Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2015, Article ID 806708, 8 pages
http://dx.doi.org/10.1155/2015/806708

Research Article

A Phoenix++ Based New Genetic Algorithm Involving Mechanism of Simulated Annealing

Luokai Hu,[1,2] Jin Liu,[1,3,4] Chao Liang,[2] Fuchuan Ni,[5] and Hang Chen[5]

[1]*Hubei Co-Innovation Center of Basic Education IT Services, Hubei University of Education, China*
[2]*Lenovo Mobile Communication Technology Co., Ltd., China*
[3]*Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, China*
[4]*State Key Laboratory of Software Engineering, Computer School, Wuhan University, Wuhan, China*
[5]*Department of Computer Science, Huazhong Agricultural University, China*

Correspondence should be addressed to Jin Liu; mailjinliu@yahoo.com

Received 16 January 2015; Accepted 27 March 2015

Academic Editor: Houbing Song

Genetic algorithm is easy to fall into local optimal solution. Simulated annealing algorithm may accept nonoptimal solution at a certain probability to jump out of local optimal solution. On the other hand, lack of communication among genes in MapReduce platform based genetic algorithm, the high-performance distributed computing technologies or platforms can further increase the execution efficiency of these traditional genetic algorithms. To this end, we propose a novel Phoenix++ based new genetic algorithm involving mechanism of simulated annealing. Simulated annealing genetic algorithm has two distinctive characteristics. First, it is the synthesis of the conventional genetic algorithm and the simulated annealing algorithm. This characteristic guarantees our proposed algorithm has a higher probability of getting the global optimal solution than traditional genetic algorithms. The other is that our algorithm is a parallel algorithm running on the high-performance parallel platform Phoenix++ instead of a conventional serial genetic algorithm. Phoenix++ implements the MapReduce programming model that processes and generates large data sets with our parallel, distributed algorithm on a cluster. The experiments indicate that the convergence speed of GA algorithm is significantly faster after adding the simulated annealing algorithm on Phoenix++ platform.

1. Introduction

Genetic algorithms (GA) are a subclass of evolutionary algorithms that use the principle of evolution in order to search for solutions to optimization problems. Evolutionary algorithms are by their nature very good candidates for parallelization, and genetic algorithms do not make an exception [1]. Moreover, researchers have stated that genetic algorithms with larger populations tend to obtain better solutions with faster convergence [2]. These are the main reasons why they can benefit from a MapReduce implementation.

The high-performance distributed computing technologies or platforms can further increase the execution efficiency of these traditional genetic algorithms [3]. As we know, there are at least two ways to improve the performance of genetic algorithms. One way to enhance genetic algorithms is to reconstruct the original genetic algorithm by synthesizing itself with another algorithm to get global optimal solutions. The traditional genetic algorithms are easy to achieve local optimal solution because they tend to fall into the early constringency and low efficient search in the late stage of evolutionary [4]. A simulated annealing algorithm achieves nonoptimal solution at a certain probability to jump out of local optimal solution [5]. The combination of the genetic algorithm and the simulated annealing algorithm can overcome the harsh selection of parameters and improve the performance of algorithm. The other way to further increase the performance of a traditional serial genetic algorithm is to transform this algorithm into a parallel algorithm running on a high-performance distributed computing platform [6]. Based on the parallel characteristic of genetic algorithms, the parallelized genetic algorithms can generally be categorized

into 3 types: the master-slave type, the single population fine-grain type, and the multiple population coarse-grain type [7]. The difficulties for common programmers to parallelize a genetic algorithm include the lack of knowledge in computer architecture and details in computer network communication. Moreover, when the scale of problem grows, the proportion of the communicating time for procedure execution also grows rapidly.

