Vol.2012-MPS-89 No.5
2012/7/16

GAROP: Genetic Algorithm framework for Running On Parallel environments

Hiroyasu Tomoyuki[1] Yamanaka Ryosuke[2] Yoshimi Masato[3] Miki Mitsunori[3]

Abstract: In this research, a Genetic Algorithms framework for Running On Parallel environments, which is named GAROP, is proposed. The GAROP provides the library for a parallel processing, so that users should only describe codes for genetic algorithms (GA) programs, utilizing the library implemented for the part requiring a parallel processing. In the GAROP framework, GA research provides only program codes which are concerned with GA algorithm and GAROP library supports other codes which are concerned with parallel processing. The advantage of using GAROP is to increase the user's productivity by making it possible to develop the program, which can execute a parallel processing. In this paper, the broad description of the GAROP is provided, and the development of the GAROP, corresponding multi-core CPU and GPU environments, is described. The libraries are implemented with GA which finds quasi-optimum solutions using meta heuristics, and its productivity and its parallelism are evaluated. As a result, only adding four descriptions to the program, the acceleration of the processing speed is confirmed in both of the environments; 5.26 times speed-up on multi-core CPU, and 3.0 times speed-up on GPU.

1. Introduction

Several types of genetic algorithms (GA) are applied to solve optimization problems and some of them are large-scale optimization problems. One of the problems confronted, when using the GA, is that the excessive amount of computing time is required. It may be difficult for some problems to be solved within a realistic time. To solve this problem, the amount of computation itself should be reduced, or the processing should be accelerated. GA attains to a global search, using multipoint searches by many candidate solutions. It requires much iteration to find a solution, and results in high calculation cost. As GA searches solutions maintaining many candidate solutions, it implicitly has parallelism. As the architectures of calculators, on the other hand, hardware having various architectures has been prevailed, such as PC clusters, multi-core processor, and GPU. Therefore, to obtain environments of parallel calculations is not as hard as ever. Although an environment is easily obtainable, the programming skills are required to bring out its performance efficiently, such as the programming for heterogeneous processors, the architectural optimization for hierarchical memories, and the programming which overlaps connection and calculation to achieve the scalability. In addition, these parallel architectures have the different configurations. Thus, even using the same algorithms, it is necessary to prepare different implementation codes suitable for different parallel architectures. This complicated and disturbing programming lacks productivity. In this research, the Genetic Algorithms framework for Running On Parallel environments, which is named GAROP, is proposed. The purpose of the GAROP is to increase the user's productivity by reducing the processing time without the specific knowledge regarding parallel architecture and parallel processing. Implementing the master-slave model, the GAROP enables to execute any logical models. In this paper, the libraries for the GAROP are implemented using C language. Target parallel processing architectures are multi-core CPU and GPU. As one of the GA technique, Simple GA is implemented using the libraries, and it is evaluated in terms of an amount of codes and execution time.

[1] Faculty of Life and Medical Sciences, Doshisha University, Kyoto, Japan
[2] Graduate School of Engineering, Doshisha University, Kyoto, Japan
[3] Faculty of Science and Engineering, Doshisha University, Kyoto, Japan

2. Background

2.1 Genetic Algorithms

GA is a multipoint search algorithm with many candidate solutions and generate-and-test algorithm. GA is powerful algorithm in all kinds of research field, because GA makes it possible to search globally, and it does not require continuity and differentiability of target problems [1–5]. Under GA, a combination of variables, which is to be a candidate solution, is termed an individual. GA searches suboptimal solutions with a group of individuals. **Fig. 1** shows a general flow of GA. The first group of individuals is generated randomly. Generated individuals are evaluated on their objective function. Then, iteration searches are started. Parent individuals are selected from the group of individuals. Individuals are generated from selected parents as candidate children by genetic calculations, such as crossover and mutation. Generated individuals as candidate children are evaluated on their objective function. Selecting from these individuals, a group of individuals for next generation is created. Presently, various types of GA algorithms have been proposed, and these algorithms are applied to real-world problems. However, an excessive amount

© 2012 Information Processing Society of Japan

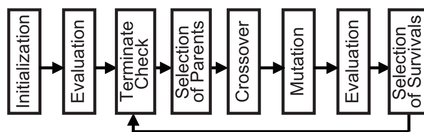


Fig. 1 A flowchart of GA.

of computation is required to find suboptimal solutions in large-scale problems. Thus, it may be difficult for some problems to be solved within a realistic time, so that the amount of computation should be reduced, or the processing should be accelerated.

