Scaling Genetic Programming for Data Classification using MapReduce Methodology

Nailah Al-Madi and Simone A. Ludwig
Department of Computer Science
North Dakota State University
Fargo, ND, USA
{nailah.almadi,simone.ludwig}@ndsu.edu

***Abstract*—Genetic Programming (GP) is an optimization method that has proved to achieve good results. It solves problems by generating programs and applying natural operations on these programs until a good solution is found. GP has been used to solve many classifications problems, however, its drawback is the long execution time. When GP is applied on the classification task, the execution time proportionally increases with the dataset size. Therefore, to manage the long execution time, the GP algorithm is parallelized in order to speed up the classification process. Our GP is implemented based on the MapReduce methodology (abbreviated as MRGP), in order to benefit from the MapReduce concept in terms of fault tolerance, load balancing, and data locality. MRGP does not only accelerate the execution time of GP for large datasets, it also provides the ability to use large population sizes, thus finding the best result in fewer numbers of generations. MRGP is evaluated using different population sizes ranging from 1,000 to 100,000 measuring the accuracy, scalability, and speedup.**

Keywords—Evolutionary computation, genetic programming, data classification, Parallel Processing, MapReduce, Hadoop

I. Introduction

Optimization is the process of finding the best solution from all feasible solutions. One way to solve optimization problems is through evolutionary computation algorithms, which are inspired by the processes of biological evolution. Evolutionary algorithms start by creating a population of solutions and apply natural operations to the individuals in order to produce a new population. This iterative process continues until a good solution is found or a predefined number of generations is reached. Natural operations include the selection process which implies the survival of the fittest concept; the best candidates are chosen to undergo recombination and mutation. Selecting two candidates and recombining them, results in one or more new candidates; whereas mutation is only applied to one candidate and results in one new candidate. The fitness of the individuals is calculated using a fitness measure which is based on the problem to be solved.

Genetic Programming (GP) [18] is one of the evolutionary computation algorithms that proved its effectiveness since it automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance. GP provides solutions to problems by creating computer programs. The same process of evolutionary algorithms is adopted by GP whereas the individuals are computer programs, and the crossover and mutation processes are applied to these programs to exchange or modify parts of the programs.

Classification is one of the important data analysis tasks, which is used to find a model from previous data in order to predict the class of new data. Classification model effectiveness is measured by calculating the accuracy outlining how accurate the model can predict a new data class. GP can be used to solve classification problems and produce a classification model. This classification model is a computer program that takes unseen data values and predicts the class label. The goal of GP as an optimization process is to maximize the number of correctly classified records (accuracy).

GP proved its effectiveness in many classification problems [1, 2, 3], however, it suffers from the long run time since it is an iterative process (large number of generations are needed to find a good solution), in addition to the grammar used to define chromosome representations, and the consequent large search space. Moreover, the execution time of GP to build the classification model depends on the size of the dataset, large datasets require long execution time. Moreover, as mentioned in [4], many typical GP problems do not have large sets of fitness cases for two reasons: first, evaluation has always been considered computationally expensive, and secondly, it is very difficult to evolve solutions to harder problems. For all these reasons, the acceleration of the GP process is needed, which can be achieved by the parallelization of the GP process.

All nature-inspired algorithms are intrinsically parallel and distributed. Three main factors allow to easily parallelize Evolutionary Computation (EC) algorithms [5]: 1) Individual programs are evaluated using multiple independent fitness cases; 2) Populations consist of individuals which could be evaluated on independent hardware in parallel; 3) Independent runs can be executed simultaneously on different hardware.

Based on these factors three parallelization strategies (levels) can be implemented for EC algorithms:

- Fitness evaluation level: individuals are distributed among other nodes to compute the fitness values of the individuals. Based on the fact that evaluating a fitness function for every individual is the most costly operation of the algorithms.
- Population level: divides the population into several sub-

populations then executes each sub-population.

- Individual level: each individual is placed on a grid and all algorithm operations are performed in parallel, evaluating simultaneously the fitness and applying local selection and algorithm-specific operations to a small neighboring group.

Algorithm parallelism can be done using different methodologies like Message Passing Interface (MPI) [6], or MapReduce [7] and many others. MapReduce is a prominent parallel data processing tool which has been gaining significant interest from both industry and academia. It is a new methodology proposed by Google in 2004, which is a programming model and an associated implementation for processing large data sets [7]. MapReduce enables users to easily develop large-scale distributed applications by supporting fault tolerance, load balancing, and data locality. In MapReduce, the user expresses the computation as two functions: Map and Reduce where the inputs and outputs are represented as a set of key/value pairs. Map takes an input pair and produces a set of intermediate key/value pairs. The MapReduce framework then groups all intermediate values associated with the same key and passes them to the Reduce function. The Reduce function accepts an intermediate key and a set of values for that key, and merges these values together to form a possibly smaller set of values. The intermediate values are supplied to the user's Reduce function via an iterator. This allows the model to handle lists of values that are too large to fit in main memory.

