$ become available in an order that is different from the generation of offspring. In this setting we argue that evaluated individuals should not be viewed as members of populations in a discrete cycle, but rather as a continuous stream of individuals.

This view offers the opportunity to use all information as soon as they become available for updating the search distribution from which new offspring are generated. Now assume that a compute node has just returned the fitness value of an individual. Put in a nutshell, our approach amounts to applying a $1/n$ fraction of the strategy update of a standard generational ES, to account for the higher update frequency, using the n most recently arrived individuals from the stream as the current population. This rule needs minor corrections to account for noise induced by delayed arrival of individuals due to varying evaluations times.

Our simple yet efficient scheme can deal with variation in the runtimes of fitness evaluations without wasting computation time. Interestingly it also improves sequential search on a single core. The reason is that the first individual of a hypothetical offspring population is readily evaluated before the second one needs to be generated. The information contained in the first fitness value can already guide the search to better points, even within a single generation.

The remainder of this paper is organized as follows: Natural evolution strategies in general and the xNES algorithm in particular are presented in the context of information geometric optimization. Then we introduce the asynchronous update. The new algorithm is benchmarked in a number of different sequential and parallel settings against the population-based ES. We close with our conclusions.

2. NATURAL EVOLUTION STRATEGIES

Natural evolution strategies (NES) [11] are a class of evolutionary algorithms for real-valued optimization. The principal idea is to adapt the search distribution to the problem at hand by following the natural gradient of expected fitness in parameter space. NES exist in different variants, mostly with fully adaptive covariance matrix [9, 4], but the very same principle can be applied to any class of distributions with continuous parameters (see, e.g, [8, 3]).

The recently developed framework of information geometric optimization (IGO) [3] offers a modern perspective on natural evolution strategies. IGO algorithms update their search distribution by a rule that follows a Monte Carlo approximation of a continuous flow on the statistical manifold formed by the class of search distributions in use. The flow direction points towards better fitness. A scalar objective function is constructed from the distribution of fitness values under the current search distribution as a weighted combination of fitness quantiles. The IGO flow is defined locally as the natural gradient field of this objective (see [3] for details). Thus, distribution updates of IGO algorithms can be understood as stochastic gradient ascent steps. However, the IGO flow is designed such that these steps can be

Algorithm 1: The xNES Algorithm

Input: $d \in \mathbb{N}$, $f : \mathbb{R}^d \to \mathbb{R}$, $m \in \mathbb{R}^d$, $A \in \mathbb{R}^{d\times d}$
$\sigma \leftarrow \sqrt[d]{|\det(A)|}$
$B \leftarrow \sigma^{-1} \cdot A$
while *stopping condition not met* **do**
  for $i \in \{1, \ldots, n\}$ **do**
    $z_i \leftarrow \mathcal{N}(0, I)$
    $x_i \leftarrow m + \sigma B \cdot z_i$
  end
  end
  sort $\{(z_i, x_i)\}$ with respect to $f(x_i)$
  $G_m \leftarrow \sum_{i=1}^n u_i \cdot z_i$
  $G_A \leftarrow \sum_{i=1}^n u_i \cdot (z_i z_i^T - I)$
  $G_\sigma \leftarrow \mathrm{tr}(G_A)/d$
  $G_B \leftarrow G_A - G_\sigma \cdot I$
  $m \leftarrow m + \eta_m \cdot \sigma B \cdot G_m$
  $\sigma \leftarrow \sigma \cdot \exp(\eta_\sigma \cdot G_\sigma)$
  $B \leftarrow B \cdot \exp(\eta_B \cdot G_B)$
end
end

performed with a fixed rather than a decaying learning rate. We refer the reader to the excellent paper [3] for further details and for a more gentle introduction to the topic.

The most prominent application of IGO to evolution strategies are strategy updates for Gaussian search distributions, parameterized by mean $m \in \mathbb{R}^d$ and covariance matrix $C \in \mathbb{R}^{d\times d}$. The approach guarantees convenient invariance properties, since the flow direction is independent of the parameterization of the underlying distributions.