Following this thought, this paper proposes a novel Phoenix++ based simulated annealing genetic algorithm that has two corresponding characteristics. The first is that the proposed algorithm is the synthesis of the conventional genetic algorithm and the simulated annealing algorithm. While most traditional genetic algorithms are easy to fall into local optimal solution, the simulated annealing algorithm accepts nonoptimal solution at a certain probability to jump out of local optimum and achieve the global optimal solution. This characteristic guarantees our proposed algorithm has a higher probability of getting the global optimal solution than traditional genetic algorithms. The second characteristic is that our algorithm is a parallel algorithm running on the high-performance parallel platform Phoenix++ other than a conventional serial genetic algorithm [8, 9]. Phoenix++ is a parallel platform that was developed by Stanford University to paralyze traditional serial genetic algorithms. Phoenix++ implements the MapReduce parallel model based on shared memory. So Phoenix++ can process and generate large data sets with our parallel algorithm on distributed clusters. By doing so, Phoenix++ makes up for the lack of communication between the genes in traditional Hadoop platform and improves the platform performance. Before our work, Jin et al. proposed an adapted model of MapReduce with an additional reduce step, in order to deal with iterative algorithms like evolutionary algorithms [10]. With two reduce processes, MRPGA enables MapReduce computing framework applying the iterative process of genetic algorithm [10, 11].

To summarize, we mainly make the following contributions.

(i) We increase the probability of obtaining the global optimal solution of the generic algorithm by synthesizing the generic algorithm with the simulated annealing genetic algorithm. We take advantage of the characteristic of the simulated annealing algorithm in jumping out of local optimal.

(ii) We parallelize the traditional generic algorithm with the platform Phoenix++ to overcome the deficiency of communication among two genes.

(iii) Phoenix++ implements a parallel MapReduce model that can further enhance the performance of our proposed algorithm.

The remainder of the paper is organized as follows. Section 2 presents necessary background of our work. Section 3 explains the main work about implementation of the simulated annealing genetic algorithm on Phoenix++. Section 4 illustrates the experiment of the proposed algorithm. Section 5 concludes this paper.

2. Background

2.1. MapReduce Model. Google engineers pioneered the implementation of the MapReduce model [12]. They applied hundreds of dedicated calculation algorithms to handle massive raw data, for document capture (similar to the Web crawler procedures), Web request log, and so forth. To handle these complex computing issues, Google engineers designed the new abstract MapReduce model to isolate data analysts from technical computing operations such as the parallel computing, fault tolerance, data distribution, and load balancing. By doing so, analysts can focus on handling data processing tasks. MapReduce provides users service of the parallel computation and the large scale distributed computing with a set of exposed interfaces.

Hadoop as open source computing platform is the implementation of MapReduce model. Computing nodes in Hadoop work independently and reduce time cost by depositing data locally to save time of data transmission. So Hadoop is weak in communication among computing nodes. One salient characteristic of genetic algorithms is that individuals can communicate with each other randomly in each generation. Obviously, it is hard for Hadoop to support this characteristic naturally. Accordingly, although Hadoop is of high performance to speedup ratio, it can also result in decreasing the probability of getting optimum solution. For this reason, we turn to study the parallelization of genetic algorithms on Phoenix++ platform.

2.2. Phoenix++ Parallel Computing Platform. With the rapid development of computer hardware, the applications of multicore and multi-CPU machine become the trend of current computing techniques. Many parallel platforms have developed to increase the system resource utilization and reduce programming complexity, for example, OpenMP and Phoenix/Phoenix++ [13]. A platform takes care of the internal thread management, provides operators a set of unified application programming interfaces (API), and improves execution performance of parallel programs. Phoenix++ is this sort of platform that was developed by Stanford University. As an updated parallel computing platform of Phoenix system, Phoenix++ implements both the MapReduce model and a shared memory model [14]. With multiple-thread technique, Phoenix++ can produce a large number of parallel Map tasks and reduce tasks to enhance the performance of parallel algorithms. Phoenix++ also applies a shared memory buffer to promote the exchange of data between two threads but avoid frequent data copy operation. Phoenix++ not only automatically manages threads but also executes dynamic task scheduling, data segmentation, and fault tolerance between two processor nodes. By doing so, Phoenix++ can provide efficient parallel support for multicore systems.