This MapReduce model is used for many evolutionary computation algorithms such as Genetic Algorithms (GA) [8, 9], Particle Swarm Optimization (PSO) [10], Ant Colony Optimization (ACO) [11] and many others, which emphasize its effectiveness and efficiency to be used for big data calculations. To our knowledge, GP was not implemented using the MapReduce model until now. In this paper, we parallelize the GP process using MapReduce methodology (MRGP) in order to accelerate the execution time of GP, by using large population sizes, hence, less number of generations may be needed to find good results. The main contributions of this paper are the following: it demonstrates the transformation of GP into the Map and Reduce primitives, and confirms its ability to use large population sizes in order to find a good solution in shorter execution times (tackling GP's drawback of long execution time), and supports the scalability property to solve large problem sizes such as used for the classification of big data.

The organization of this paper is as follows: Section II presents some related works. In Section III, we describe the GP process and MapReduce model, and introduce the proposed MRGP approach. Then, we report on our experiments, and show the results in Section IV. We conclude this paper in Section V.

II. Related Work

GP has been used for many problems, such as classification, regression, and many other optimization problems. All of them were implemented as sequential versions, and based on our knowledge, GP was parallelized only using two approaches: Graphics Processing Unit (GPU) [12], and OpenCL [13]. In [12], the authors used GPUs to accelerate the GP by running the GP program on several GPUs in parallel, thereby demonstrating the benefit of GPUs to accelerate the GP approach. The authors showed that it is possible to get speed increases of several hundred times compared to a typical CPU implementation. In [13] the authors presented a detailed high-performance GP implementation in OpenCL for accelerated tree evaluation on the CPU and GPU architectures. OpenCl is an open standard [14] for uniform and portable parallel programming across heterogeneous computing platforms. They concluded that GPU is considerably faster and more power efficient than CPU.

Since GA is the closest EC algorithm to GP, therefore, we will review some parallel GA approaches first. Different parallel approaches have been used to parallelize the GA process, and the first attempt of implementing GA with MapReduce was done as described in [9], where the authors presented an extension to the MapReduce model featuring a hierarchical reduction phase (MRPGA: MapReduce for Parallel GAs), which can automatically parallelize GAs. The authors described the design and implementation of the extended MapReduce model on a .NET-based enterprise Grid system, and claim that GAs cannot be directly expressed by MapReduce, therefore, they offer their own implementation to extend the model to MapReduce. Several shortcomings of this MRPGA approach are pointed out in [8], however, showing that GA can be implemented with MapReduce without the need of modifying the structure of the MapReduce concept. In [8], the authors described the algorithm design and implementation of GAs on Hadoop [20] and investigated the convergence and scalability of the implementation on the BitCounting problem, where the results showed that their implementation converged after 220 generations, taking an average of 149 seconds per generation, and scaling well up to problems with 10^5 variables. In [15], the paper showed how GAs can be modeled with the MapReduce model. The authors describe the algorithm design and implementation of simple and compact GAs on Hadoop.

A practical application of GA modeled with the MapReduce was proposed in [16]. The authors implemented GA for the job shop scheduling problems using MapReduce, running experiments with various population sizes (i.e., up to 10^7), and on clusters of various sizes. Moreover, a parallel GA for the automatic generation of JUnit test suites was proposed in [17]. The proposed solution is based on Hadoop MapReduce since it is well supported on cloud platforms and on graphic cards, thus, being an ideal candidate for high scalable parallelization of GAs. Although related work found in the literature have shown the remarkable power of GPUs in speeding up the execution of GP using different frameworks, so far no one has implemented and evaluated parallel GP using the MapReduce methodology supporting the properties of fault tolerance, load balancing, and data locality.

In this paper, we are proposing a MapReduce GP approach (MRGP), which transforms the GP implementation into a parallel model using MapReduce methodology. MRGP aims to speed up the execution time of GP by providing the ability to use large population sizes in order to find a good solution in early generations, and to support the ability of GP to solve the classification of large scale data problems.

III. Proposed Approach

Given that our proposed approach is based on GP as well as MapReduce methodology, we first briefly introduce GP and MapReduce before outlining the details of our proposed MRGP algorithm.

A. Genetic Programming

GP [18] is an evolutionary computation technique that solves optimization problems by offering a solution through the evolution of computer programs by methods of natural selection. GP is distinguished from other evolutionary computation techniques in that it automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance [19]. Each program in GP is composed of mathematical and logical functions ($+$, $-$, $\times$, $/$, if, $lessThan$, etc.) and terminals (variables or constants), and is represented as a tree. Before starting the run of GP, several settings have to be defined, such as the functions to be used in the programs, the fitness function needed to evaluate the goodness of the program, and other settings like the crossover and mutation probabilities. The GP process starts by generating an initial population of programs. In the second step, GP evaluates each program and calculates its fitness by executing it and using the defined fitness function. After that, natural operations are applied by first selecting programs, then performing crossover and mutation to generate new programs. This process is repeated until the new population has the same size as the initial population, and this new population is then used in the next generation. The evolution of generations continues until the termination criterion (maximum number of generation is reached, or a pre-specified accuracy) is satisfied. The result of the run is the program with the best fitness value found during the whole evolution.

