Asynchronously Evolving Solutions with Excessively Different Evaluation Time by Reference-based Evaluation

Tomohiro Harada  
The University of Electro-Communications  
W6-309, 1-5-1 Chofugaoka, Chofu, Tokyo, Japan  
Research Fellow of the Japan Society for the Promotion of Science DC1  
harada@cas.hc.uec.ac.jp

Keiki Takadama  
The University of Electro-Communications  
W6-309, 1-5-1 Chofugaoka, Chofu, Tokyo, Japan  
keiki@inf.uec.ac.jp

ABSTRACT
The asynchronous evolution has an advantage when evolving solutions with excessively different evaluation time since the asynchronous evolution evolves each solution independently without waiting for other evaluations, unlike the synchronous evolution requires evaluations of all solutions at the same time. As a novel asynchronous evolution approach, this paper proposes Asynchronous Reference-based Evaluation (ARE) that asynchronously selects good parents by the tournament selection using reference solution in order to evolve solutions through a crossover of the good parents. To investigate the effectiveness of ARE in the case of evolving solutions with excessively different evaluation time, this paper applies ARE to Genetic Programming (GP), and compares GP using ARE (ARE-GP) with GP using \((\mu + \lambda)\)-GP as the synchronous approach in particular situation where the evaluation time of individuals differs from each other. The intensive experiments have revealed the following implications: (1) ARE-GP greatly outperforms \((\mu + \lambda)\)-GP from the viewpoint of the elapsed unit time in the parallel computation environment, (2) ARE-GP can evolve individuals without decreasing the searching ability in the situation where the computing speed of each individual differs from each other and some individuals fail in their execution.

Categories and Subject Descriptors
I.2.8 [Problem Solving, Control Methods, and Search]: Heuristic methods

General Terms
Algorithms

Keywords
Asynchronous evolution, Genetic Programming

1. INTRODUCTION
Recently, a parallel computation technique to quickly compute using many distributed computer nodes has attracted attention. In the evolutionary computation community, many researches have studied to parallelize an evolution process for efficiently searching solutions [12, 3]. In general, Evolutionary Algorithms (EAs) typified as Genetic Algorithm (GA) [5] and Genetic Programming (GP) [7] evolve individuals (solutions) by repeating a generation step. This approach waits for evaluations of all individuals in a population and generates a new population through the parent selection and the individual deletion depending on the evaluations of all individuals. This requires that all individuals be evaluated at the same time, i.e., requires waiting for an individual with the slowest evaluation, where the evaluation time of individuals differs from each other. This causes to consume a computational time. Such situation easily occurs in the parallel computation, for example, because of (1) the difference of property of individuals and (2) the difference of computing speed of distributed computer nodes. Additionally, in some case, evaluations of some individuals could never complete, for example, because of the communication error between computer notes or an infinite loop in the case of evolving computer programs in GP. When some individual do not complete their evaluation, in general, EAs cut off the evaluation of an individual that exceeds a certain limitation and evaluate its fitness as the minimum value. However, since it is difficult to determine an ideal limitation, it is hard for conventional EAs to efficiently evolve individuals in such situation.

To tackle this problem, asynchronous approaches are recently proposed [8, 4], that evolves individuals independently, i.e., individuals do not have to wait for the evaluations of other individuals. The asynchronous evolution can continue to evolve individuals without waiting for other individuals such as slower one, which enables to efficiently evolve individuals without wasting the computational time. However, when some individual do not complete their evaluation, these methods [8, 4] also require the execution time limitation as same as the synchronous EAs. In contrast to this, we have proposed GP employing the asynchronous approach, named as TAGP (Tierra-based Asynchronous Genetic Programming) [6], as one kind of machine-code GP based on the idea of a biological simulator, Tierra [11]. An advantage of TAGP is that it can continue to evolve individuals (programs) without any limitation even if some of them do not complete their evaluation. Our previous re-
search [6] reported that TAGP could asynchronously evolve the machine-code programs even if they include loop structure. However, TAGP has the following two essential problems: (1) Unlike general EAs, TAGP do not guarantee to select the good individuals as parents from a population, which prevents from improving its performance; and (2) since TAGP selects individuals as parents or deletes them depending on a threshold based on an absolute evaluation, it is difficult to properly evolve individuals in the case that settings of a fitness function and a threshold are not proper.

