ISSN 2442-6571

A coarse-grained parallelization of genetic algorithms

Muhamad Radzi Rathomi [a,b,1], Reza Pulungan [a,2,*]

[a] Department of Computer Science and Electronics, Universitas Gadjah Mada, Yogyakarta, Indonesia
[b] Department of Informatics, Universitas Maritim Raja Ali Haji, Tanjungpinang, Indonesia
[1] radzi@umrah.ac.id; [2] pulungan@ugm.ac.id
* corresponding author

ARTICLE INFO

Article history
Received December 26, 2017
Revised February 15, 2018
Accepted February 22, 2018

Keywords
Genetic algorithms
Parallelization
Coarse-grained
MPI
GPU

ABSTRACT

Genetic algorithms are frequently used to solve optimization problems. However, the problems become increasingly complex and time consuming. One solution to speed up the genetic algorithm processing is to use parallelization. The proposed parallelization method is coarse-grained and employs two levels of parallelization: message passing with MPI and Single Instruction Multiple Threads with GPU. Experimental results show that the accuracy of the proposed approach is similar to the sequential genetic algorithm. Parallelization with coarse-grained method, however, can improve the processing and convergence speed of genetic algorithms.

This is an open access article under the CC–BY-SA license.

1. Introduction

Genetic algorithms (GA) are frequently used to solve scheduling, shortest paths, machine learning, and modeling problems. Genetic algorithms are basically a search and optimization technique. The working mechanism of GA is based on the principles of genetics and natural selection. On the other hands, the must be solved problems become more complex and bigger. Consequently, it takes much longer times and more advance objective functions to find optimal solutions. Until now, time complexity analysis is still developed to get a good performance estimation of genetic algorithms [1]. Since the complexity may certainly affect its processing time, a novel approach should be done to improve the GA performance.

This paper investigates a new method to increase the GA speed of genetic algorithms in finding the optimal solutions by parallelizing the processing of subpopulations. Splitting the population into subpopulations may prevent premature convergence since each subpopulation finds a different genetic combination. The proposed method employs two levels of parallelization: message passing and Single Instruction Multiple Threads (SIMT). On the first level, message passing is used because of its ability to connect more than one computer, and hence to provide, in principle, unlimited scalability. Previous researches, such as Liu and Wang [2], have shown the feasibility of this parallelization. On the second level, SIMT is used because it can generate a large number of threads and one individual can, therefore, be processed by one thread, as shown in Zhang and He [3]. Rapid advances in the technology of general purpose Graphics Processing Units (GPU) have allowed for massive numbers of threads in SIMT parallelization.

The proposed coarse-grained genetic algorithm consist of several GAs, which perform concurrent computations on different subpopulations. These genetic algorithms communicate with each other to exchange their best individual information; this technique is often called *migration*. According to Skolicki and De Jong [4], migration in a coarse-grained method a definite impact to data convergence.

This method is relatively easier to implement compared to other methods, and furthermore, it may result in shorter processing time in finding the optimal solutions.

Previous works on parallelization of genetic algorithms are plentiful. Li and Huang [5] conducted research on parallel genetic algorithms, analyzed occurred shortcomings and solved them with an adaptive migration strategy technique by adjusting the number of migrants using individual function similarity. Falahiazar et al. [6] made modifications on the parallel genetic algorithms for a better migration strategy. The technique used the max-min and a hill-climbing algorithm to structure the genetic algorithm migration. Hedar et al. [7] proposed gene matrix technique; a parallel genetic algorithm based on a distributed model to solve high-dimensional problems. Gene matrix of one subpopulation migrates to other subpopulations to create an individual diversity in each subpopulation.

Umbarkar et al. [8] solved a premature convergence problem in a genetic algorithm with two populations. Migration is done to perform r crossovers between two individuals in the two populations. Processing is done by using multithreaded parallelization. Liu and Wang [2] used a parallel genetic algorithm to solve non-deterministic polynomial-time hard problems; generalized assignment problems. The developed asynchronous migration strategy allowed efficient interaction between processes, improved communication, and reduced computing overlap significantly. On the other hands, other researches emphasized their work on the diversity of the population. They modified and classified individuals into three-category subpopulations: “good”, “moderate” and “bad”. The probability of crossovers and mutations adapts itself based on a self-adaptive formula [9].

Wang et al. [10] implemented a hybrid parallel genetic algorithm based on two layers of parallelism: process and thread. Their proposed coarse-grained method uses a hardware processing master-slave model by integrating message-passing parallelism using Message Passing Interface (MPI) and shared-memory parallelism using OpenMP [11]. Wahib et al. [12] parallelized genetic algorithms by using SIMT architecture with general purpose GPUs. They discussed the features of the GPU and the relevant issues when implementing parallel genetic algorithms. Johar et al. [13] conducted an analysis of genetic algorithms implemented in parallel both CPU and GPU using CUDA [14] architecture. The analysis was performed by comparing the operations performed in both implementations.