B. MapReduce Methodology

MapReduce is a highly scalable model and can be used across many computer nodes, and is mostly applicable for data intensive applications and when there are limitations on multiprocessing and large shared-memory machines.

The surpass of MapReduce moves the processing to the data not vice versa, and processes data sequentially to avoid random access that requires expensive seeks and disk throughput. MapReduce solves the problem by formulating it into two main operations, Map and Reduce. Both Map and Reduce operations take inputs and produce outputs in the form of <key, value>. The Map operation goes over a large number of records and extracts interesting information from each record, and then all values with the same key are sent to the same Reduce operation. However, the Reduce operation aggregates intermediate results, generated from the Map function that has the same key, then generates the final results.

A well-known and commonly used implementation of MapReduce is Apache Hadoop [20]. It is an open source software framework that supports data-intensive distributed applications licensed under Apache. It enables applications to work with petabytes of data using thousands of independent processors. One of the main components of Hadoop is the storage component, Hadoop Distributed File System (HDFS). HDFS provides high-throughput access to the data and maintains fault tolerance by creating multiple replicas of the target data blocks. HDFS and MapReduce work together to support the ability of moving computation to the data, and not vice versa.

C. Proposed MRGP Approach

This paper proposes a MapReduce GP approach (MRGP), which transforms the GP implementation from sequential into a parallel model using MapReduce. The goal is to accelerate the execution time of GP, providing the ability to use larger population sizes in order to find a good solution in early generations, and to give GP the ability to be used in the classification of large scale data.

GP's long execution time for solving classification problems arise from the fitness calculations of each program in the population. Each program is executed on every record of the data then the number of incorrectly classified records is counted. This number is then used as the fitness of that program. Hence, for example, if we have a population size of 100 programs, and a dataset of 1,000 records, then 100×1000=10^5 computations are executed for each generation. Therefore, MRGP distributes and parallelizes these computations to accelerate the fitness ranking process.

MRGP parallelizes the GP process based on the fitness evaluation parallelization level by transforming the GP process into the Map and Reduce operations. The Map is responsible for the fitness evaluation, while the Reduce is responsible for performing the remaining GP process (selection, crossover, and mutation).

MRGP works as follows: at the beginning, the GP algorithm generates the initial population based on the given settings. Then, this population is written to the HDFS, thus, the mappers can access it. After that, the population is distributed on the mappers based on the number of mappers and the size of the population. Each mapper receives its part of the population and reads the programs from that part in the form of <Key, Value>, where the Key is the program id, and the Value is the program information. Then, it calculates the fitness of that program based on the dataset, and updates the program information to include the fitness, and then emits this program. The program is emitted also in the form of <Key, Value>, where the Key is the population number and the Value is the program information.

Using the population number as the output key of the mapper implies that all programs are going to the same reducer. The reducer recollects the population and then performs all

GP steps of selection, crossover and mutation to produce the new population. The Reducer then emits this population in the form of <Key, Value>, where the Key is the program id, and the Value is the program information, which is used in the next mappers. Each Map and Reduce job is considered as a generation of GP. The jobs continue iteratively until the GP reaches its stopping criteria (either the maximum number of generations or a good solution is found). The result of the GP run from all generations is the program that has the best fitness value; therefore, we need to save this program throughout the generations. This process is done by the reducer also, where the best program found so far is saved to a file on the HDFS. The reducer updates this file whenever a better program than the one previously saved is found by collecting the programs from the mappers and comparing the best program saved in the file with the best program given by the mappers. The detailed process of MRGP is shown in Figure 1, where the structure displays that MRGP has several mappers to perform the fitness calculation for the individuals in the population, and one reducer to combine the programs and proceed with the GP iteration process.

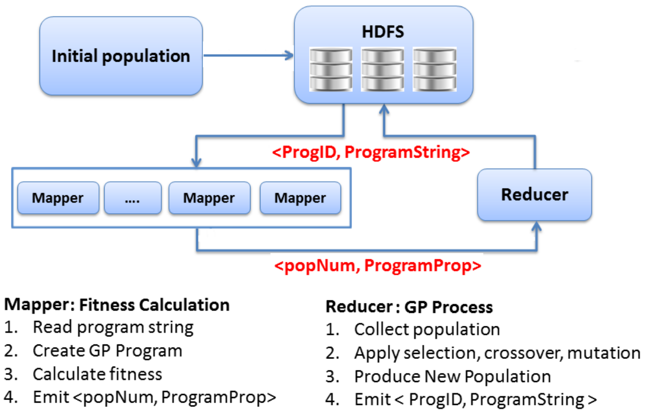


Fig. 1. MRGP Process Architecture

IV. Experiments and Results

In this section, we describe the experiments done to evaluate the MRGP algorithm. First, the GP settings, execution environment, and information of the datasets used for the classification problem are given. Then, the experimental results are provided and discussed.

