Parallel Evolutionary Algorithm with Interleaving Generations

Martin Pilát
Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 25
Prague, Czech republic 118 00
Martin.Pilat@mff.cuni.cz

Roman Neruda
Institute of Computer Science
The Czech Academy of Sciences
Pod Vodárenskou věží 271/2
Prague, Czech republic 182 07
roman@cs.cas.cz

ABSTRACT

We present a parallel evolutionary algorithm with interleaving generations. The algorithm uses a careful analysis of genetic operators and selection in order to evaluate individuals from following generations while the current generation is still not completely evaluated. This brings significant advantages in cases where each fitness evaluation takes different amount of time, the evaluations are performed in parallel, and a traditional generational evolutionary algorithm has to wait for all evaluations to finish. The proposed algorithm provides better utilization of computational resources in these cases. Moreover, the algorithm is functionally equivalent to the generational evolutionary algorithm, and thus it does not have any evaluation time bias, which is often present in asynchronous evolutionary algorithms.

The proposed algorithm is tested in a series of simple experiments and its effectiveness is compared to the effectiveness of the generational evolutionary algorithm in terms of CPU utilization.

CCS CONCEPTS

•**Computing methodologies** → *Parallel algorithms; Artificial intelligence; Discrete space search; Continuous space search; Distributed algorithms;*

KEYWORDS

evolutionary algorithms, parallelization, evaluation-time bias, complex optimization, interleaving generations

ACM Reference format:
Martin Pilát and Roman Neruda. 2017. Parallel Evolutionary Algorithm with Interleaving Generations. In *Proceedings of GECCO '17, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3071178.3071309

1 INTRODUCTION

With growing computational power, evolutionary algorithms start to be applied to problems with more complex fitness functions which take longer to evaluate. A popular approach to deal with the problem of slowly evaluating fitness functions is to use a version of a parallel evolutionary algorithm [1]. Probably the simplest option is to use the master-slave version, in which only the fitness function is evaluated in parallel. However, in some of the cases, evaluation of each individual can take a different amount of time. The result of the variance in evaluation time is sub-optimal use of computational resources in case multiple processors are used, as the algorithm has to wait for the individuals that evaluate longer in each generation before the next generation can start. Such situation often appears when evolutionary algorithm are used to tune the parameters of machine learning methods. For example, Pilát *et al.* [3] used genetic programming to evolve machine learning workflows. In this case, small machine learning workflows evaluate quickly, while the evaluation of more complex machine learning workflows can take a considerable amount of time. Another example is the routing in Vehicular ad-Hoc Networks presented by Said and Nakamura [5].

One of the popular possibilities how to increase the utilization of the computational resources is to use the so called asynchronous evolutionary algorithms [6]. In asynchronous evolution, the algorithm does not use any generations, but instead, it generates new individuals as soon as some other individuals are evaluated – in a sense this resembles the $(\mu + 1)$ selection from evolution strategies [2]. This removes the necessity to wait for evaluation of the slowest individuals in each generation and allows 100 percent utilization of computation resources regardless of the number of CPUs used. On the other hand, asynchronous evolutionary algorithms can have considerable evaluation-time bias as discussed by Scott and de Jong [7]. The evaluation-time bias causes the algorithm to converge to areas of the search space with individuals that evaluate faster more often than to areas with more slowly evaluated individuals. The evaluation-time bias can thus hinder the convergence of the algorithm in cases where better individuals evaluate longer than worse individuals. On the other hand, if the better individuals are faster, the evaluation-time bias can in fact help the algorithm. However, we consider the latter case to be much less common than the former one as often the better solutions are more complex.

Here, we present an algorithm that tries to avoid the evaluation-time bias of the asynchronous algorithms by interleaving the generations of the standard generational EA. The idea of the algorithm comes from the observation that some individuals from the next generation can be in fact generated before the whole current generation is evaluated. These individuals can then be evaluated by the idle CPUs that when the algorithm otherwise had to wait for the slower evaluations to finish. In this way the algorithm improves the utilization of computation resources without adding the evaluation-time bias inherent to asynchronous evolution.

GECCO '17, Berlin, Germany

DOI: http://dx.doi.org/10.1145/3071178.3071309

2 EVOLUTIONARY ALGORITHM WITH INTERLEAVING GENERATIONS

In this section, we describe two version of the evolutionary algorithm with interleaving generations (IGEA), these would be called (λ, λ) and $(\lambda + \lambda)$ in the evolution strategy notation. The (λ, λ) version (which will also be called the comma version), has a population of λ parents and in each generation creates λ offspring which are used in the next generation. On the other hand, the $(\lambda + \lambda)$ version (also called the plus version) generates λ offspring from λ parents, and the best λ individuals from the combined population of offspring and parents are selected to the next generation.

There are some features, which are common to both versions of the algorithm and these are described here, the details of each of the version of the algorithm will be described in separate subsections.

The main idea of the IGEA is that one can actually generate some individuals from the next generation even before all the individuals from the current generation finish evaluating. Each individual depends only on a limited number of individuals from previous generation. In the case described bellow, we use binary tournament as the mating selection and only unary and binary genetic operators, therefore, each individual has two parents, each of whom is selected from two random individuals from the previous generation. Thus, the individual depends at most on four individuals from the previous generation and once these individuals are evaluated, it can be generated and submitted for evaluation.

