Genet Program Evolvable Mach (2010) 11:205–225
DOI 10.1007/s10710-010-9105-2

CONTRIBUTED ARTICLE

Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm

Ting Hu · Simon Harding · Wolfgang Banzhaf

Received: 15 May 2009 / Revised: 10 February 2010 / Published online: 14 March 2010

Abstract With current developments of parallel and distributed computing, evolutionary algorithms have benefited considerably from parallelization techniques. Besides improved computation efficiency, parallelization may bring about innovation to many aspects of evolutionary algorithms. In this article, we focus on the effect of variable population size on accelerating evolution in the context of a parallel evolutionary algorithm. In nature it is observed that dramatic variations of population size have considerable impact on evolution. Interestingly, the property of variable population size here arises implicitly and naturally from the algorithm rather than through intentional design. To investigate the effect of variable population size in such a parallel algorithm, evolution dynamics, including fitness progression and population diversity variation, are analyzed. Further, this parallel algorithm is compared to a conventional fixed-population-size genetic algorithm. We observe that the dramatic changes in population size allow evolution to accelerate.

Keywords Variable population size · Population bottleneck · Evolution acceleration · Parallel computing · GPU

1 Introduction

It is well known that evolutionary algorithms are often computationally expensive and benefit from parallelization [1]. In other research, we have investigated the use

T. Hu (✉) · S. Harding · W. Banzhaf
Department of Computer Science, Memorial University, St. John's, NL, Canada
e-mail: tingh@mun.ca

S. Harding
e-mail: simonh@mun.ca

W. Banzhaf
e-mail: banzhaf@mun.ca

of Graphical Processor Units (GPUs) to accelerate the evaluation of individuals in genetic programming [5]. One of our current implementations attempts to optimize the use of multiple GPUs within a single computer. To do this a number of processes in the evolutionary algorithm are required to work asynchronously. The algorithm results in the population changing size during evolution. This variable population size is an unintentional side effect of the algorithm. However, it gave us the opportunity to investigate the behavior of variable size populations in a unique "real world" scenario. The full algorithm is described in Sect. 3.

Population size is crucial to evolution in both natural and artificial systems. From the biological point of view, population size is linked to the likelihood of preserving genetic variations through reproduction [10]. The relationship between population size and genetic variation is different under different conditions. Natural selection can either promote genetic variation (i.e., diversifying selection) or reduce genetic variation (i.e., purifying selection). Under diversifying selection, genetic variation can be fixed faster in larger populations, while under purifying selection, smaller populations show higher evolution rate [24]. Moreover, genetic variations may also be lost due to genetic drift. A positive relationship between population size and genetic variation is prevalent for neutral or nearly neutral mutations, because these are more likely to be lost during genetic drift in smaller populations [10]. In most cases of natural systems, larger populations are more stable and have slower rates of accepting new phenotypic variations. In smaller populations, however, change of the phenotype can happen relatively faster, but these populations have a higher risk of extinction due to the limited genetic diversity.

In computational evolution, population size is directly related to the search space and computation overhead [21]. A carefully designed population sizing scheme can improve the performance of an evolutionary algorithm whilst balancing search capability and evaluation overhead. In previous work in the literature, the general approach was to implement the dynamic population size adjustment during evolution according to a feedback loop that is able to reflect the requirement of population diversity over time. These approaches are reviewed in the next section.

In this article, the variable population size results from the parallelization of our algorithm rather than by intentional design. The dynamics of this variable population size and its effect on evolution is analyzed. We particularly examine the fitness progression and population diversity variation. When compared to a conventional fixed-population-size Genetic Algorithm (GA), it is shown that our new algorithm results in faster evolution.

2 Background

2.1 Population size variation in biology

In biology, population size has been shown to play an important role in evolution. Both theoretical hypotheses and empirical verifications can be found in areas such as *conservation biology* [27], *evolutionary biology* [9, 24], and *population genetics* [15, 29].

In nature, the size of a population often changes dramatically during evolution. Biologists use the term *population bottleneck* to describe an event in which the size of a population has been significantly reduced. Natural populations can be affected considerably by these bottlenecks in evolution. For instance, genetic variability is expected to decline rapidly after bottlenecks and to increase gradually during recovery periods [23]. The rate of such genetic change depends critically on the intensity of population bottlenecks and the rate of population growth in recovery.

In research on human evolution, it is reported that there existed several major population bottlenecks that contributed significantly to human evolution [2, 6, 16]. These population bottlenecks are empirically identified through investigating genetic diversity in the human genome. After each bottleneck event, the genetic diversity of human populations was found to have decreased. Recently, Gherman et al. [11] report that population bottlenecks are a potential major shaping force of human genome architecture.

The drastic growth of a population can also affect its evolution. In a study on the recent, rapid molecular evolution in human genomes, Hawks et al. [17] hypothesize that the current dramatically growing human population may be the major driving force of new adaptive evolution. They state that a growing population size can provide the potential for rapid adaptive innovations if a population is highly adaptive to the current environment. Therefore, significant fluctuations in population size can contribute a great deal to evolution.

2.2 Population size control in evolutionary algorithms

In computational evolution, population size is closely related to an algorithm's search capability. A small population size may not be able to provide the necessary search diversity. However, larger population sizes require to pay greater computational costs [21]. Many efforts have been devoted to control this critical parameter in evolutionary computation. A method of theoretically estimating optimal population size in GAs is proposed by Goldberg et al. [13], where the population size can be determined based on the complexity of the given optimization problem. This approach suggests a general principle for population size optimization, although its application still needs further work since estimating the complexity of many problems can be challenging if not impossible.

In addition to initializing a proper population size beforehand, it has been shown later on that the dynamic adjustment of population size during evolution can also improve the performance of an evolutionary algorithm. Some empirical methods for variable population size have been proposed.

Arabas et al. [3] propose the *Genetic Algorithm with Variable Population Size* (GAVaPS) where the algorithm considers the *age* and *lifetime* of each individual. Population size fluctuates as a result of removing over-aged individuals and reproducing new ones. Back et al. [4] extend this lifetime notion to steady-state GAs in their *Adaptive Population size Genetic Algorithm* (APGA). Fernandes and Rosa [8] propose the *Self-Regulated Population size Evolutionary Algorithm* (SRP-EA) to enhance APGA using a diversity-driven reproduction process.

In other work, Smith [26] adjusts the population size based on the probability of selection error. Goldberg [12] proposes the “serial and parallel GA” by re-initializing the population with preserved elite and randomly generated individuals to restart the search process when the algorithm converges. Schlierkamp-Voosen and Muhlenbein [25] demonstrate a sub-population competition scheme, where better performing sub-populations are rewarded with increasing sizes. Harik and Lobo [14] introduce *parameter-less GA*, where several populations with different sizes evolve in parallel, starting with small population sizes. By inspecting the average fitness of these populations, less fit undersized populations are replaced by larger ones. Koumousis and Katsaras [20] propose a variable population size GA that works with a periodically re-initialized population, the size of which change with a saw-tooth-like function.