A. Environment

To evaluate the proposed MRGP approach, we ran the experiments on the NDSU Hadoop cluster. The NDSU Hadoop cluster consists of 18 nodes, containing 6GB of RAM, 4 Intel cores (2.67 GHz each) with HDFS of 2.86 TB aggregated capacity. For the MapReduce framework, Hadoop version 0.20 was used.

Experiments were performed using the Java Genetic Algorithms Package (JGAP) [21] with the following settings:

- Population size = 1,000, 5,000, 10,000, 50,000.
- Number of generations = 100
- Crossover probability = 0.5
- Mutation probability = 0.1
- Maximum initial depth = 8
- Maximum crossover depth = 8
- Function probability = 0.7
- Dynamize arity probability = 0.05
- New chromosome percentage = 0.2

The functions used are $+, -, *, /, Exp, Pow$, and Log. Given the stochastic nature of GP, ten independent runs were performed.

The experiments are applied on six datasets [22], each was partitioned into two parts; 66% of the dataset for training the GP and building the classifier, and the remaining 34% for testing the classifier. All details including the number of features and records of these datasets are shown in Table I.

TABLE I
Datasets

	Dataset	Classes	Features	Records
D1	Iono	2	34 (14)	351
D2	Vertebral-2C	2	6 (6)	310
D3	Blood	2	5 (4)	748
D4	Balance	3	4 (4)	625
D5	Vertebral-3C	3	6 (6)	310
D6	CTG-NSP	3	22 (7)	2126

A pre-processing stage was performed on the datasets, where a feature selection process using the WEKA software [23] was performed to choose the best features of the datasets (a supervised attribute filter that allows various search and evaluation methods to be combined [23]). The resulting number of features is shown in the brackets in Table I. The experiments are applied on two types of datasets (binary and multi-class). The benchmark classification problems are:

- Ionosphere: Classification of radar returns from the ionosphere. The class label is either “Good”, which means that radar returns are those showing evidence of some type of structure in the ionosphere, or “Bad”, which means the returns are those that do not (their signals pass through the ionosphere).
- Vertebral Column: Data set containing values for six biomechanical features used to classify orthopaedic patients into 3 classes (normal, disk hernia or spondilolysthesis), or 2 classes (normal or abnormal).
- Blood Transfusion Service Center: Data taken from the Blood Transfusion Service Center in the Hsin-Chu City of Taiwan. Class labels represent whether or not the person donated blood in March 2007.
- Balance Scale: generated to model psychological experimental results. Each example is classified as having the balance scale tip to the right, tip to the left, or be balanced.
- Cardiotocography: fetal cardiotocograms (CTGs) were automatically processed and the respective diagnostic features measured. Classification is categorized with respect to a fetal state (N, S, P).

B. Results

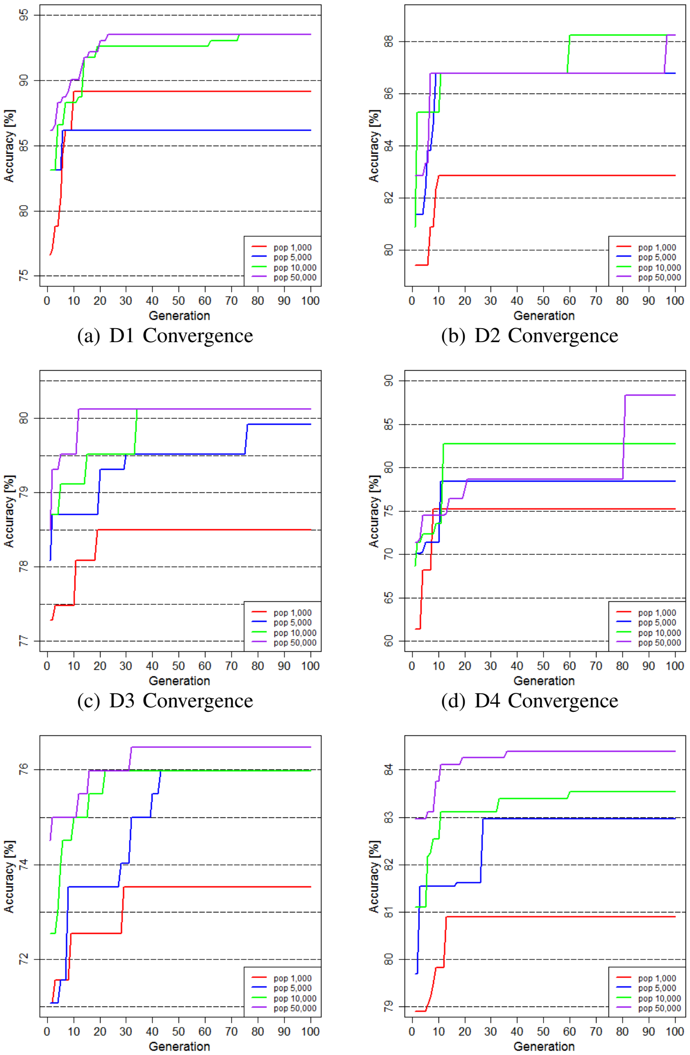
To evaluate MRGP, the experiments measure the speed of convergence and the impact of population sizes, testing accuracy, average time for mapper, speedup and scalability.