To overcome these problems, this paper proposes a novel asynchronous-reference-based evaluation (named as ARE) for asynchronous EA, which not only inherit the advantage of TAGP but also overcome its weak points. Concretely, in ARE, the archive mechanism is employed to preserve good individuals for improving the performance, and the parent selection and the individual deletion are asynchronously executed to evolve individuals through a comparison of two parents with the tertiary parent randomly selected from the archive as a reference individual. This is called as a reference-based evaluation as a relative evaluation, which does not require any threshold like TAGP.

To investigate the effectiveness of ARE, this paper applies it to the Linear GP (LGP) [1, 2] problems and compares GP using ARE (ARE-GP) with GP using \((\mu + \lambda)\)-select as the synchronous GP. Experiments simulate the following situations: (1) The computing speed of each individual differs from each other; and (2) some individuals do not complete (fail in) their evaluations.

This paper organized as follows. Section 2 introduces TAGP that is the base method of ARE and explains its problems. Section 3 proposes the novel asynchronous-reference-based evaluation for asynchronous EAs. Section 4 shows the experimental settings and cases, and Section 5 shows the experimental results. The experiment is firstly conducted in situation where all individuals have the same evaluation time, and then, is conducted in situation where each individual has the different evaluation time and some individuals do not complete their evaluations. Section 6 discusses the experimental results, and finally, our conclusion is given in Section 7.

2. TIERRA-BASED ASYNCHRONOUS GENETIC PROGRAMMING

2.1 Overview

TAGP (Tierra-based Asynchronous Genetic Programming) [6] is one kind of machine-code GP employing the asynchronous approach. TAGP is based on the idea of a biological simulator, Tierra [11]. To apply Tierra to evolving programs that solves given tasks, TAGP introduces fitness into Tierra and executes the parent selection and the individual deletion mechanisms depending on fitness.

2.2 Algorithm

Fig 1 illustrates the image of TAGP, which simple flow is shown in Algorithm 1. In Algorithm 1, \(ind\) indicates an individual just after evaluation, \(ind.f\) and \(ind.f_{acc}\) indicate the fitness and an accumulated fitness of \(ind\) respectively, \(f_{max}\) indicates the maximum fitness, and \(N_p\) indicates the maximum population size. All individuals are stored in a queue named as reaper queue. Firstly, each individual is evaluated in (pseudo-)parallel (step 1), and it accumulates its fitness to \(ind.f_{acc}\), when its evaluation completes (step 2). Because of the use of the accumulated fitness, the fitness function is designed as the maximizing function. If its accumulated fitness exceeds a certain threshold (e.g., \(f_{max}\) in Algorithm 1), the threshold is subtracted from the accumulated fitness and it generates offspring (step 3-6). For example, if the accumulated fitness of \(ind_1\) and \(ind_3\) exceed the threshold \((f_{max})\), they generate offspring in the genetic operators as shown in Fig. 1. Depending on whether an individual can generate offspring or not, its position in the queue changes. If an individual generate offspring, its position in the queue shifts toward lower (step 5), otherwise its position shifts toward upper (step 11). For example, in Fig. 1, positions in the reaper queue of \(ind_1\) and \(ind_3\) that generate their offspring shift toward lower, while one of \(ind_2\) that cannot generate offspring shifts toward upper. If the population size exceeds \(N_p\), the individuals located at the top in the reaper queue are removed (step 7-9). For example, when three offspring are added as shown in Fig. 1, three individuals at the top of the reaper queue are removed.