3. Implementation of the Simulated Annealing Genetic Algorithm on Phoenix++

3.1. Implementation of the Simulated Annealing Genetic Algorithm. The typical process on genetic algorithm includes

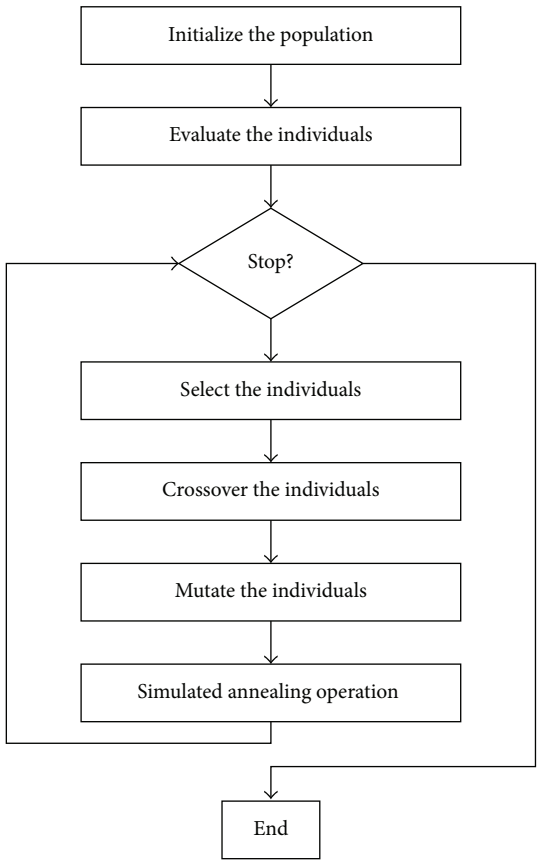

Figure 1: The whole process of simulated annealing genetic algorithm.

selection, crossover, mutation, and screening the best gene. We insert the simulated annealing algorithm into the genetic algorithm in stages of crossover and mutation. The whole process of simulated annealing genetic algorithm is shown as Figure 1.

3.1.1. Crossover. The individuals of population can pairwise couple using a random method. If the result of a two-two cross operation on any gene segments of two individuals is below the crossover probability (we usually take the empirical crossover probability as 0.8 in experiments), the fitness of two individual genes after crossover will be reappraised. We compare the fitness of new individual with its father individual. In this way, we can accept individuals of poor fitness as the next generation's new individual at a certain probability according to the simulated annealing model. This work is beneficial to reducing the possibility of falling into a local adaptive optimal.

3.1.2. Mutation. We usually make genes in chromosome to mutate with a certain probability (we usually take the empirical mutation probability as 0.2 in experiments). We use Guotao operator to reverse the mutation procedure. For the new individuals after a round of the mutation, it is necessary to reappraise their fitness. If the fitness of a new individual has been improved, this individual will be taken into the next generation. By contrast, if the fitness of a new individual has been declined, we accept the individual of poor fitness at a certain probability according to the simulated annealing model. In this way, it is beneficial to reduce the possibility of falling into local adaptive optimal. The basic principle of reverse mutation is to randomly select two gene positions and reverse all gene orders between these two genes.

3.1.3. Retain Elites. To guarantee that the best genes in each generation can continue the population reproduction, we need to screen the whole population of genes after each crossover. That is to make the next generation inherits the highest fitness in this generation. For a serial genetic algorithm in the following experiments, we take a traverse of the whole individuals of the population to find the best gene. However, when there are a large number of individuals in population, this operation may become a bottleneck to improve the performance of the serial program. So the proposed parallel algorithm will solve this problem with the parallel technique.