The speed of convergence of MRGP per generation, i.e., highest accuracy that can be achieved for each dataset is measured. In addition, a comparison of the impact of the population size on the MRGP convergence is performed. The experimental results for different populations sizes (1,000, 5,000, 10,000 and 50,000) are displayed in Figure 2, where it is shown that larger population sizes yield better solutions, which is intuitive since more function evaluations are performed. For example, for D1, the population size 1,000, starts with an accuracy of 76.6%, while population 5,000 starts with 83.12%, and population 50,000 starts with 86.14%. Moreover, at the end of 100 generations, population 1,000's best accuracy is 89.18%, while population 50,000's best accuracy is 93.51%. On the other hand, the accuracy of 89.18% is achieved with population size 50,000 in the 9^{th} generation unlike the 100 generations needed for population size 1,000. In general, the same observation can be obtained with the other datasets. Therefore, larger population sizes achieve better results when the same number of generation is used. This can be explained by having larger population sizes implies that more individuals search the solution space, and therefore, the possibility to find better results is higher than using smaller population sizes.

After the GP has trained the classifier by finding the best program using the training dataset, this classifier is then tested using the testing dataset by measuring its quality using the accuracy metric. The accuracy results of the MRGP classifier for the six datasets and by using different population sizes are shown in Table II. The tables illustrate the average accuracy of ten independent runs for each dataset. From the tables we can infer that for each dataset, the larger the population size the better the accuracy. In general, the difference between the accuracy of using a population size of 1,000 and 5,000 is somehow smaller, while using larger population sizes such as 10,000 or 50,000 increases the difference. For example, for D1, the accuracy for population size 1,000 is 87.66%, while for population size of 5,000 it is 90.25% (with +2.59% difference), nevertheless, the accuracy of population 50,000 is 92.91% (the difference compared to population size 1,000 is +5.25%). The same trend is seen for D4, where pop1000's accuracy is 71.22%, while pop5000's is 73.89% (with difference +2.67%), while pop10,000's accuracy is 75.30 (difference equal to +4.08%), moreover, pop50,000's accuracy is 81.59% (with the highest difference +10.37%).

The detailed accuracy results for the six datasets using different population sizes are shown by the box plots in Figure 3, where the circle represents the average of 10 runs, the solid bar depicts the median, the lower end of the dashed line represents the minimum accuracy observed by the GP for a given population size, and the upper end of the dashed line depicts the maximum accuracy value. It can be observed from the figure that using a population size of 50,000 leads to better average accuracy. Moreover, it has smaller box sizes (especially for D1, D2, D3, and D4), which means that the accuracy values are within a small range, hence it is more reliable than using a population size of 50,000.

However, since the scaling of the population size by keeping the number of generations fixed is not a fair comparison, we have investigated MRGP in terms of numbers of fitness evaluations. In particular, we kept the number of fitness evaluations fixed to 100,000, using different populations sizes of 1,000, 2,500, 5,000, 7,500, and 10,000 and using different number



