Improved Parallel Genetic Algorithm Based on Resetting Strategy and Hash Table

Peng Lu, Xiaoqiang Xiao*, Jijin Wang, and Weixun Ning

Department of Computer Science, National University of Defence and Technology, Changsha, Hunan, 410000, China

*Corresponding author’s e-mail: xqxiao@nudt.edu.cn

Abstract. In the process of parallel genetic algorithm (PGA) convergence, the probability of generating repeated individuals in a new generation is gradually increasing, which leads to repetitive computations of fitness values. In order to improve the time-efficiency and quality of solutions, an improved algorithm based on resetting strategy and hash table (RHPGA) is proposed. On the one hand, RHPGA takes advantage of the low time complexity of hash table lookup to reduce fitness values’ calculation of repeated individuals. On the other hand, RHPGA uses a resetting strategy to improve the optimal solution searching ability. By solving a set covering problem, PGA and RHPGA are compared. The experiment proves that RHPGA can almost increase the time-efficiency by 300% and quality of solutions by 15%.

1. Introduction

Genetic Algorithm (GA) was officially proposed by Professor Holland in the United States in 1975. Inspired by natural selection, GA represents a generic heuristic method for finding near-optimal or optimal solutions to difficult search and optimization problems [1].

GA has inherent parallelism embedded in the evolutionary process [2]. For example, a population can be naturally divided into a set of sub-populations (also called demes) that evolve and converge with a significant level of independence. As a well-known approach to increase the computational efficiency of GA, Parallel genetic algorithms differ in where parallelization is exploited: fine-grained PGAs [3] parallelize the selection operator to select parents from directly connected neighbours on a PGA topology in each iteration; coarse-grained PGAs migrate a portion of local solutions to connected demes periodically [4]; global parallelization based PGAs parallelize the computation of the fitness evaluation function if the function is computing intensive [5]. PGA combines the high-speed parallelism of parallel computers with the natural parallelism of genetic algorithms. In recent years, high performance and parallel computing has been extensively studied and various types of PGAs have been developed and broadly applied in a rich set of application domains [6-9]. More interestingly, previous work by Alba and Troya [10] showed that PGA computation not only improves computational efficiency over sequential GAs, but also facilitates parallel exploration of solution space for obtaining more and better solutions. In fact, Hart et al. showed that running PGA even on a single processor core outperformed its sequential counterpart. Therefore, PGA is often considered and evaluated as a different algorithm rather than just the parallelization of its corresponding sequential GA.

This paper describes a resetting-based and hash-table-based PGA (RHPGA) to improve time-efficiency and quality of solutions compared with PGA for solving large problem instances of a classic combinatorial optimization problem – the Set Covering Problem (SCP). SCP belongs to the class of
NP-hard 0–1 integer linear programming problems [11]. Various exact and heuristic algorithms have been developed to solve SCP instances of modest sizes. However, in practice, problem instances often have larger sizes while the problem solving requires quick solution time and specified quality, which compounds the computational challenges. RHPGA is a coarse-grained PGA that searches solution space in parallel based on independent deme evolution and periodical migrations among connected demes.

2. Algorithm

2.1 Set Covering Problem

Set covering problem is a classic problem in combinatorial mathematics, computer science, and computational complexity theory. The decisive problem of set coverage is one of Karp’s twenty-one NP-complete problems [12].

Suppose $S$ is a set and contains $m$ elements. $S$’s subsets $S_1, S_2 \ldots S_k$ form one feasible solution of SCP if and only if the following equation is satisfied:

$$\bigcup_{i=1}^{k} S_i = S$$  \hspace{1cm} (1)

The SCP is to find the minimum sets from $S$’s given subsets $S_1, S_2 \ldots S_n$ that satisfy (1). It can be modelled mathematically by the following formula:

$$\min \quad 1^T x$$
$$\text{s.t.} \quad A x \geq 1^T$$
$$x \in \{0,1\}^n$$
$$A \in \{0,1\}^{m \times n}$$  \hspace{1cm} (2)

$A_{m \times n}$ is a binary matrix and $a_{ij} = 1$ indicates that subset $S_j$ contains element $i$. And $x_i = 1$ indicates that subset $S_i$ is chosen.

