Comparison of Different Computing Platforms for Implementing Parallel Genetic Programming

Ruihua Zeng, Zhixing Huang, Yongliang Chen, Jinghui Zhong
School of Computer Science and Engineering
South China University of Technology
GuangZhou, China
jinghuizhong@scut.edu.cn

Liang Feng
College of Computer Science
Chongqing University
Chongqing, China

***Abstract*—Genetic programming (GP) is a powerful tool for knowledge discovery and data mining. Over the past decades, GP has been implemented in various parallel computing platforms to reduce its search time. However, these parallel GPs have different design principles and performance characteristics, which makes it difficult for users to choose the proper parallel GP in practice. To address this issue, this paper focuses on comparing and analyzing the characteristics of parallel GPs implemented in different computing platforms, in terms of running time, the speedup ratio, and the scalability. Based on the empirical results, the guidance of selecting different parallel GPs is concluded.**

***Index Terms*—Genetic Programming, parallel platform, MPI, GPU, Spark, Parallel Computing**

I. Introduction

Genetic Programming (GP) is a powerful evolutionary computation (EC) algorithm that can automatically design computer programs to solve user-defined tasks. It uses genetic operators such as crossover, mutation and selection, to evolve a population of individuals. Each individual in GP can be decoded to a parsing tree to represent a computer program and solve the user-defined tasks. Nowadays, GP has undergone a rapid development and a number of enhanced GP variants have been proposed [1], [2], such as Cartesian Genetic Programming (CGP) [3], Geometric Semantic GP (GSGP) [4], and Gene Expression Programming [5]–[8].

However, GP requires a high computation time due to its iterative nature, which limits its utilization in real-world applications [9]–[11]. One of the effective methods to improve the efficiency of GP is parallelization. Currently, there are many parallel computing platforms for the GP parallelization such as Message Passing Interface (MPI) [12], OpenMP [13], and Spark [14]. Different parallel platforms have different characteristics. For instance, MPI is a language-independent communication protocol for multiple processes. Every process in MPI owns an independent stack and a shared code segment. The processes communicate with each other by message transmission. Similar with MPI, OpenMP is another parallel programming technique using shared memory. However, limitation exists as a single host is necessary. And because of this limitation, the programs of OpenMP are usually simpler with a lower memory overhead than those of MPI. Unlike MPI and OpenMP, Spark is a computing engine specially designed for big data processing. It is suitable for iterative calculations because of the introduction of the Resilient Distributed Dataset (RDD) [15]. Spark offers two types of APIs (i.e., transform APIs and action APIs) to operate the RDD. In addition to the technologies mentioned above, there are many other parallel computing technologies such as Graphics Processing Unit (GPU) [16], MapReduce [17], Hadoop [18] and so on. Though the mentioned studies of parallel computing platforms can facilitate the researchers and industries to implement the GP on different platforms, it is still confused for researchers and industries to choose the proper parallel computing platform because of the different performance characteristics of different platforms. To facilitate real-world applications, a comprehensive comparison between GPs on different parallel computing platforms is made in this paper.

In this paper, we select three typical parallelization technologies, which are MPI, GPU, and Spark, to make the comparison. Without loss of generality, a recently published GP variant named Self-Learning Gene Expression Programming (SL-GEP) is adopted as the based GP solver and it is implemented on MPI, GPU, and Spark respectively for comparison analysis. It is worth to mention that the SL-GEP can be replaced by other GP variants because these parallel computing platforms are algorithm-independent. The contributions of our work are listed as follows.

1)A series of experiments are set up to comprehensively compare the performance (i.e., running time and speedup ratio) of the representative parallel GPs

2)Based on the empirical results, the performance characteristics of different parallel GPs are concluded.

3)The guidance for designing parallel GPs is drawn out to help users select parallel technologies in real world according to the designed principles and the empirical results of parallel GPs.

II. Related Work

In recent years, parallel GP is one of the most effective methods to reduce the computation time of GPs on large-scale and complicated optimization problems. Existing parallel GPs can be generally classified into the following classes based on the parallel computing platform.

The first category is based on the MPI. The representative work was contributed by M. Tomassini et al. [19] and Du

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

et al. [20]. In [19], M. Tomassini et al. implemented GP on MPI and designed a graphical user interface (GUI) to facilitate the applications. In [20], Du et al. implemented a parallel GEP algorithm based on MPI, and the experimental results showed that the speedup ratio approached linear increasing. In addition, there are some other related work on GP with MPI. Salhi, Glaser et al. [21] combined GP with island model based on MPI. Stoffel and Spector [22] studied the MPI version of high-performance genetic programming system (HiGP) and found out that the speedup ratio can be linear with the number of processors.

The second category focuses on using GPU. This category is likely to be established by Harding and Banzhaf [23], who parallelized the GP using GPU. In addition, they also proposed a data parallel approach for GP by using multiple computers to mitigate the overhead of program compilation and implemented it in Compute Unified Device Architecture (CUDA) C code [24]. Besides, Robilliard et al. [25] also implement the parallel GP on the G80 GPU, they designed a parallel scheme which was able to instantiate several GP programs on the GPU. Later, to further improve the efficiency of the GPU implementation, Shao et al. [26] proposed a parallel GEP (pGEP), which utilized the post-order visiting array to decode the tree structure of GP individuals and used the constant memory of GPU to store the chromosomes. These two measures successfully further improve the efficiency of the algorithm. Cano et al. [11] combined the GP with GPU to solve association rule mining task and achieved good results. In recent years, Chitty [27] tried to improve the performance of parallel GP based on GPU by considering L1 cache and shared memory. And he implemented a GPU-based GP with a two-dimensional stack model to improve the performance of parallel GP [28]. Huang et al [29] proposed a fast parallel GP by utilizing both GPU and multiple core CPUs.