Our general proceeding is applicable to any IGO algorithm, and to any NES algorithm in particular. For demonstration purposes we apply it to the xNES (exponential NES) algorithm [4]. Its name originates from the locally exponential parameterization of the covariance matrix. Pseudo code is presented in algorithm 1.

In each generation, xNES samples a population of $n \in \mathbb{N}$ individuals $x_i \sim \mathcal{N}(m, C)$, $i \in \{1, \ldots, n\}$, i.i.d. from its search distribution, which is represented by the center $m \in \mathbb{R}^d$ and a factor $A \in \mathbb{R}^{d\times d}$ of the covariance matrix $C = AA^T$. These points are obtained by sampling $z_i \sim \mathcal{N}(0, I)$ (where $I \in \mathbb{R}^{d\times d}$ is the unit matrix) and setting $x_i = m + A \cdot z_i$. Let $p(x \,|\, m, A)$ denote the density of the search distribution $\mathcal{N}(m, AA^T)$. Then

$$J(m, A) = \mathbb{E}[f(x) \,|\, m, A] = \int f(x)\; p(x \,|\, m, A)\; dx$$

is the expected fitness under the current search distribution. The so-called 'log-likelihood trick' enables us to write

$$\begin{aligned}
&\nabla_{(m,A)} J(m, A) \\
&= \int \Big[f(x)\; \nabla_{(m,A)} \log(p(x \,|\, m, A)) \Big]\; p(x \,|\, m, A)\; dx \\
&\approx \frac{1}{n} \sum_{i=1}^{n} f(x_i)\; \nabla_{(m,A)} \log(p(x \,|\, m, A)) \;\; .
\end{aligned}$$

Replacement of raw fitness values $f(x_i)$ with rank-based weights or utilities u_i turns this quantity into the (Monte Carlo estimate of the) gradient of the local IGO objective function. The ranked offspring fitnesses are nothing but a Monte Carlo sample of the true fitness quantiles. Thus, with utility values resembling IGO's quantile weights,

$$\frac{1}{n} \sum_{i=1}^{n} u_i \cdot \nabla_{(m,A)} \log(p(x_i \,|\, m, A)) \tag{1}$$

is a Monte Carlo sample of the gradient of the IGO objective. When multiplied with the inverse of the Fisher matrix (the metric tensor of the inner geometry of the statistical parameter manifold), this vector turns into the natural gradient, which is the canonical ascent direction, since it is invariant under changes of parameterization.

parameter	default value
n	$4 + \lfloor 3 \log(d) \rfloor$
η_m	1
$\eta_\sigma = \eta_B$	$\frac{3}{5} \cdot \frac{(3+\log(d))}{d\sqrt{d}}$
u_i	$\frac{\max\left(0, \log(\frac{n}{2}+1) - \log(i)\right)}{\sum_{j=1}^{n} \max\left(0, \log(\frac{n}{2}+1) - \log(j)\right)} - \frac{1}{n}$

Table 1: Default parameter values for xNES as a function of the problem dimension d. The number of samples n and the utilities u_i have been taken over from CMA-ES [6].

The xNES algorithm implements the IGO update rule for the special case of Gaussian search distributions. It uses a local parameterization of the manifold of Gaussian distributions. Coordinates are chosen so that the current search distribution becomes a standard normal distribution. This simple trick turns the Fisher metric into the identity matrix, which saves its explicit computation, and moreover its inversion. The positive definite, symmetric covariance matrix is represented by unconstrained parameters in the form $C = \exp(M)$, with M being symmetric, but not necessarily positive definite. Then, w.l.o.g., we can assume $A = \exp\left(\frac{1}{2}M\right)$. The covariance factor A is further split into a step size parameter σ and a shape matrix B of unit determinant, so that it holds $A = \sigma B$. We refer the reader to [4] for a detailed discussion of the rationale behind this representation. Straightforward application of equation (1) to the state variables m, σ, and B results in the IGO flow gradient components G_m, G_σ, and G_B as presented in algorithm 1. The last three lines of the algorithm perform a gradient step with learning rates η_m, η_σ, and η_B, in the non-linear coordinates induced by the exponential map. The default settings of all parameters are summarized in table 1.