Later, Eiben et al. [7] suggest to use the pace of fitness improvements as the signal to adjust population size dynamically in the *Population Resizing on Fitness Improvement GA* (PRoFIGA). Similarly in genetic programming systems, Tomassini et al. [28] implement a dynamic population size algorithm using fitness progression as the signal to delete over-sized and worse-fit individuals or to insert mutated best-fit individuals under certain conditions. Hu and Banzhaf [18] adopt a biologically motivated measurement technique for the evolution rate as an indicator to adjust population size and demonstrate that this method can accelerate evolution in genetic programming.

The existing adaptive population size schemes in evolutionary algorithms are mainly designed by intentionally adjusting the population size based on a certain feedback loop. The common finding from these approaches is that the performance of an evolutionary algorithm can be improved by varying the population size during evolution.

In the technique presented here, the variable population size scheme does not result from intentional adjustment but is a natural consequence of the asynchronous behavior of the parallel algorithm. This raises a number of questions, with the most important one being whether this variable population size property can improve the algorithm’s performance. This motivates us to take a look at how this unintentional but real scenario can affect the pace of evolution.

3 Asynchronous parallel evolutionary algorithm (APEA)

To understand the overall goal of this research, and not just the variable population size issues detailed here, it is important to understand what motivated this algorithm.

Recently there has been much interest in accelerating evolutionary algorithms using Graphical Processing Units (GPUs) [5]. GPUs may contain several hundreds simple processors, and typically run a single program on many different sections of data in parallel (the processors are usually considered as a Single Instruction Multiple Data (SIMD) processor).

Many modern GPU setups contain multiple GPUs. In evolutionary algorithms, and particularly in genetic programming, it is difficult to keep all the GPUs busy, i.e., supplied with new individuals for evaluation. With genetic programming (the primary focus of our research, but not this paper) there is a significant overhead in the compilation stage of programs which is necessary before execution.

Our approach aims at efficiently exploiting all of the GPU processing available in a multi-GPU scenario. Individuals are evaluated in a fitness case parallel fashion, with each GPU device processing a different individual.

Individuals need to be first converted from their genotype to CUDA C source code, which is the programming language used by the GPUs, and then compiled to programs that can be run on the GPUs. The CPUs are responsible for this task. Executing individuals and evaluating their fitness is the other and even more time consuming component of an evolutionary algorithm, which will be operated on GPUs. The entire system is designed to act in an asynchronous way. Compiling individuals is performed asynchronously on CPUs, with the execution of other individuals on GPUs. Further, each GPU device executes the individuals asynchronously, so that several different individuals are evaluated at the same time.

The evolutionary algorithm itself is also designed to operate in an asynchronous manner in order to ensure that there are always unevaluated individuals available for a GPU device to process. For convenience we will refer to this algorithm as APEA (Asynchronous, Parallel Evolutionary Algorithm).

The entire process in APEA can be considered divided into a number of different tasks:

- Generation of new individuals and compilation;
- Execution of individuals;
- Maintenance tasks.

Figure 1 illustrates the process in APEA. The following sections describe these steps in more details. Note that the size of the shared population varies as a result of the operations by multiple threads. The GPU thread increases the size of the shared population when it posts new compiled individuals. The housekeeping thread decreases the size of the shared population after it removes less fit individuals. The CPU thread has no effect on the size of the shared population.

3.1 New individual generation and compilation

In APEA, each of the CPU processors available is responsible for generating new individuals and compiling them from their representations to GPU executable programs.

Each processor selects—using tournament selection—a number of individuals from the population of individuals (“shared population”). From these, it generates new individuals by genetic variation, e.g., mutation and crossover. These individuals are then converted to CUDA C source code. Each compiled program also acts as a fitness function, and therefore also includes all the code required to implement a test case.

The individuals are then compiled by a CPU using the CUDA compiler. Once an individual has been compiled, it is placed into a list (“compiled individuals”). This

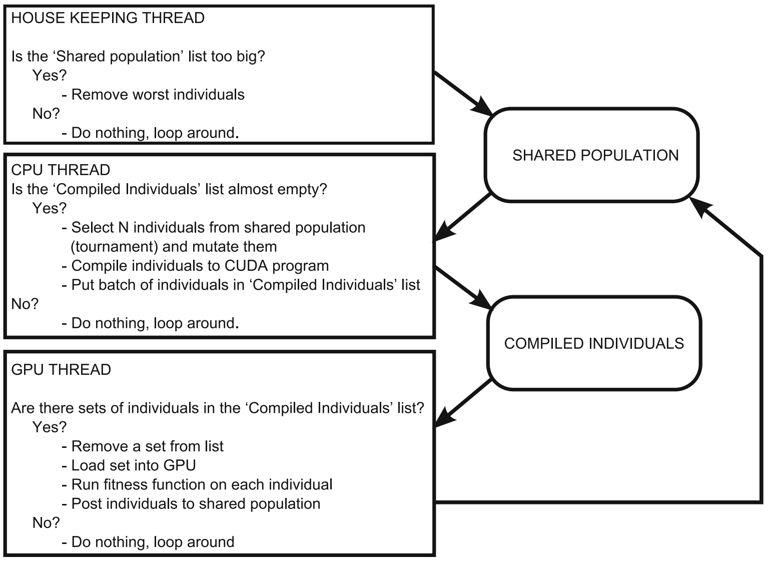
HOUSE KEEPING THREAD
Is the 'Shared population' list too big?
Yes?
- Remove worst individuals
No?
- Do nothing, loop around.
CPU THREAD
Is the 'Compiled Individuals' list almost empty?
Yes?
- Select N individuals from shared population (tournament) and mutate them
- Compile individuals to CUDA program
- Put batch of individuals in 'Compiled Individuals' list
No?
- Do nothing, loop around.
GPU THREAD
Are there sets of individuals in the 'Compiled Individuals' list?
Yes?
- Remove a set from list
- Load set into GPU
- Run fitness function on each individual
- Post individuals to shared population
No?
- Do nothing, loop around
SHARED POPULATION
COMPILED INDIVIDUALS

Fig. 1 Overview of APEA. The house keeping thread maintains the shared population. The CPU thread generates new individuals and compiles them. The compiled individuals are executed and evaluated by the GPU thread and then inserted back to the shared population

process is repeated continuously. The process also occurs asynchronously across the available CPU processors.

3.2 Individual execution

Each GPU device has a corresponding process running on the host computer. If the GPU device is idle, the process examines the list of compiled individuals and selects one to process. It removes this individual from the list to make it unavailable for other GPU devices. The process then loads this individual into its GPU to execute.

As in previous work, the function encoded by each individual is then executed on the test cases in a SIMD style, with all the test cases evaluated in parallel against a single individual's program [5]. The output from the function on each test case is an error, i.e., the deviation of actual behavior from target behavior. Errors are summed (again on the GPUs) to provide a fitness score for the associated individual.

Individuals that have been evaluated are written back to the shared population. Here, they can be used by a CPU thread to generate new individuals. Then these individuals are unloaded from the GPUs.

The process then looks to see if there are other compiled individuals to process, and the execution process repeats.

3.3 Maintenance tasks

This process is primarily responsible for maintaining the shared population. As GPU processes finish executing individuals, these individuals are put back into the shared population where they can be used to generate new ones. If not maintained, however, the population would keep growing in size. This would not only reduce the efficiency of the tournament selection but also increase the memory requirements of the algorithm.