2.2 Parallel Genetic Algorithms

Since GA processes a search by iterative execution of an excessive amount of samplings on multiple candidate solutions, it can be accommodated to parallelism. Thus, many methods of parallelization have been proposed for GA [6–12]. In this section, basic parallelizing methods of GA are described. There are two principal types of parallel GA. One is parallel processing in a population. This method has the same characteristic as serial GA. The other is to split a population into multiple subpopulations. Today, the latter method is frequently used, because it has the higher parallelism than the former method. There are GA methods performing very efficient parallelism, which combine the both methods. On changing the method of parallel GA, it is important to consider that it may happen to change the amount of calculations and the accuracy of solutions. That brings to the point that Parallel GA has the following two meanings.

- Parallel algorithm for increasing search performance
- Parallel implementation for reducing execution time

For example, Pospichal [13] has proposed the distributed population GA based on GPU, and achieved a high efficiency. Although this method has achieved high efficiency, GA other than the distributed population GA cannot be implemented, since GA and parallel implementation are inseparable Also, it is difficult to implement this method into architecture other than GPU. The most basic parallel models are introduced as the following.

2.2.1 The Master-Slave Model

Under GA, there is a tendency that the time consumed on evaluation calculations assumes a large share of total execution time, and the tendency strengthens as the complexity of the problem is increased. Then, the master-slave model accommodates this tendency based on the general idea of parallelization. Under the master-slave model, all the operations except evaluations are executed by a master processor. A master processor sends individuals, which are to be evaluated, to slave processors. Slave processors execute evaluating calculations on these individuals, and return the results to the master processor. **Fig. 2** shows the flow of the master-slave model. It is considered that this model is inferior to coarse-grained model, because this model requires relatively much communication, and a CPU is requisite as a master processor. The purpose of this model is to reduce the execution time, so it cannot increase the searching performance compared to a serial algorithm.

2.2.2 The Island Model

This model splits up a population into multiple sub popula-

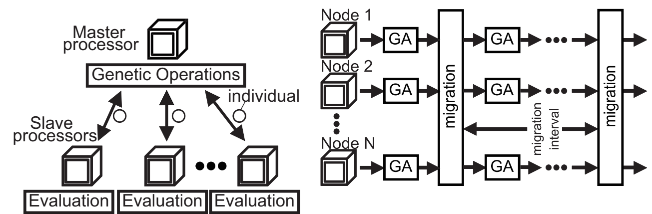


Fig. 2 A master-slave model.

Fig. 3 An island model.

tions and executes searching within each sub population. Then, it transports some individuals in a sub population to the other sub population. This operation is called the migration. **Fig. 3** shows the flow of the island model. Since this model makes communications between nodes only at the migrations, this model utilizes computational resources effectively. This model reduces its execution time and changes the performance of the search compare to a serial algorithm.

2.3 Problems

As previously mentioned, parallel GA has two objectives; those are to improve searching performance and to reduce execution time. Some of parallel GA models are depend on specialize particular parallel environment and these cannot be performed on other parallel environments. These types of GA models can use the calculation resources fully and high effectiveness. However, most of parallel calculation resources have difference architecture. Thus, when GA is tried to apply to other parallel architecture, GA researchers have to implement their algorithms for new architecture. To know configurations of various architectures and to implement suitable GA are the heavy burden on GA researchers. With these defects, even research have good parallel environments, it may take time to implement their algorithm.

3. Systematization of Parallel Model

As previously mentioned in chapter II, the expression of "parallel GA" has two models; one is a parallel algorithm to increase searching performance, and the other is a parallel implementation to reduce the execution time. These two models have to be distinguished clearly. In this research, these two models are defined as the followings:

- The logical model: parallel algorithm to increase searching performance
- The implementation model: a parallel implementation to reduce the execution time