The main feature of TAGP is summarized as follows: (1) Each individual in TAGP can be asynchronously evolved without waiting for other evaluation because the parents are selected to evolve their offspring only depending on their accumulated fitness, i.e., such parent selection is executed when the accumulated fitness of an individual exceeds a certain threshold; and (2) the reaper queue mechanism in TAGP can remove an individual that requires huge evaluation time or does not complete its evaluation (e.g., because of an infinite loop) before completing its evaluation.
3. ASYNCHRONOUS REFERENCE-BASED EVALUATION

This paper proposes a novel asynchronous reference-based evaluation (named as ARE) for an asynchronous EA, which not only inherits the advantages of TAGP but also overcomes the problems of TAGP. In this section, we firstly explain the main concept of ARE, and then we describe its algorithm.

3.1 Concept

The main concept of ARE is based on (1) the good individuals preservation (i.e., archive) and (2) the deletion of bad individuals with quick evaluation with the relative evaluation which does not require a threshold as an absolute evaluation like in TAGP. Regarding (1) the good individuals preservation, it is generally difficult to guarantee to preserve the good individual in a population due to an asynchronous manner, which means that good individuals are not always to be selected as parents for an evolution. In ARE, the archive mechanism is employed to preserve good individuals.

Regarding (2) the deletion of bad individuals with quick evaluation with the relative evaluation, what should be noted here is that, in the asynchronous evaluation, the individuals that quickly complete their evaluation preferentially get selection chance as parents regardless of its fitness. This causes to fill the population with offspring generated by such individuals, and it is easy to fall into local optimal through this kind of an evolution. To avoid such a situation, in ARE, all individuals are compared with the tertiary one that has already completed its evaluation with high fitness, which is named as reference individual. If they are worse than the reference individual, they are not selected as parents. This contributes to selecting the good individuals by excepting for bad individuals with quick evaluation. In ARE, an evaluation based on a comparison with a reference individual is called as relative evaluation, which does not require a threshold as an absolute evaluation. Concretely, the parent selection and the individual deletion are asynchronously executed to evolve individuals through a comparison of the offspring with the reference individual.

Finally, the main difference between ARE and TAGP is summarized as follows: (1) TAGP requires the accumulated fitness and the selection threshold, while ARE does not require these conditions, i.e., any fitness function in ARE can be employed like general EAs. This means that ARE can be applied to the problems where an optimal solution is unknown; and (2) ARE has the archive mechanism to guarantee to maintain the good individuals in the population, unlike TAGP does not have such mechanism. This mechanism also contributes to selecting better individuals for an evolution by excepting for bad individuals.

3.2 Algorithm

Fig. 2 illustrates the image of ARE, which flow is shown in Algorithm 2. In Algorithm 2, ind indicates an individual just after evaluation. All individuals are stored in either the reaper queue or the archive. As same as TAGP, in ARE, the individuals are evaluated in (pseudo-)parallel (step 1). The parent individuals are selected when two individuals complete their evaluations (here we call them temporarily selected individuals). As a unique point of ARE, the parent selection is done by the tournament selection from the temporally selected individuals (T in Algorithm 2) and the reference individual (ind_{ref} in Algorithm 2) that randomly selected from the archive (step 4-5). This mechanism guarantees to mate with the individuals that have a high fitness in the genetic operators. After the parent selection, two offspring are generated from two selected parents (step 6), and they are added into the bottom of the reaper queue (step 7).

As the other unique point of ARE, the archive mechanism preserves good individuals, while the deletion mechanism keeps the number of the archive size. In order to determine the individuals that should be archived or should be deleted, the temporally selected individuals are compared with the reference individual. If one of the temporarily selected individuals is better than the reference individual, it is archived and the reference individual change its position to the bottom of the reaper queue alternatively (step 8). To maintain the diversity of the individuals in the archive, if the better temporally selected individual already exists in the archive and the reference individual is unique in the archive, they are not replaced each other. For example, if ind_1 is better than ind_{ref}, ind_1 is archived and ind_{ref} is added to the bottom of the reaper queue as shown in Fig. 2. This mechanism guarantees to preserve the good individuals in the archive. On the other hand, if the temporally selected individuals are worse than the reference individual, they are removed from the reaper queue with a certain probability P_d (step 9). Afterward this deletion is called fitness deletion, and the probability P_d is called fitness deletion probability.
sample, if ind$_2$ is worse than ind$_{ref}$, ind$_2$ is removed from the reaper queue with the probability $P_d$ as shown in Fig. 2.

