Parallel and Distributed Architecture of Genetic Algorithm on Apache Hadoop and Spark

Hau-Chun Lu[a], F.J. Hwang[b], Yao-Huei Huang[c,a,*]

[a]*Department of Information Management*,
Fu Jen Catholic University, *New Taipei City*, *Taiwan*
[b]*School of Mathematical and Physical Sciences*,
University of Technology Sydney, *Ultimo*, *Australia*
[c]*School of Economics and Management*,
Southwest Jiaotong University, *Chengdu*, *China*
E-mail: haoclu@gmail.com; feng-jang.hwang@uts.edu.au; yaohuei.huang@gmail.com

ABSTRACT

The genetic algorithm (GA), one of the best-known metaheuristic algorithms, has been extensively utilized in various fields of management science, operational research, and industrial engineering. The efficiency of GAs in solving large-scale optimization problems would be enhanced if the iterative processes required by the genetic operators can be implemented in a parallel and distributed computing architecture. Apache Hadoop has recently been one of the most popular systems for distributed storage and parallel processing of big data. By integrating highly the GA into Apache Hadoop, this study proposes an advanced GA parallel and distributed computing architecture that achieves the effectiveness and efficiency of GA evolution. The developed computing framework is characterized by the sophisticated mechanism of dispatching the GA core operators into Apache Hadoop and fits well the cloud computing model. The presented GA parallelization architecture outperforms the state-of-the-art reference architectures according to the computational experiments where the testing instances of travelling salesman problems are employed. Our numerical experiments also demonstrate that the proposed architecture can readily be extended to Apache Spark.

Keyword: Genetic algorithm; parallel and distributed computing; Apache Hadoop; travelling salesman problems; Apache Spark

1. Introduction

The global optimization of real-world problems in management science and operational research, e.g. the container loading problems (Tsai et al., 2015; Huang et al., 2016; Huang and Hwang, 2017), vehicle routing problems (Grefenstette et al., 1985; Victor et al., 2013; Teobaldo et al., 2018), cutting stock problems (Dyckhoff, 1990; Lu and Huang, 2015; Wuttke et al., 2018), production scheduling problems (Murata et al., 1996; Pongchairerks and Kachitvichyanukul, 2009; Maenhout and Vanhoucke, 2018), are difficult to achieve due to their nature of NP-hardness (Scheithauer, 1992). Considering those optimization problems on large scale, it is relatively realistic to produce a suboptimal or quality feasible solution rather than a global optimum. It thus is popular in practice to develop heuristic/metaheuristic solution techniques that are capable of yielding acceptable solutions within a reasonable time. One of the most common evolutionary algorithms is the genetic algorithm (GA) (Holland, 1975), which has been extensively utilized for tackling nonlinear optimization problems in various fields (Gallagher and Sambridge, 1994; Hartmann, 2002; Lu and Huang, 2015; Ren et al., 2018). To enhance the computational performance of GAs for the large-scale NP-hard problems, the parallelization of GA in a distributed computing architecture has been proposed in this study.

1.1 Three baseline models of GA parallel and distributed architecture

Considering the stochastic solution searching mechanism of GAs, Luque and Alba (2011) developed a parallel and distributed architecture integrated with the GA for exploring solution spaces. Referring to Luque and Alba (2011), the parallel and distributed architecture for the GA processing can be classified into the following three models:

(i) **Master-slave model** (also called global parallelization model): In the master-slave model (Martino et al., 2012; Geronimo et al., 2012), there is a master node to manage all sub populations and distribute individuals among slave nodes. Subsequently, the fitness values of individuals are calculated in the corresponding slave nodes. Actually, the redesign of GA framework is not required in this model since the individual fitness evaluation operator can be independent from one another in the population.

(ii) **Distributed model** (also called coarse-grained parallelization model or island model): In the distributed model (Ferrucci et al., 2013), a population is divided into several subpopulations located in several islands and the GA runs independently in each island. Since each island only contains partial individuals of the population, those islands periodically exchange information by migrating some individuals for injecting diversity

into the converging population (Herrera and Lozano, 2000; Yu and Zhang, 2006; Melab and Talbi, 2010; Yu et al., 2011). The model can perform all operators of GA in the parallel and distributed computing such that the different islands can explore different portions of the search space periodically.

(iii) **Cellular model** (also called fine-grained parallelization model or grid model): The cellular model (Arora and Chana, 2014; Camacho, 2015; Gong et al., 2015) is to structure a population into neighborhoods and put a part of individuals in a node (i.e. grid). The GA is performed in parallel computing to evaluate the fitness value of each chromosome and apply locally GA operators (i.e. selection, crossover and mutation operators) to the small adjacent neighboring. On the basis of the massively parallel architecture, the model can significantly speed up the evaluation of all chromosomes. However, it requires a massive clustering system to deal with this model. Vidal and Alba (2010) demonstrated that this model can be implemented based on the graphical processing units (GPUs) for fast computations when the operators are defined as matrix operations.

1.2 GA parallelization based on MapReduce

MapReduce is a software framework for distributed computing proposed by Dean and Ghemawat (2008) in the Google system. The framework is an associated implementation for processing large-scale data sets using a distributed algorithm on a cluster.

In the master-slave model implemented on the MapReduce (Martino et al., 2012; Geronimo et al., 2012), the evaluation operator of GA can be delegated by utilizing multiple mappers for evaluating the fitness value of each chromosome in the parallel and distributed environment. Subsequently, the single reducer is to collect the results and perform the other GA operators including the selection, crossover, and mutation operators. One generation means an execution round on the MapReduce, and the whole computation is a sequence of executions. The master-slave model performs the evaluation operator in the parallel computing with multiple mappers, while it does not consider the parallelism of other GA operators (i.e. selection, crossover, and mutation operators), all of which are run in the reducers, and may cause some scalability issues.

For the distributed model on the MapReduce, Ferrucci et al. (2013) designed a partitioner to assign each island to a reducer. It aims to perform the other operators in parallel computing after running the evaluation operator in the mappers. Since the model has much more reducers, it is more efficient than the master-slave model. These operators are performed in the

incomplete situation such that it often causes the early convergence.

As for the cellular model, Arora and Chana (2014) and Camacho (2015) integrated the clustering technique with the GA to solve the large-scale problems in a massively parallel computing environment. Gong et al. (2015) proposed a cellular model that is slightly modified from the clustering technique of the distributed model with a partitioner using a pseudorandom function on the MapReduce. For the cellular model, individuals only interact with their neighbors, which could cause the lack of population diversity. Since the selection and crossover operators are not able to generate solutions outside the limited neighborhood, it may fall into a local optimum to incur the early convergence. Additionally, the method requires a massive powerful clustering system to deal with these operators.

Hadoop MapReduce has recently been a complete framework for processing vast amounts of data in a parallel and distributed environment and one of the most popular distributed computing architectures. There exists a well-designed structure for large-scale data processing in the distributed servers of Hadoop Distributed File System (HDFS), with which the users can easily develop a parallel computing program for large-scale optimization problems. The review of GA parallelization with MapReduce (Sachar and Khullar, 2016) indicates that Hadoop MapReduce platform has obvious advantages over others. The detailed architectures of Apache Hadoop and its extension, i.e. Apache Spark, are discussed in the next subsection.

1.3 Architectures of Apache Hadoop and Spark

The advent of big data makes the formulation of optimization problems closer to the real world while the solution finding and relevant computations become more challenging. In order to tackle the large-scale optimization problems by the metaheuristics such as the GA, it is necessary to implement the iterative procedures of GA in the distributed computing environment so that the computation time can be reduced. Apache Hadoop (2006) and Apache Spark (2017) have been the popular distributed computing architectures for the iterative algorithms such as the GA. The characteristics of Hadoop and Spark are: (i) Hadoop is designed for efficiently dealing with large-scale problems on clusters of hardware with a scalable and fault-tolerant framework; (ii) Spark, without a file system, is a cluster computing tool running in memory such that its speed is faster than Hadoop. The studies of GA parallelization using Apache Hadoop/Spark can be found in the literature.