(a) D1 Convergence (b) D2 Convergence

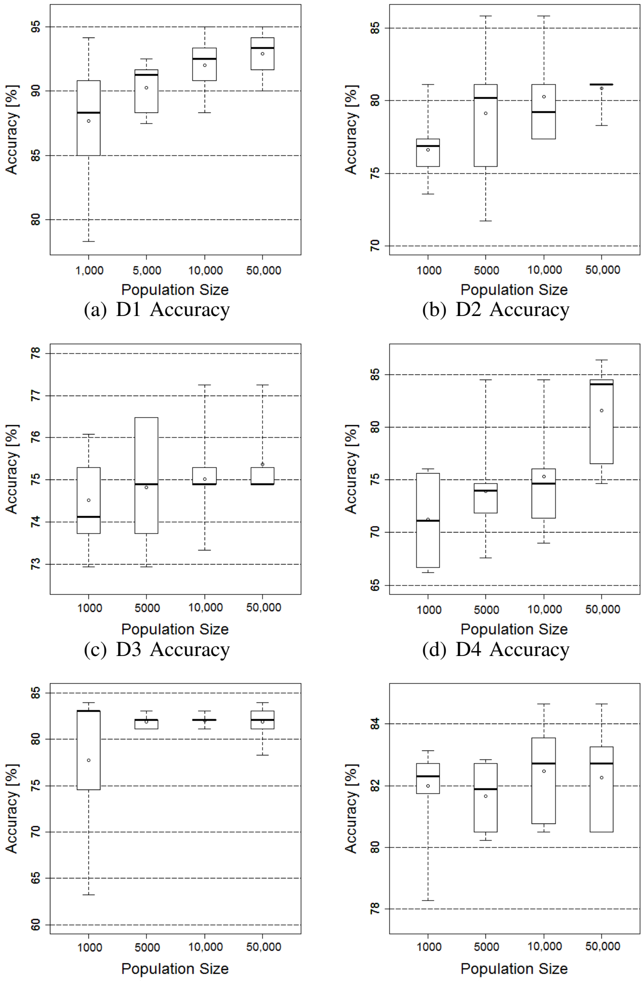
(c) D3 Convergence (d) D4 Convergence

(e) D5 Convergence (f) D6 Convergence

Fig. 2. Convergence speed with different population sizes

TABLE II
ACCURACY RESULTS - TESTING PHASE

Dataset	Pop 1000	Pop 5000	Pop 10000	Pop 50000
D1	87.66	90.25	91.998	92.91
D2	76.60	79.15	80.28	81.13
D3	74.51	74.82	75.01	75.36
D4	71.22	73.89	75.30	81.59
D5	77.73	83.20	78.87	81.88
D6	81.99	81.66	82.46	82.26



(a) D1 Accuracy

(b) D2 Accuracy

(c) D3 Accuracy

(d) D4 Accuracy

(e) D5 Accuracy

(f) D6 Accuracy

Fig. 3. Accuracy results - Testing Phase

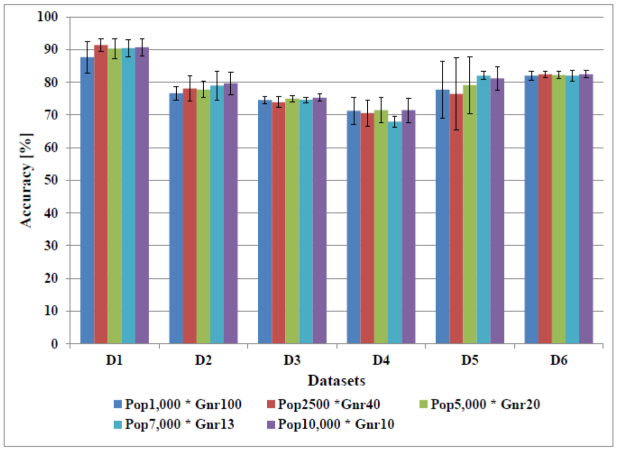


Fig. 4. Accuracy Results based on Fitness Evaluations (100,000)

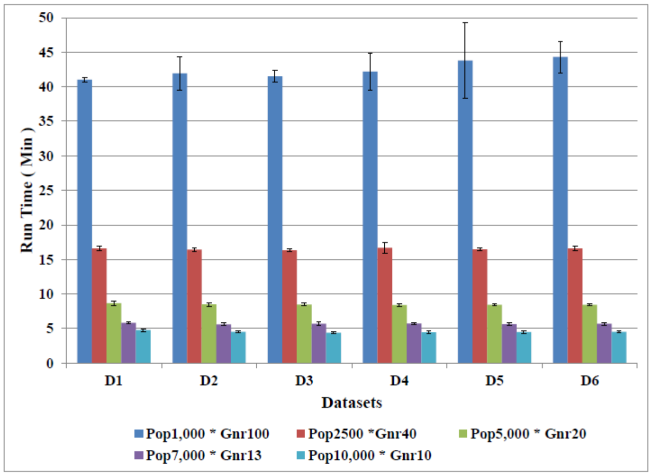


Fig. 5. Time Results based on Fitness Evaluations (100,000)

of generations with 100, 40, 20, 13, and 10, respectively. Figure 4 shows the accuracy results whereby for all columns belonging to a particular dataset all accuracy results show no significant difference as can be seen by the standard deviation bars. However, what we can clearly see in Figure 5 is that the run time for larger population sizes significantly reduces as seen by all datasets. The reason for this is using larger population sizes need smaller numbers of generations, which in turn means less MapReduce jobs.

The parallelization of MRGP provides the ability to use large population sizes such as 50,000 and obtain similar high accuracy values. Therefore, to evaluate the parallelization aspect of MRGP, first we calculated the average time needed for different numbers of mappers using the same population size of 100,000 programs and applied it on dataset D6 since it is the largest one. A large population and large dataset were chosen in order to demonstrate higher utilization rates of the MapReduce (Hadoop) framework, and to reduce the parallelization overhead such as starting MapReduce jobs, starting mappers and reducers operations.