If the temporally selected individuals are not removed and the population size exceeds the maximum population size, the reaper queue mechanism removes the individual located at the top of the reaper queue (step 10-11). Afterward this deletion is called reaper deletion. For example, when two offspring are added but only one individual (ind$_2$) is removed as shown in Fig. 2, one individual at the top of the reaper queue is additionally removed.

What should be noted here is that the fitness deletion probability $P_d$ determines the ratio between the fitness deletion and the reaper one. Concretely, when $P_d$ is high, the fitness deletion is increasingly executed, while the reaper deletion is decreasingly executed. On the other hand, when $P_d$ is low, the fitness deletion is decreasingly executed, while the reaper deletion is increasingly executed. Since many individuals are removed before their evaluations complete if the reaper deletion increases, $P_d$ can control how long the reaper deletion waits for the individuals that require long evaluation time.

4. EXPERIMENTS

4.1 Overview

To investigate the effectiveness of ARE, this paper applies ARE to Linear GP (LGP) [1, 2]. This experiment employs symbolic regression problem that is a general benchmark program in LGP, and employs four functions shown in Table 1 [10]. Eight instructions {+, −, ×, ÷, sin, cos, exp, ln} and constant value {1, · · · , 9} can be used. Each program has eight registers (variables). The number of training data is 100 in all problems. The mean square error represented in Eq. (1) is used as the fitness function, where $\bar{y}$ and $y^*$ respectively indicate the output of a program and the target value.

$$fitness = \frac{1}{n} \sum_{i=1}^{n} (\bar{y} - y^*)^2 \tag{1}$$

In this experiment, GP using (μ+λ) selection ((μ+λ)-GP) is employed as the synchronous GP because its evolution process can be easily parallelized.

4.2 Settings

The common parameter is shown in Table 2. In ARE-GP, the settings of the archive size as = 5 and the fitness deletion probability $P_d = 0.5$ are used. Each case executes 20 independent runs and the average of the minimum fitness in the population is evaluated.

4.3 Cases

This experiment simulates the situation where (Case1) all computing nodes have same speed, (Case2) the computing speed of each individual differs from each other, (Case3) some individuals do not complete (fail in) their evaluations in the parallel computation environment, and (Case4) both of Cases 2 and 3 occurs. These details are described below.

Case1: Same computing speed

In Case1, all individuals execute the same number of instructions (100 instructions) per unit time and can certainly complete their evaluation. Note that the number of executable instructions per unit time means how many instructions can be executed in one unit time. For example, when 100 instructions are executable per unit time and an individual (program) has 100 instructions, it can execute all 100 instructions in one unit time. Since the number of the training data is set as 100 in this experiment, this individual can complete its evaluation with 100 unit time. Since this experiment simulates the parallel evaluation of 100 individuals, (μ+λ)-GP requires the maximum number of required unit time in a population in one generation (shown in Fig. 3(a)). While ARE-GP spends the parallel evaluation time including a termination of an evaluation due to the reaper deletion (shown in Fig. 3(b)).

Case2: Different computing speed

Unlike Case1 where all individual execute 100 instructions per unit time, in Case2, the number of executable instructions of each individual differs from each other as shown in Table 3. Concretely, 3, 3, 20, 43, and 31 individuals in 100 individuals respectively execute 20, 40, 60, 80, and 100 instructions per unit time. This setting is used in [8]. In this setting, the slowest individual requires five times longer evaluation time than the fastest one. Since the synchronous evolution needs to wait the slowest individual, five times longer evaluation time elapse in one generation cycle. This simulates the situation where each computer node has different computing speed.
Table 3: Difference of the number of executable instructions per unit time in Case2

<table>
<thead>
<tr>
<th>#executable instruction</th>
<th>#individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>60</td>
<td>20</td>
</tr>
<tr>
<td>80</td>
<td>43</td>
</tr>
<tr>
<td>100</td>
<td>31</td>
</tr>
</tbody>
</table>

**Case3: Evaluation failure**

Unlike all individuals in Case1 complete their evaluations, some individuals in Case3 do not complete their evaluation with the certain probability of the evaluation failure $P_f$. This experiment set $P_f = 0.05$, in other words, 5% of the evaluations do not complete. Since the synchronous evolution requires all evaluation of individuals in a population at the same time, it needs to determine the execution time limitation to terminate the evaluation. This environment simulates the situation where evaluations of some individuals do not complete due to the communication error between computer notes or an infinite loop of a computer program.