3.2. Design of Simulated Annealing Genetic Algorithm on Phoenix++. A genetic algorithm abstracts the problem space as a population that consists of a large number of individuals. The loop function of the genetic algorithm enables the population evolution and constantly searches for the best individual. Each generation of evolution involves the operations as chromosome crossover, gene mutation, evaluation of the individual genetic quality, and selection of the best individual. Chromosome crossover means banning arbitrary choice of two ancestral individuals and swapping gene segments of these two ancestral individuals. Genic mutation means randomly transforming any gene fragments of an ancestral individual. Evaluation of the individual genetic quality uses the evaluation scheme for individual. Selection of the best individual means selecting the individual that passes the best gene to the next generation from the ancestral individuals according to the evaluation parameters.

Since the probability of the gene mutation is relatively small, it persists for a short period of time. However, since the chromosome crossover's operation involves multiple individuals, the level of parallel is not high. If we insert a plurality of individuals into the input queue of Map phase and randomly take out two individuals to crossover in the map function, we have to solve two problems in the implementation of a genetic algorithm on Phoenix++. The first is that Phoenix++ provides the user with interface that gets an individual from the input queue as parameters to the map function at a time. Unfortunately, the map function can only operate on this individual. But we must operate on these two individuals at the same time to realize the crossover. So it is difficult to realize this implementation if we cannot revise the API interface of the Phoenix++. The second problem is that Phoenix++ for all of the user's API interface functions is the constant function. So the API interface function is not able to change the class of any member variables. While the crossover needs to change the order of individual gene, it is difficult

Table 1: The output key value pairs in the three execution phases of the proposed parallel genetic algorithm.

The key value pair		Explanation
Key1: integer		The ID number of the individual in the population.
Value1: double		The evaluation value of the individual.
Key2: integer	→	The ID number of the best individual in all individuals of the input queue in reduce phase.
Value2: double		The evaluation value of the best individual.
Key3: integer		The ID number of the global optimal individual.
Value3: double		The evaluation value of the global optimal individual.

to do this task if we cannot revise the API interface of the Phoenix++.

In this paper, we put chromosome crossover and genic mutation out of the MapReduce framework. For a larger population, compared with the time of evaluation of individual genetic quality and selection of optimal individual, the time of chromosome crossover and genic mutation is a correspondingly smaller proportion. Consequently, the parallelization of the genetic algorithm mainly reflects the individual genetic quality's evaluation of the operation and the selection of optimal individual operation.

3.2.1. Genetic Algorithm Framework on MapReduce. Using MapReduce for running genetic algorithms has been a subject of research in the last few years. Verma et al. proposed the system framework of MRPGA in 2009 [15]. This system consisted of a master, a number of mappers, and multilevel reducers. The master is mainly responsible for the distribution of tasks and scheduling work. User enables the Mapper according to the map rules. Each reducer is responsible for the execution of reduce task. Di Geronimo et al. presented a parallel genetic algorithm for the automatic generation of test suites. The solution is based on Hadoop MapReduce since it is well supported to work also in the cloud and on graphic cards, thus being an ideal candidate for high scalable parallelization of genetic algorithms [16]. Narayanan and Krishnakumar proposed a simple genetic algorithm with optimum population size, mutation rate, and selection strategy which is parallelized with MapReduce architecture for finding the optimal conformation of a protein using the two-dimensional square HP model [17].

In our work, we introduce the mechanism of simulated annealing based on the system framework of MRPGA and implement simulated annealing genetic algorithm on Phoenix++. To meet the evolving of population, our prototype system introduces a coordination mechanism to manage the system operations as a whole.

3.2.2. The Key Value Pairs of Simulated Annealing Genetic Algorithms on Phoenix++. The execution process of our proposed parallel genetic algorithm is divided into three phases: "map," the first phase of "reduce" and the second phase of "reduce." Table 1 exhibits the output key value pairs in the three phases.

First, every individual is distinguished by its unique ID number. After the map phase, all individuals get together by the same ID number. But these results of the map phase are deposited in the current thread's processor cache. So these results are identified as a "local" data. For the standard MapReduce implementation, all individuals with the same ID number gather together as the reduce phase of the input. And the implementation of the generic algorithm on Phoenix++ is that Phoenix++ provides three optional intermediate key values of storage mode for the application: the variable length hash table implementation, the fixed length hash table implementation, and the fixed length array implementation. The combiner container is attached behind to the hash table or the array.