3. ASYNCHRONOUS STRATEGY UPDATES

Turning a generational ES into an asynchronous one is in general a challenging undertaking, in particular with non-elitist "comma" selection. Not only does the selection require a fixed generation cycle, even so does the strategy adaptation rule. For NES and IGO algorithms the task is greatly simplified by the fact that strategy updates are stochastic gradient steps. These steps do already involve (typically empirically tuned) learning rate parameters.

Our goal is to perform a search strategy update instantaneously as soon as a new fitness evaluation becomes available. The basic idea is to perform the standard update each time, but with a learning rate that is reduced by a factor of $1/n$ (with n being the number of offspring in the generational ES). Formally, let θ be the state vector of an IGO

algorithm, and let the random variable G_θ be the current Monte Carlo sample of the IGO flow vector, obtained from a sample of n offspring. Then the generational algorithm would perform the update

$$\theta \leftarrow \theta + \eta \cdot G_\theta$$

which is replaced by n updates

$$\theta \leftarrow \theta + \frac{\eta}{n} \cdot G_\theta \ ,$$

one for each fitness evaluation. In their simplicity these equations do not describe the algorithm in detail. The following questions remain open:

1. The strategy update of an ES with "comma" selection requires a whole offspring population, not only a single individual. How should the current population be formed from the stream of evaluated individuals? It surely needs to consist of already evaluated points, but these may not arrive in the order of their creation.

2. How to account for the fact that there is not a single source distribution for all offspring in the current population, but each individual is drawn from its own distribution?

3. How to deal with added variation in the fitness due to variance in the delay with which fitness evaluations become available?

A straightforward answer to all three questions at once is importance mixing. This technique has been introduced quite early into NES algorithms [9]. The original idea is to reuse offspring from recent generations by reweighting them so that their impact is proportional to the quotient of their density in the current search distribution divided by their density at the time of their creation. This technique is an a posteriori correction inspired by standard importance sampling. Applied to the problem at hand one may compose an effective population as a weighted combination of readily evaluated individuals from the stream, with importance mixing weights, so that the sum of weights equals the nominal population size n. This technique automatically accounts for delays in fitness evaluation and for individuals generated from other than the current search distribution: if an individual happens to stem from an outdated search distribution, then its weight is usually very small. However, the Gaussian density decays quickly with the distance to the mean, and so the quotient of such densities can take extreme values. For this reason the ES using reweighting can easily become unstable. We observe such instabilities in particular in high dimensional search problems, and sometimes even with dimensions as low as three. Despite its conceptual beauty we therefore discard this approach.

A conceptually much simpler and seemingly ad hoc solution is to answer question 1 as follows: always consider the n most recently arrived fitness evaluations from the stream as the current population.[1] This simple choice has the advantage that every individual takes part in exactly n strategy updates. This is why this scheme is much less prone to instability. As a consequence, answers to questions 2 and 3 above become non-trivial. Our equally naive answer to question 2 is to simply ignore the effect, instead of performing density-based corrections. This means that the Monto Carlo estimate of IGO's fitness quantiles may be systematically biased. Also, question 3 points at a newly introduced source of uncertainty, namely that the orders of offspring generation and evaluation differ. Instead of performing active corrections we rely on the working principle of IGO algorithms. We simply lower the learning rate in order to obtain a better approximation of the underling flow, which stabilizes the algorithm agains all kinds of noise effects.