Combining a distributed computing framework of Hadoop MapReduce with the GA, Verma et al. (2009) employs the mappers to perform the iterative procedures with partial

chromosomes, and the results produced by the mappers are transmitted to the reducers for the further operations such as selection, crossover, and mutation. Then the new population will be generated in the next generation. However, the chromosomes in the reducers are produced from the partial population. Since the evolution is limited in an incomplete population, the yielded solutions fall into a local optimum, thus causing the premature convergence. Kečo and Subasi (2012) proposed a distributed GA approach based on the mapper and reducer of Apache Hadoop, where each mapper performs the complete operational procedure of GA and then delivers the result the reducer. Subsequently, it is determined by the reducer whether the GA operations should be iterated. Unfortunately, this approach cannot effectively divide a whole operational procedure into serval sub-procedures for parallel computing, which results in a long solving time for large-scale problems. The parallel and distributed architecture for GAs on the basis of the mapper and reducer presented by Geronimo et al. (2012) has the similar disadvantage. Qi et al. (2016) designed the parallel GA procedure with the scheme of Apache Spark, which contains a two-phase parallelization algorithm including fitness evaluation parallelization and genetic operational parallelization.

It is found in our preliminary research that the effectiveness and efficiency of the GA parallelization can be enhanced if the evolution operations of GA can be highly integrated into Apache Hadoop or Spark. This study proposes a parallel and distributed architecture of GA on Apache Hadoop and Spark that outperforms those in the existing literature. The presented GA parallelization scheme enhances the utilization of the HDFS of Apache Hadoop by dispatching the core operators of GA with a relatively efficient parallelization mechanism, which can reduce the idle times of processing jobs in the HDFS. The similar parallelization mechanism can be applied to the Resilient Distributed Dataset (RDD) in Apache Spark, and the developed scheme can be readily extended to Apache Spark.

The remainder of this paper is organized as follows. Section 2 discusses the state-of-the-art architectures of GA parallelization on Apache Hadoop. The proposed GA parallelization architecture is presented in Section 3. The computational experiments are provided in Section 4. The extension of the proposed GA parallelization to Apache Spark is discussed in Section 5. The concluding remarks are given in Section 6.

2. Review of GA parallelization on Apache Hadoop

Apache Hadoop (2006) with MapReduce is a distributed architecture on clusters for dealing with large-scale data sets. In the environment of MapReduce, the developers can utilize the distributed computing program for solving large-scale problems using the map and reduce functions. Also, the MapReduce is also a simple programming model, where a scalable, flexible, and fault-tolerant algorithm for computations can be easily established. There are various applications using the Apache Hadoop with MapReduce to solve large-scale problems (Almeer, 2012; Dittrich and Quiané-Ruiz, 2012; Gu et al., 2014).

2.1 Procedure of MapReduce on Apache Hadoop

The procedure of MapReduce is illustrated in Figure 1. The MapReduce plays an important role to transform the input data sets into the form of <key, value> for receiving and sending. The first element, i.e. key, indicates an index of the corresponding value and is defined by the user. The second element, i.e. value, can be a set of data sets, a set of numbers, a set of specific sequential values, a chromosome in the GA, etc. There exists a mapper, i.e. function *map*, for converting original 2-tuple <key, value> into an intermediate 2-tuple. Afterward, the Hadoop merges, sorts, shuffles, and partitions the intermediate 2-tuple, and then each processed 2-tuple is transformed into the form of <key, value list> as the input of the reducer, i.e. function *reduce*. Note that the transformed key is also a user-defined index, and the value list refers to the multiple sets corresponding to the same index. Finally, the reducer outputs the result to the HDFS. Because of the required subsequent transformation from <key, value> into <key, value list> for the reducer on the MapReduce, a procedure doing appropriately the dataset dispatching in the form of <key, value> in the mapper on the basis of HDFS is the key to the efficiency of Apache Hadoop in solving the large-scale problems.

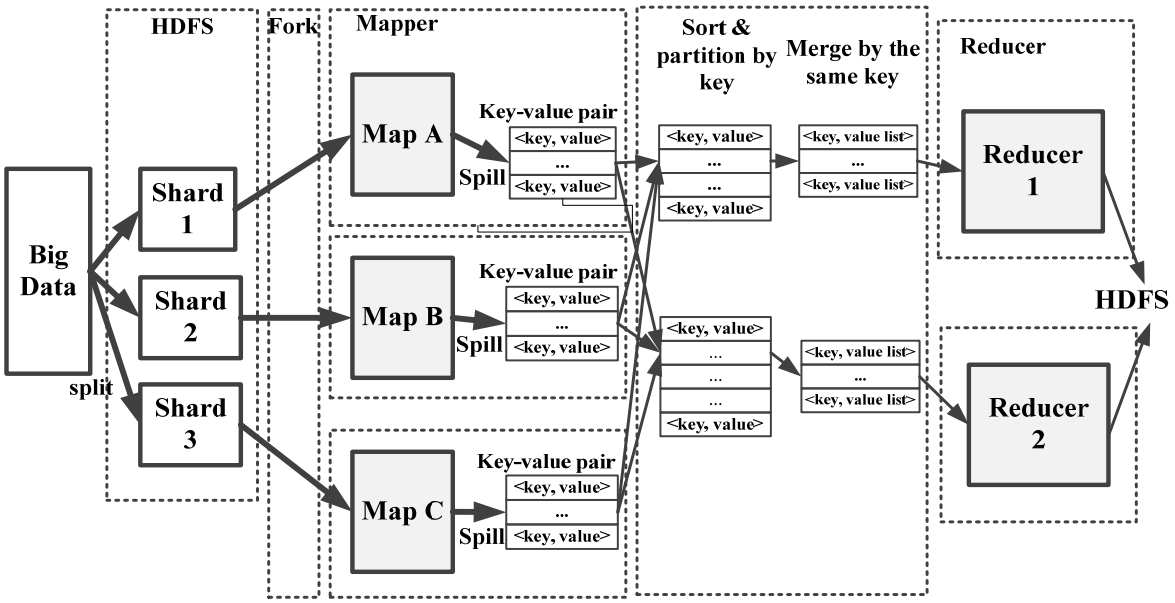


Figure 1. Procedure of MapReduce on Hadoop

2.2 Operational procedure of the GA

Gen and Cheng (2000) implies that the operational procedure of the GA can be run in a parallel and distributed system. The concept of GA is demonstrated in Table 1.

Table 1. Operational procedure of the GA

Begin
  $t = 1$
  Initialize population $P(t)$;
  Evaluate the fitness value of each chromosome in $P(t)$;
  Arrange chromosomes in $P(t)$ in non-increasing order of the fitness value.
  do begin
    Create new offspring $Q(t)$ from $P(t)$ by *Crossover*, and modify the new offspring $Q(t)$ by *Mutation*;
    Evaluate the fitness value of each chromosome in $Q(t)$;
    Sort all chromosomes in $P(t)$ and $Q(t)$ into non-increasing order according to their fitness values;
    Generate the next population $P(t+1)$ from $P(t)$ and $Q(t)$ by *Selection*;
    $t = t + 1$.
  end while (Checking the termination condition*)
end

*Note that the termination condition can be set using a limit to the number of iterations, the CPU time, or a threshold of objective value.

2.3 GA parallel architecture of Verma et al. (2009)

Verma et al. (2009) proposed a GA distributed computing architecture on Apache Hadoop

based on the selecto-recombinative GA (Goldberg, 1989; 2013). The approach resolves the issue of a long computational time when the problem size in GA becomes large. The implementation of GA on Hadoop proposed by Verma et al. (2009) is shown in Figure 2. The main procedure starts by loading data sets and then sets up the Hadoop environment according to a set of parameters. Subsequently, the Hadoop will be launched to run the operators of GA. The mapper only corresponds to the evaluation operator while the reducer corresponds to selection, crossover, and mutation operators. These four operators are designed in the main procedure. The Hadoop controls the mappers and reducers automatically, while different procedures of the mapper and reducer can be designed by various requests. After satisfying the stop condition, the iterative procedure of mappers and reducers will be stopped and the incumbent solution will be output.