Fig. 4(a) shows GA adopted an island model as a logical model and a serial model as an implementation model. **Fig. 4(b)** shows GA adopted and island model as a logical model and also as an implementation model. In **Fig. 4**, the number means the orders of processing. The logical model is a model to execute parallel searching, but it is capable to execute serial processing. Additionally, the logical model may be confined by a limitation imposed by the implementation model. For example, if the island model is adopted as the implementation model, the Simple GA cannot be adopted as the logical model. Once the implementation model is provided, only users have to do is to consider and implement the logical model. Thus, users are able to develop any algorithms

© 2012 Information Processing Society of Japan

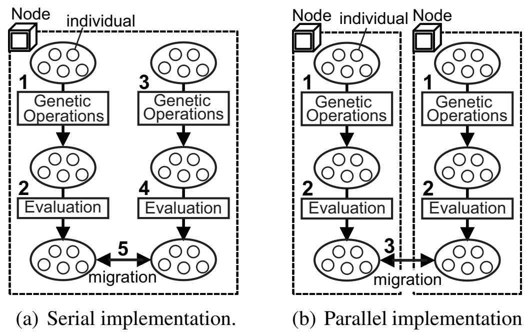


Fig. 4 Two island models as the logical model.

without the limitation of architectures.

4. GAROP

The GA framework for Running On Parallel environments (GAROP) is a framework in which any GA can be executed in various parallel environments. The purpose of the GAROP is that users can execute parallel processing, with the master-slave model as the implementation model, without having special techniques for parallel programming. The users of the GAROP are presumed as the developers of GA. Once the implementation for parallel processing is provided, users could receive the benefit of parallel processing, keeping the same level of productivities as sequential programs. Under the GAROP, users construct any logical models and implement some parts other than the evaluation part. Users designate the template suited to each parallel environment, and combine the template and the codes for evaluation of the problem, so that the evaluation part is implemented. The use of this template leads to hide the special communications and the implementation of scheduling for evaluation tasks. Thus, users can execute GA under parallel environments without having knowledge of communication and schedulers appropriate for parallel environments.

4.1 Requirements

To achieve the purpose as precious described, the GAROP is expected to meet the following requirements:

- Productivity of users
  In order to achieve the same level of productivity as the sequential programs, the framework should be provided as the form of libraries which call up a function group.
- Versatility of parallel environments
  Users should be able to use various parallel environments by using common descriptions.
- Independent implementation model
  In order to correspond to any GA, the implementation model which is implemented independently of algorithms is required.

4.2 Design of the GAROP

Fig. 5 shows the overview of the GAROP. GAROP introduces the concept of the Individual Pool as an interface to link users to parallel environments. The Individual Pool is an archive and stores individuals which should be evaluated in parallel automatically. Under this design, any GA models can be executed in parallel without change searching performance. In **Fig. 6**, the concepts of serial model and island model are demonstrated. In the same way, other parallel models can be performed using the GAROP.

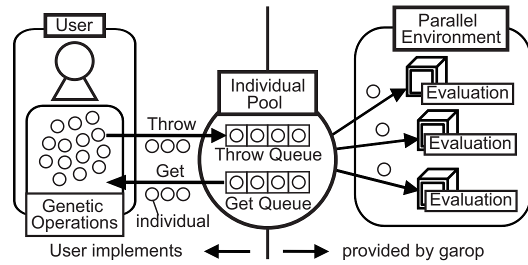


Fig. 5 Overview of the GAROP.

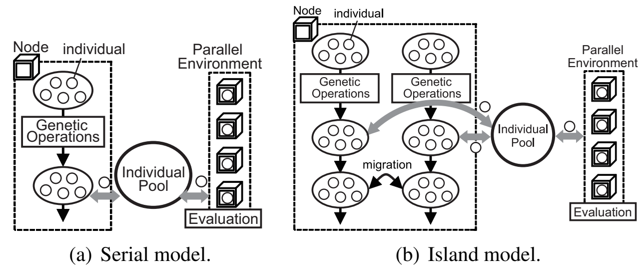


Fig. 6 GA examples with GAROP.

Table 1 Functions provided by GAROP.

Name	Action
Initialize	prepare parallel environments and the Individual Pool.
Throw	throw an individual to the Individual Pool.
Get	get an individual from the Individual Pool.
Finalize	free memories used by the Individual Pool and disconnect parallel environments.