GPU Millan et al. [15] used to improve the computation time. Hou et al. [16] built a parallel genetic algorithm that makes use of two parallel systems: multi-core CPU and many-core GPU. Furthermore, Li et al. [17] also developed a parallel genetic algorithm that runs in GPU using island model. The last three studies, however, did not employ message passing interface to migrate the best individuals. This study combines both message passing and GPU to speed up parallel genetic algorithms. Network is required for migration; hence CPU is used, but not to process genetic algorithm operations. GPU instead processes each individual in the subpopulations as [15], [16] and [17]. GPU, however, spends a lot of time to move individuals from host to device and vice versa; and this problem affects the resulting parallel genetic algorithm. Asynchronous migration technique is proposed to handle this problem. The message passing and GPU are combined to build a massive and scalable machine and to speed up parallel genetic algorithms.

The rest of the paper is organized as follows: Section II provides a brief introduction to genetic algorithms and granularity in parallel computation. Section III describes the proposed parallelization methods. Section IV presents experimental results and their analysis, and Section V concludes the paper.

2. Method

2.1. Preliminaries

2.1.1. Simple Genetic Algorithm

The generic sequential genetic algorithm is shown in Fig. 1. In general, the most time-influential part of the generic algorithm is the population size, a number of individuals in one population. If the population size is large, then genetic algorithm takes a long time to complete its iterations up to the defined maximum generation. Therefore, partitioning the population into subpopulations and then do

parallel processing on them may speed up the computation time. Each subpopulation is processed on different computers connected by a network. Beside partitioning the population into subpopulations, on certain operations, each individual in a subpopulation can also be processed in parallel. Hence, parallelization can be applied to the generic sequential genetic algorithms in two levels. Section III will discuss this parallelization scheme in detail.

Algorithm 1 (The generic sequential genetic algorithm)
Phase 1: Initialization:
  Step 1.1: Set parameters: P_c, P_m, *popsize*, and *maxgen*.
  Step 1.2: Generate *popsize* individuals randomly to build the initial population and evaluate their fitness values. *gen* = 0.
Phase 2: Main Loop. Repeat the following steps until *gen* > *maxgen*:
  Step 2.1: Select *popsize* individuals from the current population using Roulette Wheel Selection to generate mating pool.
  Step 2.2: Repeat the following operations until a new population with *popsize* individuals is generated: Select two individuals from the mating pool randomly without replacement to perform crossover with probability P_c, and perform mutation for every gene of the offspring with probability P_m. Then insert the mutant into a new population.
  Step 2.3: Evaluate the fitness value for every new individual in the new population.
  Step 2.4: Replace the current population with the new population. *gen* = *gen* + 1.
Phase 3: Submit the final *popsize* individuals as the result of the genetic algorithm.
End

Fig. 1. The generic sequential genetic algorithm.

2.1.2. Granularity in Parallel Computing

Granularity in parallel computing is a qualitative measure of the ratio of computation to communication [18]. Periods of computation are typically separated from periods of communication by synchronization events. There are two types of parallel program designs based on granularity: coarse-grained and fine-grained. In the coarse-grained design, large amount of computation work is performed between communication events. On the other hand, fine-grained design performs relatively small amount of computation work between communication events.

Parallelization with fine-grained design facilitates load balancing nicely since many communication events between processing keep the balance of the workload. Parallelization with coarse-grained design, on the other hand, depends on size of the workload being processed. If the size of the workload being worked on by each process can be divided equally, then load balancing can be achieved. Parallelization with fine-grained design is, however, vulnerable to communication overhead, which results in the overall speed that cannot be increased; and sometimes even decreases. The main cause is data flooding in communication media. Thus, the parallelization with fine-grained design is not effective when used to process large amount of data with slow communication media. The parallelization of genetic algorithms in this paper will be designed with coarse-grained model.

2.2. Proposed Method

2.2.1. Parallelization of Genetic Algorithms

In this paper, we propose a method to develop parallel genetic algorithms with two levels of parallelization as shown in Fig. 2. This proposed method is an improvement from Ratomi [19]. The first level exploits the fact that the population can be partitioned into loosely dependent subpopulations, while the second level exploits the fact that individuals in a subpopulation are independent of each other and moreover perform similar computation.

On the first level, the population is divided into subpopulations by the number of available computing nodes. This level uses of message-passing hardware with master-slave parallelism: node 0 (master) broadcasts the size of the subpopulation to all slave nodes. Each slave node generates its own

subpopulation. Subpopulations are processed using coarse-grained method: a slave node only establishes communication to exchange its best individuals, which is transferred through the network. At the end, the master node gathers all final subpopulations from all slave nodes and produces the final result. In this study, the first level is implemented using MPI. Fig. 3 shows the proposed generic parallel genetic algorithm. Each computing node runs the whole algorithm (from Phase 1 to the End) in parallel; this corresponds to "Level I" in Fig. 2.