The third category focuses on using Cloud computing techniques such as MapReduce and Hadoop. For example, Du et al. [30] implemented MR-GEP based on MapReduce to validate the effectiveness of the proposed MR-PEA model. Xu et al. [31] proposed a MapReduce based parallelized GEP to solve large-scale classification problems. Khan et al. [32] investigated the robustness of parameter settings of Hadoop in GEP implementations by running programs in two Hadoop clusters. As a kind of Cloud computing techniques, Spark is a new technology becoming more and more popular in recent years. There are few studies on combining the GP with Spark to the best of our knowledge.

Moreover, several other high-performance technologies or platforms are studied for parallel GP such as Parallel Virtual Machine (PVM) and OpenMP. For PVM, Fernndez et al. [33] designed a parallel genetic programming based on PVM to accelerate the GP algorithms. Besides, Wu et al. [34] implemented the parallel niche gene expression programming (PNGEP-MP) based on OpenMP and compared their work with conventional GEP on mining and classification function.

In summary, most methods mentioned above are only studying the implementation of one technology for parallel GP. Whereas, GP performs differently under different parallel computing technologies or platforms. Therefore, we plan to compare different parallel GPs and conclude the characteristics of different parallel GPs in this paper to facilitate the further research and industrial application. Among the existing studies, we select two representative implementations and an emerging technology as our comparing methods(i.e., MPI, GPU, and Spark).

III. Framework of Parallel GPs

In this section, the key idea of different parallel GPs and their implementation frameworks in our work are introduced. First of all, the general procedure of SL-GEP and a general parallel GP framework are given. Then, based on the SL-GEP and general parallel GP framework, parallel SL-GEP on different parallel computing platforms (i.e., MPI, GPU, and Spark) are presented.

A. Framework of SL-GEP

SL-GEP is a recently published GP variant which have been shown effective for solving various problems [7]. Generally, the procedure of SL-GEP consists of the following operations: initialization, crossover, mutation, and selection. By iteratively performing the last three steps, SL-GEP gradually finds better solution.

In SL-GEP, each chromosome can be decoded to a parsing tree to represent a computer program. Initially, the chromosomes are generated randomly based on primitive settings (i.e., terminals, functions, constants, and other user-defined structures) to form the initial population. More details of the chromosome representation of SL-GEP can be referred in the original paper [7]. Each individual is expressed as a vector of primitive symbol.

After initialization, SL-GEP applies a series of genetic operators (i.e., crossover, mutation and selection), to evolve the chromosomes and find the chromosomes with better fitness.

In crossover, two chromosomes are selected randomly as the parent chromosomes from the current population and swap a part of chromosomes to generate two new offspring (i.e., children chromosomes). Before swapping, two crossover points is randomly chosen from each chromosome. Then, the two parents exchange their genes with each other based on the crossover points.

In mutation, genes (i.e., the primitives) are randomly changed to new a value with a certain probability. It is worth to mention that both the crossover and mutation are performed following the chromosome representation. The goal of crossover and mutation operation is to bring the gene diversity into the population so that the population can search the solution space for better solutions.

The selection is performed to choose the better chromosomes between the parent individuals and the newly generated children chromosomes based on their fitness values. After evaluating the fitness of newly generated individuals, the inferior ones between the parents and children are eliminated, while the better ones are selected out to form the new population for the next generation.

B. The general Parallel Computing Framework

Generally, existing parallel GPs focus on using multiple computing units (e.g., CPU threads or GPU threads) to perform fitness evaluation parallelly to reduce the computation time because the fitness evaluation has a parallel nature and is often the most time-consuming part of computation in GPs. The abstracted framework of parallel GPs is shown in Fig. 1. After the initialization of GP, the population is divided into several groups. Each group is considered as a unit to be computed. Then, a population reproduction loop containing the fitness evaluation, is performed parallelly for every group. Besides, the genetic operators (e.g., mutation, crossover, and selection) are performed parallelly or sequentially in the reproduction loop to evolve the population. When the termination condition is met, the best individual of the population is selected as the final solution to the given problem. Based on the general parallel framework, we implement the SL-GEP on the three parallel computing platforms (i.e., MPI, GPU, and Spark) respectively.

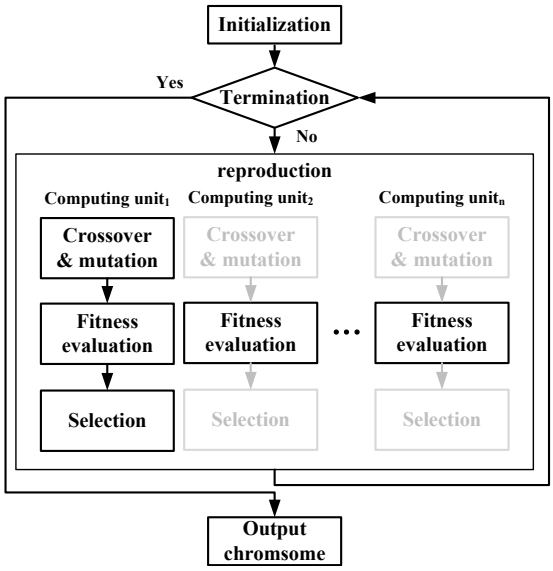

Fig. 1. The parallel framework of GP.