Both versions of the algorithm start by submitting all the individuals from the initial generation for evaluation. From this point on, the algorithm only reacts to newly evaluated individuals. The reaction always starts by assigning the fitness function to the evaluated individual and then the information about the new evaluation is propagated to the next generations. Although the propagation step is similar for both version of the algorithm, there are some differences and therefore we describe it later.

Another common part of both algorithms is the mating selection. We use binary tournament selection to this end, however, the selection needs to be divided into two steps to facilitate the interleaving of generations and to ensure the equivalence to the generational EA. First, once a new generation is started, we generate the list of pairs of indexes of individuals which will be compared in the binary tournament (we call them tournament aspirants). Later, once an evaluated individual is added to the population, the list of aspirants is checked for all pairs that contain the individual and if the other individual from the pair is also evaluated they are compared and the better one is selected for mating, thus the step of tournament selection is finished. The selected individual is placed to the same index in the mating pool as was the index of the pair of aspirants from which it was selected. Thus, the selection is equivalent to the common implementation where the selection of random two individuals is made right before their comparison and the selection is not affected by the order of evaluation of the individuals.

The genetic operators are always applied to two consecutive individuals in the mating pool, with the first individual being on an even position in the list. The genetic operators are applied as soon as both of these individuals are added to the mating pool and, if needed, are directly submitted for evaluation.

The individuals submitted for evaluation are placed into a priority queue which is sorted by the generation in which the individual was created (individuals from earlier generations are evaluated first). In the comma version, the individuals are further sorted by the order in which they are needed for the tournament selection (i.e. by the index of their first position in the aspirant list). The motivation for this is to have the consecutive individuals in the mating pool available as soon as possible.

In the implementation we need to keep track of several pieces of information. Therefore, each individual keeps track of the number of generation it is from and also its index in the generation. The evaluation pool returns a pair which contains the submitted individual and its fitness. The information contained in the individual can be used to assign the fitness to the correct individual in the correct generation. Each generation also has the populations of parents and offspring, it has the mating pool (the individuals selected by the mating selection from the parent population) and the list of tournament aspirants, which contains pairs of indexes of individuals which are compared in the tournament selection. As the algorithm works with multiple generations at once, we actually keep the list of all generations during the whole run of the algorithm. The list is expanded whenever needed. All the additional information is generated and kept track of in the program, but we omit the implementation details in the pseudocodes presented bellow in order to make them more readable and intuitive. The source codes of the implementation are provided as supplementary materials.

2.1 (λ, λ)-IGEA

The comma version of the algorithm as it is simpler than the plus version, mostly because the environmental selection always selects the offspring. This also means that once an offspring is generated, it can be added to the parent population in the next generation even without evaluating it. The individuals need to be evaluated only before their fitness is compared in the tournament selection. This has another interesting implication – some of the individuals are never used in the tournament selection and therefore do not need to be evaluated at all. Let us consider a population of μ individuals. In the tournament selection, 2μ individuals are selected randomly with repetition from the population. This means that the probability that an individual is not selected at all is $(1 - \frac{1}{\mu})^{2\mu}$. For a common setting of $\mu = 100$, this means that the fitness of approximately 13 percent of the individuals is never needed and does not need to be evaluated. As IGEA splits the tournament selection into the generation of aspirants and the selection itself, it is easy to detect which individuals do not need to be evaluated. In this way, the performance of the algorithm is improved considerably even before any interleaving of generations. [1]

The comma version of the algorithm (cf. Algorithm 1) starts by creating the random initial generation. This generation also contains the list of aspirants for the tournament selection. Then, all the individuals which are in the list of aspirants are submitted for evaluation. At this point, the algorithm enters the main loop. In

[1] A similar idea was in fact used by Poli and Langdon [4] in their backward chaining-algorithms. They evaluated only the individuals that affect the results starting from the last generation to the first.

Algorithm 1 (λ, λ)-IGEA

```
G ← [ random initial population ]                ▷ single item list
for all parents in G_0 do
    if parent in aspirants in G then
        Submit parent to pool
    end if
end for
while not happy do                               ▷ termination condition
    i, f ← next evaluation result                ▷ individual and fitness
    Assign fitness f to i in G_i.gen
    Comma-Propagate(i, G)
end while
```

Algorithm 2 Comma-Propagate(i, G)

```
g ← G_ind.gen                                    ▷ current generation
for all aspirant pairs (a, b) in g that contain i do
    if both a and b are already in g and evaluated then
        g_sel[p] ← better of a and b             ▷ p index of the asp. pair
        if p even then                           ▷ get parents
            p_1, p_2 ← g_sel[p], g_sel[p + 1]
        else
            p_1, p_2 ← g_sel[p − 1], g_sel[p]
        end if
        if both p1 and p2 in g_sel then
            g_off[p_1 : p_2] ← create offspring from p_1 and p_2
            Create next generation after g if needed
            n ← generation after g
            Copy new offspring from g to n ▷ comma selection
            for all new offspring o in n do
                if o not evaluated and in aspirants in n then
                    Submit o for evaluation
                else      ▷ The offspring is identical to parent
                    Comma-Propagate(o, G)
                end if
            end for
        end if
    end if
end for
```

each iteration of the loop, the fitness of an individual finished evaluation on the pool of CPUs. The fitness is assigned to the respective individual in the population and the information is propagated.