There are two functions, i.e. *map*() and *reduce*(), used in the mapper and reducer, respectively. The two functions generate an initial population and send it to the HDFS. Then the initial population is divided into serval shards, depending on the size of the input dataset. The shards are forked onto different mappers for the fitness evaluation in parallel. Each mapper can evaluate (denoted by E in Figure 2) the fitness of each chromosome in the shard and then output the 2-tuple <key, value>=<fitness, chromosome> to Hadoop. Afterward, any results with the same key in the mapper will be merged in the form of <key, value list>, where the value list indicates the multiple chromosomes corresponding to the same index key (i.e. fitness). Then the 2-tuple <key, value list> will be send to different reducers for selection (denoted by S in Figure 2) and crossover/mutation (denoted by CM in Figure 2) operations.

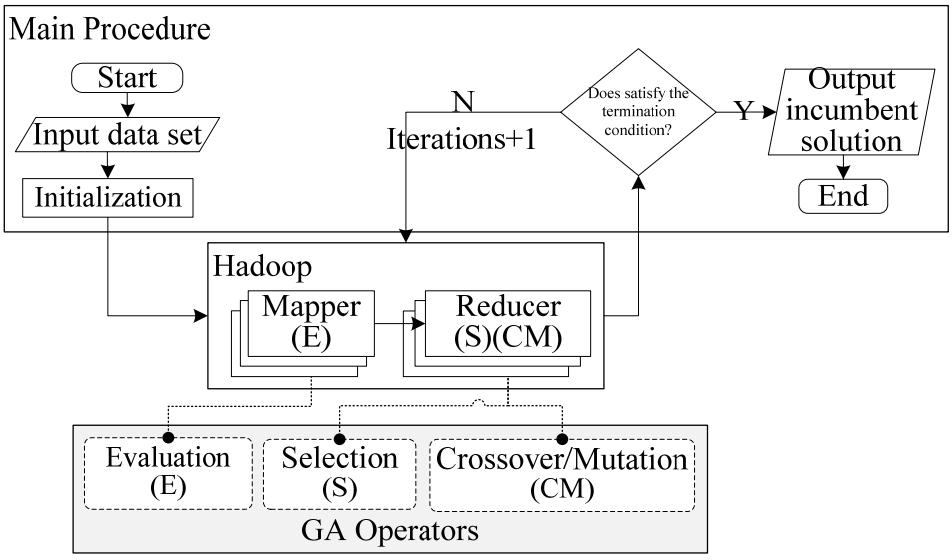


Figure 2. The GA parallelization on Hadoop developed by Verma et al. (2009)

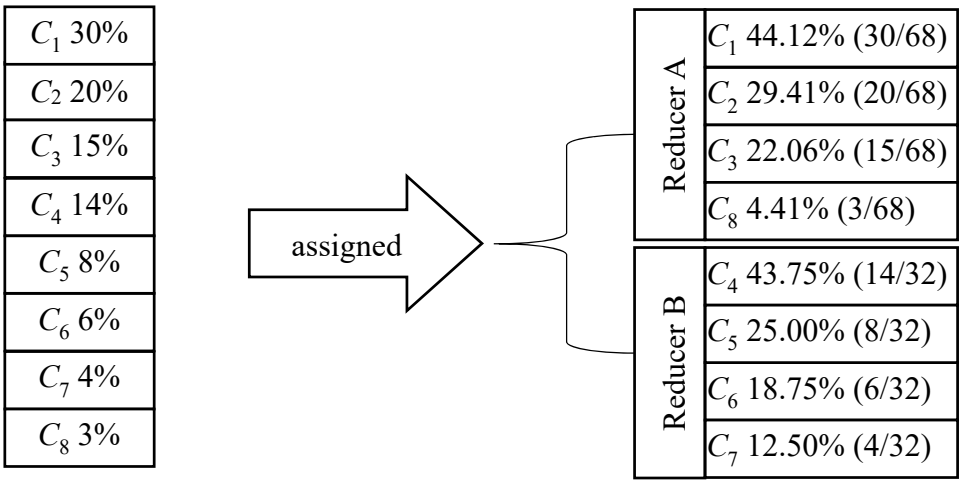
However, the architecture of Verma et al. (2009) may cause the following concerns:

(i) The evolution of chromosomes in the reducer never considers the whole population, i.e., each reducer only performs the operations with partial populations, so the evolution pool would not be large enough for maintaining genetic diversity. It thus would lead to an early convergence by ending up with a local optimum.

(ii) The GA roulette wheel selection mechanism could have the better chromosomes eliminated but the worse chromosomes reserved in the next generation. The details will be discussed with an example in the next paragraph as well as Figure 3.

(iii) The whole procedure would not be directly terminated even when an optimal solution is obtained in the mapper, i.e. the operations of selection and crossover/mutation in the reducer could be redundantly performed.

The abovementioned disadvantage of the GA roulette wheel selection is discussed. Consider eight chromosomes in the whole population as shown in Figure 3. The probabilities of all the chromosomes in the roulette wheel selection operator are 30%, 20%, 15%, 14%, 8%, 6%, 4%, and 3%, respectively, as shown in Figure 3(a). After the partitioner randomly assigns the chromosomes to the reducers A and B, the probability of each chromosome in the two partial populations is recalculated. Figure 3(b) demonstrates that the two partitioned sets (C_1, C_2, C_3, C_8) and (C_4, C_5, C_6, C_7) are assigned to the reducers A and B, respectively. The probabilities are (44.12%, 29.41%, 22.06%, 4.41%) and (43.75%, 25.00%, 18.75%, 12.50%) for (C_1, C_2, C_3, C_8) and (C_4, C_5, C_6, C_7), respectively. The selection probability of C_3 is less than that of C_4 or C_5 after the partition while the opposite situation exists in the whole population. Therefore, this approach could miss significant chromosomes.



(a) The whole population

(b) Two partial populations after the assignment to the reducers

Figure 3. Probabilities in the roulette wheel selection operator

2.4 GA parallel architecture of Kečo and Subasi (2012)

The GA parallel architecture presented by Kečo and Subasi (2012), also on Apache Hadoop, is shown in Figure 4. Note that the function *map*() is used in the mapper stage while the reducer stage is void.

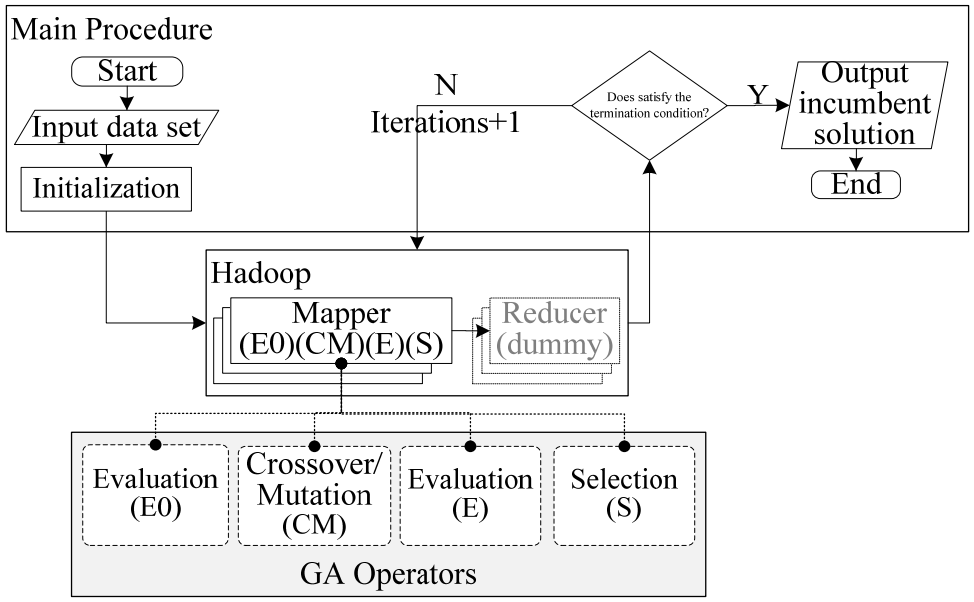


Figure 4. The GA parallelization on Hadoop developed by Kečo and Subasi (2012)

In the architecture of Kečo and Subasi (2012), the main procedure firstly configures the Hadoop environment and runs the initialization. In the first iteration, each mapper performs all

the GA operators, including initial evaluation (E0), crossover/mutation (CM), evaluation (E), and selection(S), where the setting of relevant parameters is required. After the first iteration, each mapper performs all operators but E0. The evaluation operator E0 calculates the fitness values of the initialized chromosomes, while E calculates the fitness values of those generated by the crossover/mutation operators. Then the results are delivered back to the main procedure for checking the stop condition. If the stop condition is not satisfied, then the Hadoop will repeat all operators except E0. Otherwise, the incumbent solution will be output.