In order to address this problem, the house keeping thread periodically removes the worst individuals from the population. This is done approximately every *D*ms (recovery duration) and is performed asynchronously with the other processes. This periodic removal causes the population size to vary over time in a saw-tooth-like fashion, and it is the effect of this action that we want to study here.

The house keeping thread is also responsible to monitor the population and produce statistical information—as well as terminating the experiment if a successful solution is found.

3.4 Algorithm simulation

For the work presented here a simulation of the APEA algorithm was implemented instead of employing real GPU devices. This primary simulation allows for an optimization of parameters for the actual parallel algorithm in addition to allowing the study of the behavior of the algorithm itself.

The actual algorithm is designed for genetic programming, but for simplicity this has been replaced here with a GA on two classical benchmark problems: the OneMax and the Spears multi-model problems [19]. Individuals are encoded as binary strings for both problems. Two point crossover is used. During mutation, the probability of altering one bit is (arbitrarily) set to 0.05. Tournament selection is used, with 5 individuals being considered.

The behavior of the algorithm, being asynchronous, is dependent on the time required for each component. The actual parallel algorithm was benchmarked to provide approximation of the timings required for the simulator. The simulator operates in fixed time steps, with events scheduled for processing at appropriate time steps. One time step in the simulation corresponds to 1 ms.

For compilation of individuals, the time taken to compile individuals (i.e., copy a selection of individuals from the shared population to the compiled individuals list) is computed based on estimates taken from benchmarking the actual implementation of APEA. There is an overhead of 100 time steps per set of individuals, and each individual takes 10 time steps to compile. To this time, we add a random time period to simulate the variability in compilation time. Noise of a maximum of 200 time steps (uniformly distributed) is added to the total time.

For execution, we simulate the run time of a program by making the processing time per individual proportional to the length of the program, plus an overhead. We use 1 time step per 100 bits of the genotype length. The overhead is fixed at 100 time steps, and again some random variation is added with a noise of 100 time steps. A set of 10 individuals are loaded into a GPU for each round of execution. Note that although a GPU executes individuals one by one asynchronously, a set of 10 individuals are loaded to a GPU at a time for the efficiency on the network.

For system monitoring, the house keeping thread generates the statistical information, e.g., the size of the shared population, the best and the average fitness of the current population, etc., periodically every 1,000 time steps.

4 Analysis of evolutionary dynamics

The original motivation for APEA was to utilize multiple processing units, however, the unintentional property of varying population size generates interesting questions about its effect on evolution. In this section, we perform an analysis of the algorithm's fitness progression and population diversity variation in order to take a closer look at the impact of population bottlenecks and size fluctuations. Furthermore, APEA is compared to a conventional fixed-population-size GA to see whether the algorithm can benefit from a variable population size scheme.

4.1 Test suites

Two classical benchmark problems are employed in the experiments, the OneMax problem and the Spears multi-model problem.

For the OneMax problem, each individual is initialized as a binary string containing random numbers of 0s and 1s. The objective of this algorithm is to maximize the number of 1s in the bit string and thus to eliminate 0s. The fitness of an individual is the number of remaining 0s, hence a score of 0 represents a binary string consisting purely of 1s.

The Spears multi-model problem [19] has been designed for systematic analysis of GA behavior. The problem generator creates a certain number of binary strings with length L, each of which represents a peak in an L dimensional space. The height of peaks can be defined by users with the highest peak of height 1. The goal is to find the highest peak. The fitness of a bit string s with length L is defined as,

$$f(s) = \frac{L - HammingDistance(s, Peak_n(s))}{L} \cdot height(Peak_n(s)), \tag{1}$$

where $Peak_n(s)$ is the nearest peak in Hamming space to the bit string s. It can be seen that the complexity of this benchmark can be manually configured by choosing the binary string length, the number of peaks, and the assignment function of peak heights. Here, we set 6 peaks with linearly increasing heights between 0.5 and 1.

For both benchmarks, we set the starting and default population size 100, since this is the common configuration in the literature. The OneMax problem is employed in Sects. 4.2 and 4.3 because a detailed investigation on evolutionary dynamics can benefit from such a linear search problem. Further in Sect. 4.4, the APEA and a conventional GA are tested both on the OneMax problem and the Spears multi-model problem for comparison purposes. The bit string length is set to 100 for OneMax problem and is set to 200 for the Spears multi-model problem. The goal is to test the algorithm performances on problems with different complexity.

4.2 Variable population size and fitness progression

As described in Sect. 3, the house keeping thread leads to a reduction in population size and therefore to a population bottleneck each time it is executed. In our simulation, parameter *D* controls how often this happens through setting four different recovery duration values, i.e., 2,000, 5,000, 10,000, and 20,000 ms. Here, we are particularly interested in the dynamics of the population size and that of the average/best fitness over a simulation run on the OneMax problem.

On the one hand, the degree of increase in population size is essentially determined by *D*. Specifically, the population grows as individuals are evaluated, and at the end of each recovery period the population is set back to 100 individuals. Figures 2, 3, 4, 5 each show the saw-tooth-like population fluctuations (left *y*-axis) in a typical run with one of the four values of *D*. Recall that our algorithm simulation collects system statistics every 1,000 ms, represented by each point in these figures. The data collection terminates when the best population fitness reaches 0, i.e., a perfect solution is found.

Note that the reset population size captured in those figures fluctuates at 100 and 110. This is due to the algorithm implementation with multiple threads. That is, occasionally 10 new individuals (one set of individuals loaded to a GPU for execution, see Sect. 3.4) are inserted into the shared population by a GPU thread just prior to the statistics being generated by the house keeping thread. The population size growth after bottlenecks can also vary during implementations because of the simulated processing time and noises (See Sect. 3.4).

On the other hand, we also plot the best and the average fitness development curves (right *y*-axis). From these plots, we observe that the greater *D* is, the longer it takes the algorithm to discover an optimal individual, while the number of

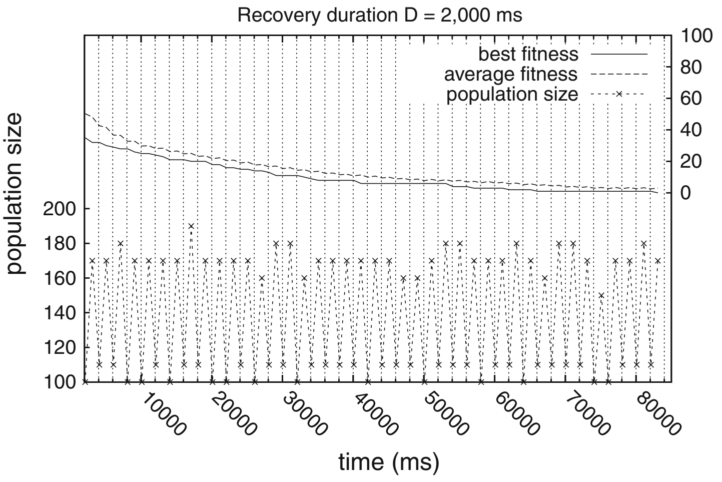

Fig. 2 Variable population size and fitness development in an example run with recovery duration $D = 2{,}000$ ms. Since the system generates status statistics every 1,000 ms, represented by a point in the figure, there is only one point between every two bottlenecks