During the propagation (Algorithm 2), first all the aspirant pairs in the generation are checked and if they contain the evaluated individual, the selection is performed in case both the aspirants are already evaluated. In case the selection was performed, the individual is put on the same index in an array of selected individuals as was the index of the aspirant pair and the algorithm checks, whether both parents are available (the parents are always consecutive individuals starting with the one on even position). If both parents are available, the genetic operators are performed. The offspring are put into an array of offspring (on the same indexes as their parents), and also copied to the parent population in the next generation (this is the comma selection). Then, if the offspring changed during the genetic operators and they are in the aspirant

Algorithm 3 $(\lambda + \lambda)$-IGEA

```
G ← [ random initial offspring population ]      ▷ single item list
for all parents in G_0 do
    Assign small dummy fitness to parent
end for
for all offspring in G_0 do
    Submit offspring to pool
end for
while not happy do                               ▷ termination condition
    i, f ← next evaluation result                ▷ individual and fitness
    Assign fitness f to i in G_i.gen
    Plus-Propagate(i, G)
end while
```

Algorithm 4 Plus-Propagate(i, G)

```
g ← G_ind.gen                                    ▷ current generation
n ← next generation after g, create if does not exist
P ← evaluated parents from g
O ← evaluated offspring from g
S′ ← best |P ∪ O| − λ individuals from P ∪ O
S ← individuals from S′ which are not among parents of n
for all individuals i from S do
    Add i to parents of n
    for all aspirant pairs (a, b) in n that contain individual i do
        if both a and b are already in n and evaluated then
            n_sel[p] ← better of a and b ▷ p index of the asp. pair
            if p even then                       ▷ get parents
                p_1, p_2 ← n_sel[p], g_sel[p + 1]
            else
                p_1, p_2 ← n_sel[p − 1], g_sel[p]
            end if
            if both p1 and p2 in n_sel then
                n_off[p_1 : p_2] ← create offspring from p_1 and p_2
                Create next generation after g if needed
                for all new offspring o in n do
                    Submit o for evaluation
                end for
            end if
        end if
    end for
    Comma-Propagate(o, G)
end for
```

list in the next generation, they are submitted for evaluation in the evaluation pool, otherwise (the offspring is a clone of one of the parents) the offspring is propagated further by calling the propagation recursively, which finishes the propagation step.

An interesting feature of the algorithm is that it submits the individuals in the pool in the beginning and later it only reacts to new evaluations. The propagation of the information in some of the steps leads to the generation and submission of new individuals. The implementation is thus more event-driven than the common implementations of evolutionary algorithms.

2.2 $(\lambda + \lambda)$-IGEA

The $(\lambda+\lambda)$ version of IGEA is similar to the comma version, however, the plus selection requires some non-trivial changes. First of all, in this version, the offspring actually need to be evaluated before they can be placed to the next generation. Also, all of the offspring need to be evaluated in each generation as without their fitness it is not possible to perform the plus selection.

The selection of individuals to the next generation is also more complex, this time, we need to select the best λ of the 2λ parents and offspring. To facilitate the interleaving, we use the fact, that once $\kappa > \lambda$ individuals (parents and offspring) are evaluated, we know that the best $\kappa - \lambda$ individuals from the combined population will definitely be selected to the next generation. We therefore add these individuals to the next generation as soon as possible. We only need to remember which of the parents and offspring were already added to the population and only add the new ones. The new individuals are added to the first free slot in the population.

The plus version of the algorithm (see Algorithm 3) starts again by generating a random initial population. To ease the implementation, we generate dummy parents in this initial population which have small dummy fitness (worse than any other possible individual), and we also generate the population of offspring which is in fact the real initial population (we do this as in the rest of the algorithm, we always evaluate the offspring of the current generation, while the parents are evaluated in the previous generation). After the initialization, all the offspring are submitted to the evaluation pool and the main loop starts. It is in fact the same as in the comma case, the only difference is in the propagation function.

The propagation function in the plus case (Algorithm 4) starts by finding the evaluated parents and offspring from the current generation. If the number of the evaluated individuals (κ) is larger than λ, those from the $\kappa - \lambda$ best individuals which have not yet been added to the next generation are added to the first free slots in the next generation. From this point on, the propagation function continues in a very similar way to the one in the comma case. It first checks all the aspirant pairs and if both of the aspirants in a pair are already evaluated, the step of tournament selection is performed. Afterwards, the function checks, whether both parents are available for the genetic operators, and if they are, new offspring are generated. In case the new offspring need to be evaluated, they are submitted to the evaluation pool. The propagation is called again recursively on the next generation whenever any individual was added to the set of parents (i.e. even in case the offspring were submitted for evaluation), as every time the set of parents change, it is possible that another individual can be selected in the next generation.

3 EXPERIMENTS

In order to evaluate the performance of the IGEA we performed a series of experiments, most of the settings are the same in all the experiments, therefore we present them only once here to save space. The genetic operators, however, do not affect the experiments in Section 3.1. All the experiments use a simple Gaussian mutation with standard deviation of 1 and with probability 0.1 and a two point crossover with probability 0.8. The population size is in all cases set to 100 individuals and the algorithms are run until they make 10,000 fitness evaluations. The experiments are repeated 20 times for each of the algorithms and settings used and, unless otherwise noted, we report the median values of the quantity of interest (displayed by the lines in the plots) and also the first and third quartile (shown as the shaded areas around the line). The differences in the two quartiles are actually pretty small in most of the cases, so the shaded areas are hard to see in many of the plots as they are too close to the lines.