The main disadvantage of the architecture is that the parallelization effectiveness exists only in the GA parameter tuning but not in the GA operator processing. Consider an example of the GA procedure in the mapper. Assume that the running times of these four operations, i.e. evaluation, crossover, mutation, and selection, for the population $P(t)$ are α_1, α_2, α_3, and α_4, respectively, as shown in Figure 5. Suppose the total run time of the four operators (i.e. $\alpha_1+\alpha_2+\alpha_3+\alpha_4$) is 20 seconds in each iteration t. If the whole process of GA runs 180 iterations, it would take at least $20\times180=3,600$ seconds to fulfill this convergent evolution. We note that around 95% of the total elapsed time is used for performing these four operators. If these four operators can be performed in parallel in a distributed system, then it will significantly reduce the total run time of GA.

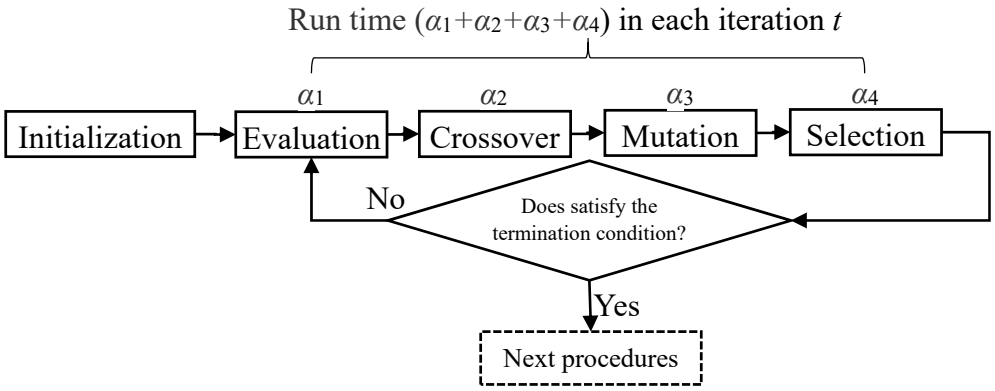


Figure 5. The GA procedure in some iteration t in the mapper

3. Proposed GA parallel and distributed architecture

How to parallelize the four major operators of GA, i.e. evaluation, crossover, mutation, and selection, is crucial to the effectiveness and efficiency of a developed GA parallel and distributed architecture. Before proposing the parallelization scheme, this study analyzes the characteristics of these four operators one by one as follows:

(i) **Evaluation:** The evaluation operator plays an important role in the GA evolution since evaluating all chromosomes' fitness in each generation is necessary. The fitness evaluations actually can be distributed for parallel computing since evaluating the fitness of an arbitrary chromosome needs no information from other chromosomes and is an independent job. Besides, the computational time for the whole GA procedure grows rapidly when the sizes of population $P(t)$ and offspring $Q(t)$ become large. Thus, the evaluation operator is suitable for and shall be developed for parallelization in a parallel and distributed system.

(ii) **Crossover:** The crossover operator generates an offspring from two parent chromosomes of the population. Firstly, the frequency of performing the crossover operation is not higher than that of running the evaluation operator in each generation. Secondly, this operator requires two parent chromosomes selected from the whole population to avoid the same situation as Figure 3. In other words, it may cause premature convergence if the crossover operator selects the two parent chromosomes from different sub-populations. Moreover, because a generated offspring chromosome could be an infeasible solution, designing a procedure to avoid or fix the infeasibility of offspring chromosomes is critical. From the above viewpoints, a sophisticated design that any infeasible chromosome yielded can be independently adjusted in the parallel computation and any two parent chromosomes for producing an offspring are selected from the whole population is needed for the crossover operator.

(iii) **Mutation**: The main task of the mutation operator is adjusting the offspring chromosome generated by the crossover operator for keeping the diversity. The frequency of performing the mutation operation is nearly identical to that of the crossover operation. Similarly, a mechanism to fix the infeasibility of chromosomes generated from the mutation operation is needed. The mutation operation is also suitable for the parallelization in distributed computing.

(iv) **Selection:** The selection operation for each *t*-th generation is to generate the new population for the (*t*+1)-th generation, i.e. $P(t+1) = P(t) \bigcup Q(t)$. The desired GA principle "survival of the fittest" could fail provided that the whole population is divided into sub-populations for the selection operation as demonstrated in Figure 3. It is thus necessary to generate a new population from all sub-populations in the reducers. In addition, the selection operation is performed just once for each generation in the GA procedure. Therefore, it is unnecessary and unsuitable to run the selection operator in

parallel and distributed computing.

The aforementioned discussions of the four GA operators are summarized in Table 2.

Table 2. Characteristics of the four GA operators

Operator	Frequency per generation	The whole population requirement	Parallelization
Evaluation	High	No	Suitable
Crossover	Middle	Yes	Conditionally suitable
Mutation	Middle	No	Suitable
Selection	Once	Yes	Unsuitable

These characteristics inspire this study to develop a GA parallelization scheme that retains the following properties:

Property 1: To enhance the efficiency of exploring the solution space, the three operators including evaluation, crossover, and mutation in GA are implemented in a parallelization architecture.

Property 2: Any new child chromosome generated from the crossover or mutation is immediately evaluated in a parallelization architecture, i.e. the operators of crossover, mutation, and evaluation are executed simultaneously.

The GA parallel and distributed architecture on Apache Hadoop proposed by this study is shown in Figure 6.