2.2 Parallel Genetic Algorithm

2.2.1 Genetic Algorithm

Genetic algorithm is a global iterative, evolutionary, and widely adaptive global search algorithm based on the principles of natural selection and population genetics.

The relevant content of the genetic algorithm in this paper is as follows.

Individual’s code: Each individual is encoded into a bool array gene of $n$ bits. Where $gene[i] = 1$ indicates that the solution contains the subset $S_i$, and vice versa.

![Figure 1. Representation of an individual.](image)

**Fitness function:** In a general fitness function, better individual fitness is as large as possible, and this paper makes some changes. The lower the fitness of an individual, the better. The function is as follows:

$$fit(x) = \beta \cdot 1^T \cdot (\text{neg}(Ax))$$
$$\beta \geq 1$$

$$\text{neg}(a) = \begin{cases} 0, & a = 0 \\ 1, & \text{otherwise} \end{cases}$$  \hspace{1cm} (3)

$\beta \cdot 1^T \cdot (\text{neg}(Ax))$ indicates $\beta$ times of the number of uncovered elements, and $1^T x$ indicates the number of subsets. By setting the fitness function in this way, on the one hand, the punishment for the
uncovered elements is strengthened to ensure that the subset can cover \( S \); on the other hand, the fitness of the algorithm after convergence is able to equal to the solution of (2).

**Crossover operator:** In each iteration, every individual has a fixed probability of crossover. A single individual \( r \) is randomly selected from the population, and the cut position is generated randomly, and the two genes cut and splice themselves to generate two new individuals - child1 and child2. Then we just choose one child into selection operator from these two children by equal probability.

![Crossover operator](image)

**Mutation operator:** In each iteration, every individual has a fixed probability of mutation. The mutation position is generated randomly, and the value of the mutation bit is inverted to generate a child. Then we perform the selection operator.

![Mutation operator](image)

**Selection operator:** After crossover or mutation operators. Each parent generates only one child, which is selected to the next generation with the probability \( P \), as shown in (4). Such a selection operator slows the convergence speed, but improves the ability of finding optimal solutions.

\[
P(child) = \frac{\text{fit}^2(father)}{\text{fit}^2(father) + \text{fit}^2(child)} \quad \text{fit}^2(father) + \text{fit}^2(child)
\]

\[
P(father) = \frac{\text{fit}^2(child)}{\text{fit}^2(father) + \text{fit}^2(child)}
\]

The GA flow chart is shown in Area I of figure 4.

2.3 **Thread parallelism**

This paper takes a thread-parallel coarse-grained model. A shared area is created to record the fittest individuals of each thread. Each child thread runs GA, and synchronize its own fittest individual to the shared area and introduce other threads’ fittest individuals periodically from the shared area.

In order to prevent read-write errors, we introduced a global `timed_mutex` from C++ STL to ensure that each thread read and write safely. The `timed_mutex` class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In a manner similar to `mutex`, `timed_mutex` offers exclusive, non-recursive ownership semantics [13]:

- When a thread owns a `timed_mutex`, all other threads will block or receive a false return value if they attempt to claim ownership of the `timed_mutex`.
- A calling thread must not own the `timed_mutex` prior to calling `lock()` or `try_lock_for()`.
- `Timed_mutex` provides the ability to attempt to claim ownership of a `timed_mutex` with a timeout via the `try_lock_for()` and `try_lock_until()` method.
The specific PGA flow chart is shown in Area II of figure 4.

2.4 Hash-table-based optimization method

A general hash function saves the corresponding key-value pair in the hash table.

![Algorithm flow chart](image-url)

*Figure 4. Algorithm flow chart: Area I shows GA, Area II shows PGA and Area III shows HPGA. Dotted arrow lines represent data interactions.*
Find process: Firstly, a hash address Hash(key) is calculated according to a given key, and if the key of key-value stored in the address equals to the given key, the function returns the corresponding value. Otherwise, the hash address is calculated again according to collision resolution strategies. If it still fails, it will return failure.

Insertion process: When the table does not contain the given key, calculate Hash(key). If there is no key-value stored in the address, given key-value can be directly inserted. Otherwise, the hash address is calculated again according to collision resolution strategies. If it still fails, the hash table capacity expansion is performed or returns failure.