As baseline algorithms we use $(100, 100)$-EA, $(100 + 100)$-EA and an asynchronous EA. The $(100, 100)$-EA starts by generating a random initial population of 100 individuals. Those of these individuals which are needed in the tournament selection are submitted for parallel evaluation and evaluated. Then, the main loop starts. In each generation, the tournament selection is finished, afterwards new individuals are generated and put directly to the next generation. Those that are new (i.e. are not copies of their parents) and are needed in the tournament selection in the next generation are again evaluated in parallel. Here, we use the observation that some of the individuals are never needed in the tournament and thus the algorithm does not need to evaluate them. With the above described settings, the algorithm evaluates approximately 70 new individuals in each generation.

The $(100 + 100)$-EA works in a similar way. It generates the initial population of 100 individuals and submits it for parallel evaluation. Once the whole population is evaluated, the tournament selection is used to create the mating pool, the genetic operators are used to create the offspring and the newly created individuals are evaluated. Then, the best 100 of the combined parents and offspring are selected to the next generation and the algorithm continues in the same way until the termination condition is met. With the settings described above, the algorithm submits slightly more than 80 individuals for evaluation in each generation.

Finally, the asynchronous EA starts also by generating 100 random initial individuals and submit them for evaluation. Once an individual is evaluated its fitness is set in the population. As soon as the algorithm has more than half of the population evaluated, it starts creating new individuals. To this end, it performs the tournament selection on the set of evaluated individuals (it selects two random individuals, compare their fitness and the better one is selected as the parent, this is repeated twice to obtain two parents), the genetic operators are applied to the selected pair of individuals and the offspring is submitted for evaluation, if it is new. If it is not new, it is directly added to the population and the selection and application of genetic operators is repeated again. Once the population size exceeds 100 individuals, the environmental selection starts to operate and always removes the worst individual from the population, when a new individual is added.

We report mainly the CPU utilization achieved by the algorithm in a series of simulated parallel runs. The CPU utilization is computed as $u = \frac{t_e}{N t_w}$, where N is the number of CPUs used, t_w is the wall-clock time of the experiment, and t_e is the sum of all fitness evaluation times. As the CPU utilization depends both on the number of CPUs (it is generally better with lower number of CPUs as we show later), and the distribution of evaluation times, we provide the results of the experiments for varying number of CPUs (1, 10, 20, ..., 100) and with five different distributions of the

evaluation time – constant evaluation time, uniformly random evaluation time between 1 and 100 seconds, exponentially distributed evaluation time with 1 second mean evaluation time, evaluation time positively correlated to the fitness function, and evaluation time negatively correlated with the fitness function. The largest number of CPUs used in the experiments (90 and 100) is actually higher than the number of individuals generated by the algorithms in each generation, therefore 100 percent utilization in this case is not possible, however, we include these experiments to investigate whether the IGEA is able to use such a high number of CPUs more effectively.

We do not show the convergence graphs of the algorithms (apart from the comparison with the asynchronous EA), as there would be to many of them for the different settings. Moreover, the algorithms are functionally equivalent, thus any increase in the CPU utilization directly translates to faster convergence.

3.1 Constant Fitness and Uncorrelated Evaluation Time

As the CPU utilization does not depend on the fitness function in the three cases, where the evaluation time is not correlated with the fitness function, we use a constant fitness in this case. This means that the tournament selection is basically random and its outcome depends on the order of individuals in the aspirant pair.

Before we start discussing the CPU utilization of the algorithm in all the cases, we show how the interleaving of generations looks in a more graphical way. Figure 1 shows how 30 CPUs are used by the $(100, 100)$-EA (top) and $(100, 100)$-IGEA (bottom). They show, how many CPUs are used for evaluating individuals from different generations. The graphs are stacked, so the overall height shows the utilization of the CPUs by the algorithm. As is expected, the $(100, 100)$-EA wastes quite a lot of computation resources as it waits for the slowest evaluations to finish in each generation. On the other hand, the $(100, 100)$-IGEA is able to utilize the CPUs better as it starts evaluating next generations before the evaluation of the previous one is finished. In some cases (e.g. slightly before the 15 second mark) individuals from four different generations are evaluated at the same time. It is also interesting, how the number of CPUs used drops around the 18 second mark. Here, the algorithm has to wait for the evaluation of the last individual from the fourth (teal) generation. However, in the meantime, between the 10 and 17 second marks, the algorithm almost finished the evaluation of the next (violet) generation and evaluated a significant number of individuals from the next two generations (yellow and black). Once the individual is evaluated, the last five individuals from the fifth generation are generated and evaluated and the evolution is able to continue to use more CPUs again. The $(100, 100)$-EA would wait for the evaluation of the last individual from the fourth generation and the computational resources would be wasted.