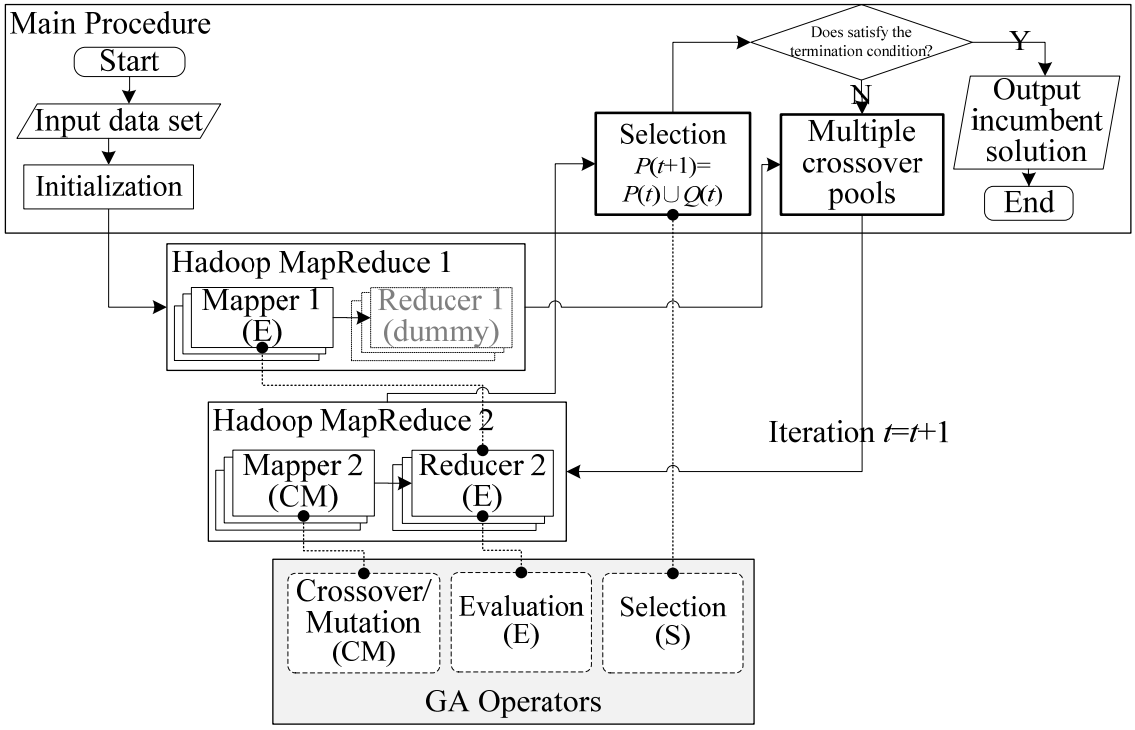


Figure 6. Proposed parallel and distributed architecture on Apache Hadoop

The three procedures in the proposed GA parallelization architecture are described below:

(i) **Main procedure** is a main program to dispatch the data into the MapReduce on Hadoop and control the processes of MapReduce 1 and MapReduce 2. It also loads the dataset and configures the initial environment of Hadoop. The four GA operators are designed in the main procedure. The operators of evaluation, crossover, and mutation are controlled by the Hadoop, while the selection operator is governed by the main procedure to collect the solutions of all sub-populations on the Hadoop. Besides, to avoid the disadvantage of selecting parent chromosomes in partial populations (as shown in Figure 3), we design a sub-procedure to select multiple parent chromosome sets (i.e. multiple crossover pools), from the whole sub populations. Then, the multiple parent chromosome sets will be sent to the mappers for the crossover operation. The design mechanism can achieve the parallelization of the crossover/mutation operations and retain the aforementioned Property 1 and 2.

(ii) **MapReduce 1** is the initial MapReduce procedure on Hadoop. Mapper 1 in this MapReduce procedure evaluates each chromosome in the initial population, while all reducers are dummy.

(iii) **MapReduce 2** is the follow-up iterative MapReduce procedure, where each mapper (i.e.

Mapper 2) runs the crossover/mutation operators and its corresponding reducer (i.e. Reducer 2) evaluates each chromosome assigned from the mapper.

To increase the efficiency of MapReduce 1 and MapReduce 2, the Hadoop dynamically adjusts the numbers of mappers and reducers. In the initial stage, each mapper in the MapReduce 1 calls the function *map*_1(key, value), where the first parameter 'key' is an index of the second parameter 'value' representing the chromosome. The function *map*_1 (key, value) performs the following operations (see Figure 7):

(i) Call the function *Representation*(value) to convert 'value' into a chromosome. Then save it temporarily as 'chromosome'.

(ii) Call the function *Evaluation*(chromosome) to evaluate the fitness of 'chromosome'. Then save it temporarily as 'fitness'.

(iii) Call the function *Emit*() to send the 2-tuple <fitness, chromosome> back to the Hadoop.

```
function map_1(key, value)
begin
      chromosome = Representation(value);
      fitness = Evaluation(chromosome);
      Emit(fitness, chromosome);
End
```

Figure 7. The proposed function *map*_1(key, value)

In the iterative stage, each mapper in MapeReduce 2 calls the other function *map*_2(key, values). The first parameter 'key' is similar to that in *map*_1(). The second parameter 'values' is defined as a string representing multiple chromosomes. The function *map*_2(key, values) performs the following operations (see Figure 8):

(i) Call the function *RepresentationBuf*(values) to convert 'values' into a pair of chromosomes, and then save them temporarily as 'chromosomeBuffer'.

(ii) Call the function *CrossoverMutation*(chromosomeBuffer) to generate a new child chromosome.

(iii) Call the function *Emit*() to send the 2-tuple <_index, new_chromosome> back to the Hadoop, where the parameter '_index' is the new index for each new offspring 'new_chromosome'.

```
static int new _index=0;
function map_2(key, values)
begin
    chromosomeBuffer = RepresentationBuf(values)
    chromosomeArray = CrossoverMutation(chromosomeBuffer)
```

```
    for each (new_chromosome in chromosomeArray)
        Emit(_index, new_chromosome);
        _index += 1;
    end for each
end
```

Figure 8. The proposed function *map*_2(key, value)

Additionally, the chromosomes with the same fitness value will be grouped into a string named 'value_list'. Each reducer calls *reduce*_2(), a function of the two parameters 'key' and 'value_list'. The function *reduce*_2(key, value_list) performs the following operations (see Figure 9):

(i) Call the function *Representation*(value) to convert 'value' into a chromosome. Then save it temporarily as 'chromosome'.

(ii) Call the function *Evaluation*(chromosome) to evaluate the fitness of 'chromosome'. Then save it temporarily as 'fitness'.

(iii) Call the function *Emit*() to send the 2-tuple <fitness, chromosome> back to the Hadoop.

```
function reduce_2(key, value_list)
begin
      for each (value in value_list)
         chromosome = Representation(value);
         fitness = Evaluation(chromosome);
         Emit(fitness, chromosome);
      end for each
End
```

Figure 9. The proposed function *reduce_2*(key, value_list)

The proposed GA parallel and distributed architecture on Apache Hadoop mainly contains three components, i.e. main procedure, Hadoop MapeReduce 1 and Hadoop MapeReduce 2. The detailed steps of the designed procedure are provided below:

(i) Initial procedure (running only once):

Step 1: The main procedure generates the initial population according to the user-defined parameters.

Step 2: The main procedure converts the whole population into sub-populations and then activates Hadoop MapeReduce 1.

Step 3: Each sub-population is assigned to one of the mappers for the fitness evaluation by Hadoop MapeReduce 1 and then sent back to the main procedure.

Step 4: The main procedure calls a sub-procedure to prepare multiple crossover pools for the mappers of Hadoop MapeReduce 2 and then activates Hadoop MapeReduce

2.

- Note that each mapper corresponds to one reducer, but the reducer is dummy in the initial procedure.

(ii) Iterative procedure:

Step 5: Each mapper coupled with the corresponding reducer performs sequentially the crossover, mutation, and evaluation operators for generating new child chromosomes and then sends them back to the main procedure.

Step 6: The main procedure performs the selection operation for all the new child chromosomes collected from the reducers.

Step 7: The main procedure checks the stop condition. If the stop condition is not satisfied, it calls a sub-procedure to prepare multiple crossover pools for the mappers of Hadoop MapeReduce 2 and then goes to Step 5 to activate MapeReduce 2. Otherwise, it goes to the output phase.

- Note that the stop condition can be the maximal number of iterations, the maximal run time or the objective value accepted.

(iii) Output: The incumbent solution is yielded in the output stage.

The advantages of the proposed GA parallelization scheme includes:

(i) The GA procedure is sophisticatedly embedded in Apache Hadoop.

(ii) The selection operator is designed in the main procedure so that the selection probabilities can be calculated in accordance with the rule "survival of the fittest".

(iii) Any new child chromosome generated from the crossover/mutation operations can be directly evaluated in the parallel and distributed computing system since the two operators are run simultaneously, which can achieve the sufficient solution diversity for exploring the solution space.