Original hash table and hash function will cause the following problems if an individual’s binary code is used as a key and its fitness value is used as a value:

- Too much memory will be occupied if the length of an individual’s gene is very large.
- key-value needs to be compared in every find or insertion process, which calls extra time cost.
- General collision resolution strategies have a bad effect on time and space costs.

Therefore, according to the upper problems, the following methods are used:

- Using a fixed size HASHSIZE hash table instead of an expandable hash table to control the space cost.
- Using hash function based on random seed method from C++ standard template library.

Random number engine default_random_engine (abbreviated to e) is used in this paper. Function e.seed(int) can set a seed. The random number generator uniform_int_distribution < HASHSIZE − 1 > (abbreviated to u), function u(e) can randomly generate integers in the range [0,HASHSIZE-1].

The individual binary code gene is segmented by 32 bits, and those 32-bit integer values are \( n_1, n_2, \ldots, n_x \) (\( n_x \) may be less than 32 bits, and its integer value is taken according to the remaining number of bits). The initial seed is set to \( n_1 \), then calls the e.seed(\( n_2 + u(e) \)), and loops. The process and the final hash value hash(key) are shown in figure 5:

\[
\begin{align*}
\text{Addr:0} & \quad \begin{array}{|c|c|}
\hline
\text{timestamp} & \text{fitness value} \\
\text{timestamp} & \text{fitness value} \\
\text{timestamp} & \text{fitness value} \\
\text{......} & \\
\hline
\end{array} \\
\text{Addr: HASHSIZE-1} & \quad \begin{array}{|c|c|}
\hline
\text{timestamp} & \text{fitness value} \\
\text{timestamp} & \text{fitness value} \\
\text{timestamp} & \text{fitness value} \\
\hline
\end{array}
\end{align*}
\]

Figure 5. Hash function based on random method.

Figure 6. Hash table.
• Using timestamps instead of keys to reduce the space cost.
  In find and insertion process, calculate the individual’s hash address. If there is no data or
  \( current\_generation - timestamp \geq gate \), calls fit() to calculate the fitness and overwrite current
  \( timestamp - value \). Otherwise, return the fitness value.

  Obviously, such a find and insertion process has a possibility that different individuals return error
  fitness values due to the same hash address. But this is feasible in the algorithm because all fitness
  values in the table are calculated by the fitness function, so the returned error value does not affect
  the correctness of the solution. The form of the hash table is shown in figure 6.

  The hash-table-based parallel genetic algorithm (HPGA) flow chart is shown in Area II of figure 4.

2.5 Resetting-based optimization method

Individuals will gradually converge and result in near-optimal solution in the iterations. HPGA only
  reduces the time cost compared with PGA, but its ability to search for optimal solutions is not
  improved. Therefore, we introduce a resetting-based optimization method to improve the quality of
  solutions. During the iterations, using the following rules to generate new populations:

  • Reset the population periodically.
  • Every individual is reset. Its bool array gene is replaced by an array of randomly distributed
    \((a-1)\) Trues where \(a\) means the number of Trues in current best individual’s gene.

3. Experimental verification and results

3.1 Experimental platform and algorithm parameters

Experimental platform and algorithm parameters are shown in table 1.

| Algorithm | Parameter                  | Detail                  |
|-----------|----------------------------|-------------------------|
| CPU       |                           | Intel i7-9700KF 8C8T    |
| RAM       |                           | 32GiB DDR4 2133MHZ      |
|           | Programming Language       | C++ 14                  |
|           | Operating System           | WINDOWS 10              |
| PGA       | \( \beta \)                | 10                      |
|           | Number of Generation       | 1 million               |
|           | Number of Population of    | 500                     |
|           | Each Threads               |                         |
|           | Number of Child Threads    | 4                       |
|           | Rate of Crossover          | 0.9                     |
|           | Rate of Mutation           | 0.05                    |
|           | Synchronize Best Individuals| Every 100 generations   |
| HPGA      | Size of Hash Table\(^1\)   | \(2^{24} = 16,777,216\) |
|           | Update Rate of Hash Table’s| \(\Delta Timestamp \geq 200\) |
|           | Values                     |                         |
|           | Synchronize Hash Table     | Every 100 Generations   |
| RHPGA     | Reset Rate                 | Every 5000 Generations  |

\(^1\) With the size of hash table, we can calculate the extra memory consumption:

\[8 \times 2^{24} \times \text{Number of Hashtables Bytes} = 625\text{MiB}].\]
3.2 Test cases
We randomly generate five test samples (sample No.0 to No.4), each of which contains 400 elements and 400 subsets. In each subset, the probability of each element being generated is 5%. Test samples and source codes have been uploaded to GitHub [14].