Figure 6 displays the average time spent in a mapper, showing that using more mappers reduces the execution time needed in each mapper. This is because the population is divided into smaller parts based on the number of mappers, hence, using more mappers leads to the division of the population into smaller portions, and in turns leads to a faster average execution time in the mapper. However, when the number of mappers increases, the overhead of initializing the mappers also increases, therefore, in Figure 6 the average time per mapper does not reduce according to the number of mappers.

The scalability measures the impact of using different population sizes when the same number of mappers is used. Figure 7 shows the average time per mapper, for each population size, using the six datasets, and 10 mappers. It can be inferred that MRGP scales well between population sizes 1,000 to 10,000 on all the datasets. However, it increases drastically as the population size reaches 50,000. Therefore, a recommendation that can be made here is that using a population size between 1,000 to 10,000 and adding more mappers would enable MRGP to scale well for larger population sizes.

We note in this figure, that the average mapper running time when using larger population sizes does not increase by the

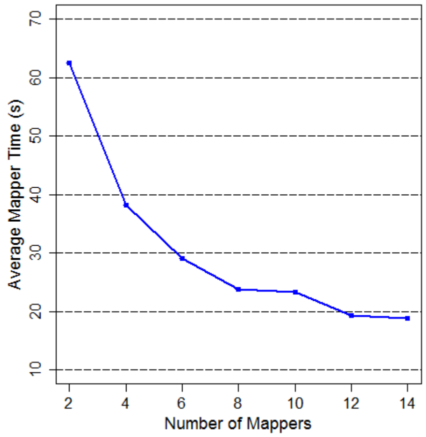


Fig. 6. Average time per Mapper

same ratio as the size of population, such as when using a 50 times larger population size (comparing 1,000 to 50,000), the running time increases approximately twice. The reason for this is that the MapReduce processes of copying and sorting of the larger populations causes extra overhead.

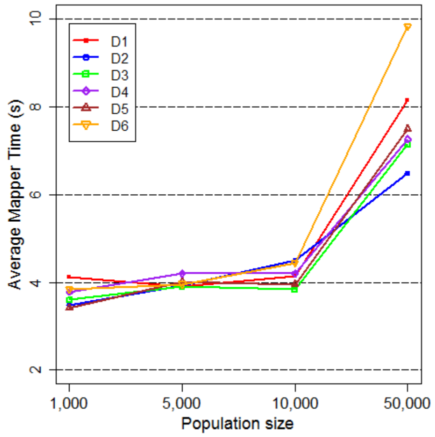


Fig. 7. Scaleup - Mapper Time

Figure 8 presents the speedup results of the MRGP algorithm compared to the optimal linear speedup. To measure the speedup, we fixed the population size to 100,000 by increasing the number of mappers by a certain ratio (factor of 2). Then, the speedup measure is calculated as:

$$Speedup = \frac{Time_2}{Time_n} \tag{1}$$

where $Time_2$ is the time using 2 mappers, $Time_n$ is the time using n mappers, where n= {2, 4, 6, 8, 10, 12, 14}. We started with 2 mappers and used the factor of 2, because running the MRGP using one mapper will not be efficient due to the high overhead of initializing the MapReduce framework compared to running the standard GP.

The figure illustrates that the speedup was very close to the linear speedup (optimal scaling) using 4, 6 mappers, but after that (with more mappers) it diverges from the linear speedup quite dramatically. This is because of the overhead of the MapReduce (Hadoop) framework, which results from starting the mappers and storing the outputs to the distributed file system, in addition to other management tasks. We have to note that the Hadoop framework has proved its efficiency of using very large input/data (which is in our MRGP the population size) and that MapReduce's speedup scales much better with larger population sizes. To proof this we ran MRGP with 200,000 or larger population sizes, however, the JGAP framework did not work due to garbage collection issues which need to be addressed.

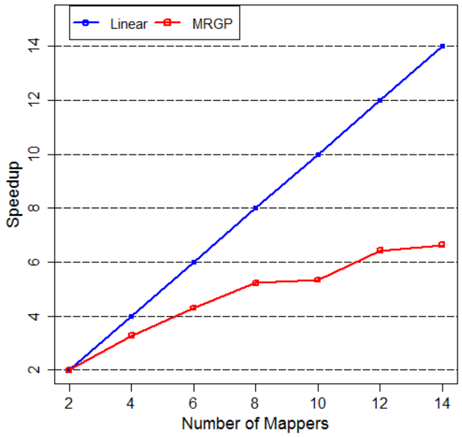


Fig. 8. MRGP Speedup

In summary, MRGP confirms the ability of implementing GP with the MapReduce framework by providing the ability of using large population sizes, such as 50,000, which is somehow impossible for the sequential GP. Also, using larger population sizes result in a faster convergence and a higher accuracy. Moreover, the structure of MRGP guarantees its scalability, and supports its speedup based on the provided resources. This supports the ability of applying MRGP to the classification of large scale data, which is not possible with the sequential version.