It turns out that for the xNES algorithm these effects are most pronounced for small problem dimensions d. This is because of the sub-linear growth of the population size with the problem dimension, and the resulting decay of the covariance matrix learning rate. Therefore, in high dimensions subsequent search distribution are usually close. Another factor of influence is the number c of available compute nodes. A single CPU results in a sequential algorithm, which removes the variability due to delayed arrival of fitness values, and leaves us only with the effect of offspring being sampled from different distributions. With increasing number of CPUs the variation in delays starts to grow.

We have found empirically that asynchronous updates can result not only in a slowdown, but even in a complete breakdown of the algorithm. A decrease of the IGO step learning rates by the factor

$$\nu = \left(\frac{2}{3}\right)^{(2c)/(nd)}$$

seems sufficient as a counter measure. It results in a reliable stabilization of the algorithm. In many realistic cases the correction factor is close to one, in particular for high dimensional problems.

A minor side effect of asynchronous strategy updates is that the first update can be performed already before a full population of size n has been evaluated. The only difference is that the effective population cannot consist of n individuals, which requires a corresponding correction of the utility weights.

Putting everything together, the pseudo code of the asynchronous xNES algorithm is summarized in algorithm 2.

4. EXPERIMENTS

The following questions guide our experimental evaluation of the asynchronous xNES algorithm:

1. What is the effect of asynchronous strategy updates on the sequential algorithm? How does asynchronous xNES perform compared to the original generational xNES algorithm?

2. How do the two algorithms compare when running fitness evaluations in parallel?

3. How close does asynchronous xNES come to linear speed-ups w.r.t. the number of processors when running fitness computations in parallel?

For the assessment of the first two points we have reproduced the experimental setup of [4]. In that study, the xNES algorithm was compared to CMA-ES on nine standard benchmark functions in dimensions $d \in \{2, 4, 8, 16, 32, 64\}$ (refer to [4] for more details). The initial search distribution is $\mathcal{N}(x, I)$, with center component $x \sim \mathcal{N}(0, I)$ being itself a random variable that varies across trials. The assessment

[1] A related approach has been introduced in [10].

Algorithm 2: The asyncronous xNES Algorithm

Input: $d \in \mathbb{N}$, $f : \mathbb{R}^d \to \mathbb{R}$, $m \in \mathbb{R}^d$, $A \in \mathbb{R}^{d \times d}$
$\sigma \leftarrow \sqrt[d]{|\det(A)|}$
$B \leftarrow \sigma^{-1} \cdot A$
$P \leftarrow \emptyset$
forall the *compute nodes* **do**
  $z \leftarrow \mathcal{N}(0, I)$
  $x \leftarrow m + \sigma B \cdot z$
  start computation of $f(x)$
end
end

while *stopping condition not met* **do**
  wait for compute node to finish; result: $\big(z, x, f(x)\big)$
  $P \leftarrow P \cup \big(z, x, f(x)\big)$
  If $|P| > n$ **Then** remove oldest individual from P
  sort P with respect to $f(x)$
  $G_m \leftarrow \sum_{i=1}^{|P|} u_i \cdot z_i$
  $G_A \leftarrow \sum_{i=1}^{|P|} u_i \cdot (z_i z_i^T - I)$
  $G_\sigma \leftarrow \mathrm{tr}(G_A)/d$
  $G_B \leftarrow G_A - G_\sigma \cdot I$
  $m \leftarrow m + \frac{\nu}{n} \cdot \eta_m \cdot \sigma B \cdot G_m$
  $\sigma \leftarrow \sigma \cdot \exp\big(\frac{\nu}{n} \cdot \eta_\sigma \cdot G_\sigma\big)$
  $B \leftarrow B \cdot \exp\big(\frac{\nu}{n} \cdot \eta_B \cdot G_B\big)$
  $z \leftarrow \mathcal{N}(0, I)$
  $x \leftarrow m + \sigma B \cdot z$
  start computation of $f(x)$ on the idle node
end
end

was assuming a standard sequential scheme, using the number of fitness evaluations for reaching a target fitness value as its evaluation criterion.