In Case3, $(\mu + \lambda)$-GP cuts off the evaluation of an individual if its execution time exceeds the limitation (described in Section 5.3) and calculates its fitness as $-\infty$.

**Case4: Both of Case2&3**

In Case4, both situations of Case2 and Case3 occur. In particular, the number of executable instructions is set as shown in Table 3, while $P_f$ is set as 0.05.

5. RESULTS

5.1 Case1: Same computing speed

Fig. 4 shows the change of the average fitness of ARE-GP and $(\mu + \lambda)$-GP from the viewpoint of the number of evaluations until $1.0 \times 10^6$ evaluations in Case1. In Fig. 4, the horizontal axis indicates the number of the evaluations, while the vertical axis indicates the average fitness of 20 independent runs. The solid line with circles shows the result of ARE-GP, while the dash line shows one of $(\mu + \lambda)$-GP. Fig. 4 indicates the ARE-GP requires the larger number of evaluations than $(\mu + \lambda)$-GP in the early term, while ARE-GP and $(\mu + \lambda)$-GP have the equivalent fitness value after the maximum number of evaluations.

On the other hand, Fig. 5 shows the change of the average fitness of ARE-GP and $(\mu + \lambda)$-GP from the viewpoint of the elapsed unit time until $1.0 \times 10^7$ unit time in Case1. In Fig. 5, the horizontal axis indicates the elapsed unit time, while the vertical axis indicates the average fitness of 20 independent runs. The solid line with circles shows the result of ARE-GP, while the dash line shows one of $(\mu + \lambda)$-GP. From the viewpoint of the elapsed unit time, ARE-GP outperforms $(\mu + \lambda)$-GP in all test problems regarding both of the convergence speed and the final fitness value. This is because $(\mu + \lambda)$-GP needs to wait the latest evaluation (i.e., the largest program in the population) to generate the next population, it causes to waste a lot of evaluation time, while ARE-GP can asynchronously generate a lot of offspring by efficient use of the evaluation time since ARE-GP needs not to wait for other evaluation of individuals.

Fig. 6 shows the accumulated number of generation of offspring in Ex. 1 in Case1. In Fig. 6, the horizontal axis indicates the elapsed unit time, while the vertical axis indicates the accumulated number of generation of offspring. The solid line with circles shows the result of ARE-GP, while the dash line shows one of $(\mu + \lambda)$-GP. Note that although Fig. 6 only shows the result of Ex. 1, we confirm same trend in other test problems. As shown in Fig. 6, ARE-GP can execute ten times generation of offspring than $(\mu + \lambda)$-GP. This indicates that ARE-GP can generate many offspring in one unit time, which contributes to improve its searching ability.

From these results, it is revealed that although ARE-GP is not better than $(\mu + \lambda)$-GP from the viewpoint of the number of evaluations, ARE-GP greatly outperforms $(\mu + \lambda)$-GP from the viewpoint of the elapsed unit time in the parallel computation environment.

5.2 Case2: Different computing speed

Fig. 7 shows the change of the average fitness of ARE-GP and $(\mu + \lambda)$-GP from the viewpoint of the elapsed unit time until $1.0 \times 10^7$ unit time in Case2. Axes and lines in Fig. 7 have the same meaning as Fig. 5. From the result of $(\mu + \lambda)$-GP, the convergence speed decreases from Case1 where all individuals have the same computing speed in all test problems. This is because $(\mu + \lambda)$-GP elapses five times longer unit time in one generation than Case1 since the synchronous evolution needs to wait the slowest evaluation that can execute only 20 instructions per unit time. In particular, in this experiment, a few slow computer nodes causes to waste many computer resources despite the number of the slowest computer notes is very few (only three nodes). On the other hand, ARE-GP achieves the equivalent convergence speed and the final fitness value as Case1 and outperforms $(\mu + \lambda)$-GP in all test problems.