C. MPI-Based Implementation

Massage Passing Interface (MPI) is a communication protocol rather than a programming language [35]. In general, MPI supports not only point-to-point communication but also collective communication. Programs can be adapted into the MPI framework in different programming languages such as C, C++, and Fortran. In this paper, we use C/C++ to implement the MPI+SL-GEP, denoted as MPIGP. The framework of MPIGP is shown in Fig. 2 and the introduction of the implementation are as follows.

Firstly, the initialization procedure is performed to initialize the processes in the communication domain of MPI and a unique ID is assigned to every process. After the initialization, the master process (e.g., process$_0$) divides the GP population and sends groups of individuals to the other processes uniformly. The genetic operations and fitness evaluation are performed parallelly in these processes. It is worth to mention that, some genetic operations, such as crossover and selection, may require the global information of the whole population (i.e., the parent chromosomes may come from different groups of the different processes), so the population is put into the shared memory of MPI so that every process can access the whole population. The independent genetic operations and fitness evaluation in each process are the same as those introduced in Section III-A. Following the reproduction, the computing results of processes (i.e., the new-born individuals and their fitness value) are required to send back to the master process to make a global update. Based on the computed results from the other processes, the master process will make a new division on the new population and repeat the reproduction. This loop of parallel reproduction will repeat until the termination condition is met. Finally the chromosome with the best fitness value will be outputted as the final solution.

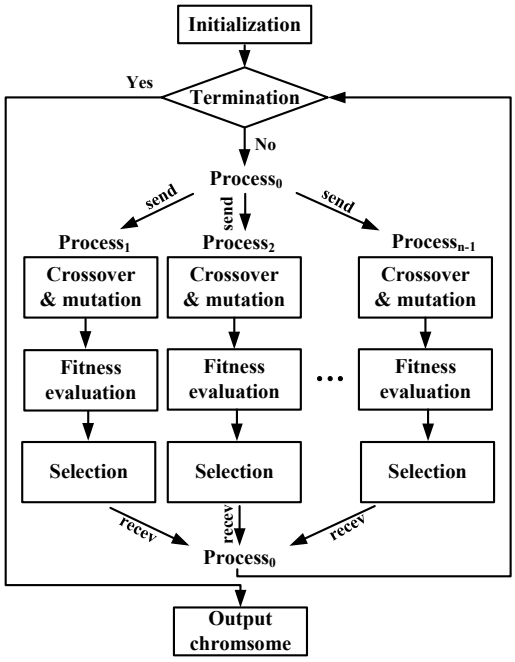

Fig. 2. The parallel framework of MPIGP.

D. GPU-Based Implementation

Graphic Processing Unit (GPU) is a parallel processing hardware that is used for graphics processing in the early years. Nowadays, GPU has undergone a rapid development and it can handle computation work of large-scale data efficiently. CUDA is a computing framework designed for the implementation of GPU programs, which is developed by NVIDIA corporation. Nowadays, CUDA has successfully solved many problems in academic researches and practical applications such as medical image, computed fluid dynamics and so on [16]. Developers can easily use C, C++, and FORTRAN language to write programs under the CUDA framework. We implement

SL-GEP on GPU platform with C language under CUDA framework to form the GPU+SLGEP denoted as GPUGP. The framework of GPUGP is shown in Fig. 3 and the details of GPUGP are described as follows.

Firstly, GPUGP performs the initialization to initialize the GP population and the GPU platform. The initialization of GPU platform includes the GPU memory allocation and the memory initialization. There are two important issues in designing the memory allocation of GPU [36]. On the one hand, the data computed by GPU is required to be organized as an array structure to fully utilize the GPU computing resources. Therefore, the tree structure for parsing trees of chromosomes is decoded into the pre-order visiting array in GPU memory. On the other hand, to reduce the times of memory access, the fitness evaluation of each chromosome needs to be finished by a single block. It is worth to introduce that the GPU has a hierarchical computing structure consisting of stream multiprocessor (SM), block, and thread. By evaluating each chromosome using a distinct block, the GPU only needs to read the chromosomes from memory to cache once and finish fitness evaluation efficiently. After the initialization, the GPU iterates the reproduction of GP (i.e., crossover and mutation, fitness evaluation, and selection) until the termination condition is met. Similar with MPIGP, to allow blocks to perform genetic operations, the population is stored in the global memory of GPU which can be accessed by all blocks. In each iteration, there is also a synchronization barrier to all blocks in GPU so that the selection and the update of GP population can be performed correctly. Finally, the chromosome with best fitness value is outputted as the final solution.