For the present study we have modified the setup in two respects. First, we simulate parallel fitness evaluations with an adjustable number of processors. Second, we introduce a distribution of running times for fitness evaluations. This runtime is on an abstract scale, and it is uniformly distributed on logarithmic scale in the form $t = T^u$, where $T \geq 1$ is an adjustable parameter controlling variability, and $u \sim U([0, 1])$ in drawn uniformly from the unit interval. This gives random simulated runtimes in the interval $t \in [1, T]$. Importantly, the runtime t of an evaluation of the fitness function is known to the algorithm only after the process has finished, and the random quantity t is independent of the search point x and the fitness value $f(x)$. We use $T = 3$ and $T = 10$ in our experiments.

Unpredictable runtimes are an issue that is not tested by standard benchmark problems. The above mechanism allows us to randomize runtimes of established benchmarks, and thus to use them as surrogates of simulation-based fitness functions with runtime variability.

When running the original xNES on multiple processors, we need to parallelize the process for comparison. We make optimal use of available resources as follows. If the number c of processors is greater or equal to the population size n then we start all fitness evaluations in parallel. Otherwise we fill all available processors. Once the first evaluation finishes, the processor is given a new task, until the whole population has been scheduled for evaluation. Then all we can do is wait for all evaluations to finish.

In this study we report the median over 100 independent runs for all performance metrics. The distributions turn out to be sufficiently concentrated, so that reporting other quantiles or the mean does not change the general picture.

4.1 Asynchronous, Sequential Updates

In a first experiment we compare two sequential algorithms, running on a single processor ($c = 1$), namely generation based xNES and asynchronous xNES. In this case the parameter T does not matter at all and we resort to the number of fitness evaluations as our evaluation criterion. The only difference between the two algorithms is that the asynchronous version makes smaller updates more often. Since the two algorithms are very similar, differences are expected to be small. The results are presented in the first row of figure 1a. It turns out that the asynchronous algorithm is always faster than the generation-based one, with a surprisingly stable speed-up across all tested benchmark problems. We obtain a very clear answer to question 1: the asynchronous scheme gives a reliable speed-up. The amount of speed-up depends on the dimension, but it is rather independent of the problem. In low dimensions the effect is most pronounced, with speed-ups of about $20\% - 30\%$.