In the first phase of "reduce," the implementation of the content is to select the best individual from its input queue and put it into the output queue through the evaluation function. In the second phase of "reduce," Phoenix++ merges the output queue of the first "reduce" into a queue and finds the global optimal individual by sorting the value in the queue and gets the global optimal individual as the final result.

```
Function mapper(p, out)
  /* perform evaluation */
  x = Evaluation(p)
  /* Find out the individual p's ID in a population */
  id = find_id(p)
  /* Submit intermediate results */
Emit(out, id, x)
```

Algorithm 1

3.2.3. The Map Phase. Each individual of the Map operation in each generation cycle will call a map function. As the input of map function, datatype int &*p* represents an individual and map container &out represents the container object of the storage map output. And the output of the map function includes the ID number of the individuals in the population and the individual evaluation value. The Pseudo-code of the map function is represented as in Algorithm 1.

When there is a big population size, the time of paired chromatids and genetic mutation is substantially less than the time on estimating individual genes and selecting optimal individual genes. To compensate the cost of communication and computation on the execution of genetic algorithm, Map component in MapReduce optimizes the operations on estimating individual genes and selecting optimal individual genes, that is, the estimation function evaluation(int k)

```
double Evaluation(int k)
{
   double fit;
   fit = 1.0/group_dist[k];
   return fit;
}
void dist()
{
   int i, j;
   for (i = 0; i < Pop_Num; ++i) {
      group_dist[i] = 0.0;
      for (j = 0; j < City_Num − 1; j++) {
           group_dist[i] += distan[Sumgen[i].gen[j]][Sumgen[i].gen[j + 1]];
      }
           group_dist[i] += distan[Sumgen[i].gen[City_Num − 1]][Sumgen[i].gen[0]];
   } //The total distance Pop_Num populations are stored in group_dist[], waiting the calculation of the evaluation function.
}
String find_id(int Sumgen)
{
   int i, j;
   double tmpfit;
   for (i = 1; i < Pop_Num; i++) {
        tmpfit = (double)(rand()%1000000 + 1)/1000000.0;    //1000000
        if ((tmpfit − Sumgen[1].refit) < eps) {
           tmpgen[i] = Sumgen[1];
        }
        else {
           for (j = 1; j < Pop_Num; j++) {
              if ((tmpfit − Sumgen[j].refit) < eps) {
                 tmpgen[i] = Sumgen[j];
                 for (int k = 0; k < City_Num; ++k)
                    tmpgen[i].gen[k] = Sumgen[j].gen[k];
                 tmpgen[i].id = i;
                 break;
              }
           }
        }
     } //Using a random function to re-select a population
     for (i = 1;  i < Pop_Num; i++)
        Sumgen[i] = tmpgen[i];
     Return tmpgen[i].id;
}
```

Algorithm 2

and the select function find_id(int Sumgen), as shown in Algorithm 2.

We insert all the outputs of the "map" phase into the local output queue and take all results that are in accordance with ID as an index into a fixed length hash container. The operation find_id() can record the ID by adding a ID domain in a structure definition of individual body. By doing so, we can avoid re-putting individual p back into the population search and time waste.

3.2.4. The First Reduce Phase. The output results of "map" phase can be integrated and redistributed into each queue as the input of the first phase of "reduce" phase through the container and combiner. The first phase of "reduce" aims to find out the optimal individual from the input queue and insert the ID number and the assessment value of optimal individual as the output of this phase into the output queue. The first phase of "reduce" also aims to find out the local optimal individual. The corresponding Pseudo-code of this procedure can be described as a function "reduce," as shown in Algorithm 3.