4. Numerical experiments on Apache Hadoop

This section conducts the computational experiments to compare the performance of the proposed GA parallelization architecture (cf. Figure 6) with those of the two reference architectures, i.e. Verma et al. (2009) (cf. Figure 2) and Kečo and Subasi (2012) (cf. Figure 4), in solving the classical NP-hard combinatorial problem. To test the effectiveness of GA parallelization, we employed the asymmetric travelling salesman problem (ATSP) to generate three sets of testing instances as shown in Table 3. Instance 1 (denoted by Small-ATSP),

Instance 2 (Medium-ATSP) , and Instance 3 (Large-ATSP) are the small-scale ATSP composed of 256 cities, middle-scale ATSP composed of 512 cities, and large-scale ATSP composed of 1024 cities, respectively. The dataset of the three testing instances of ATSPs are provided in the supplementary information which can be found online (http://drive.google.com/open?id=18DI8tGhraf15ib9uIia70TUI-SX3t160).

Table 3. Three testing instances of ATSP

Instance	Type	Number of cities
1	Small-ATSP	256
2	Medium-ATSP	512
3	Large-ATSP	1024

In the experiments, the GA programs were implemented with Java standard edition version 9 on eight PCs (i.e. a master and seven slaves) equipped with Intel Core i7 CPU, CentOS 7, and Apache Spark 2.2. The detailed information of our equipment is listed in Table 4. Referring to Verma et al. (2009) and Kečo and Subasi (2012), we had the following parameters and settings for the three testing instances.

(i) Population size: 2048

(ii) Crossover rate: 0.8

(iii) Considering the fairness, the mutation rate was set to 0 since the mutation operator was not performed in Verma et al. (2009).

(iv) Considering the fairness, we adopt the roulette wheel selection operator for the three testing architectures.

Table 4. Experimental equipment

Node name	Storage	Operating system	CPU	Memory
Master	64 GB	CentOS 7	Intel Core i7	12 GB
Slaves1-7	32 GB	CentOS 7	Intel Core i5	8 GB

In each of the three GA parallelization architectures, we run 30 rounds for all the three testing instances. The best, average, and worst results are tabulated in Table 5. The experiments employed the maximal run time as the stop condition. Three time thresholds (900, 1400, 9000) in seconds were used for Instance 1. For Instance 2, we adopted the three thresholds (900, 3000, 18000) in seconds. The three thresholds (900, 4200, 36000) in seconds applied to Instance 3. The evolution diagrams of each GA parallelization architecture for the three instances are depicted in Figures 10-12, where the *x*-axis stands for the computational time (in seconds) and

the *y*-axis indicates the objective value (i.e. fitness value). The computational results can be summarized as follows:

(i) It is convergent at the 1643rd, 2351st, and 4531st second to solve Small-ATSP, Medium-ATSP, and Large-ATSP, respectively, with the proposed GA parallelization architecture. The trends of convergence illustrate that the proposed architecture outperforms the two reference architectures.

(ii) Figures 10-12 clearly show that the premature convergence arises in the architecture of Verma et al. (2009), which in general is the worst in performance from each aspect. It converges at the 1235th, 1314th, and 1740th second to solve Small-ATSP, Medium-ATSP and Large-ATSP, respectively, in the architecture of Verma et al. (2009). The aforementioned disadvantage of the GA roulette wheel selection is evidenced by this computational result. Each reducer in the architecture of Verma et al. (2009) performs the selection and crossover operations in which sub-populations instead of the whole population are used. It leads to an evolution pool that is not large enough and cannot contain a genetically stable population.

(iii) The evolution diagrams show that the required evolution time in the architecture of Kečo and Subasi (2012) is the longest among these three. The main reason is that the four GA operators in their designed architecture basically are not processed in parallel.

Table 5. Computational results of ATSPs on Apache Hadoop

Instance	Time (sec.)	Solution Situation	Verma et al. (2009)	Kečo and Subasi (2012)	Proposed architecture	Evolution (Average)
Small-ATSP	900	Best	105,648.0	99,130.0	**70,151.0**	Please refer to Figure 10
		Average	106,252.3	103,040.8	**72,654.3**	
		Worst	106,652.0	106,716.0	**74,605.0**	
	1,400	Best	100,780.0	94,669.0	**57,557.0**	
		Average	101,698.7	98,742.6	**59,077.7**	
		Worst	102,230.0	103,828.0	**60,362.0**	
	9,000	Best	100,780.0	54,411.0	**54,352.0**	
		Average	101,698.7	58,659. 8	**55,293.0**	
		Worst	102,230.0	68,887.0	**56,102.0**	
Medium-ATSP	900	Best	219,901.0	220,598.0	**189,748.0**	Please refer to Figure 11
		Average	222,977.7	227,739.2	**194,268.0**	
		Worst	226,303.0	237,122.0	**197,975.0**	
	3,000	Best	219,071.0	197,480.0	**139,990.0**	
		Average	219,683.7	208,742.1	**141,599.7**	
		Worst	219,994.0	216,693.0	**142,852.0**	
	1,8000	Best	219,071.0	138,410.0	**139,990.0**	
		Average	219,683.7	148,110.6	**141,599.7**	

		Worst	219,994.0	171,265.0	**142,852.0**	
Large-ATSP	900	Best	477,883.0	478,314.0	**455,583.0**	Please refer to Figure 12
		Average	481,609.3	485,423.1	**458,238.0**	
		Worst	488,797.0	492,386.0	**462,033.0**	
	4,200	Best	459,034.0	431,302.0	**341,551.0**	
		Average	462,036.3	450,598.4	**351,031.0**	
		Worst	467,460.0	483,973.0	**357,725.0**	
	36,000	Best	459,034.0	342,014.0	**334,118.0**	
		Average	462,036.3	361,409.5	**344,822.3**	
		Worst	467,460.0	410,463.0	**351,340.0**	