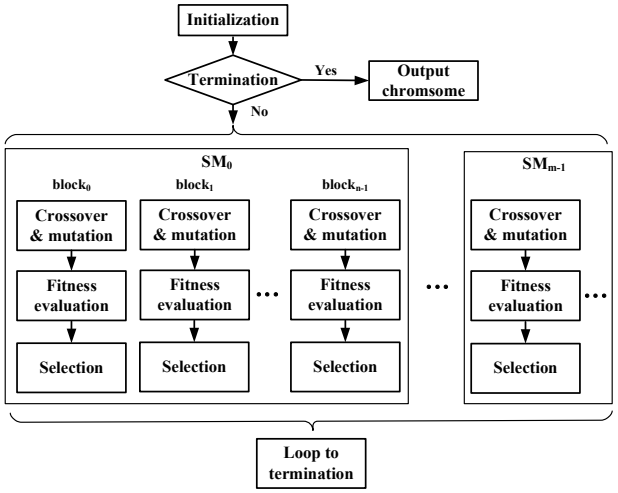

Fig. 3. The parallel framework of GPUGP.

E. Spark-Based Implementation

Spark is a popular and fast parallel computing engine. Because Spark can access diverse data sources (e.g. HDFS and HBase) and has higher efficiency than Hadoop in iterative problems [37], Spark has been widely applied in big-data problems currently. In Spark, one of the most important data structures is the resilient distributed dataset (RDD). RDD is a programable read-only multiset-of-data item for fault tolerance and parallel computing. In this paper, we implement the parallel SL-GEP on the Spark platform with Java language and this parallel GP is denoted as SparkGP. The framework of SparkGP is shown in Fig. 4.

At the beginning, besides the GP population initialization, the Spark platform are initialized to allocate the memory of Spark and launch the Spark computing units. Similar with the decoding of pre-order visiting array in GPU, the GP population is required to be encapsulated into RDD to facilitate Spark to compute before performing the GP reproduction. RDD is further divided into several partitions. The number of partitions is equal to the number of computing units of Spark (i.e., the executor). After that, the executors in Spark start to perform the reproduction loop parallelly with the shared program code. In each iteration of the GP reproduction, the RDD holding the population will be transformed into a new RDD (denoted as RDD') holding the new population by executors after genetic operations and fitness evaluation. And these new population will be transformed into another RDD in next iteration. However, because of the lazy evaluation mechanism of Spark which limits the transform APIs to be performed only when the action APIs are called, the parallel computation of executors is truly performed only when the SparkGP goes through all the program code and reaches the action API. In SparkGP, the action API is regarded as a kind trigger and it is performed after the selection and the update of GP population. The reproduction loop of SparkGP will also be terminated when it reaches a termination condition.

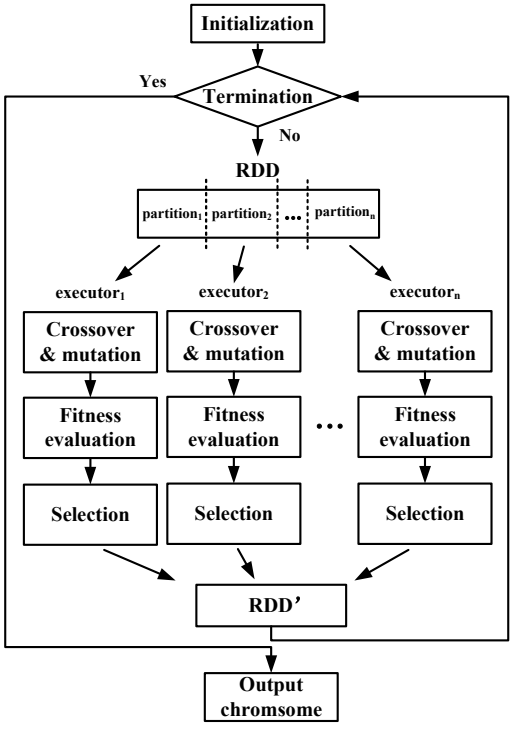

Fig. 4. The parallel framework of SparkGP.

TABLE I
THE BENCHMARK PROBLEMS IN GP AREA

Problem	Objective Function	Data Setting
F1	$x^5 + x^4 + x^3 + x^2 + x$	$U[-1, 1, 200]$
F2	$sin(x) + sin(x + x^2)$	$U[-1, 1, 200]$
F3	$sin(x) + sin(y^2)$	$U[0, 1, 1000]$

TABLE II
THE PARAMETER SETTING OF THE ALGORITHMS

Parameter	Value
NP	50
h	10
h'	3
K	2
$EvaluationTimes$	6000

IV. EXPERIMENTS AND COMPARISONS

A. Experimental Settings

The overall purpose of the experiments is to compare different parallel GPs and analyze their characteristics. Since the experiments mainly focus on the non-functional metrics (i.e., the running time, the speedup ratio, and the scalability) instead of the searching efficiency of different parallel GPs, only three symbolic regression problems with different properties are selected to simplify the experiment but without loss of generality. Table I lists the three benchmark problems for experimental study. In Table I, the second column shows the objective functions, while the last column describes the sample data from a uniform distribution in the form of $U[u, v, r]$, where r represents the sample number and u and v are the upper and lower bounds of the sample. The function set of SL-GEP is set as $\{+, -, \times, \div, sin, cos, e^x, ln(|x|)\}$. Table II describes the parameter settings of all the compared algorithms. The first three parameters h, h' and K are three important hyper-parameters in SL-GEP. h represents the length of main program, h' represents the head length of ADFs, and K means the number of ADFs in each chromosome. Each algorithm is run for 30 times independently on each problem, and the average results are used for comparison analysis. For all problems, the maximum fitness evaluations is set to be 6000. The running environment of MPIGP and SparkGP is a computer with an Intel(R) Core (TM) i7-7820HQ CPU (4 cores and 8 threads) and 16GB memory and GPUGP is run on a computer which is integrated with a graphics card of NVIDIA GeForce GTX 1070.