4.2.1 The Individual Pool

The Individual Pool consists of two queues as shown in Fig. 5. One of these queues is the "throw queue", which stores thrown individuals, and the other queue is the "get queue", which stores evaluated individuals. As soon as individuals are stored in the throw queue, they are sent to calculation resources and evaluated. Evaluated individuals are stored in the get queue. Users throw individual which should be evaluated to the Individual Pool one by one. Users can get the evaluated individual from the get queue, whenever they need. The Individual Pool is a useful concept which makes it possible to execute evaluating calculations and other processing simultaneously.

4.2.2 Programming Interface

The GAROP provides 4 functions as shown in **Table 1**. This won't be changed even though the environment of the parallel computation is changed. However, the types of four functions and arguments vary depending on the environment.

4.2.3 The Template of evaluation

It is impossible to provide implementations of every single evaluation, because the evaluation depends on the problem to be solved. In the GAROP, users implement the evaluation part. This means that the implementation of the evaluating calculation exists on the memory of the master machine. However, the evaluation is executed by slave processors. Therefore, the evaluation part must be handed over from the master machine to the slave processor.

© 2012 Information Processing Society of Japan

It is realistic to hand over using a form of a function, which is easy to describe. In order to figure out how much memory region is needed, the types and the number of arguments and the type of return value should be known, when the function is handed over. The GAROP provides the template of an evaluate function as the following. Users can solve any problems by limiting the arguments and the return value of the function.

```
void evaluate(unsigned char* indata,
              unsigned char* retdata);
      infdata: individual data
      retdata: evaluated individual data
```

This way of description is suitable for the C language. The important thing is to allocate an individual to the argument and to allocate another individual to the return value. In order for a description of an individual to be free, it employs unsigned char as a pointer. The template of the evaluating function varies depending on each of the parallel environment and the language used.

4.3 How to run algorithms with the GAROP

The libraries to substantiate the GAROP are provided in the form of source codes. Multiple libraries are prepared for each of the execution environments, such as compilers and languages. The user's flow is shown as the following.

- Obtaining the library (source codes) corresponding to the executing environment
- Implementing evaluate function using the templates
- Implementing GA with API of the library
- Compiling source codes
- Placing executable file into calculation resources
- Executing

List 4 is an example of evaluate function using a template. List 1 is an implementation of the serial model with the GAROP. List 2 is an implementation of the island model with the GAROP. Like this way, not only the master-slave model but also the island model can be implemented. Using the GAROP, other models such as cellular model can be implemented in the same way. Under the GAROP, the parallel environment is set up by users. For example, if users use PC cluster with MPI, users prepare cluster by themselves and give a description of machine file etc.

5. Implementations of Libraries for the GAROP

The libraries are implemented to get individuals from Individual Pool and to evaluate in parallel. The environments implemented in this paper are the multi-core CPU by pthread of the C language and the GPU by CUDA. The multi-core CPU is a shared memory environment, and the GPU is a distributed memory environment. Policies of these environments are as the following.

5.1 Multi-core CPU

The characteristics of multi-core CPU are having not many cores and having a shared memory environment. Considering these two characteristics, the library is implemented with a constitution shown as **Fig. 7**. A thread is regarded as a calculation resource and assigned as a slave processor. Each thread monitors the throw queues. When the throw queue has one or more data,

List 1 Serial model with the GAROP.

```
1  population = InitPopulation();
2  Initialize(); // initialization of framework
3  FOR j = 0 to generation limit DO
4    FOR i = 0 to population num DO
5      // throw individuals to GA Pool
6      Throw(population[i]);
7    ENDFOR
8    FOR
9      // get individuals from GA Pool
10     Get(population[i]);
11   ENDFOR
12   selection(population);
13   crossover(population);
14   mutation(population);
15 ENDFOR
16 Finalize(); // finalization of framework
```

List 2 Island model with the GAROP.

```
1  population1 = InitPopulation();
2  population2 = InitPopulation();
3  Initialize(); // initialization of framework
4  FOR j = 0 to generation limit DO
5    FOR i = 0 to population num DO
6      // throw individuals to GA Pool
7      Throw(population1[i]);
8      Throw(population2[i]);
9    ENDFOR
10   FOR
11     // get individuals from GA Pool
12     Get(population1[i]);
13     Get(population2[i]);
14   ENDFOR
15   selection(population1); selection(population2);
16   crossover(population1); crossover(population2);
17   mutation(population1); mutation(population2);
18   IF j % 10 == 0 THEN
19     migration()
20   ENDIF
21 ENDFOR
22 Finalize(); // finalization of framework
```

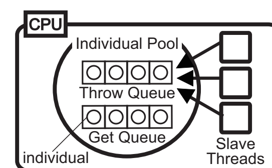


Fig. 7 An implementation of the GAROP for multi-core CPU.

each thread gets an individual and executes evaluating calculations. Arguments of an initialize function are number of threads, size of an individual, and pointer of an evaluate function.