V. Conclusion

This paper proposed a parallelized version of the GP algorithm applied to the fitness evaluation level, referred to as MRGP. MRGP is implemented based on the MapReduce methodology which is a new methodology that manages fault tolerance, load balancing, and data locality. MRGP accelerates the execution time of GP by distributing the population to different mappers, which perform the fitness evaluation of the programs, and then the reducer combines these mappers and continues the GP process. MRGP also provides the ability to use large population sizes, thus, finding the best result in fewer numbers of generations. Experiments evaluated MRGP measuring the speed of convergence, accuracy, average time per mapper, scalability and speedup. The results confirm the goals of MRGP by accelerating the execution time, supporting the usage of large population sizes, and obtaining higher accuracy while maintaining the speedup and scalability properties.

Future work will involve investigating MRGP with much larger population sizes by solving the issues of the current GP implementation. Moreover, we plan to parallelize GP on the population level, and run experiments especially for the classification of large data sets.

Acknowledgment

The authors acknowledge the support of the NDSU Advance FORWARD program sponsored by NSF HRD-0811239 and ND EPSCoR through NSF grant EPS-0814442.

References

[1] S. M. Winkler, M. Affenzeller, and S. Wagner, "Using enhanced genetic programming techniques for evolving classifiers in the context of medical diagnosis - an empirical study," in *MedGEC 2006 GECCO Workshop on Medical Applications of Genetic and Evolutionary Computation*, S. L. Smith, S. Cagnoni, and J. van Hemert, Eds., Seattle, WA, USA, 8 July 2006.

[2] H. Jabeen and A. R. Baig, "Review of classification using genetic programming," *International Journal of Engineering Science and Technology*, vol. 2, no. 2, pp. 94–103, 2010.

[3] W. Smart and M. Zhang, "Using genetic programming for multiclass classification by simultaneously solving component binary classification problems," in *Proceedings of the 8th European conference on Genetic Programming*, ser. EuroGP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 227–239.

[4] 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*, May, pp. 2–2.

[5] W. Banzhaf, S. Harding, W. Langdon, and G. Wilson, "Accelerating genetic programming through graphics processing units," in *Genetic Programming Theory and Practice VI*, ser. Genetic and Evolutionary Computation. Springer US, 2009, pp. 1–19.

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*. MIT Press Cambridge, MA, USA, 1995.

[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," 2004, pp. 137–150.

[8] A. Verma, X. Llora, D. Goldberg, and R. Campbell, "Scaling genetic algorithms using mapreduce," in *Intelligent Systems Design and Applications, 2009. ISDA '09. Ninth International Conference on*, 30 2009-Dec. 2, pp. 13–18.

[9] C. Jin, C. Vecchiola, and R. Buyya, "MRPGA: An extension of mapreduce for parallelizing genetic algorithms," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, Dec., pp. 214–221.

[10] A. W. McNabb, C. K. Monson, and K. D. Seppi, "MRPSO: Mapreduce particle swarm optimization," in *GECCO*, 2007, p. 177.

[11] B. Wu, G. Wu, and M. Yang, "A mapreduce based ant colony optimization approach to combinatorial optimization problems," in *Natural Computation (ICNC), 2012 Eighth International Conference on*, May, pp. 728–732.

[12] S. Harding and W. Banzhaf, "Fast genetic programming on GPUs," in *Proceedings of the 10th European conference on Genetic programming*, ser. EuroGP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 90–101.

[13] D. A. Augusto and H. J. C. Barbosa, "Accelerated parallel genetic programming tree evaluation with OpenCL," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 86–100, Jan. 2013.

[14] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.2*, 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[15] D. E. G. Abhishek Vermay, Xavier Llora and R. H. Campbelly, "Scaling simple, compact and extended compact genetic algorithms using mapreduce by," 2010.

[16] D.-W. Huang and J. Lin, "Scaling populations of a genetic algorithm for job shop scheduling problems using mapreduce," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 30 2010-Dec. 3, pp. 780–785.

[17] L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro, "A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 785–793.

[18] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[19] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[20] (2011) Apache software foundation, hadoop mapreduce. [Online]. Available: http://hadoop.apache.org/mapreduce

[21] K. Meffert. (2012) et al. JGAP - Java Genetic Algorithms and Genetic Programming Package. [Online]. Available: http://jgap.sf.net

[22] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. [Online]. Available: http://archive.ics.uci.edu/ml

[23] I. Witten, E. Frank, and M. Hall, *Data Mining: Practical Machine Learning Tools And Techniques, 3rd Edition*. Morgan Kaufmann, 2011.