B. Performance Metrics for Comparison

First, we investigate the running time of each parallel GP on the test problems. Then, the parallel speedup ratios of the three GPs are analyzed. The speedup ratio is calculated by

$$Speedup = \frac{T_1}{T_n} \tag{1}$$

where T_1 is the average running time of the sequential SL-GEP and T_n is the parallel executing time with n cores or threads. It should be noticed that MPIGP and GPUGP are implemented in C language, but SparkGP is implemented in Java. Therefore, to guarantee the fairness of the comparison and erase the interior differences between different languages, the speedup ratios of the algorithms are calculated based on the same executive environment (both T_1 and T_n are results obtained from C or Java). We also tested the performance of these three algorithms by varying the data scale and population size. In the experiment of data scale, the size of dataset is set to 200, 1000, 5000, 10000, 20000 and 50000. And the influence of population size is investigated by altering the size of population to 50, 128, 256, 512 and 1024. In these two experiments, the number of processes in MPIGP is set to 4, the number of cores in SparkGP is set to 4 and there are 60 blocks with 256 threads in every block in GPUGP.

C. Experimental Results and Analysis

The results are presented in five seperated parts. The first three parts show the properties of three parallel GPs. The last two parts make comparisons between the three parallel GPs using different population sizes and data scales.

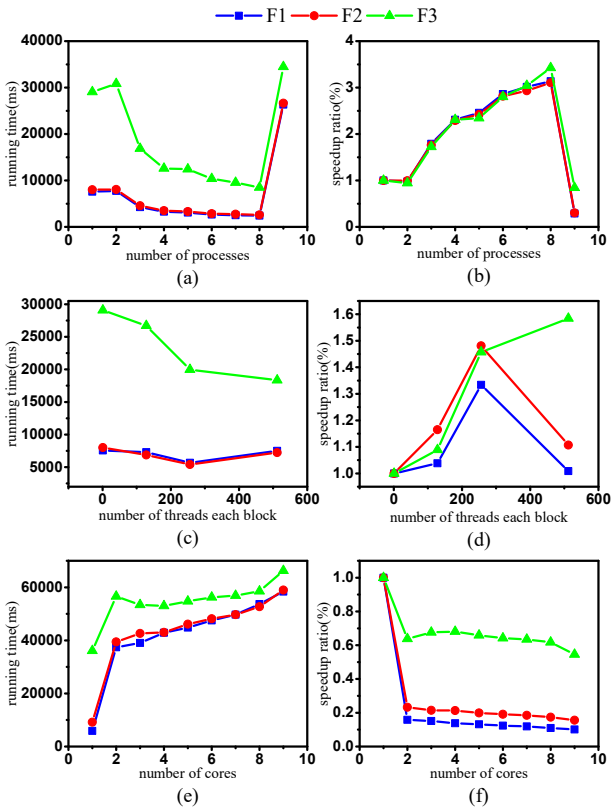

Fig. 5. The running time and speedup ratio on three parallel GPs: (a) the running time of MPIGP; (b) the speedup ratio of MPIGP; (c) the running time of GPUGP; (d) the speedup ratio of GPUGP; (e) the running time of SparkGP; (f) the speedup ratio of SparkGP

1) Results of MPIGP: Fig. 5.(a) shows the relationship between running time and the number of processes in MPIGP. When the number of processes is 1, the experiment is performed by the sequential algorithm. The running time of F3 is higher than F1 and F2. This is mainly because the dataset size

of F3 is 1000, which is five times of the size of F1 and F2. It can be observed that the performance deteriorates when there are two or nine processes. The reasons for the deterioration are different. When there are two processes, the program is actually serial, because there is only one process performing calculation, while the one is responsible for message delivery. Therefore, the time consumption in this case is greater than serial program with extra communication overhead. On the other hand, when there are nine processes, the population will be divided into eight equal parts. Since there are only eight physical threads in the experimental computer, the 9th process has to remain waiting until one of the other processes finish its task.

Fig. 5.(b) shows the relationship between the number of processes and the speedup ratio in MPIGP. The speedup ratio increases as process increases. The situation is related with Fig. 5.(a) which shows a deterioration of performance when the numbers of processes are two and nine. It can be found that when there are 8 processes, the speedup ratio reaches the maximum, and the maximum of speedup ratio is more than 3. In ideal situation, the speedup ratio of parallel program can reach 6 or 7 when the number of processes is 8. But in MPI, the more processes there are, the greater overhead of the inevitable communication expense is needed. When the number of processes increases, the burden of allocating individuals for the main processes ($process_0$) becomes larger, which slows down the whole evolutionary process of SL-GEP.