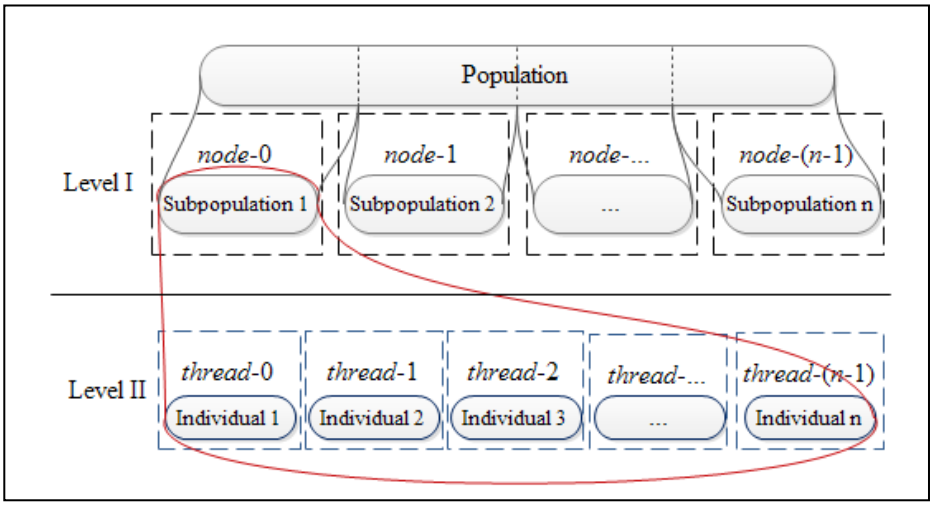


Fig. 2. The general parallelization design.

Algorithm 2 (The generic parallel genetic algorithm)
Phase 1: Initialization:
  Step 1.1: Receive *spopsize* from node 0.
  Step 1.2: Set parameters: P_c, P_m, and *maxgen*.
  Step 1.3: Generate *spopsize* individuals randomly to build the initial subpopulation and evaluate their fitness values and elitism. *gen* = 0.
Phase 2: Main Loop. Repeat the following steps until *gen* > *maxgen*:
  Step 2.1: Generate the mating pool from the current subpopulation using Tournament Selection.
  Step 2.2: Repeat the following operations until a new subpopulation with *spopsize* individuals is generated: Select two individuals from the mating pool randomly without replacement to perform crossover with probability P_c, and perform mutation for every gene of the offspring with probability P_m. Then insert the mutant into a new subpopulation.
  Step 2.3: Evaluate the fitness value for every new individual in the new subpopulation.
  Step 2.4: Replace the current subpopulation with the new subpopulation. *gen* = *gen* + 1.
  Step 2.5: Perform elitism and migration.
Phase 3: Submit the final *spopsize* individuals to node 0.
End

Fig. 3. The generic parallel genetic algorithm.

On the second level, parallelization will be performed in the processing of each individual (*cf.* "Level II" in Fig. 2). However, not all of genetic operations is parallelized. Looping in each generation is performed sequentially. The parts that are parallelized are genetic algorithm operations that have no individual dependency, namely selection, crossover, mutation, individual evaluation, and updating the population's individuals. These correspond to Phase 2, Steps 2.1 until 2.4 in Fig. 3. Steps 2.1 until 2.4 in Phase 2 are all similar data updating operations that are performed lockstep for all individuals in the subpopulation. Hence, these steps are further parallelized as a thread in each computing node's GPU.

Since modern general-purpose GPUs have a large number of threads, this allows for speedy computation of a subpopulation having a large number of individuals.

Threads' IDs in the GPU are used as indices of the individuals. Hence, threads are generated as many as the number of individuals in the subpopulation. Unbiased tournament is used in the selection method because this selection method can be easily executed in parallel. Random permutation values are generated only at the beginning of the execution of algorithm, and each thread builds its own mating pool based on these random permutation values. In the crossover operation, each thread selects parents (namely two individuals) randomly from the mating pool, and then based on P_c, the thread performs crossover for these individuals. Afterward, each thread obtains offsprings produced by the crossover operation, and based on P_m, applies mutation operations on these offsprings. Subsequently, all mutated offsprings are simultaneously evaluated to obtain fitness values, and then individuals in the old subpopulation are simultaneously replaced by new individuals. After previous sequence of operations is completed, elitism operation is executed sequentially to obtain the best individual. The best individual obtained is then copied to host CPU. Subsequently, MPI sends this individual to other nodes . All of these steps are repeated until the predefined maximum generation is reached.

Memory allocation for data required in the genetic algorithm operations is carried out in the host as well as in the GPU device. Although genetic algorithm operations are only performed on the device, these data also need to be copied to the host so that the best individual can be delivered by MPI. At the end of the genetic algorithm operations, all individuals at each node can be combined.