The simplest case we investigate here is the one, where all the evaluation times are the same, namely 1 second. This should be the simplest case for the traditional version of the EA as there is no need to wait for slow evaluations and the utilization depends only on the number of CPUs and number of individuals generated in each generation. However, as the number of individuals that need evaluation is slightly different in each generation (depending on the

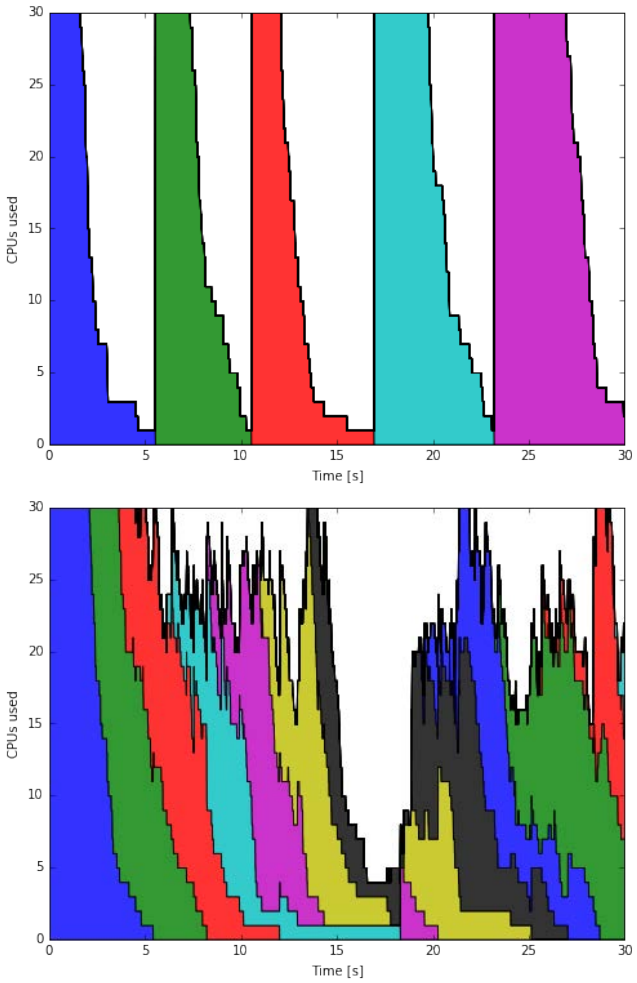

Figure 1: The number of CPUs used for evaluation of individuals from different generations for the classical generational $(100, 100)$-EA (top) and the $(100, 100)$-IGEA (bottom) while optimizing the constant fitness function with random exponentially distributed evaluation times and running on 30 CPUs. Different colors show different generations.

genetic operators and whether they change and individual or simply copy it), the traditional EA cannot reach 100 percent utilization even in this case. Figure 2 (top) shows the results of all four versions of the algorithm. The CPU utilization drops to 80 percent with 40 CPUs for both versions of EA, while it stays at 100 percent for the IGEA. This shows that IGEA actually improves the performance even in cases where all the evaluations take the same time. The EA has some peaks in the utilization, these peaks are in areas where the number of generated individuals that require evaluation (around 70 for the comma version and around 80 for the plus version) is divisible by the number of CPUs. Once the number of CPUs is higher than the number of individuals for evaluation, the interleaving stops to improve the performance as all the individuals can be evaluated in parallel, each on a single CPU. We would recommend using

around 60 CPUs in this case if one wants to have a reasonable CPU utilization. More CPUs should always improve the speed of the algorithm, however their utilization starts to drop and thus the speed improvement is also smaller.

In a more complex case (see Figure 2 (middle)), the evaluation time is a uniform random variable between 1 and 100. In this case the interleaving generations provide even more advantage over the standard EA. We can see that the CPU utilization drops to less than 70 percent for the EA with 40 CPUs, while it stays at almost 100 percent for the IGEA. The interleaving of generations is able to fill the CPUs while the algorithms waits for the slower evaluations. In this case, using around 50 CPUs seems a reasonable number with good overall CPU utilization (more than 90 percent).

The most complex case with the random evaluation time uses an exponentially distributed evaluation time. In this case the probability of an individual which is much slower than the rest of the generation is quite large, and it implies that it is much harder to provide reasonable CPU utilization. We can see (cf. Figure 2 (bottom)) that the utilization drops quite quickly for the standard EA (it is around 50 percent for 20 CPUs and around 40 percent for 40 CPUs). Although the utilization for IGEA is much lower than in the previous cases, it is still much better than the one for EA. The utilization stays at 100 percent for 20 CPUs, and drops to around 70 percent for 40 CPUs. A reasonable number of CPUs in this case seems to be around 30 as it should ensure approximately 90 percent utilization.

3.2 Rastrigin Fitness and Correlated Evaluation Time

In a more realistic case, the evaluation time can be correlated with the fitness. For minimization problems, we assume the correlation should be more often negative, i.e. better individuals evaluate longer, as they are more complex. In this part of the paper, we investigate the behavior of IGEA in both cases – with the evaluation time positively and negatively correlated with the fitness function. We use the Rastrigin function

$$f_R(x) = 10n + \sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i)),$$

where n is the length of the individual set to $n = 5$ in the experiments, as a simple example of a fitness function. The positively correlated evaluation time is defined as

$$T_p(x) = 1 + f_R(x),$$

while negatively correlated evaluation time is defined as

$$T_n(x) = 1 + \max(100 - f(x), 0).$$

In most cases, the negatively correlated time is $101 - f(x)$, the maximum only ensures that both of the time functions are always greater than 1 second. This is mostly to ensure reasonable evaluation times. Without this limitation we often got evaluation time very close to 0 which complicates the visualization of some of the results, also, we believe that in the intended applications, the evaluation time would only rarely be close to 0.