These results reveal that ARE-GP can evolve individuals in the situation where the computing speed of each individual differs from each other without decreasing the searching ability.

5.3 Case3: Evaluation failure

Fig. 8 shows the change of the average fitness of ARE-GP and $(\mu + \lambda)$-GP from the viewpoint of the elapsed unit time until $1.0 \times 10^7$ unit time in Case3. Axes and lines in Fig. 8 have the same meaning as Fig. 5. In this case, since the maximum number of instructions, the number of executable instructions per unit time, and the number of training data is respectively set as 256 instructions, 100 instructions, 100, the ideal execution time limitation is 256 unit time. In all test problems, the execution time limitation is set as 256.
and 2560 unit time, the former value is ideal while the latter one is 10 times larger than the ideal value, which result is additionally shown as the dotted line in Fig. 8.

The comparison of the different time limitation indicates that the searching ability of \((\mu + \lambda)\)-GP decreases depending on the setting of the execution time limitation. From this result, it is verified that the setting of the execution time limitation affect to the searching ability of \((\mu + \lambda)\)-GP. It is, however, difficult to determine the ideal time limitation because the properties of the solved problem are generally unknown, so that it is difficult to keep the searching ability of \((\mu + \lambda)\)-GP in such situation. On the other hand, the results of ARE-GP indicate that it outperforms \((\mu + \lambda)\)-GP with the ideal time limitation, and achieves the equivalent convergence speed and the final fitness value as Case1. What should be noted here is that ARE-GP can achieve this performance without the time limitation unlike \((\mu + \lambda)\)-GP.

These results reveal that ARE-GP can evolve individuals in the situation where some individuals do not complete their execution and do not require the execution time limitation, which is required in the synchronous evolution.

**5.4 Case4: Both of Case2&3**

Fig. 9 shows the change of the average fitness of ARE-GP and \((\mu + \lambda)\)-GP from the viewpoint of the elapsed unit time until \(1.0 \times 10^7\) unit time in Case4 that includes the both situations of Case2 and Case3. Axes and lines in Fig. 9 have the same meaning as Fig. 5. In this case, since the slowest evaluation requires five times longer than the fastest one, the ideal value of the execution time limitation is 1280 \((= 256 \times 5)\) unit time, and \((\mu + \lambda)\)-GP employs this limitation in all test problem.

In contrast to the convergence speed of \((\mu + \lambda)\)-GP greatly decreases as same as the result of Case2 and Case3 in all test problems, ARE-GP achieves the equivalent convergence speed and the equivalent final fitness value as Case1.

These results reveal that ARE-GP can also evolve individuals in the situation where each individual has the different computing speed and some individuals do not complete their evaluation without decreasing the searching ability, unlike the searching ability of the synchronous evolution greatly decreases in such situation.

### 6. DISCUSSION

ARE-GP can evolve individuals without any setting in the situation of the different evaluation time, and achieves efficient search and better performance than \((\mu + \lambda)\)-GP. Table 4 shows the ratio of the elapsed unit time when ARE-GP achieves the average fitness that \((\mu + \lambda)\)-GP finally achieves. Table 4 indicates that ARE-GP needs less than 10% elapsed time to achieve the equivalent performance as \((\mu + \lambda)\)-GP.

To check the statistically difference between ARE-GP and \((\mu + \lambda)\)-GP, the non-parametric Mann-Whitney U test [9] is conducted with the level of significance \(\alpha = 0.05\) regarding the fitness achieved after \(10^7\) unit time. Table 5 shows the \(p\) value in all cases and all test problems. This result reveals that the significant difference between ARE-GP and \((\mu + \lambda)\)-GP is found in all cases and all test problems, which means ARE-GP significantly outperforms \((\mu + \lambda)\)-GP.