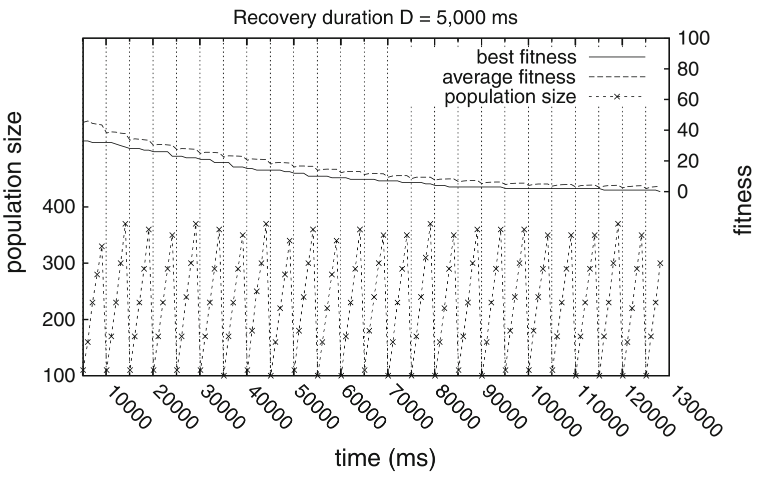

Fig. 3 Variable population size and fitness development in an example run with recovery duration $D = 5{,}000$ ms. There are four data points between every two bottlenecks. The top population size can be more than twice as that of the run with $D = 2{,}000$ ms

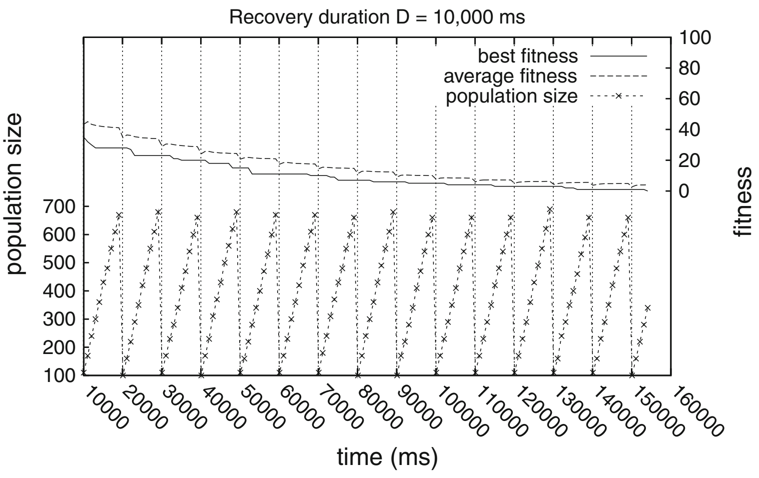

Fig. 4 Variable population size and fitness development in an example run with recovery duration $D = 10{,}000$ ms. With a longer recovery duration, population size can grow by a much higher magnitude

bottlenecks experienced is smaller. The figures also suggest a correlation between varying population size and fitness progression. It can be observed that, for all four cases, both fitness metrics develop in synchrony with the population size variation cycle. There is always a considerable improvement of the average fitness at the population bottleneck. This behavior is fairly self-explanatory because of the massive removal of inferior individuals by the house keeping thread. Second, best population fitness improvement mostly occurs after each bottleneck, but over an

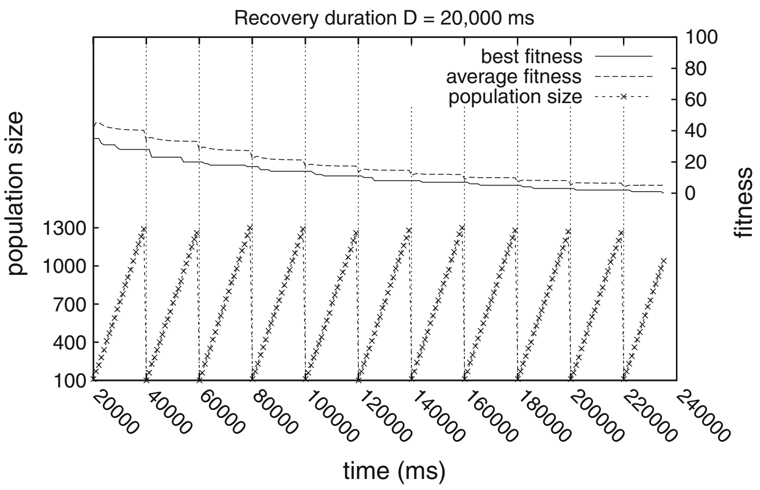

Fig. 5 Variable population size and fitness development in an example run with recovery duration $D = 20{,}000$ ms. In general, greater recovery duration needs longer time in an entire evolution process, but interestingly, requires less number of bottlenecks

extended period of time. This indicates that population bottlenecks accelerate the search for elites.

To further verify the above observation, we focus on the change of fitness between two consecutive bottlenecks, D ms apart. For a given recovery duration of D ms, we divide the evolutionary process into k micro-steps ($1000 \times k = D$). For each micro-step, we measure the best fitness improvement over the last 1,000 ms. We calculate the mean and variance of these measurements over the entire evolution of 200 runs. These statistics are depicted as bar charts with error intervals in Figs. 6, 7, 8, 9. From these figures we conclude that the rate of best fitness improvement is

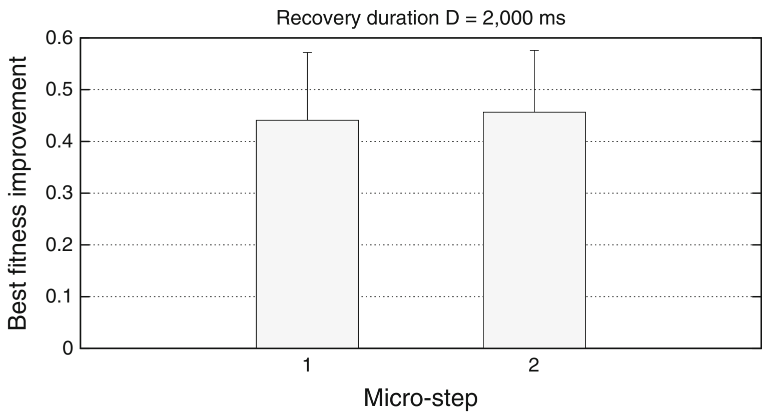

Fig. 6 The best fitness improvement in 2 ($k = 2$) different micro-steps with recovery duration $D = 2{,}000$ ms. Data are collected and averaged over 200 runs. In this configuration, the second micro-step has slightly higher fitness improvement than the first one

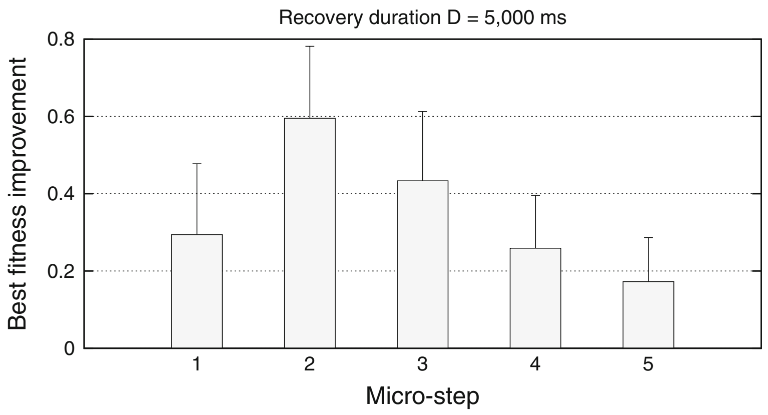

Fig. 7 The best fitness improvement in 5 ($k = 5$) different micro-steps with recovery duration $D = 5{,}000$ ms. Data are collected and averaged over 200 runs. It is clear to see that the second micro-step has the highest fitness improvement among all five