Figure 3 (top) shows the result with the negatively correlated fitness function. These are very similar to those with constant fitness function as the algorithm is able to converge close to the optima

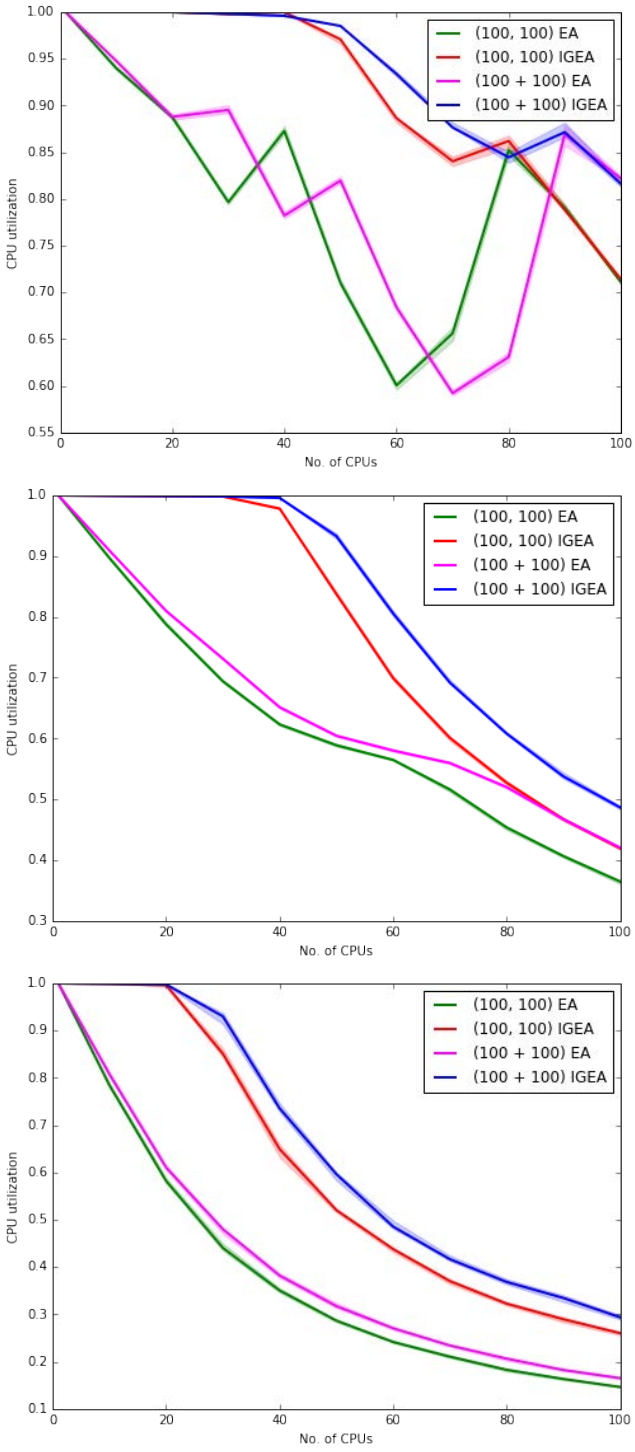

Figure 2: The CPU utilization for the four versions of the EA in the case of a constant fitness function and (top) constant evaluation times, (middle) uniformly random evaluation time, and (bottom) exponentially distributed evaluation time. The line is the median of 20 runs, the shaded areas represent the first and third quartile.

and thus all the evaluations take almost the same time and are long, there is of course still a possibility that a poor individual is generated, however, in this case such an individual is evaluated quickly and does not affect the utilization much. Therefore, we also present (Figure 3 (middle)) the results after only 1,000 evaluations, before the algorithm converges (this is roughly after 2,000 seconds with 40 CPUs). The effect similar to the case with the constant evaluation time is also slightly visible here, however it is also apparent that the CPU utilization is lower than in the previous case. Still, the IGEA is able to reach over 95 percent utilization for 40 CPUs, while the standard EA has only around 80 percent in this case.

The negatively correlated case is also the one where the asynchronous evolution suffers from the evaluation-time bias. To show the effect of the bias, we compared the asynchronous evolution to the (100 + 100)-EA and (100 + 100)-IGEA in the case of 40 CPUs. In this case, the IGEA still has almost 100 percent CPU utilization in the first thousand evaluations (cf. Figure 3 (middle)), the EA has around 77 percent and the asynchronous evolution has (as always) 100 percent. Without the evaluation- time bias, the IGEA should converge approximately as fast as the asynchronous EA, while the standard EA should be the slowest. The results of first 5,000 seconds of the experiment are presented in Figure 4. Indeed, the standard EA is the slowest of the algorithms, however the IGEA is significantly faster than the asynchronous EA, the asynchronous EA suffers from the evaluation time bias. After the algorithms converge to a reasonable value of the fitness function (around the 2,000 second mark), both the asynchronous EA and the IGEA equalize and their convergence continues to be the same. The standard EA is still a bit slower. This experiments shows the advantage of the IGEA over the asynchronous EA which comes from the fact that IGEA does not suffer from the evaluation-time bias. Of course, in the case of positively correlated evaluation time and fitness the evaluation-time bias would actually help the asynchronous EA. The asynchronous EA would be faster if more CPUs were used, as it would be able to still utilize all the CPUs, while the utilization for IGEA would be lower. However, given that IGEA utilizes all CPUs even in cases with tens of them, it may be preferable in many practical applications.