4.2 Parallel Fitness Evaluations

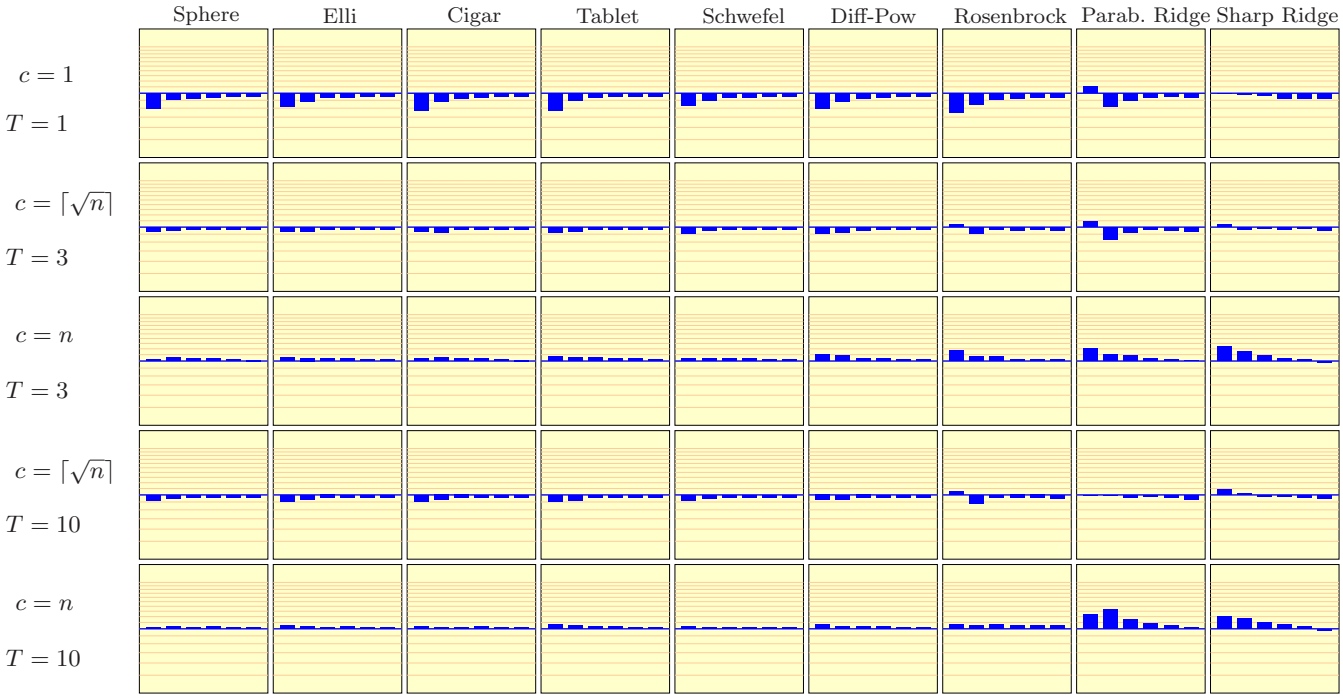
The second question is a bit harder to answer. The generation-based version can make use of at most n processors, which usually is a function of the problem dimension. We have varied the number of processors $c \in \{1, \lceil\sqrt{n}\rceil, n\}$ and the time variability $T \in \{3, 10\}$. The resulting number of fitness evaluations and the required wall clock time are presented in figure 1. Note that this comparison to biased towards the generation based algorithm, which cannot benefit from $c > n$ processors at all.

It is expected that the number of fitness evaluations increases as a price for delayed arrival of fitness evaluations in the parallel, asynchronous algorithm. This is indeed the case when comparing to the sequential, asynchronous algorithm. In comparison to the synchronous baseline the effect is compensated in most cases by the performance gain that originates from using available information immediately. The only notable exceptions are the "ridge" benchmarks for relatively large numbers of parallel processors, where the number of fitness evaluations increases in some cases by about 30%.