2) Results of GPUGP: Fig. 5.(c) illustrates the relationship between running time and the number of threads in GPU. The number of blocks is set to 60 in these cases. It can be observed that in F1 and F2, the running time of GPUGP remains declining and reaches the bottom when the number of threads is 256 in every block. But when the number of threads is more than 256, the performance on F1 and F2 deteriorates. This is because the fitness of every individual is computed by all threads in a single block in one fitness evalulation. If the number of threads is larger than the input data, a part of threads will keep waiting and be wasted. On the contrary, if the number of threads is much less than data scale, each thread is required to perform many fitness evaluation to process all input data. Therefore, using 256 threads is better than the case using 512 threads in F1 and F2. And because that the number of threads is always less than the number of data, the running time of F3 keep decreasing as the number of threads increases from 1 to 512. Besides, based on some previous work [38], the performance of algorithms roughly maintains the same level with different number of blocks.

Fig. 5.(d) indicates how speedup ratio is influenced by the number of threads. In this figure, the speedup ratio of F2 exceeds F1 in all these testing cases. This is because the computational ability of one thread is fixed when the number of threads remains static. And the only difference between F1 and F2 is the computational complexity. As for F1, it is not a complicated problem, which makes each thread only need to contribute part of its computational capability to solve the problem. But when it comes to F2, each thread may exhaust its capability to solve the problem. Therefore, compared with F1, F2 can make full use of the GPU, so that the speedup ratio of F2 exceeds that of F1.

3) Results of SparkGP: Fig. 5.(e) shows the relationship between running time and the number of cores in Spark. The results of SparkGP are different from MPIGP. It can be observed that the running time increases with the increment of cores, so that the serial program runs the fastest in these testing cases. It's abnormal that the parallel program is slower than the serial program. One of the reasons for this situation is the implementation mechanism. When SparkGP starts running, the program will start up a Diver, which will initialize a variable called SparkContext. SparkContext transforms population to RDD and send it to executors in each generation. When the fitness values are calculated and new individuals are sent back, action APIs will be executed, which costs most of the time. As the number of cores increases, this scheduler delay will be higher and higher. That's why the polyline shows an increasing tendency. Another reason is time consumption of the deserialization task. In Spark, SparkContext uses RDD to send individuals to each executor through transform APIs. The transform operators need to be executed in a serial manner, demanding a lot of extra executive time. Thus, the task deserialization also results in the unsatisfying performance of SparkGP.

Fig. 6 shows the details of time consumption of a job where the x-axis is the task index in a job. The reason why there are only four tasks on x-coordinate is that SparkGP only runs in 4 cores. It is clear that the program spends most of the time in scheduler work. The total time consumption is 13ms with 11ms cost by scheduler delay. The computing time only cost 1ms, accounting for one thirteenth of the total executive time. Therefore, though the program is parallel, the overhead for parallelization cost a lot. Thus, why the serial program runs faster than parallel program in F1 and F2.

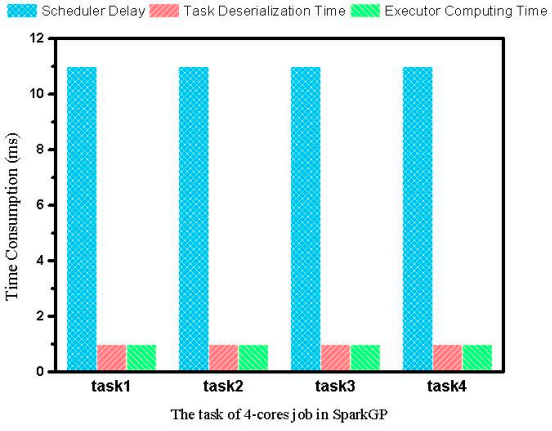

Fig. 6. The event timeline of 4-cores job.

Fig. 5.(f) shows the relationship between speedup ratio and the number of cores. It can be observed that the speedup ratio of F2 is higher than F1. The reason is that F2 includes $+$, sin and e^x, which is more complicated than F1 with only $+$

and e^x. As a result, the computing time of F2 will be more than F1 with the constant scheduler delay. Suppose that the running time of sequential program is T_1, the scheduler delay is T, and the parallel number is n. The speedup ratio can be expressed as:

$$Speedup = \frac{T_1}{T + T_1/n} = \frac{1}{T/T_1 + 1/n} \quad (2)$$

where both T and n of F1 is equal to F2 because of the same number of used cores. Since the T_1 of F2 is larger than F1, there is a higher speedup ratio of F2 compared with F1.

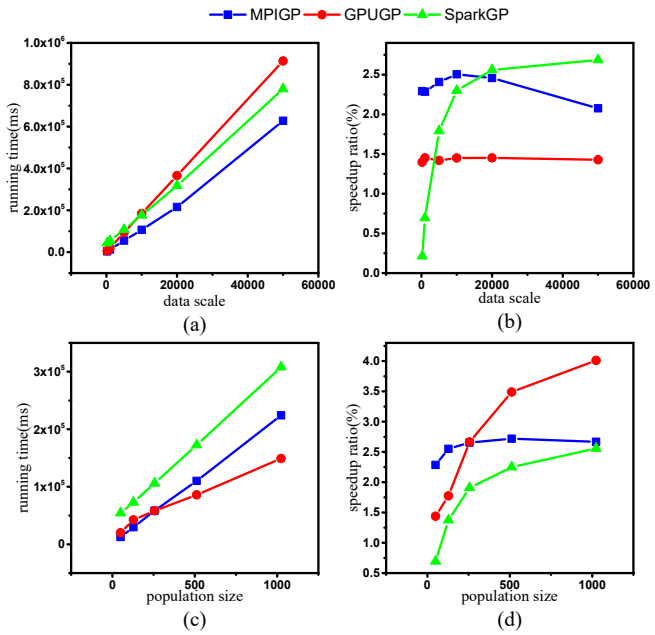

Fig. 7. The comparison results of three parallel GPs on various data scales and population sizes: (a) the running time vs. data scale; (b) the speedup ratio vs. data scale; (c) the running time vs. population size; (d) the speedup ratio vs. population size

4) Impacts of data size: In this experiment, we test the performance of these three parallel GPs on data with different size. The results of this experiment are shown as follows.