Finally, Figure 3 (bottom) shows the CPU utilization in case the evaluation time is positively correlated with the fitness function, i.e. better individuals (with lower fitness) evaluate evaluate faster. Again, the CPU utilization for the IGEA is much better than for the classical EA, however, compared to the previous results, the utilization drops much faster. The classical EA has less than 70 percent utilization even in if it uses only 10 CPUs. The IGEA is able to have almost 100 percent utilization in this case, but it also falls quickly and is only around 80 percent for 20 CPUs. In this case, once the algorithm converges, the evaluation times are also almost similar as in the previous case, however, the evaluations are fast. Therefore any poor individual generated by the algorithm takes much longer to evaluate and there is not much that can be done to speed the evaluation up. The interleaving of generations helps, however, the algorithm is unable to generate enough individuals to fully utilize CPUs. As this is also the case where the evaluation-time bias should help the asynchronous EA, we believe that using the asynchronous EA should be preferred in cases where better individuals evaluate faster.

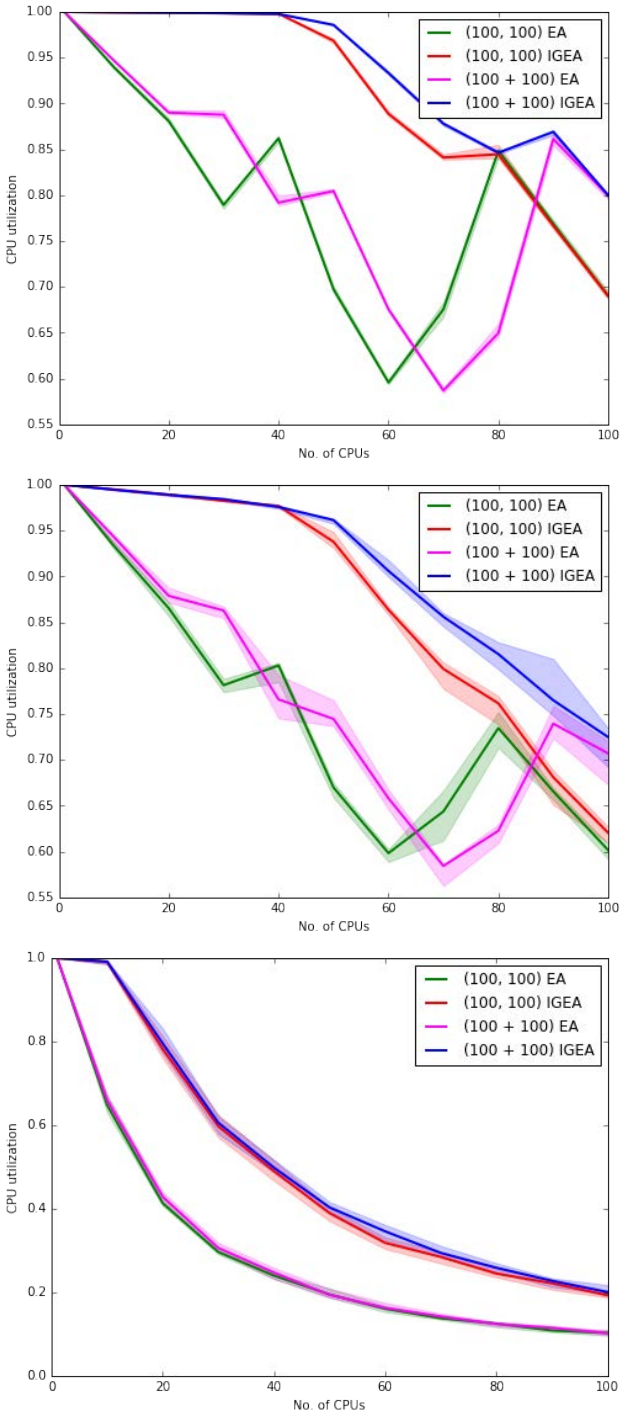

Figure 3: The CPU utilization for the four versions of the EA in the case of a the Rastrigin fitness function and evaluation times (top) negatively correlated (10,000 evaluations), (middle) negatively correlated (1,000 evaluations), and (bottom) (positively correlated) with the fitness value. The line is the median of 20 runs, the shaded areas represent the first and third quartile.