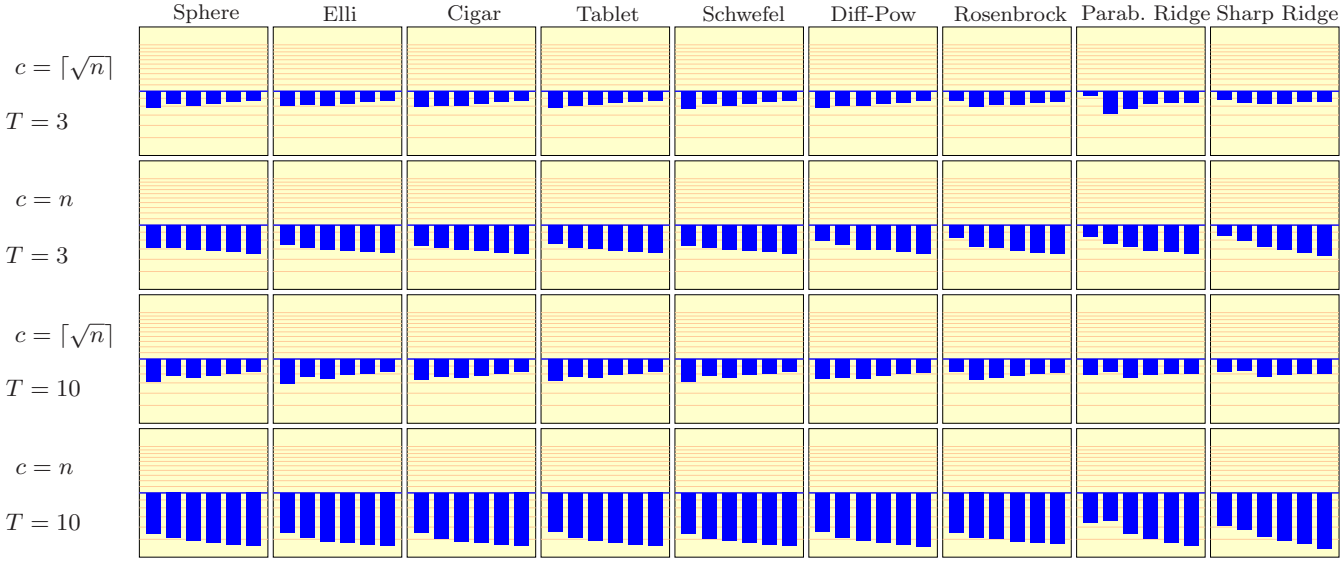
In a parallel computation we are usually most concerned with the overall wall clock time. Here we ask specifically for the effect of asynchronous strategy updates. It turns out that the parallel, asynchronous algorithm provides extensive savings over the also parallel, but synchronous one. Although both algorithms use the same number of processors, the asynchronous scheme saves between 10% and more than 50% of the wall clock time, despite requiring slightly more fitness evaluations. This effect is due to better processor use over time, enabled by the asynchronous update scheme.

4.3 Linear Acceleration

To assess the evolution of the speed-up with increasing number of processors we vary the number of processors in the range $c \in \{1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 25\}$, for the Rosen-



(a) fitness evaluations



(b) runtime (wall clock time)

Figure 1: The figure presents relative (a) number of function evaluations and (b) running times of asynchronous versus generations xNES. Five test configurations (rows, first configuration is only in (a)) and nine fitness functions (columns) are shown. The six bars in each square represent relative resource demands of the asynchronous scheme as compared to the generational baseline, in dimensions 2, 4, 8, 16, 32, and 64. Bars upwards and downwards from the center line indicate a relative increase or decrease, respectively, on logarithmic scale. Thus, downwards bars represent improvements of asynchronous xNES over its predecessor. Each small red line indicates 10% of difference. The indicators range from halved (50%) to doubled (200%) computational demands.

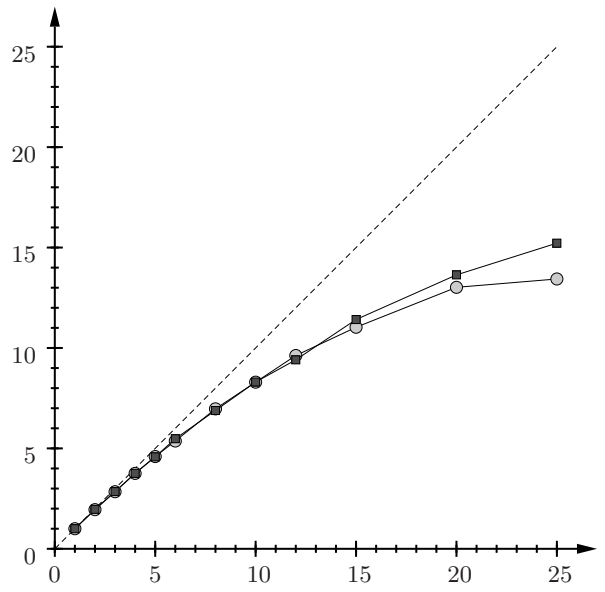


Figure 2: Plot of the speed-up over the number of processors. Let $\tau(c)$ denote the runtime with c processors in parallel, then the graph represents the function $\tau(1)/\tau(c)$. The plot is based on median times over 100 independent runs on the Rosenbrock benchmark in dimension $d = 8$. The two curves refer to the time variabilities $T = 3$ (light circles) and $T = 10$ (dark squares). For comparison, the dashed line indicates linear speed-up.

brock problem in dimension $d = 8$ with nominal population size of $n = 10$. The resulting speed-up as functions of the number of processors is illustrated in figure 2.