5.2 GPU

The GPU has many cores and has a distributed memory environment. As the GPU has a distributed memory environment, the library is implemented with a constitution shown as **Fig. 8**. In CPU, there are main thread which run GA and sub thread which communicate data to GPU. Arguments of initialize function are number of blocks and thread, size of an individual, and number of individuals that are processed at a kernel function call. Individuals sent to GPU are stored in the constant memory. Each CUDA thread acquires individual information from constant memory, and writes evaluated individual data to global memory.

6. Evaluation

The following is the evaluation of a Simple GA [14] imple-

© 2012 Information Processing Society of Japan

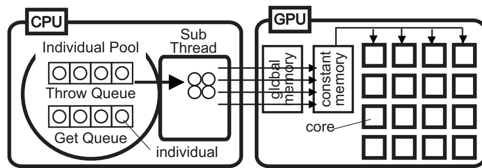


Fig. 8 An implementation of the GAROP for GPU.

Table 2 Parameters of the Simple GA

population size	64
chromosome length	64
size of an individual	68 bytes
max generations	100
optimization problem	onemax

List 3 an implementation of an individual

```
1 typedef struct __individual {
2   char* chromosome;
3   int fitness;
4 } Individual;
```

List 4 an evaluate function used in this experiment

```
1  void evaluate( unsigned char* indata,
2                 unsigned char* retdata ) {
3    int i, j;
4    int sum = 0;
5    Individual* individual = (Individual*)indata;
6    for( j = 0; j < 100000; j++ )
7      for( i = 0; i < CHROMOSOME_LENGTH; i++ )
8        sum += individual->chromosome[i];
9    individual->fitness = sum;
10   retdata = indata; }
```

Table 3 Specifications of a master machine

OS	Debian 4.1.2
memory	6 GB
CPU	Intel Xeon W3530 2.80 Ghz
# of physical cores	4
# of logical cores	8

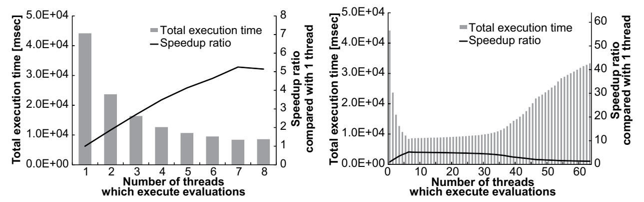
Table 4 Specifications of Tesla C2050

total amount of global memory	2.68 GB
number of multiprocessors	14
number of cores	448
total amount of constant memory	65536 bytes
total amount of shared memory per block	49152 bytes
warp size	32
clock rate	1.15 GHz

mented using the GAROP, in terms of productivity and parallelism. **Table 2** shows a parameter of the Simple GA. List 3 shows a description of an individual in this evaluation. In this experiment, onemax problem is iterated 100,000 times to mimic a large-scale problem. List 4 shows a function on the evaluation. In this GA, the evaluating calculation takes up more than 99 % of total execution time. The maximum number of parallelism is 64, since the population size is 64. Thus, number of individuals to be calculated at a kernel function call is 64 in the GPU library.

6.1 Environments

The architectures to be evaluated in this experiment are multi-core CPU and GPU. **Table 3** shows the specifications of the machine used in this evaluation. This machine is mounted Tesla C2050 as shown in **Table 4**.



(a) The number of threads (1 to 8). (b) The number of threads (1 to 64).

Fig. 9 Speedup and execution time on multi-core CPU depending on number of threads.

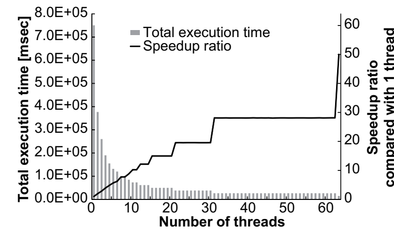


Fig. 10 Speedup and execution time on GPU depending on number of threads.