3.2.5. The Second Phase of "Reduce". The second phase of "reduce" is to complete the selection of a global optimal individual to make the output of the second phase of "reduce" for each reduce queue as the input of second "reduce" queue input. For the second phase of "reduce," the Phoenix++ provides a merge mechanism that can rank all the outputs

```
function reducer(key, values, out)
    eval = INF
  foreach value in value_list
    eval = min(value) or max(value)
/* To find out the best individual evaluation value from value queue */
    Emit(key, eval)
```

Algorithm 3

```
function final_reducer(key, value)
  sort()
  /* To rank all the input (using the ranking mechanism inside phoenix++) */
  Emit(id, val);
  /* submint global optimum individuals */
```

Algorithm 4

of the first phase of "reduce" in global according to a certain value. Consequently, this algorithm can easily achieve the global optimal individual search by using this merge mechanism. The corresponding Pseudo-code can be presented as function "final_reducer," as shown in Algorithm 4.

4. Algorithm Experiment

4.1. Experimental Setup. We setup an experiment that applies the proposed parallel distributed algorithm to the classical TSP problem that solves combinatorial optimization issues. The experiment runs the serial simulated annealing genetic algorithm and the simulated annealing genetic algorithms on Phoenix++, respectively. The experiment takes the CPU as Intel (R) Xeon (R) E5630 @ 2.53 GHz, 16 Cores; the cache size 12288 KB/core Memory, 16 GB; and the population size as 3001; the genetic Algebra as 10000. The experiment uses TSP data set at http://www.tsp.gatech.edu/index.html, beta site test data. The number of test of the city is 38. Data are deposited as text that can be exported in the city number/horizontal/vertical coordinate with information, as described in Table 2.

4.2. Experimental Result. After several round experiments, the final path is obtained as follows: $0 \to 9 \to 13 \to 20 \to 28 \to 29 \to 31 \to 34 \to 36 \to 37 \to 32 \to 33 \to 35 \to 30 \to 26 \to 27 \to 23 \to 21 \to 24 \to 25 \to 22 \to 19 \to 14 \to 12 \to 15 \to 16 \to 17 \to 18 \to 10 \to 11 \to 8 \to 7 \to 6 \to 5 \to 4 \to 2 \to 3 \to 1 \to 0$. The length of the path is 6659.431533 that is only 3.431533 times length than the optimal beta results 6656 in TSP data set as http://www.tsp.gatech.edu/index.html and is the same as the optimal results of the serial programs. Besides the optimal global results, the proposed algorithm also aims at improving the execution performance of parallel algorithm. Therefore, we compare our parallel algorithm with its serial correspondence at the same computing environment.

Table 3 exhibits the results of the algorithm executions after 5 rounds, respectively.

Table 2: Details information of experimental data set.

City number	Horizontal ordinate	Vertical ordinate
1	11003.611100	42102.500000
2	11108.611100	42373.888900
3	11133.333300	42885.833300
4	11155.833300	42712.500000
5	11183.333300	42933.333300
6	11297.500000	42853.333300
7	11310.277800	42929.444400
8	11416.666700	42983.333300
9	11423.888900	43000.277800
10	11438.333300	42057.222200
11	11461.111100	43252.777800
12	11485.555600	43187.222200
13	11503.055600	42855.277800
⋮		
20	11690.555600	42686.666700
27	11963.055600	43290.555600
28	11973.055600	43026.111100
29	12058.333300	42195.555600
30	12149.444400	42477.500000
31	12286.944400	43355.555600
32	12300.000000	42433.333300
33	12355.833300	43156.388900
34	12363.333300	43189.166700
35	12372.777800	42711.388900
36	12386.666700	43334.722200
37	12421.666700	42895.555600
38	12645.000000	42973.333300

According to Table 3, the speed-up ratio of the serial program to the parallel program is $S_p = T_l/T_p = 2.74$. The standard efficiency of the parallel system is $E_p = S_p/_p = 2.74/16 = 0.17$. While Figure 2 exhibits the convergence

Table 3: The results of the algorithm executions after 5 rounds, respectively.