2.2.2. Distribution of Population

The distribution of population also influences the speed of parallel genetic algorithm because the speed of algorithm depends on the speed of the computing node processing the most individuals. The population is distributed, as far as possible, in equal size in order to achieve load-balanced nodes, and the processing speed of each node is then relatively equal. The value distributed to each node is the size of the node's subpopulation that has been calculated in the master node. Each node then generates its own subpopulation. If the size of the population is less than the number of nodes, each node processes only one individual. If the size of the population is more than the number of nodes, population is divided equally by rounding it up

2.2.3. Individual Migration Strategy

In this research, the migration process is integrated into elitism process, which derives the best individual resulted from individual comparison in elitism memory. Individual migration among the slave nodes is performed in one direction with a ring topology. The best individual in slave node i is sent to slave node i+1, and so on until the last node sends its best individual to the first node. Two transfer modes are available during the migration process: *asynchronous* and *synchronous* modes. In synchronous mode, migration is carried out by directly sending out the best individual to another computing node. On the other hand, in the asynchronous method, migration is carried out by first copying it to a buffer. Individual's migration in the asynchronous mode is performed by a separate thread, and the individual that is sent and received is accessed through the buffer.

Fig. 4 depicts the working boundaries of threads in migration and elitism processes in asynchronous mode. Genetic algorithm operations and individual migration work concurrently via a global shared buffer that stores the best individuals. In order to avoid collisions in accessing the buffer, the buffer is built with a handler, namely a status flag. If a migration thread wishes to send an individual, then it first checks the status flag of the sender buffer; whether it is free or in use. If the status is in use, then the migration thread must wait until the buffer's status is free. Similar to elitism process in the genetic algorithm operation, if the corresponding thread wishes to copy the individual from the receiver buffer, then it must first check the status of the buffer whether it is free or in use. If the status is in use, then the thread must wait until the status of the receiver buffer is free.

There is no migration on the second level, because the threads in the GPU work with Single Instruction Multiple Threads model, and therefore there is no communication between the threads. Communication only occurs between the host CPU and device GPU, namely to copy the best individual

produced by elitism in the device memory to the elitism memory on the host, which then is migrated using MPI.

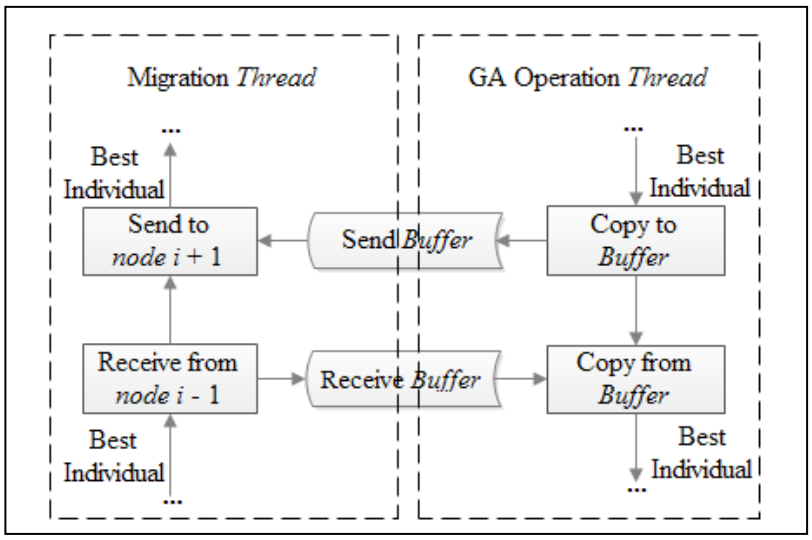


Fig. 4. The working boundaries of migration threads and genetic algorithm operations threads in asynchronous mode where communication proceeds via a shared buffer.

3. Results and Discussion

Experiments are conducted in computers with an Intel Core-i5 (4 CPU), 4 GB RAM, Linux operating system Ubuntu LTS 14:04. The computer is connected by a network using Ethernet LAN, each with an NVidia GeForce GT 420 GPU. In the experiments, the parameters of the genetic algorithm use crossover probability 0.9, mutation probability 0.05, and population size 100 individuals [20]. The number of copies for selection with unbiased tournament is 5 individuals. Experiments have been conducted in sequential and parallel implementations; for parallel one, we have used 5 and 10 computer nodes.

The datasets used in the experiments are related to Job Shop Scheduling Problem (JSSP) and are derived from OR-Library [21]. The library contains 82 cases, among which the experiments only make use of 10, namely ft10 (10x10), ft20 (20x5), la02 (10x5), 1a06 (15x5), la21 (15x10), la29 (20x10), la31 (30x10), la36 (15x15), orb01 (10x10), and orb04 (10x10). The numbers in brackets in the name of each case indicate the size of the case. For example, ft10 (10x10) is a problem named ft10 with 10 products, and to make one product, it must pass through 10 stages of work.