Fig. 7.(a) shows the relationship between running time and data scale of these parallel GPs. It can be observed that when there is a small amount of data, MPIGP and GPUGP outperform SparkGP. Especially when the size of dataset is smaller than 10000, SparkGP is slower than GPUGP, and MPIGP is the best one. As the data scale is larger than 10000, MPIGP still dominates the comparison and SparkGP exceeds GPUGP to become the second fastest parallel GP.

Fig. 7.(b) shows the relationship between speedup ratio and data scale. The speedup ratio of MPIGP reaches the top when the data scale is 10000. And the curve declines when the data scale increases. But in GPUGP, the speedup ratio is steady when the data scale ranges from 200 to 50000. Different from MPIGP and GPUGP, the speedup ratio of SparkGP keeps rising when the data scale is increasing. In the interval of 200 to 10000, MPIGP always performs the best. Whereas when the data-size approaches to 20000 or more than 20000, the speedup ratio of SparkGP exceeds MPIGP and becomes the best one among these three technologies.

5) Impact of population size: In this experiment, we test the performance of these three technologies with different population size. The results of this experiment are shown as follows.

Fig. 7.(c) shows the relationship of these three technologies between running time and population size. When the population size is equal to 50 and 128, MPIGP performs the best. When the population size is more than 256, GPUGP becomes the best. The figure also shows that the running time of MPIGP and sparkGP grow linearly with the population size and SparkGP always performs the worst. As for MPI and Spark, though the cost of communication will increase, it is not large enough to eliminate the increasing trend of running time. But as for GPU, as the population size increases, the resource of every block will be made full use of, which makes it outperform other technologies when the population size is large enough.

Fig. 7.(d) shows the relationship between speedup ratio and population size of these three technologies. It can be observed that when the population size is equal to 50 and 128, the speedup ratio of MPIGP is the highest. As the population size reaches or exceeds 256, the speedup ratio of GPUGP becomes the best and increases steadily. When the population size is larger than 1024, the speedup ratio of SparkGP approaches MPIGP and the trend of these two curves tends to be smooth. The scheduling time of SparkGP will increases with the increment of population size, while the growth of data scale can play the performance of SparkGP. This is the reason why the SparkGP performs well with the increasing data scale, while performs worse with the increase of population size.

V. Conclusions

In this paper, we compare the performance of parallel GPs developed on different computing platforms (i.e., MPI, GPU and Spark). Three versions of parallel GPs are implemented accordingly and tested on symbolic regression problems with different features such as the dimension of problem and the size of training data. Based on the comparison studies, we obtained the following conclusions.

1) Parallel GPs with MPI generally can perform well on data set with different scales. The number of processors in the computer has a significant influence on performance of the GPs. The search efficiency of parallel GPs with MPI increases as the number of processors increases. However, the programming complexity and communication overhead of GPs with MPI are relatively high.

2) GPU contains a large number of computing units. Thus, parallel GPs with GPU is quite suitable for evolving a large number of fitness evaluations, especially the applications which need a large population size to maintain the population diversity. Besides, the parameter settings in GPU have significant influence on the performance of the algorithm. Therefore, parallel GPs with GPU should be carefully implemented and

tuned based on the physical computing environment and the application problem.

3) Parallel GPs implemented in Spark performs poorly in small scale problems. However, when the training dataset becomes large, GPs with Spark can perform much better. The speedup ratio of GPs with Spark in large dataset can even be superior to that of MPI. Besides, the program complexity of GPs with Spark is much simpler than those using GPU and MPI.

ACKNOWLEDGMENT

This work is supported by the Program for Guangdong Introducing Innovative and Entrepreneurial Teams (Grant No. 2017ZT07X183), the Guangdong Natural Science Foundation Research Team (Grant No. 2018B030312003), and the Fundamental Research Funds for the Central Universities (Grant No. D2191200).

REFERENCES

[1] J. R. Koza, *Genetic programming*. MIT Press, Cambridge, Massachusetts, 1992.

[2] E. J. Vladislavleva, G. F. Smits, and D. Den Hertog, "Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 2, pp. 333–349, 2009.

[3] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *European Conference on Genetic Programming*. Springer, 2000, pp. 121–132.

[4] A. Moraglio, K. Krawiec, and C. G. Johnson, "Geometric semantic genetic programming," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2012, pp. 21–31.

[5] C. Ferreira, "Algorithm for solving gene expression programming: a new adaptive problems," *Complex Systems*, vol. 13, no. 2, pp. 87–129, 2001.