The plot indicates that the speed-up from parallelization of the asynchronous xNES algorithm is nearly linear in the number of processors up to about the nominal population size. Then it drops to a sub-linear speedup. The speed-up increases consistently even for numbers of processors that exceed the nominal population size n. For comparison note that for $c > n$ the generation-based algorithm is unable to achieve any further speed-up.

We can answer question 3 as follows: asynchronicity helps to use multiple processors well. Up to about $c = n$ processors, the speed-up is nearly linear. Beyond this point the speed-up is sub-linear, but importantly consistently increasing with increasing number of parallel processors.

5. CONCLUSION

In this paper we have introduced a generic, asynchronous strategy update for a class of non-elitist evolution strategies. The natural gradient interpretation of evolution within natural evolution strategies and the information geometric optimization framework provide a canonical way of turning the classic population-based update into an asynchronous one.

We have demonstrated that using available information immediately—not only after a full offspring generation has been evaluated—improves performance of sequential algorithms. The same mechanism opens the door for efficient parallel fitness evaluation, even if the evaluation time differs vastly among individuals. We have shown that nearly linear speed-ups can be achieved for small numbers of processors. While the speed-up naturally saturates, our method can still benefit from a number of processors that exceeds the nominal offspring population size.

6. REFERENCES

[1] P. Adamidis. Parallel evolutionary algorithms: A review. In *Proceedings of the 4th Hellenic-European Conference on Computer Mathematics and its Applications (HERCMA 1998), Athens, Greece*. Citeseer, 1998.

[2] E. Alba and J.M. Troya. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems*, 17(4):451–465, 2001.

[3] L. Arnold, A. Auger, N. Hansen, and Y. Ollivier. Information-geometric optimization algorithms: a unifying picture via invariance principles. *arXiv preprint arXiv:1106.3708*, 2011.

[4] Tobias Glasmachers, Tom Schaul, Yi Sun, Daan Wierstra, and Jürgen Schmidhuber. Exponential Natural Evolution Strategies. In *Genetic and Evolutionary Computation Conference (GECCO)*, Portland, OR, 2010.

[5] N. Hansen, S. D. Müller, and P. Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary Computation*, 11(1):1–18, 2003.

[6] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[7] F. Krüger, O. Maitre, S. Jimenez, L. Baumes, and P. Collet. Speedups between x70 and x120 for a generic local search (memetic) algorithm on a single GPGPU chip. In C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. Esparcia-Alcazar, C.-K. Goh, J. Merelo, F. Neri, M. Preuß, J. Togelius, and G. Yannakakis, editors, *EvoNum 2010*, volume 6024 of *LNCS*, pages 501–511. Springer Berlin / Heidelberg, 2010.

[8] T. Schaul, T. Glasmachers, and J. Schmidhuber. High Dimensions and Heavy Tails for Natural Evolution Strategies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2011.

[9] Y. Sun, D. Wierstra, T. Schaul, and J. Schmidhuber. Stochastic Search using the Natural Gradient. In *International Conference on Machine Learning (ICML)*, 2009.

[10] Daan Wierstra, Tom Schaul, Jan Peters, and Jürgen Schmidhuber. Fitness expectation maximization. In *Lecture Notes in Computer Science, Parallel Problem Solving from Nature - PPSN X*, pages 337–346. Springer-Verlag, 2008.

[11] Daan Wierstra, Tom Schaul, Jan Peters, and Jürgen Schmidhuber. Natural Evolution Strategies. In *Proceedings of the Congress on Evolutionary Computation (CEC08), Hongkong*. IEEE Press, 2008.