Three series of experiments are carried out in this study. The first series is intended to investigate the effect of parallelization on the accuracy of the results of the genetic algorithm. Two tools are used for this, namely the standard deviation (STD) and the Average Relative Percentage of Error (ARPE). Standard deviation is used to measure the variability of the objective values produced by the genetic algorithm, which in this experiment corresponds to the variability of the obtained *makespan* for each JSSP case. A *makespan* is the total length of the time required to complete all tasks in a JSSP case. Given a JSSP case study, it is run n times with different initializations for each parallelization methods. Each run produces a *makespan*. Based on the obtained *makespans*, their standard deviation is defined in (1).

$$STD = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}, \quad (1)$$

where x_i is the i-th *makespan* and $\bar{x}$ is the average value of all makespans. ARPE, on the other hand, is used to quantify the error obtained by the genetic algorithm searching in this study compared to the best makespan ever obtained in the previous researches for each JSSP case study. Let x_o be the *makespan* produced by running the current implementation, and x_b be the best *makespan* ever obtained by earlier researches, then ARPE is defined in (2).

$$ARPE = \frac{(x_o - x_b)/x_b}{n} \times 100. \quad (2)$$

The second and third series are conducted to investigate the effect of parallelization on the computation time and the transformation of the objective values, respectively. In the second series, the computation times of the implementation of the proposed parallelization method on many scenarios and case studies are compared to those of sequential implementation. In the third series, the transformation of the objective values in each generation of a particular JSSP case study for different parallelization methods is studied.

3.1. Accuracy

In the first series of experiments we investigate the effect of parallelization on the accuracy of the results of the genetic algorithm. Table 1 and Table 2 show the results of the first series of experiments.

Table 1. Standard Deviation Comparison

No.	Case	Sequential	Parallel sync. migration		Parallel async. migration	
			5 nodes	*10 nodes*	*5 nodes*	*10 nodes*
1	ft10 (10x10)	0.516	0.000	0.316	0.483	0.422
2	ft20 (20x5)	8.260	7.367	6.736	1.989	9.055
3	la02 (10x5)	0.000	9.698	9.698	15.830	18.490
4	1a06 (15x5)	0.000	0.000	0.000	0.000	0.000
5	la21 (15x10)	2.530	3.162	2.898	3.098	3.098
6	la29 (20x10)	0.000	0.000	0.000	0.000	0.000
7	la31 (30x10)	14.200	4.111	4.909	23.510	10.630
8	la36 (15x15)	1.581	1.265	1.449	0.949	1.449
9	orb01 (10x10)	0.000	0.000	0.000	0.000	0.000
10	orb04 (10x10)	0.000	0.000	0.000	0.000	0.000

Table 1 presents the standard deviations of the obtained makespans for all 10 JSSP cases under sequential and parallel implementations. For each JSSP case, 20 executions of the genetic algorithm have been run with different initialization values. Each of these executions produces a certain makespan as a solution. The standard deviations of all produced makespans for each JSSP case basically measure the variability of the results. Table 1 indicates that the standard deviations of the results of sequential and parallel implementations are close to each other and their difference is not significant. Even the highest standard deviation is still relatively small, namely for the la36 case, with parallel genetic algorithm using asynchronous migration model that shows the standard deviation value of 23.51. This suggests that genetic algorithms with sequential and parallel implementations converge to a solution. The only anomaly is case la02; but even in this case, the obtained makespans are not really divergent.

In this series of experiments, we also investigate the distance between the obtained makespans to makespans already known in previous researches; Table 2 shows the results. Columns with heading "***x***" contain the smallest makespan ever obtained for each case in various settings. Negative ARPE values occur in cases la06 and la31; this means that the makespans obtained in this research for these two cases are smaller than the makespans ever obtained in earlier researches. For case la31, the parallel implementation can even achieve smaller makespan than that of the standard genetic algorithm.

However, most of ARPE values in Table 2 are greater than 0 because they do not reach the makespans already known, unlike the Hybrid PGA (the fifth and sixth columns) that obtained ARPE values less than 1 [20]. This is because JSSP coding used in this research is a direct encoding. Direct encoding depends on the order of jobs in a given problem, so the set of possible combinations formed is small. In contrast, previous studies mostly used indirect encoding, whose set of possible combinations is more than that of direct one. Our purpose in using direct encoding is that we are more interested in observing the impact of migration in the parallel genetic algorithm on the structure of chromosomes permutation migrated. This is because direct encoding is highly dependent on the structure of chromosomes permutation. If the migration process damages the structure of chromosomes permutation, it will certainly result in larger ARPE values and the search results will not satisfy the given genetic algorithm

case. In general, the performance of the searching for optimal values using genetic algorithms that run in sequential and parallel is similar. Therefore, the proposed parallel genetic algorithm is not worse at finding the smallest *makespan* on the JSSP cases used in this study than the standard genetic algorithm.