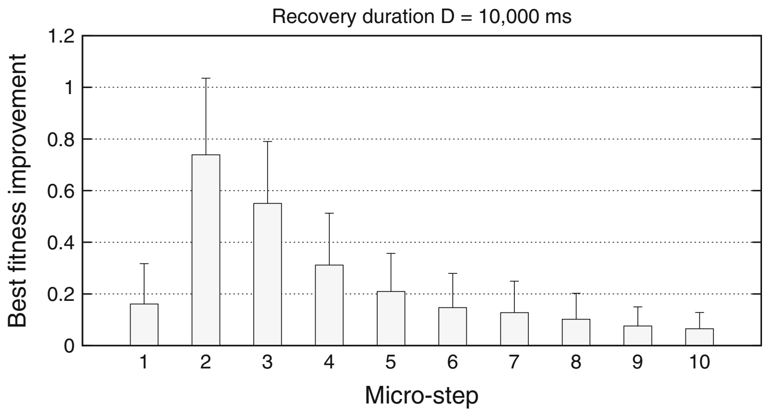

Fig. 8 The best fitness improvement in 10 ($k = 10$) different micro-steps with recovery duration $D = 10{,}000$ ms. Data are collected and averaged over 200 runs. With longer recovery duration, the effect of the bottlenecks can be seen more clearly

highly correlated with the population size fluctuation, and the second micro-step always has the greatest boost in best fitness improvements. Such performance leap can be attributed to the enhanced local search in a population with less diverse but more fit individuals that have survived after a bottleneck.

4.3 Variable population size and population diversity

With a binary string representation for the OneMax problem, the genotypic diversity is calculated by a measure proposed by Morrison and De Jong [22]. This measure is originally from the concept of *moment of inertia* for measurement of mass

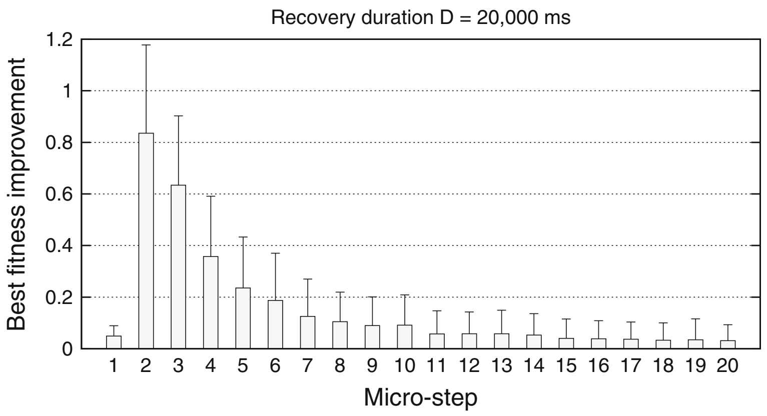

Fig. 9 The best fitness improvement in 20 ($k = 20$) different micro-steps with recovery duration $D = 20{,}000$ ms. Data are collected and averaged over 200 runs. From this and the three previous figures, it is seen that the best fitness improves drastically after bottlenecks with about 1,000 ms lag

distribution in physics and is transferred by the authors to measure the population diversity of high-dimensional GA genotypes. Morrison and De Jong further verify that their measure is equivalently functional as but more computationally efficient than the Hamming distance method, a common measurement for binary string genotype space diversity.

In Morrison and De Jong's measure, for P binary strings with length L, the coordinates of the centroid of these P strings $C = (c_1, c_2, c_3, \cdots, c_L)$ are computed as,

$$c_i = \frac{\sum_{j=1}^{P} x_{ij}}{P}, \tag{2}$$

where $x_{ij} \in 0, 1$ $(1 \leq i \leq L,\ 1 \leq j \leq P)$ is the value of the i-th allele on the j-th binary string. The moment of inertia I of the P strings about the centroid is,

$$I = \sum_{i=1}^{L} \sum_{j=1}^{P} (x_{ij} - c_i)^2. \tag{3}$$

Therefore, the genotypic diversity of a population independent of its size can be obtained as $\frac{I}{P}$.

Figures 10, 11, 12, 13 show the change of genotypic population diversity over time in the four example runs that we looked into previously. It can be observed that genotypic diversity has a generally decreasing trend as evolution proceeds, and each population bottleneck leads to a considerable reduction in genotypic diversity.

In terms of the phenotypic/fitness diversity, fitness variance is adopted as our proxy measurement. Figures 14, 15, 16, 17 depict the fitness variance changes as the population size fluctuates. Although not having exactly the same pattern as genotypic diversity, fitness variance also has a strong correlation with population size fluctuations. That is, not surprisingly, population bottlenecks reduce the fitness diversity a great deal.

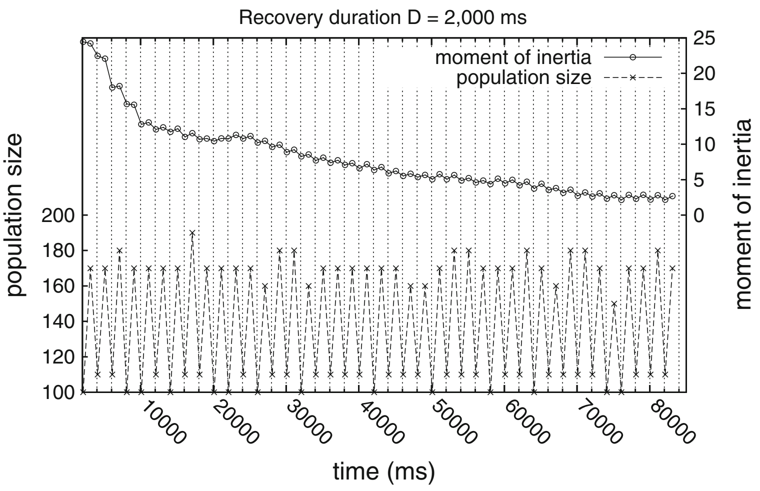

Fig. 10 The phenotype diversity measure changes with the population size fluctuates. The recovery duration $D = 2{,}000$ ms. Again each point represents the status data collected every 1,000 ms, so there is only one point between every two bottlenecks

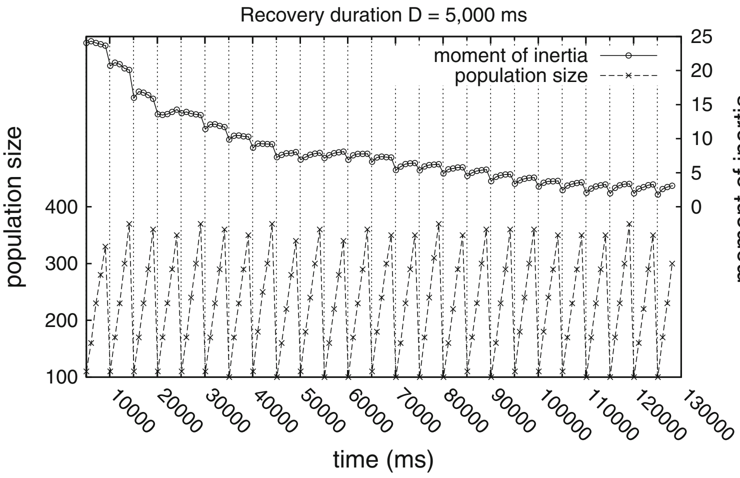

Fig. 11 The phenotype diversity measure changes with the population size fluctuates. The recovery duration $D = 5{,}000$ ms. Each bottleneck reduces the genetic diversity considerably

Table 1 shows the average diversity reduction over 200 collected runs. The higher the recovery duration *D*, the greater is the bottleneck effect, and thus the larger both genotypic and fitness diversity reductions are.