Times	Serial program	Parallel program
1	11″9.663′	4″8.599′
2	11″4.912′	4″7.484′
3	11″17.875′	4″13.968′
4	11″27.801′	4″4.421′
5	11″13.150′	4″6.222′
Average running time	11″14.6802′	4″6.1368′

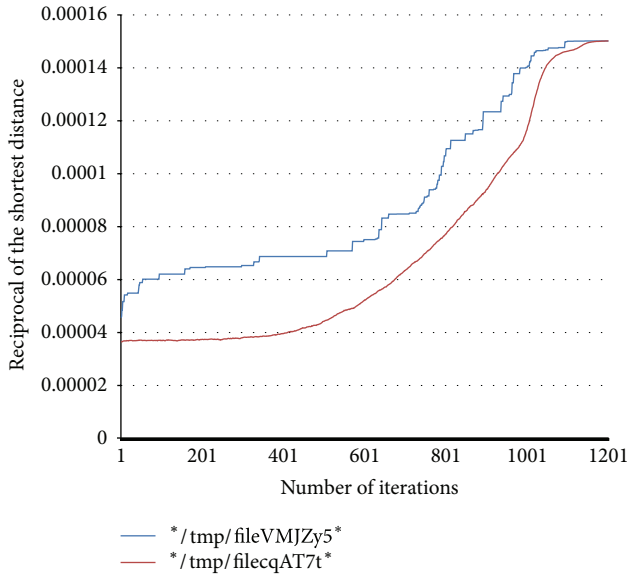

Figure 2: Renderings in the case without emerging of the simulated annealing algorithm.

curve of the GA algorithm without simulated annealing, Figure 3 illustrates the first curve after adding the simulated annealing algorithm. According to the comparison of these curves, we can find that the convergence speed of the GA algorithm significantly increases after emerging the simulated annealing algorithm into the proposed algorithm in this paper.

We further deploy the same program on the Hadoop platform that consists of 10 PCs and get the optimal result as 6549.43. Compared with traditional parallel computing frameworks such as MPI OPMAP, MapReduce simplifies the design of communication within a parallel genetic algorithm and enhances the robustness. Thus it cannot interrupt executing computing tasks due to the computing equipment failure. This characteristic is very important in the execution of complex tasks. So compared with Hadoop, the Phoenix++ is more efficient in the parallel communication because it adopts the way of shared memory to implement MapReduce.

5. Conclusion

This paper proposes a simulated annealing genetic algorithm based on Phoenix++ and the shared memory, which

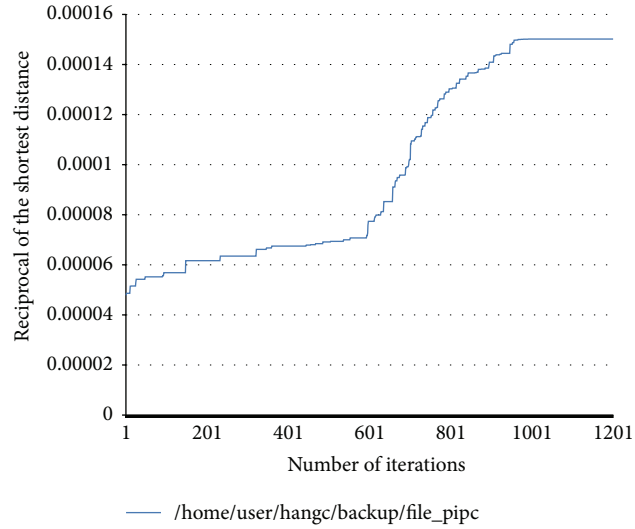

Figure 3: Renderings in the case of emerging of simulated annealing algorithm.

synthesizes the advantages of both the simulated annealing algorithm and the genetic algorithm. Phoenix++ provides a unified programming interface (API) for users. Users only need to call its API without taking consideration of the management of its internal thread. It can easily achieve the program parallelization which greatly improves the efficiency of programming. The proposed algorithm also takes advantage of the parallel distributed computing architecture and meets the natural parallel characteristic of swarm intelligence algorithm. So this proposed algorithm is suitable for handling the issue with the large scale data optimization and modeling. The proposed algorithm also keeps the best individual in each subgroup. By keeping the excellent individual evolutionary stability, our algorithm accelerates the evolution and avoids the phenomenon of premature convergence during the single species evolutionary process. Our algorithm exhibits its excellent performance in dealing with the classical combinatorial optimization problem TSP.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was partially supported by grant from the Hubei Provincial Department of Education scientific research programs for youth project (no. Q20133003), the Natural Science Foundation of Hubei Province (no. 2014CFB568), the grants of the National Natural Science Foundation of China (61070013 and U1135005), and Guangxi Key Laboratory of Trusted Software (no. kx201421).