Table 2. ARPE Comparison

Case	Known make-span	Sequential		Hybrid PGA		Parallel sync. migration				Parallel async. migration			
						5 nodes		*10 nodes*		*5 nodes*		*10 nodes*	
		x	*arpe*	*x*	*arpe*	*x*	*arpe*	*x*	*arpe*	*x*	*arpe*	*x*	*arpe*
ft10	930	1,133	2.183	930	0.00	1,134	2.194	1,133	2.183	1,133	2.183	1,133	2.183
ft20	1,165	1,495	2.833	1,165	0.00	1,495	2.833	1,495	2.833	1,495	2.833	1,495	2.833
la02	660	758	1.485	662	0.30	758	1.485	758	1.485	758	1.485	758	1.485
la06	945	927	-0.190	945	0.00	927	-0.190	927	-0.190	927	-0.190	927	-0.190
la21	1,046	1,229	1.750	1,048	0.19	1,229	1.750	1,229	1.750	1,229	1.750	1,229	1.750
la29	1,153	1,420	2.316	1,153	0.00	1,420	2.316	1,420	2.316	1,420	2.316	1,420	2.316
la31	1,888	1,829	-0.310	1,890	0.11	1,784	-0.550	1,784	-0.550	1,784	-0.550	1,784	-0.550
la36	1,268	1,491	1.759	1,272	0.32	1,491	1.760	1,491	1.759	1,491	1.759	1,491	1.759
orb01	1,059	1,279	2.080	1,055	0.28	1,279	2.080	1,279	2.080	1,279	2.080	1,279	2.080
orb04	1,005	1,037	0.318	600	0.50	1,037	0.318	1,037	0.318	1,037	0.318	1,037	0.318

3.2. Computation Times

In this series of experiments, we observe the computation times of the implementation of the proposed parallelization method on many scenarios and case studies and compare them to those of sequential implementation. For this purpose, the computation time of each scenario is obtained by running the corresponding implementation for 10,000 generations. Table 3 shows the resulting computation times. Note that the columns "***S(n)***" derived speedup when using *n* nodes.

Table 3. Computation Time Comparison

Case	Sequential	Hybrid PGA	Parallel sync. migration				Parallel async. migration			
			5 nodes	*S(5)*	*10 nodes*	*S(10)*	*5 nodes*	*S(5)*	*10 nodes*	*S(10)*
ft10	7.2074	11.20	9.4300	0.76	6.3377	1.14	9.8424	0.73	5.6769	1.27
ft20	7.1077	10.54	10.0318	0.71	6.6469	1.07	9.9139	0.72	5.9356	1.20
la02	3.6237	8.87	5.4549	0.66	3.8780	0.93	1.7461	2.08	1.6172	2.24
la06	5.2901	13.30	7.4948	0.71	5.2459	1.01	7.4625	0.71	4.6139	1.15
la21	10.5859	10.20	13.3214	0.79	9.0921	1.16	12.3286	0.86	8.1389	1.30
la29	14.5084	-	17.7133	0.82	11.9485	1.21	16.6437	0.87	10.8637	1.34
la31	21.8751	-	27.5067	0.80	18.1841	1.20	26.1276	0.84	16.8045	1.30
la36	16.1710	-	19.4508	0.83	13.0613	1.24	18.2899	0.88	12.0839	1.34
orb01	7.1614	-	9.2501	0.77	6.3660	1.12	8.4875	0.84	5.6760	1.26
orb04	7.2171	11.98	9.3505	0.77	6.3579	1.14	8.5109	0.85	5.6900	1.27
	Average	***6.609***	***Average***	***0.76***	***Average***	***1.14***	***Average***	***0.94***	***Average***	***1.37***

Table 3 shows that speed increase is obtained only when the genetic algorithm is executed in parallel using 10 nodes. Genetic algorithms running using 5 parallel nodes is not faster than genetic algorithms executed sequentially. The main cause of the lack of speed increase is due to the ineffective use of the GPU memory. The size of data processed in the high-speed memory is relatively small, since most of the data is processed using memory of slower access speed. Table 3 also shows that genetic algorithms running in parallel with asynchronous migration are faster than synchronous migration. The use of buffers and assignments of different threads to handle the migration process and the operation of genetic algorithms can reduce the waiting time required for sending and receiving the best individual through the network. Only one case is solved longer by the parallel implementation using 10 nodes (with synchronous migration) than the sequential one, namely case la02. This is because the time required to initialize MPI on the first test is longer than the time required to initialize MPI on subsequent tests.