From investigations both at the population fitness progression and diversity variation, it can be observed that population bottlenecks considerably reduce the population diversity but effectively preserve highly fit individuals. Therefore, we interpret the results in the following way: Reduced population diversity enhances

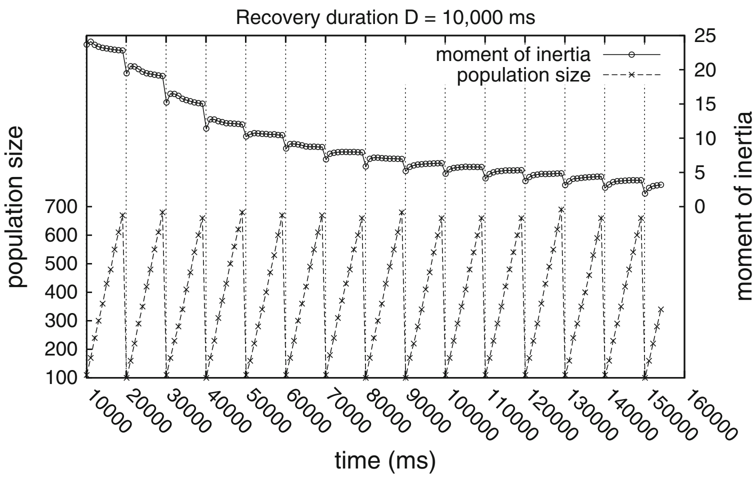

Fig. 12 The phenotype diversity measure changes with the population size fluctuates. The recovery duration $D = 10{,}000$ ms in this run

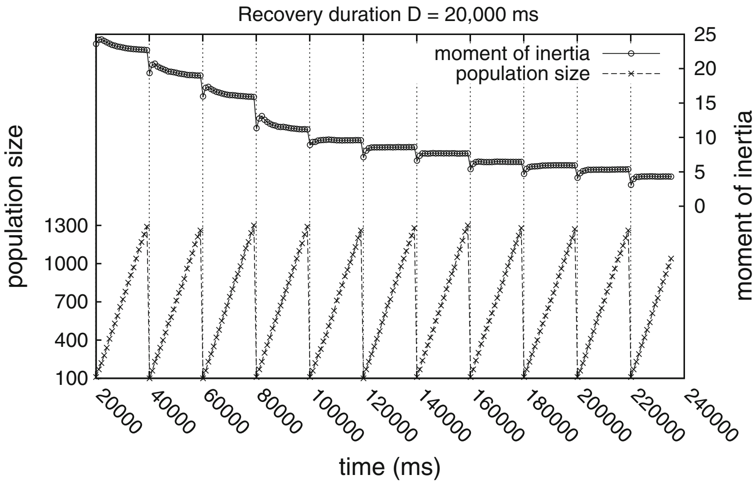

Fig. 13 The phenotype diversity measure changes with the population size fluctuates. With the longest recovery duration $D = 20{,}000$ ms, the effect of population size fluctuation on diversity reduction can be observed very clearly here

local search with those adaptive individuals. Thus, the search for fitter individuals can be accelerated after each population bottleneck, at least for this well-behaved problem. Such an observation is also in line with biological discoveries (see Sect. 2.1).

4.4 Comparative studies

A further motivation for this work was to perform comparative studies between our new parallel algorithm and a conventional GA on the same problem. For the APEA

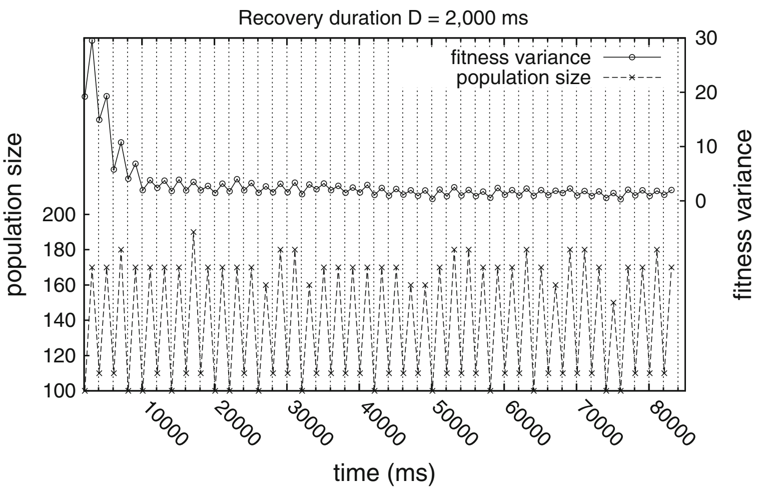

Fig. 14 The fitness variance changes with the population size fluctuates. The recovery duration $D = 2{,}000$ ms. Again each point represents the data collected every 1,000 ms

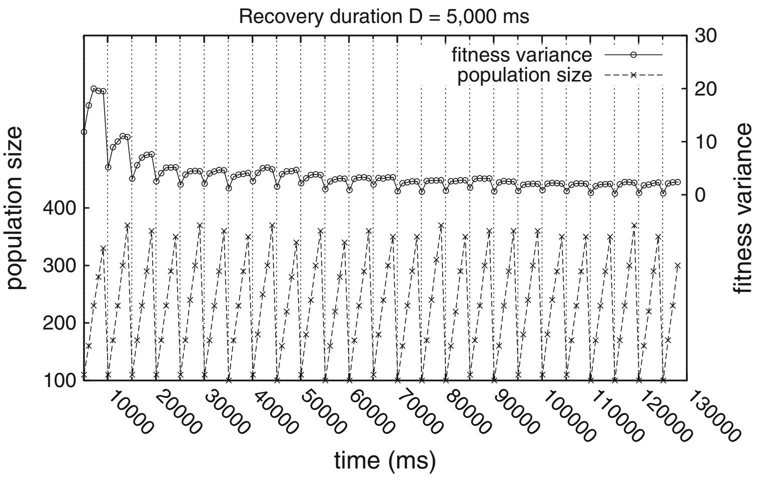

Fig. 15 With the recovery duration $D = 5{,}000$ ms, the fitness variance changes as the population size fluctuates

implementation it was important to verify that the devised evolutionary algorithm worked and performed comparably to a conventional approach. In practice, a small performance decrease can be tolerated because of the GPU acceleration which provides a massive decrease in clock time, it would even be acceptable if the number of evaluations required was less.