6.2 Results

Fig. 9 shows results of multi-core CPU, and **Fig. 10** shows results of GPU, respectively. List 5 shows the source codes which described by users on multi-core CPU. List 6 shows the source codes which described by users on GPU. List 5 and List 6 show that these descriptions are common among different architectures. It is verified that descriptions for parallel processing can be completely hidden, even though these descriptions are essentially required on parallel processing. **Fig. 9(a)** shows that the execution time reduces when the number of threads is 1 to 7. **Fig. 9(b)** shows that the number of threads that had the least processing time is 7, and it is improved 5.26 times compared to the situation when a thread is processed. When the number of thread is 8 or more, execution time takes longer. Fig. 10 shows that the number of threads that had the least processing time is 64, and it is improved 50.01 times compared to the situation when a thread is processed. When the numbers of thread are 2, 3, 4, 6, 8, 16, 32 and 64, the execution time is extremely reduced.

7. Discussion

The C language and CUDA are the similar languages, but methods of implementation in parallel are substantially different. C is used for multi-core architecture and CUDA is used for GPU architecture. Usually, users need the parallel programming knowledge of C and CUDA for these architectures. However, using the GAROP framework and its library, users need not to prepare the codes for parallel processing. Both of the codes which GA users prepare can be used as the common descriptions. Therefore, the productivity of coding is increased using the GAROP. Reviewing the result of multi-core CPUs, when the number of threads is 8, the execution time is the shortest. Among the 8 threads, 7 of them are the threads for the slave processors and the rest is the main thread for GA operations. The calculation server for this experiment has 8 logical cores. Therefore, when 7 threads are used for the slave processors, all of the logical cores are occupied. When 8 or more threads are used, the number of

© 2012 Information Processing Society of Japan

List 5 Simple GA with a library for pthread in C.

```
1  Individual population[POPULATION_SIZE];
2  // create population
3  InitPopulation( population );
4  // initialization of framework
5  Initialize(sizeof(Individual), THREAD_NUM, evaluate);
6  for(i = 1; i <= MAX_GENERATION; i++) {
7     // throw individuals to GA Pool
8     for(j = 0; j < POPULATION_SIZE; j++)
9        Throw( (unsigned char*)&population[j],
10             sizeof(Individual));
11    // get individuals from GA Pool
12    for(j = 0; j < POPULATION_SIZE; j++)
13       Get((Individual*)&population[j],
14           sizeof(Individual));
15    population = selection(population);
16    crossover(population);
17    mutation(population); }
18 Finalize(); // finalization of framework
```

List 6 Simple GA with a library for GPU in CUDA.

```
1  Individual population[POPULATION_SIZE];
2  // create population
3  InitPopulation(population);
4  // initialization of framework
5  Initialize(sizeof(Individual), BLOCK_NUM,
6             THREAD_NUM, POPULATION_SIZE);
7  for(i = 1; i <= MAX_GENERATION; i++) {
8     // throw individuals to GA Pool
9     for(j = 0; j < POPULATION_SIZE; j++)
10       Throw((unsigned char*)&population[j],
11            sizeof(Individual));
12    // get individuals from GA Pool
13    for(j = 0; j < POPULATION_SIZE; j++)
14       Get((Individual*)&population[j],
15           sizeof(Individual));
16    population = selection(population);
17    crossover(population);
18    mutation(population); }
19 Finalize(); // finalization of framework
```

threads is more than the number of logical cores, so that it prevents the speed up. When the number of threads is more than the number of cores, it may happen that CPU has to switch executing threads one after another, and this operation of switching threads produces the waiting time. In the experiment in GPU, the execution time is the shortest with 64 threads. Tesla C2050, which is the GPU used in this experiment, has 448 cores. Thus, it can use 64 cores for 64 threads. From the Fig. 10, it is observed that the stagnation of speed up is existed from 17 to 31 threads and from 33 to 63 threads. The reason of this stagnation is that the population size 64 is not the multiple of these numbers. Because of this reason, the fraction of the individuals is existed and the fraction itself should be calculated, too. This takes time and leads to the stagnation.