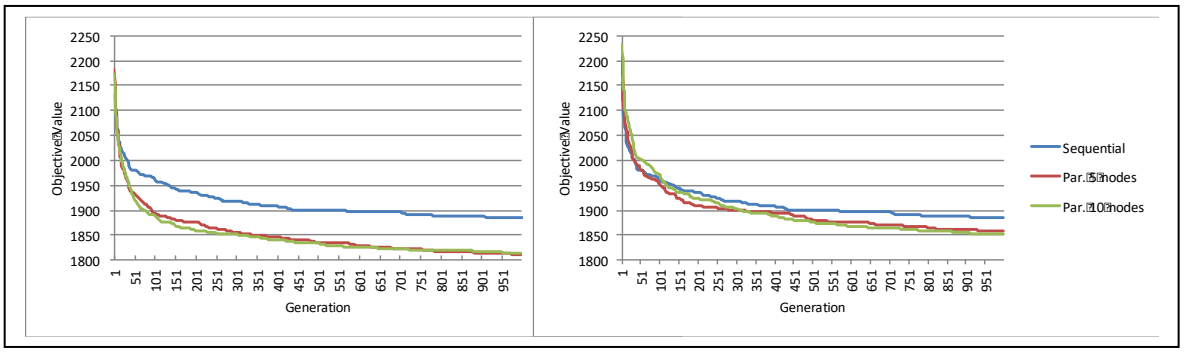
However, speedups obtained in this research are more stable than previous studies [20]. As can be observed in Table 3, using Hybrid PGA (the third column) only six cases end up with speedups in processing time, although they amounted to an average of 6.6. In this research, speedups are obtained

in all of the parallel, although most of them have speedup less than 2. This is because JSSP direct encoding used in this research are simpler than indirect encoding. Moreover, the migration technique of asynchronous model in this research is easily implemented than Liu and Wang [2]. This research uses only buffer to save the individual to be sent and received, and consequently the technique can reduce the migration processing time.

3.3. Transformation of Objective Values

In this series of experiments, we investigate the transformation of the objective values in each generation of a particular JSSP case study for different parallelization methods. Comparison of the objective values transformation is carried out to observe and compare the speed of convergence of the genetic algorithm, sequential and parallel. The selected JSSP case study is la31 and the transformation is observed for the first 1,000 generations in finding the smallest *makespan*. Fig. 5(a) shows the transformation of the objective values of the sequential and parallel implementations with synchronous migration. The parallel genetic algorithm, both with 5 nodes and 10 nodes, converges faster than the sequential one. At the first 100 generations, the objective values obtained by the parallel genetic algorithm decrease more steeply than the sequential genetic algorithm. The performance of parallel genetic algorithm that uses 5 nodes or 10 nodes with synchronous migration model is relatively the same.

Parallel genetic algorithm with asynchronous migration model also converges faster than the sequential one as shown by Fig. 5(b). However, the parallel genetic algorithm with asynchronous migration model takes longer to converge than with synchronous migration model. In the first 1,000 generations, the objective value obtained by the parallel genetic algorithm with asynchronous migration model is still higher, above 1,850, compared to the parallel genetic algorithm with synchronous migration model, which only reaches 1,810.



(a) Synchronous migration (b) Asynchronous migration

Fig. 5. Transformation of objective values in 1,000 generations.

4. Conclusion

Parallelization of genetic algorithms using coarse-grained method can be done by combining two models of processing hardware: MPI and GPU. Based on the standard deviation and ARPE obtained from the experiments, the precision of the results obtained by the parallel genetic algorithm and sequential genetic algorithm is relatively the same, with the biggest standard deviation difference of approximately 9.31. The computation time in finding a solution using parallel genetic algorithm is not yet optimal when compared to sequential genetic algorithm. Nevertheless, the proposed parallel implementation reach the convergence result faster than the sequential one. Furthermore, parallel genetic algorithm with asynchronous migration model is faster than the synchronous migration model.

In the future, we would like to investigate the use of GPU memory processing techniques to reduce data transfer time to improve the performance of the parallel genetic algorithm. For the migration process, it is also worth looking into a client-server software that is lighter to carry out the message-passing operations. Thus, the data transmission process can be efficiently shortened.

References

[1] P. S. Oliveto and C. Witt, "Improved time complexity analysis of the simple genetic algorithm," *Theoretical Comput. Sci.*, vol. 605, pp. 21-41, 2015, doi: https://doi.org/ 10.1016/j.tcs.2015.01.002.

[2] Y. Y. Liu and S. Wang, "A scalable parallel genetic algorithm for the generalized assignment problem," *Parallel Computing*, vol. 46, pp. 98–119, 2015, doi: https://doi.org/10.1016/j.parco.2014.04.008.

[3] S. Zhang and Z. He, "Implementation of parallel genetic algorithm based on CUDA," in *Advances in Computation and Intelligence (ISICA 2009)*. Springer Berlin Heidelberg, 2009, pp. 24–30, doi: http://dx.doi.org/10.1007/978-3-642-04843-2_4.

[4] Z. Skolicki and K. De Jong, "The influence of migration sizes and intervals on island models," in Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation (GECCO'05). ACM, 2005, pp. 1295–1302, doi: http://doi.acm.org/10.1145/1068009.1068219.