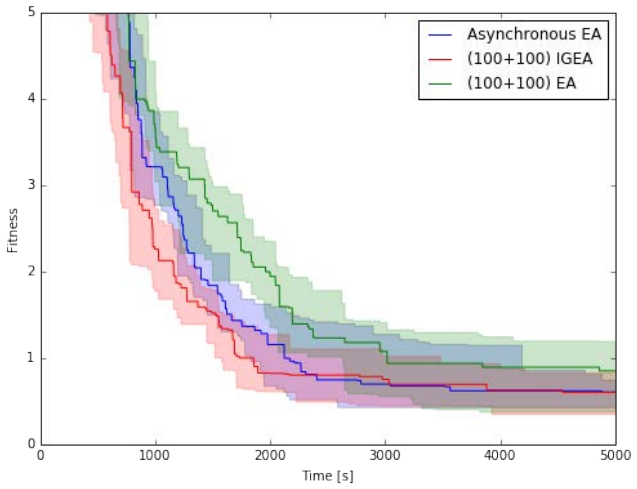

Figure 4: The comparison of $(100 + 100)$-EA, $(100 + 100)$-IGEA and asynchronous EA on the Rastrigin function with evaluation time negatively correlated to the function value, i.e. better individuals evaluate longer. The line is the median of 20 runs, the shaded areas represent the first and third quartile.

4 DISCUSSION

The presented IGEA algorithm has some significant advantages compared to both classical generational EAs and to asynchronous EA. The IGEA has better parallelization potential and, thus, CPU utilization than the generational EA, and it does not have the evaluation-time bias inherent to the asynchronous EA.

On the other hand, IGEA also has some disadvantages. The interleaving of generations can work only when the selection does not need to evaluate all the individuals before it can be executed. This means, that the popular roulette wheel selection cannot be used with the IGEA. Moreover, both the mating and environmental selection are hard-coded in the particular versions of the algorithm and their change may require more extensive changes of the algorithm. The IGEA also does not allow unlimited parallelization like the asynchronous EA, however its parallelization potential is much larger than that of standard EAs.

Despite the disadvantages, we believe IGEA can be useful to solve practical problems with long-running fitness evaluations which also have highly variable evaluation times. The tournament selection is very often used, and we do not consider the impossibility to use the roulette wheel selection too critical. We have also demonstrated that the two most common types of environmental selection can be implemented in IGEA. Other types, e.g. weaker forms of elitism (a given number of best parents is selected, while the rest of the population is selected from the offspring) can also in principle be implemented in a similar way to the plus selection.

5 CONCLUSION

We presented an evolutionary algorithm with interleaving generations – IGEA. The algorithm allows for much better utilization of computation resources by interleaving the fitness evaluations from different generations, while remaining equivalent to the standard generational evolutionary algorithm. This means that the algorithm does not suffer from the evaluation-time bias like the more common asynchronous evolutionary algorithms do. Interestingly, the algorithm improves the CPU utilization also in cases with constant evaluation time.

The experiments show that the algorithm is able to achieve almost 100 percent CPU utilization as long as the number of CPU is less than half the number of individuals evaluated in each generation. This is much better than the utilization achieved by more traditional evolutionary algorithms. We also provided an example, where IGEA can beat the asynchronous evolution, which is slowed down by the evaluation time bias.

We presented two versions of the algorithm: (λ, λ) and $(\lambda + \lambda)$ that differ in the environmental selection. The generalization to (μ, λ) and $(\mu + \lambda)$ is mostly trivial. Generalization to situations with different version of tournament selection – such as tournaments with more individuals, is also quite simple. The algorithms are general with respect to the genetic operators, however if one wanted to use genetic operators with more than two parents, the generalization is also very simple, only in case the parents should be selected randomly from the population, like in differential evolution, one may need to make similar trick to the one presented with the tournament selection and aspirant lists.

There are also further possibilities for improvement, it could be interesting to create new individuals in a speculative way, i.e. use a surrogate model that would sort the individuals and decide which of them is most likely to be selected to the next generations. The most likely individuals could then be provisionally selected, their offspring created and evaluated during the times when the CPU utilization is not 100 percent. In case the individuals are not selected in the end, their offspring can be discarded, but if they are selected, the offspring would actually be evaluated earlier and therefore the CPU utilization would be better. The generalizations from the previous paragraph and the ideas from this one are left as a future work.

ACKNOWLEDGMENT

This work has been supported by Czech Science Foundation project no. P103-15-19877S.

REFERENCES

[1] Erick Cantu-Paz. 2000. *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA.
[2] Nikolaus Hansen, Dirk V. Arnold, and Anne Auger. 2015. *Evolution Strategies.* Springer Berlin Heidelberg, Berlin, Heidelberg, 871–898.
[3] M. Pilát, T. Křen, and R. Neruda. 2016. Asynchronous Evolution of Data Mining Workflow Schemes by Strongly Typed Genetic Programming. In *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI).* 577–584.
[4] Riccardo Poli and William B. Langdon. 2006. Backward-chaining evolutionary algorithms. *Artificial Intelligence* 170, 11 (2006), 953 – 982.
[5] S. M. Said and M. Nakamura. 2014. Master-Slave Asynchronous Evolutionary Hybrid Algorithm and ITS Application in VANETs Routing Optimization. In *2014 IIAI 3rd International Conference on Advanced Applied Informatics.* 960–965.
[6] Eric O Scott and Kenneth A De Jong. 2015. Understanding simple asynchronous evolutionary algorithms. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII.* ACM, 85–98.
[7] Eric O. Scott and Kenneth A. De Jong. 2016. Evaluation-Time Bias in Quasi-Generational and Steady-State Asynchronous Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016 (GECCO '16).* ACM, New York, NY, USA, 845–852.