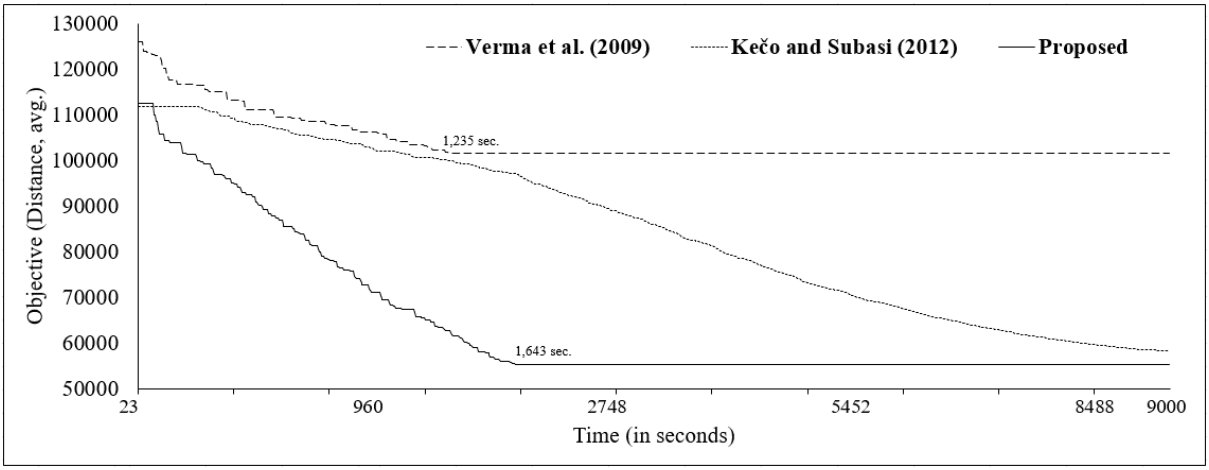


Figure 10. The evolution diagrams for Small-ATSP on Apache Hadoop

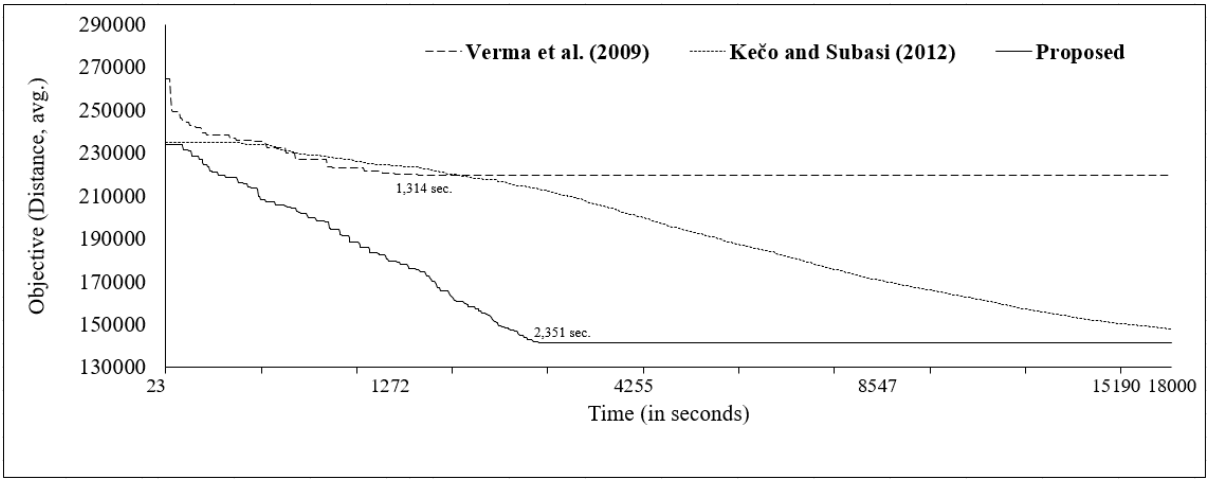


Figure 11. The evolution diagrams for Medium-ATSP on Apache Hadoop

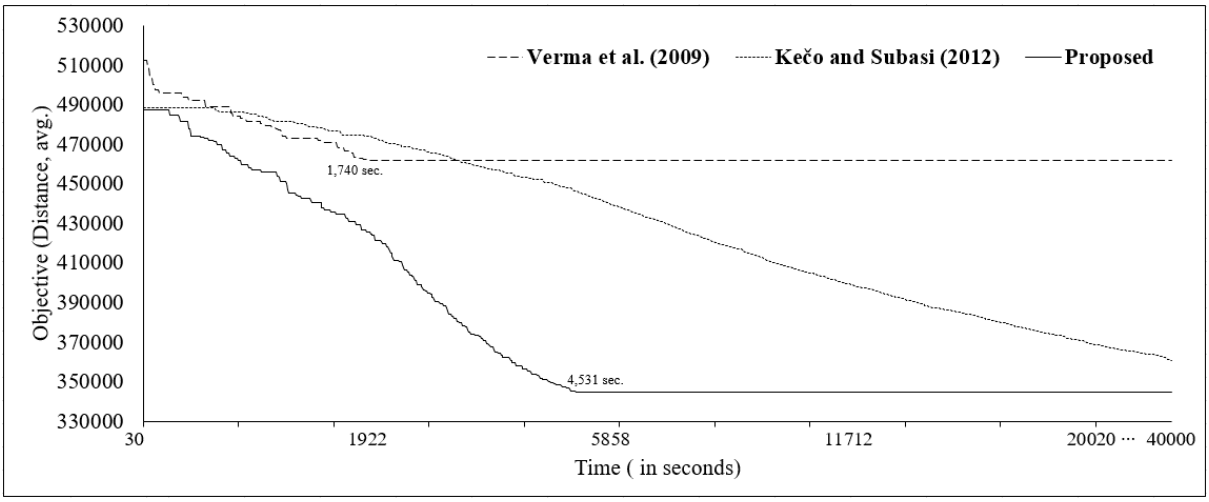


Figure 12. The evolution diagrams for Large-ATSP on Apache Hadoop

5. An extension to Apache Spark

Apache Hadoop, based on the HDFS, is suitable for solving the large-scale problems, while it may lead to a long input/output processing time. The iterative algorithm accessing data from the database of Hadoop, i.e. HDFS, is relatively inefficient (Zaharia et al., 2012). Hence, Zaharia et al. (2012) presented a scheme of RDD, which is a distributed memory abstraction such that procedures efficiently perform in-memory computations on large clusters in a fault-tolerant manner. Noting that keeping data in memory can significantly improve the performance by an order of magnitude, Zaharia et al. (2012) implemented the RDDs in Apache Spark, which is an advanced cluster computing technology for fast computation. Apache Spark modifies the computational model based on the technique of Hadoop MapReduce to unify efficiently the interactive queries and stream processing. Especially, Apache Spark allows the iteration processes, i.e. MapReduce, to contain only the mappers without reducers, and vice versa. This section extends the proposed GA parallelization architecture to Apache Spark. The computational experiments will be conducted again to compare the performance of the proposed architecture with those of the reference architectures of Verma et al. (2009) and Kečo and Subasi (2012) on Apache Spark.

5.1 Proposed GA parallelization architecture on Apache Spark

The concept of the proposed GA procedures embedded in Apache Spark is shown in Figure 13. The GA operators including evaluation, crossover, mutation, and selection can be designed in one of the two stages of mapper/reducer. The stage of reducer on Apache Spark can be omitted such that it can save the input/output (I/O) time in an iterative procedure, while

the reducer cannot be omitted on Apache Hadoop. It is thus obvious that Apache Spark is more suitable for implementing GA parallelization than Apache Hadoop.

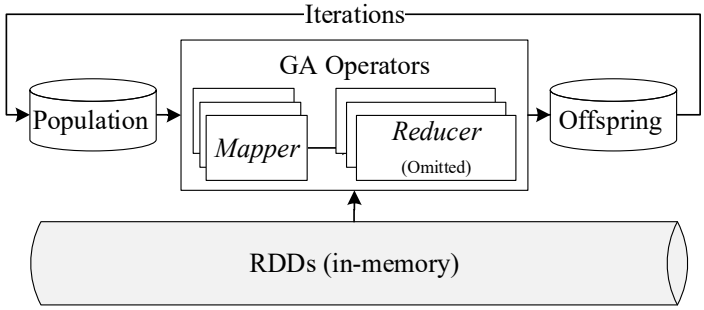


Figure 13. GA procedures on Apache Spark

For enhancing the efficiency, the master procedure on Apache Spark can dynamically modify the numbers of mappers according to the considered problem. To simplify the presentation, we set two mappers in the proposed architecture on Apache Spark as follows:

Mapper 1: The function *map*() is declared in Mapper 1 for evaluating the fitness value of each chromosome. The description of function *map*(key, value) is given below (see Figure 14):

```
function map(key, value)
begin
      chromosome = Representation(value);
      fitness = Evaluation(chromosome);
      Emit(fitness, chromosome);
End
```

Figure 14. The proposed function *map*(key, value)

Mapper 2: The function *flatMap*() is declared in Mapper 2 for running the crossover and mutation operators. The crossover operator generates a new offspring chromosome by selecting a pair of chromosomes from the whole population. The description of function *flatMap*(key, values) is provided below (see Figure 15):

```
function flatMap(key, values)
begin
    chromosomeBuffer = Representation(values);
    chromosomeArray = CrossoverMutation(chromosomeBuffer);
    for each (new_chromosome in chromosomeArray)
        Emit(new_key_1, new chromosome);
        new_key_1 += 1;
    end for each
end
```

Figure 15. The proposed function *flatMap*(key, values)

The proposed GA parallel and distributed architecture on Apache Spark is shown in Figure 16, and the detailed processes of the proposed architecture are described as follows.

(i) Initial procedure:

Step 1: The master procedure generates the initial population and saves it in the RDD.

Step 2: Mapper 1 is invoked to evaluate each chromosome of the initial population, and then the results are recorded in the RDD.

- Note that the master procedure monitors Mapper 1 and Mapper 2 to avoid misinformation and performs information exchanges between them.

(ii) Iterative procedure:

Step 3: Mapper 2 is invoked to run the crossover and mutation operations. The crossover operator generates a new chromosome, which is selected from the whole population by the master procedure and then saved in the RDD.