[6] J. Zhong, L. Feng, and Y. Ong, "Gene expression programming: A survey [review article]," *IEEE Computational Intelligence Magazine*, vol. 12, no. 3, pp. 54–72, Aug 2017.

[7] J. Zhong, Y.-S. Ong, and W. Cai, "Self-learning gene expression programming," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 65–80, 2016.

[8] J. Zhong, L. Feng, W. Cai, and Y. Ong, "Multifactorial genetic programming for symbolic regression problems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–14, 2018.

[9] M. Oussaidene, B. Chopard, O. V. Pictet, and M. Tomassini, "Parallel genetic programming: An application to trading models evolution," in *Proceedings of the 1st annual conference on genetic programming*. MIT Press, 1996, pp. 357–362.

[10] W. B. Langdon, "Large scale bioinformatics data mining with parallel genetic programming on graphics processing units," in *Parallel and distributed computational intelligence*. Springer, 2010, pp. 113–141.

[11] A. Cano, J. M. Luna, and S. Ventura, "High performance evaluation of evolutionary-mined association rules on gpus," *The Journal of Supercomputing*, vol. 66, no. 3, pp. 1438–1461, 2013.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[13] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008, vol. 10.

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[16] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[17] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[18] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[19] M. Tomassini, L. Vanneschi, L. Bucher, and F. Fernández, "An mpi-based tool for distributed genetic programming," in *Proceedings IEEE International Conference on Cluster Computing. CLUSTER 2000*. IEEE, 2000, pp. 209–216.

[20] X. Du, L. Ding, and L. Jia, "Asynchronous distributed parallel gene expression programming based on estimation of distribution algorithm," in *Natural Computation, 2008. ICNC'08. Fourth International Conference on*, vol. 1. IEEE, 2008, pp. 433–437.

[21] A. Salhi, H. Glaser, and D. D. Roure, "Parallel implementation of a genetic-programming based tool for symbolic regression," *Information Processing Letters*, vol. 66, no. 6, pp. 299–307, 1998.

[22] K. Stoffel and L. Spector, "High-performance, parallel, stack-based genetic programming," in *Proceedings of the 1st annual conference on genetic programming*. MIT Press, 1996, pp. 224–229.

[23] S. Harding and W. Banzhaf, "Fast genetic programming and artificial developmental systems on gpus," in *High Performance Computing Systems and Applications, 2007. HPCS 2007. 21st International Symposium on*. IEEE, 2007, pp. 2–2.

[24] S. L. Harding and W. Banzhaf, "Distributed genetic programming on gpus using cuda," in *Workshop on Parallel Architectures and Bioinspired Algorithms*, 2009, pp. 1–10.

[25] D. Robilliard, V. Marion-Poty, and C. Fonlupt, "Population parallel gp on the g80 gpu," in *European Conference on Genetic Programming*. Springer, 2008, pp. 98–109.

[26] S. Shao, X. Liu, M. Zhou, J. Zhan, X. Liu, Y. Chu, and H. Chen, "A gpu-based implementation of an enhanced gep algorithm," in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. ACM, 2012, pp. 999–1006.

[27] D. M. Chitty, "Improving the performance of gpu-based genetic programming through exploitation of on-chip memory," *Soft Computing*, vol. 20, no. 2, pp. 661–680, 2016.

[28] ——, "Faster gpu-based genetic programming using a two-dimensional stack," *Soft Computing*, vol. 21, no. 14, pp. 1–20, 2016.

[29] Z. Huang, J. Zhong, L. Feng, Y. Mei, and W. Cai, "A fast parallel genetic programming framework with adaptively weighted primitives for symbolic regression," *Soft Computing*, Sep 2019. [Online]. Available: https://doi.org/10.1007/s00500-019-04379-4

[30] X. Du, Y. Ni, Z. Yao, R. Xiao, and D. Xie, "High performance parallel evolutionary algorithm model based on mapreduce framework," *International Journal of Computer Applications in Technology*, vol. 46, no. 3, pp. 290–295, 2013.

[31] L. Xu, Y. Huang, X. Shen, and Y. Liu, "Parallelizing gene expression programming algorithm in enabling large-scale classification," *Scientific Programming*, vol. 2017, 2017.

[32] M. Khan, Z. Huang, M. Li, G. A. Taylor, and M. Khan, "Optimizing hadoop parameter settings with gene expression programming guided pso," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 3, p. e3786, 2017.

[33] F. Fernández, J. M. Sánchez, M. Tomassini, and J. A. Gómez, "A parallel genetic programming tool based on pvm," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 1999, pp. 241–248.

[34] J. Wu, T. Li, B. Fang, Y. Jiang, Z. Li, and Y. Liu, "Parallel niche gene expression programming based on general multi-core processor," in *Artificial Intelligence and Computational Intelligence (AICI), 2010 International Conference on*, vol. 3. IEEE, 2010, pp. 75–79.

[35] W. D. Gropp, W. Gropp, E. Lusk, A. Skjellum, and A. D. F. E. E. Lusk, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[36] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[37] M. Zaharia, *An Architecture for Fast and General Data Processing on Large Clusters*, 2016.

[38] Z. Huang, J. Zhong, L. Feng, Y. Mei, and W. Cai, "A fast parallel genetic programming framework with adaptively weighted primitives for symbolic regression," *Soft Computing*, pp. 1–17, 2019.