We compare to a conventional GA that has the same operations and the same parameter configurations as APEA, except that the population size is fixed. The termination criterion is again the reaching of the optimum. Since in the APEA

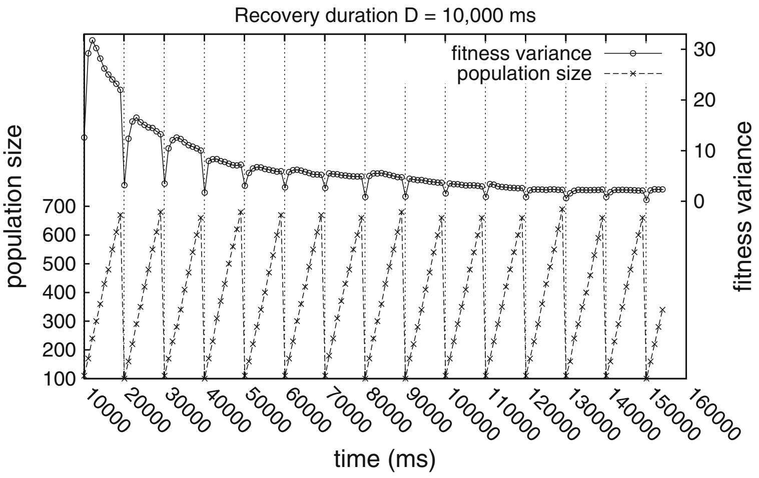

Fig. 16 With the recovery duration $D = 10{,}000$ ms, the fitness variance changes as the population size fluctuates

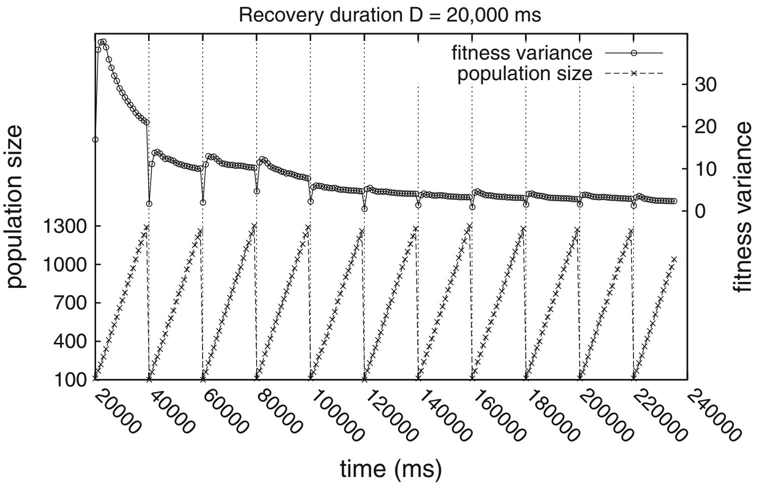

Fig. 17 With the recovery duration $D = 20{,}000$ ms, the fitness variance changes as the population size fluctuates. It can be seen from this and previous three figures that the fitness variance drops considerably after each bottleneck event

Table 1 Population bottleneck effect on genotypic diversity and fitness variance

D	Average genotypic diversity reduction	Average fitness variance reduction
2,000 ms	0.89	1.96
5,000 ms	1.65	3.65
10,000 ms	2.08	4.43
20,000 ms	2.42	4.52

population size fluctuates depending on recovery duration *D*, we calculate the average population size during an entire evolutionary process over 200 runs for each choice of *D*. This average population size will be set as the default and fixed population size for the conventional GA. Therefore, it is possible and fair to compare the new algorithm with four *D* values to four fixed-population-size GAs. For each algorithm, 200 runs are performed and for each run the number of evaluations is recorded.

Tables 2 and 3 show the results from two sets of algorithms on two benchmarks. The mean value and standard deviation of the total number of individual evaluations in each algorithm are compared. Both benchmarks benefit from small population sizes by looking at the number of evaluations.

For the OneMax problem, APEA always has less computational costs than a conventional GA with corresponding fixed population size. The obtained *p*-values from the Kolmogorov-Smirnov test, which indicate the probability that two sets of sample data have no significant difference, also confirm the superiority of APEA against conventional GAs.

For the more difficult Spears multi-model problem (see Sect. 4.1), both algorithms need much more computational efforts to reach the optima. Interestingly, however, the APEA shows substantial superiority by requiring much less individual evaluations than conventional GAs.

Table 2 Comparison on the computation costs (number of evaluations) to find the optima between APEA and conventional GAs on OneMax problem (*APS* stands for the average population size, and *SDV* stands for standard deviation)

APEA				Conventional GAs			*p*-value
D	APS	Mean	SDV	Popsize	Mean	SDV	
2,000 ms	135.59	5,258	628	136	5,707	1,185	<0.001
5,000 ms	227.29	6,654	703	227	8,364	1,562	<0.001
10,000 ms	379.62	9,418	818	380	12,443	2,307	<0.001
20,000 ms	684.70	14,819	1,036	685	20,588	3,430	<0.001

Table 3 Comparison on the computation costs (number of evaluations) to find the optima between APEA and conventional GAs on Spears multi-model problem (*APS* stands for the average population size, and *SDV* stands for standard deviation)

APEA				Conventional GAs			*p*-value
D	APS	Mean	SDV	Popsize	Mean	SDV	
2,000 ms	151.87	31,756	8,562	152	172,571	79,876	<0.001
5,000 ms	226.35	33,816	8,319	226	288,141	148,145	<0.001
10,000 ms	350.64	38,281	8,614	351	481,658	208,038	<0.001
20,000 ms	598.43	47,580	8,902	598	784,316	223,113	<0.001

These comparative studies further verify that the variation of population size in APEA can accelerate the evolutionary process. Moreover, this acceleration is considerably more noticeable when the variable population size scheme is applied on the more complex problem.

5 Concluding remarks and future work

In this article, we studied how the property of variable population size can accelerate evolution in the context of the asynchronous parallel evolutionary algorithm (APEA), designed to function with multiple CPUs and GPUs. The variable population size property of the algorithm results from the asynchronous behavior of this algorithm, rather than by intentional design.

From the analysis, it is seen that variation in population size leads to a varying pace of search in the evolutionary algorithm. That is, the search for new better individuals is accelerated each time after a population bottleneck. This effect is further investigated by looking into the population diversity variation. It is shown that the enhanced local search for fitter individuals is a result of the drastic population diversity reduction as well as elite individual preservation after population bottlenecks. The results also suggest that the degree of fluctuation in the population size can have different effects.

In this preliminary work, we simplify our parameter choices by using a fixed tournament selection size. The size of the tournament selection decides the pressure on choosing parent individuals. In a size-fluctuating population, the parent selection pressure changes with a fixed tournament size. That is, a larger population may have less chances to pick up better parents and thus to discover fitter offspring. This may also contribute in part to the observation of the bottleneck effects, as well as the major force of enhanced elite search with limited population diversity. The fixed tournament size is implemented here for the sake of simplicity. The balance among selection pressure, population diversity, and search ability is a topic that needs further examination. It would be interesting to test a varying tournament size, for instance, proportional to the population size, and we intend to examine this question in future work.