[5] W. Li and Y. Huang, "A distributed parallel genetic algorithm oriented adaptive migration strategy," in *Natural Computation (ICNC), 8th International Conference on*, 2012, pp. 592–595, doi: https://doi.org/10.1109/ICNC.2012.6234584.

[6] L. Falahiazar, M. Teshnehlab, and A. Falahiazar, "Parallel genetic algorithm based on a new migration strategy," in *Recent Advances in Computing and Software Systems (RACSS), International Conference on*, 2012, pp. 37–41, doi: https://doi.org/10.1109/RACSS.2012.6212694.

[7] A.-R. Hedar, A. Abdelsamee, A. Fouad, and S. T. Amin, "Advanced parallel genetic algorithm with gene matrix for global optimization," *Advanced Machine Learning Technologies and Applications (AMLTA 2012)*, Springer Berlin Heidelberg, 2012, pp. 295–303, doi: http://dx.doi.org/10.1007/ 978-3-642-35326-0_30.

[8] A. J. Umbarkar, M. S. Joshi, and W.-C. Hong, "Multithreaded parallel dual population genetic algorithm (MPDPGA) for unconstrained function optimizations on multi-core system," *Appl. Math. Comput.*, vol. 243, pp. 936–949, 2014, doi: http://dx.doi.org/10.1016/j.amc.2014.06.033.

[9] F. Lu, Y. Ge, and L. Gao, "A novel genetic algorithm with multiple sub-population parallel search mechanism," in *Natural Computation (ICNC), 2010 Sixth International Conference on*, vol. 5, 2010, pp. 2249–2253, doi: https://doi.org/10.1109/ICNC.2010.5584437.

[10] Z. R. Wang, T. Ju, D. W. Cui, and X. H. Hei, "A study of hybrid parallel genetic algorithm model," in *Natural Computation (ICNC), 7th International Conference on*, vol. 2, 2011, pp. 1038–1042, doi: https://doi.org/10.1109/ICNC.2011.6022186.

[11] OpenMP, "The OpenMP API specification for parallel programming," 2017, available at: http://openmp.org/.

[12] M. Wahib, A. Munawar, M. Munetomo, and K. Akama, "Optimization of parallel genetic algorithms for nVidia GPUs," in *Evolutionary Computation (CEC), 2011 IEEE Congress on*, 2011, pp. 803–811, doi: https://doi.org/10.1109/CEC.2011.5949701.

[13] F. M. Johar, F. A. Azmin, M. K. Suaidi, A. S. Shibghatullah, B. H. Ahmad, S. N. Salleh, M. Z. A. A. Aziz, and M. M. Shukor, "A review of genetic algorithms and parallel genetic algorithms on graphics processing unit (GPU)," in *Control System, Computing and Engineering (ICCSCE), 2013 IEEE International Conference on*, 2013, pp. 264–269, doi: https://doi.org/10.1109/ICCSCE.2013.6719971.

[14] NVidia, "CUDA parallel computing platform," 2017, available at: https://developer.nvidia.com/cuda-zone.

[15] J. O. C. Millan, V. H. A. Calvo, R. M. P. Chaves, "Quadratic assignment problem (QAP) on GPU through a master-slave PGA," *Vision Electronica*, 2016, vol. 10, issue 2, pp. 220-231, 2016, available at: https://dialnet.unirioja.es/descarga/articulo/6081873.pdf.

[16] N. Hou, F. He, Y. Zhou, Y. Chen, X. Yan, "A parallel genetic algorithm with dispersion correction for HW/SW partitioning on multi-core CPU and many-core GPU", *IEEE Access*, vol. 6, pp. 883-898, 2018, doi: https://doi.org/10.1109/ACCESS.2017.2776295.

[17] C. Li, C. Lin, J. Liu, "Parallel genetic algorithm on the graphics processing units using island model and simulated annealing", *Advances in Mechanical Engineering*, 2017, vol. 9, issue 7, 2017, doi: https://doi.org/10.1177/1687814017707413.

[18] B. Barney, Introduction to Parallel Computing. Lawrence Livermore National Laboratory, 2017, available at: https://computing.llnl.gov/tutorials/parallel_comp/.

[19] M. R. Rathomi, "Peningkatan kecepatan pemrosesan algoritma genetika dengan paralelisasi menggunakan metode coarse-grained," Master's thesis, Universitas Gadjah Mada, Indonesia, 2015, available at: https://goo.gl/Bsi7s5.

[20] R. Yusof, M. Khalid, G. T. Hui, S. M. Yusof, and M. F. Othman, "Solving job shop scheduling problem using a hybrid parallel micro genetic algorithm," *Applied Soft Computing*, vol. 11, no. 8, pp. 5782–5792, 2011, doi: https://doi.org/10.1016/j.asoc.2011.01.046.

[21] D. C. Mattfeld and R. J. Vaessens, "OR-library: a set of 82 JSP test instances," 2017, available at: http://people.brunel.ac.uk/~mastjjb/jeb/info.html.