3.3 Experimental results and analysis

Table 2. Comparisons in 5 test samples.

| Algorithm | No.0 | No.1 | No.2 | No.3 | No.4 |
|-----------|-----|-----|-----|-----|-----|
| RHPGA     | 35  | 13551 | 35  | 10469 | 34  | 13709 | 37  | 12168 | 36  | 12280 |
| HPGA      | 39  | 4133 | 40  | 3832 | 41  | 3813 | 38  | 3917 | 39  | 3995 |
| PGA       | 42  | 43016 | 43  | 42367 | 39  | 40649 | 42  | 64198 | 42  | 50614 |

In each test, we use the same initial parameters and conditions, including seeds. Table 2 shows:
- Hash-table-based optimization method can improve the speed of iterations without affecting PGA’s convergence.
- Resetting-based optimization method can improve the ability of finding optimal solutions compared with PGA. Although RHPGA is slower than HPGA, its solution is still better than HPGA from a time perspective.
- Compared with PGA, RHPGA only consumes one-third of the time and gets about 15% improvement of solutions.

4. Conclusions
This paper proposes an improved parallel genetic algorithm based on resetting strategy and hash table. Test examples for the SCP problem are solved by PGA, HPGA and RHPGA respectively, which verifies the improvement in the optimization ability and calculation speed. We are convinced that these optimization methods can be applied to solve a variety of problems, not limited to SCP. However, due to limited time, we will discuss the scenarios in which the algorithm is applicable in the next study.

Acknowledgments
This research was supported by No.61872372 from the National Natural Science Foundation of China. We wish to thank the anonymous referees for their insightful and constructive comments.

References
[1] Holland J H. Adaptation in Natural and Artificial System[M]// Adaptation in natural and artificial systems. MIT Press, 1992.
[2] Alba E, Tomassini M. Parallelism and evolutionary algorithms[J]. IEEE Transactions on Evolutionary Computation, 2002, 6(5):0-462.
[3] Whitley L D. Cellular Genetic Algorithms[C]// Companion Publication of the Conference on Genetic & Evolutionary Computation. ACM, 2014.
[4] M. Ruciński, Izzo D, Biscani F. On the impact of the migration topology on the Island Model[J]. Parallel Computing, 2010, 36(10-11):555-571.
[5] D'Ambrosio D, Spataro W. Parallel evolutionary modelling of geological processes[M]. Elsevier Science Publishers B. V. 2007.
[6] Pietro S, Jun O, Xin H, et al. Time Complexity of Evolutionary Algorithms for Combinatorial Optimization: A Decade of Results[J], 2007, 4(3):281-293.
[7] Zdenĕk Konřšt. Parallel genetic algorithms: advances, computing trends, applications and perspectives[C]// International Parallel & Distributed Processing Symposium. IEEE, 2004.
[8] Huang C H, Rajasekaran S. High-performance parallel bio-computing[J]. Parallel Computing, 2004, 30(9-10):999-1000.
[9] Perez J I H. Parallel Architectures and Bioinspired Algorithms[J]. Parallel Computing, 2010, 36(10):553-554.
[10] Alba E, José M. Troya. A survey of parallel distributed genetic algorithms[J]. Complexity, 1999, 4.
[11] Fisher M L, Jaikumar R, Van Wassenhove L N. A Multiplier Adjustment Method for the Generalized Assignment Problem[J]. Management Science, 1986, 32(9):1095-1103.
[12] Farahani R Z, Asgari N, Heidari N, et al. Covering problems in facility location: A review[J]. Computers & Industrial Engineering, 2012, 62(1):368-407.
[13] Cpp. (2018) Timed_mutex in STL. https://en.cppreference.com/w/cpp/thread/timed_mutex.
[14] ZWFolirly. (2019) RHPGA. https://github.com/ZWFolirly/RHPGA.