8. Conclusions

In this paper, the GAROP which is a parallel environment framework for evolutionary computation is proposed. The GAROP is the framework where the logical model and the implementation model are distinguished, and the users prepare their algorithms as the logical model and the implementation model is prepared by systems. Thus, users can implement any type of logical model on parallel environment using the GAROP. In the GAROP, user's evolutionary algorithms are performed in parallel as master-slave model. Users implement their GA operations in the master and the evaluation part is implemented in the slave using the provided template. With the provided libraries and the implemented codes, the application, which is worked on several types of parallel environment, is compiled. In this paper, the concept and the flow of the GAROP are described and the libraries for two types of parallel environments are implemented; those are multi-core CPUs and GPUs. Using the GAROP and the libraries, Simple GA is implemented and the productivity of users and parallelism are evaluated. From the results, GA applications which can be worked on parallel environments are implemented using four types of functions which are provided by the GAROP. At the same time, execution time is also reduced.

In the future work, further discussion should be held for not only Simple GA but also for other evolutionary computation algorithms. In this paper, the libraries for multi-core CPUs and GPUs are implemented. Other types of libraries for other parallel environments will be prepared. At the same time, the productivity discussions of the GAROP should be performed with researches who are working on evolutionary computation fields.

References

[1] B. Chakraborty, T. Maeda, and G. Chakraborty. *Multiobjective route selection for car navigation system using genetic algorithm*, *Soft Computing in Industrial Applications, 2005. SMCia/05. Proceedings of the 2005 IEEE Mid-Summer Workshop on*, pp. 190–195, june 2005.

[2] R. Ruiz, C. Maroto, and J. Alcaraz. Two new robust genetic algorithms for the flowshop scheduling problem. *OMEGA, The International Journal of Management Science*, Vol. 34, No. 5, pp. 461–476, 2006.

[3] P. Chann Chang, Hsieh J. Chang, and Wang C. Yuan. Adaptive multi-objective genetic algorithms for scheduling of drilling operation in printed circuit board industry. *"Applied Soft Computing*, Vol. 7, No. 3, pp. 800–806, 2007.

[4] C Poloni, A Giurgevich, L. Onesti, and V. Pediroda. Hybridization of a multi-objective genetic algorithm, a neural network and a classical optimizer for a complex design problem in fluid dynamics. *Computer Methods in Applied Mechanics and Engineering*, Vol. 186, No. 2-4, pp. 403–420, 2000.

[5] B. Ombuki, Brian J. Ross, and F. Hanshar. Multi-Objective Genetic Algorithms for Vehicle Routing Problem with Time Windows. *Applied Intelligence*, Vol. 24, pp. 17–30, 2006.

[6] T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In Schwefel, Hans-Paul and Männer, Reinhard, editor, *Parallel Problem Solving from Nature*, Vol. 496 of *Lecture Notes in Computer Science*, pp. 176–185. Springer Berlin / Heidelberg, 1991.

[7] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In J. Becker, I. Eisele, and F. Mündemann, editors, *Parallelism, Learning, Evolution*, Vol. 565 of *Lecture Notes in Computer Science*, pp. 398–406. Springer Berlin / Heidelberg, 1991.

[8] Theodore C. Belding. The distributed genetic algorithm revisited. *Proc.6th International Conf. Genetic Algorithms*, pp. 114–121, 1995.

[9] M. Miki, T. Hiroyasu, M. Kaneko, and K. Hatanaka. A Parallel Genetic Algorithm with Distributed Environment Scheme. *IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 1, pp. 695–700, 1999.

[10] D. Lim, Y. Soon Ong, Y. Jin, Sendhoff.B, and Lee B. Sung. Efficient Hierarchical Parallel Genetic Algorithms using Grid computing. *Future Generation Computer Systems*, Vol. 23, No. 4, pp. 658–670, 2007.

[11] J. Ming Li, X. Jing Wang, R. Sheng He, and Z. Xian Chi. An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, pp. 855–862, Sep. 2007.

[12] Thompson, A. Matthew and Dunlap, I. Brett. Optimization of analytic density functionals by parallel genetic algorithm. *Chemical Physics Letters*, Vol. 463, No. 1–3, pp. 278–282, 2008.

[13] P. Pospichal and J. Jaros. GPU-based Acceleration of the Genetic Algorithm. *GPU competition of GECCO competition*, 2009.

[14] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

© 2012 Information Processing Society of Japan