As the interesting feature of ARE-GP, in some cases of the different evaluation time, the convergence speed and the final average fitness improve in the comparison with the situation of the same evaluation time. To confirm this insight, Figure 10 shows the change of the ratio that is calculated as \(r(t) = \text{fCase4}(t)/\text{fCase1}(t)\), where \(\text{fCase1}(t)\) and \(\text{fCase4}(t)\) indicate the average fitness in Case1 and Case4 respectively when \(t\) unit time elapsed. This indicator explains whether
the fitness of Case1 is better (smaller) than Case4 at the evaluation time ofCase4 at t unit time, in other words, Case1 outperforms Case4 at t unit time. While if r(t) is less than 1, i.e., the results are plotted under the dotted line in Figure 10, the fitness of Case1 is smaller than one of Case4 at t unit time, in other words, Case4 outperforms Case1 at t unit time. In Figure 10, the horizontal axis indicates the elapsed unit time with the logarithmic scale, while the vertical axis indicates r(t). The solid line shows with circles the result of ARE-GP, while the dash line shows one of (μ + λ)-GP. As shown in Figure 10, the results of ARE-GP with the different evaluation time in Case1 outperforms one with the same evaluation time in Case1 in all problems but Ex. 3, while ARE-GP shows the same performance in these cases in Ex. 3. Unlike these results, (μ + λ)-GP becomes worse with the different evaluation time in all problems. Since the required evaluation time in GP is determined depending on the size of a program and ARE-GP selects parents when an individual completes its evaluation, in the situation of the same evaluation time, the small size program can get a lot of chance of a parent selection. In contrast to this, due to the difference of the evaluation time, the selection turn in ARE-GP is not determined depending on the program size because the large program can get the selection chance if its evaluation time is shorter than others. It is considered that this contributes to increase the diversity of selected parents and to improve the searching ability.

7. CONCLUSIONS

This paper proposed Asynchronous Reference-based Evaluation (ARE) that is a novel asynchronous EA to evolve each individual independently, unlike the synchronous EAs requires the evaluations of all individuals at the same time. Concretely, ARE asynchronously evolve individuals through a comparison with only three of individuals (i.e., two parents and one reference individual as the tertiary parent) unlike a synchronous evolution through a comparison with all of them in general EAs, and ARE improves its performance by archiving good individuals as the reference individual. To investigate the effectiveness of ARE in the situation where the evaluation time of each individual differs from each other, ARE is applied to Linear GP (LGP) and is compared with GP using (μ + λ) selection ((μ + λ)-GP). The experiments employ the symbolic regression problems which is well known benchmark problem in GP, and simulates the situation where (1) the computing speed of each individual differs from each other, and (2) some individuals do not complete their evaluations. The intensive experiments have revealed the following implications: (1) In the situation of the same evaluation time, although ARE-GP has slower convergence speed than (μ + λ)-GP from the viewpoint of the number of evaluations, ARE-GP greatly outperforms (μ + λ)-GP from the viewpoint of the elapsed unit time in the parallel computation environment; (2) ARE-GP can evolve individuals without decreasing the searching ability in the situation where the computing speed of each individual differs from each other and some individuals do not complete their execution without the execution time limitation, even in the situation including both of them; and (3) ARE-GP has probability to improve the search performance in the situation of the different evaluation time in comparison with the same evaluation time environment.
What should be noticed here is that these implications have only been obtained from one type of problem, i.e., Linear GP. Therefore, further careful qualifications and justification, such as an analysis of results using other general LGP problems such as symbolic regression or classification problem, are needed to extend the range of application of ARE to other EA domain. Such important directions must be pursued in the near future in addition to the following future research: (1) a verification of the validity of ARE on the actual parallel computing environment because some overheads are considered in the actual environment; and (2) an adaptation of the fitness deletion rate \( P_d \) and the archive size depending on the evolution degree or the diversity of the population because these parameters gives a big influence to the performance of ARE.

8. ACKNOWLEDGMENTS

This work was supported by Grant-in-Aid for JSPS Fellows Grant Number 249376.

9. REFERENCES