References

[1] E. Apostol, I. Băluţă, A. Gorgoi, and V. Cristea, "A parallel genetic algorithm framework for cloud computing applications," in *Adaptive Resource Management and Scheduling for Cloud Computing*, vol. 8907 of *Lecture Notes in Computer Science*, pp. 113–127, Springer International Publishing, 2014.

[2] D.-W. Huang and J. Lin, "Scaling populations of a genetic algorithm for job shop scheduling problems using mapreduce," in *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom '10)*, pp. 780–785, December 2010.

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

[4] P. Guo, X. Wang, and Y. Han, "The enhanced genetic algorithms for the optimization design," in *Proceedings of the 3rd International Conference on BioMedical Engineering and Informatics (BMEI '10)*, pp. 2990–2994, IEEE, Yantai, China, October 2010.

[5] C. Li and D. C. Coster, "A simulated annealing algorithm for D-optimal design for 2-way and 3-way polynomial regression with correlated observations," *Journal of Applied Mathematics*, vol. 2014, Article ID 746914, 6 pages, 2014.

[6] V. Roberge, M. Tarbouchi, and G. Labonté, "Comparison of parallel genetic algorithm and particle swarm optimization for real-time UAV path planning," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 132–141, 2013.

[7] D. M. Pierre, N. Zakaria, and A. J. Pal, "Master-slave parallel vector-evaluated genetic algorithm for unmanned aerial vehicle's path planning," in *Proceedings of the 11th International Conference on Hybrid Intelligent Systems (HIS '11)*, pp. 517–521, Melacca, Malaysia, December 2011.

[8] C. Cao, F. Song, and D. G. Waddington, "Implementing a high-performance recommendation system using Phoenix++," in *Proceedings of the 8th International Conference for Internet Technology and Secured Transactions (ICITST '13)*, pp. 252–257, London, UK, March 2013.

[9] Apache Software Foundation, http://en.wikipedia.org/wiki/Apache_Software_Foundation.

[10] C. Jin, C. Vecchiola, and R. Buyya, "MRPGA: an extension of MapReduce for parallelizing Genetic Algorithms," in *Proceedings of the 4th IEEE International Conference on eScience (eScience '08)*, pp. 214–221, IEEE, December 2008.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 13–24, February 2007.

[12] MapReduce, Wikipedia, http://en.wikipedia.org/wiki/MapReduce.

[13] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: modular MapReduce for shared-memory systems," in *Proceedings of the 2nd International Workshop on MapReduce and Its Applications*, pp. 9–16, ACM, June 2011.

[14] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: scalable mapreduce on a large-scale shared-memory system," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '09)*, pp. 198–207, Austin, Tex, USA, October 2009.

[15] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell, "Scaling genetic algorithms using MapReduce," in *Proceedings of the 9th International Conference on Intelligent Systems Design and Applications (ISDA '09)*, pp. 13–18, December 2009.

[16] 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 5th IEEE International Conference on Software Testing, Verification and Validation (ICST '12)*, pp. 785–793, Montreal, Canada, April 2012.

[17] A. Narayanan and U. Krishnakumar, "An enhanced MapReduce framework for solving protein folding problem using a parallel genetic algorithm," in *ICT and Critical Infrastructure: Proceedings of the 48th Annual Convention of Computer Society of India—Vol I*, vol. 248 of *Advances in Intelligent Systems and Computing*, pp. 241–250, Springer, Berlin, Germany, 2014.