The findings in this work are analogous to those known from natural evolution, where it is seen that drastic changes of a population provide great variation potential and accelerate the acceptance of new genetic variations.

The performance of our algorithm is compared to conventional fixed-population-size GAs. By investigating the required computation costs (number of individual evaluations) for these two algorithms to solve the same test problem, it is shown that variable population size can effectively improve performance.

As with other previous examples, the results of this work verify a potential advantage of using variable population size schemes in evolutionary algorithms. The results also validate that the algorithm implemented for the parallel system is not only functional, but may by itself improve the performance of the system. Therefore, variable population sizing techniques hold a lot of potential for future research, especially given the fact that variable population size is a general property in natural systems which we know to be more sophisticated and robust than computational evolutionary systems. In our forthcoming work, the algorithm will be demonstrated on genetic programming systems. It will be interesting to see if the performance gain from variable population size is also reachable for genetic programming problems.

Acknowledgments W.B. acknowledges NSERC support under Discovery Grant RGPIN 283304-07.

References

1. E. Alba, M. Tomassini, Parallelism and evolutionary algorithms. IEEE Trans. Evol. Comput. **6**(5), 443–462 (2002)
2. S.H. Ambrose, Late pleistocene human population bottlenecks, volcanicwinter, and the differentiation of modern humans. J. Human Evol. **34**(6), 623–651 (1998)
3. J. Arabas, Z. Michalewicz, J. Mulawka, GAVaPS—a genetic algorithm with varying population size, in *Proceedings of IEEE Congress on Evolutionary Computation (CEC 1994)* IEEE Press, 1994, pp. 73–78
4. T. Back, A.E. Eiben, N.A.L. van der Vaart, An empirical study on GAs “without parameters”, in *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature (PPSN VI)*, vol. 1917 of *LNCS* (Springer, 2000), pp. 315–324
5. W. Banzhaf, S. Harding, W.B. Langdon, G. Wilson, in *Accelerating Genetic Programming Through Graphics Processing Units*, ed. by, R.L. Riolo, T. Soule, B. Worzel. Genetic Programming Theory and Practice VI , Genetic and Evolutionary Computation, chapter 15 (Springer, 2008), pp. 229–249
6. G. Cochran, H. Harpending, *The 10,000 Year Explosion: How Civilization Accelerated Human Evolution* (Basic Books, New York, NY, USA, 2009)
7. A.E. Eiben, E. Marchiori, V.A. Valkó, Evolutionary algorithms with on-the-fly population size adjustment, in *Proceedings of the 8th International Conference on Parallel Problem Solving from Nature (PPSN VIII)*, vol. 3242 of *LNCS* (Springer, 2004), pp. 41–50
8. C. Fernandes, A. Rosa, Self-regulated population size in evolutionary algorithms, in *Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN IX)*, vol. 4193 of *LNCS* (Springer, 2006), pp. 920–929
9. R. A. Fisher, *Genetical Theory of Natural Selection* (Clarendon, Oxford, 1930)
10. R. Frankham, Relationship of genetic variation to population size in wildlife. Conserv. Biol. **10**(6), 1500–1508 (1996)
11. A. Gherman, P.E. Chen, T.M. Teslovich, P. Stankiewicz, M. Withers, C.S. Kashuk, A. Chakravarti, J.R. Lupski, D.J. Cutler, N. Katsanis, Population bottlenecks as a potential major shaping force of human genome architecture. PloS Genet. **3**(3), 1223–1231 (2007)
12. D.E. Goldberg, Sizing populations for serial and parallel genetic algorithms, in *Proceedings of the Third International Conference on Genetic algorithms*, (San Francisco, CA, Morgan Kaufmann Publishers Inc., 1989), pp. 70–79
13. D.E. Goldberg, K. Deb, J.H. Clark, Genetic algorithms, noise, and the sizing of populations. Complex Syst. **6**(4), 333–362 (1992)
14. G.R. Harik, F.G. Lobo, A parameter-less genetic algorithm, in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)* (Morgan Kaufmann, 1999), pp. 258–267
15. D.L. Hartl, A.G. Clark, Principles of Population Genetics, 4th edn. (Sinauer Associates Inc. Publisher, Sunderland, MA, 2007)
16. J. Hawks, K. Hunley, S.-H. Lee, M. Wolpoff, Population bottlenecks and pleistocene human evolution. Mole. Biol. Evol. **17**(1), 2–22 (2000)
17. J. Hawks, E.T. Wang, G.M. Cochran, H.C. Harpending, R.K. Moyzis, Recent acceleration of human adaptive evolution. Proc. Nat. Acad. Sci. **104**(52), 20753–20758 (2007)
18. T. Hu, W. Banzhaf, The role of population size in rate of evolution in genetic programming, in *Proceedings of the 12th European Conference on Genetic Programming (EuroGP 2009)*, vol. 5481 of *LNCS* (Springer, 2009), pp. 85–96

19. J. Kennedy, W. Spears, Matching algorithms to problems: an experimental test of the particle swarm and some genetic algorithms on the multimodal problem generator, in *Proceedings of the IEEE International Conference on Evolutionary Computation* (IEEE, 1998), pp. 74–77
20. V.K. Koumousis, C.P. Katsaras, A saw-tooth genetic algorithm combining the effects of variable population size and reinitialization to enhance performance. IEEE Trans. Evol. Comput. **10**(1), 19–28 (2006)
21. F.G. Lobo, C.F. Lima, A review of adaptive population sizing schemes in genetic algorithms, in *Proceedings of the 2005 Workshops on Genetic and Evolutionary Computation (GECCO 2005)* (ACM, 2005), pp. 228–234
22. R.W. Morrison, K.A.D. Jong, Measurement of population diversity, in *Proceedings of the 3rd International Conference on Evolutionary Algorithm*, vol. 2310 of *LNCS* (Springer, 2001), pp. 31–41
23. M. Nei, T. Maruyama, R. Chakraborty, The bottleneck effect and genetic variability in populations. Evolution **29**(1), 1–10 (1975)
24. T. Ohta, Population size and rate of evolution. J. Mole. Evol. **1**(4), 305–314 (1972)
25. D. Schlierkamp-Voosen, H. Muhlenbein, Strategy adaptation by competing subpopulations, in *Proceedings of the 3rd International Conference on Parallel Problem Solving from Nature (PPSN III)*, vol. 866 of *LNCS* (Springer, 1994), pp. 199–208
26. R.E. Smith, Adaptively resizing populations: an algorithm and analysis, in *Proceedings of the 5th International Conference on Genetic Algorithms* (San Francisco, CA, USA Morgan Kaufmann Publishers Inc, 1993) , 653 pp.
27. M.E. Soule, B.A. Wilcox, *Conservation Biology. An evolutionary-ecological perspective*. (Sinauer Associates, Sunderland, MA, USA, 1980)
28. M. Tomassini, L. Vanneschi, J. Cuendet, A new technique for dynamic size populations in genetic programming, in *Proceedings of IEEE Congress on Evolutionary Computation (CEC 2004)* (IEEE Press, 2004), pp. 486–493
29. S. Wright, *Evolution and the Genetics of Populations: Genetics and Biometric Foundations v. 4 (Variability Within and Among Natural Populations).* (University of Chicago Press, Chicago, IL, 1984)