Step 4: If there is any new offspring generated by Mapper 2, then Mapper 1 will be invoked for evaluating parallel the new offspring. The result of each chromosome is saved in the RDD.

Step 5: The master procedure selects parents to generate the population of new generation from the current population and offspring.

Step 6: The master procedure checks the stop condition, which can be the maximal number of iterations, the maximal run time or the objective value accepted. If the stop condition is not satisfied, then it goes to Step 3. Otherwise, it goes to the output phase.

(iii) Output: The best solution is yielded by selecting from the whole population.

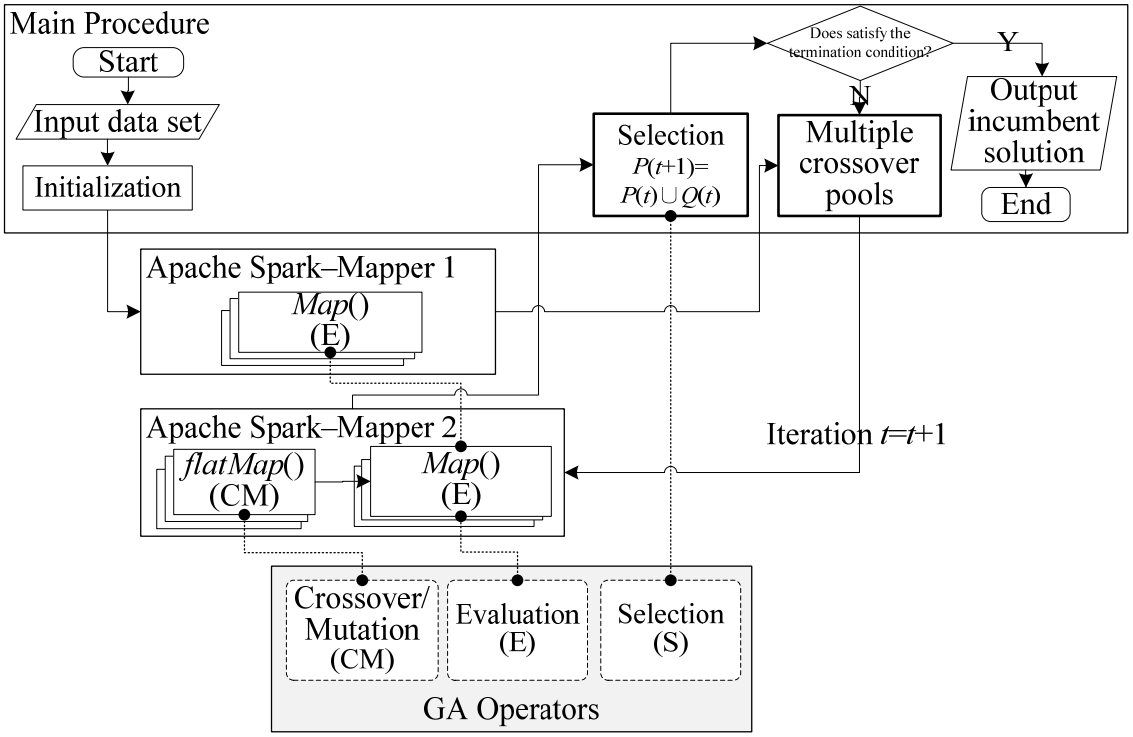


Figure 16. Proposed GA parallel and distributed architecture on Apache Spark

5.2 Numerical experiments on Apache Spark

To do the performance comparisons between the proposed GA parallelization architecture and the two reference architectures, i.e. Verma et al. (2009) and Kečo and Subasi (2012), this study utilized a variant of TSP named VTSP to generate the data sets. Each of n cities in the VTSP retains 128 attributes, i.e. city i is represented as $x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,128})$ for $i = 1, \ldots, n$. The distance between city i and j is calculated according to the Euclidean distance $\sqrt{\sum_{k=1}^{128} (x_{i,k} - x_{j,k})^2}$. There are three sets of VTSP instances, where $x_{i,k}$ is an random integer distributed uniformly over the interval [0, 30] for $i = 1, \ldots, n$ and $k = 1, \ldots, 128$. Instance 1 (named Small-VTSP), Instance 2 (named Medium-VTSP), and Instance 3 (named Large-VTSP) have 256 cities (i.e. $n = 256$), 512 cities (i.e. $n = 512$), and 1024 cities (i.e. $n = 1024$), respectively. The dataset of these three testing instances of VTSPs are provided as the supplementary information which can be downloaded online (http://drive.google.com/open?id=1qxeaw-s8-pFe96o5CMbbWTO8vNy3J7wO).

The GA programs were executed on Apache Spark 2.2. The experimental equipment, operating system parameters and settings for these three testing instances were set the same as

Section 4. These experiments were also run 30 rounds. Table 6 lists the best, average, and worst results of the three GA parallelization architectures for all the three testing instances. Figures 17-19 present the evolution diagrams of the three architectures for Small-VTSP, Medium-VTSP and Large-VTSP, respectively. The computational results are summarized as follows.

(i) The proposed architecture converges at the 663rd, 1572nd and 3539th seconds in solving Small-VTSP, Medium-VTSP, and Large-VTSP, respectively. The trends of convergence again demonstrate that the proposed architecture outperforms the two reference architectures.

(ii) The architecture of Verma et al. (2009) is convergent at the 261st, 454th, and 623rd seconds for Small-VTSP, Medium-VTSP and Large-VTSP, respectively. In Figures 17-19, the architecture of Verma et al. (2009) shows the premature convergence and again yields the worst results for all the three testing instances.

(iii) The evolution of the architecture of Kečo and Subasi (2012) is the slowest for each of the three testing instances.

Table 6. Computational results of VTSPs on Apache Spark

Instance	Time (sec.)	Solution Situation	Verma et al. (2009)	Kečo and Subasi (2012)	Proposed architecture	Evolution (Avg.)
Small-VTSP	30	Best	244,918.0	244,676.0	**244,591.0**	Please refer to Figure 17
		Average	245,684.3	246,380.1	**245,061.0**	
		Worst	246,468.0	247,208.0	**245,832.0**	
	200	Best	243,265.0	242,617.0	**232,320.0**	
		Average	244,071.1	243,977.1	**234,408.1**	
		Worst	245,340.0	245,337.0	**236,792.0**	
	800	Best	243,265.0	234,703.0	**221,597.0**	
		Average	243,522.6	237,659	**221,970.4**	
		Worst	243,695.0	240,156.0	**222,204.0**	
Medium-VTSP	60	Best	537,995.0	543,354.0	**539,000.0**	Please refer to Figure 18
		Average	540,164.8	544,162.9	**540,461.3**	
		Worst	542,362.0	545,062.2	**542,006.0**	
	400	Best	537,635.0	536,668.0	**519,008.0**	
		Average	539,215.4	541,380.3	**520,318.6**	
		Worst	540,241.0	544,351.0	**522,867.0**	
	1,600	Best	537,194.0	526,795.0	**495,784.0**	
		Average	538,321.7	531,748.9	**500,002.4**	
		Worst	540,135.0	538,233.0	**504,499.0**	
Large-VTSP	120	Best	1,179,301.0	1,182,063.0	**1,175,334.0**	Please refer to Figure 19
		Average	1,180,329.9	1,184,803.5	**1,178,384**.8	
		Worst	1,181,745.0	1,187,021.0	**1,180,706.0**	
	800	Best	1,174,666.0	1,177,063.0	**1,150,080.0**	
		Average	1,176,151.7	1,179,282.4	**1,150,484**.6	
		Worst	1,177,006.0	1,181,849.0	**1,150,921.0**	
	3,200	Best	1,174,252.0	1,158,344.0	**1,104,894.0**	

	Average	1,175,044.5	1,164,756.4	**1,108,243**.2
	Worst	1,176,213.0	1,169,072.0	**1,112,515.0**

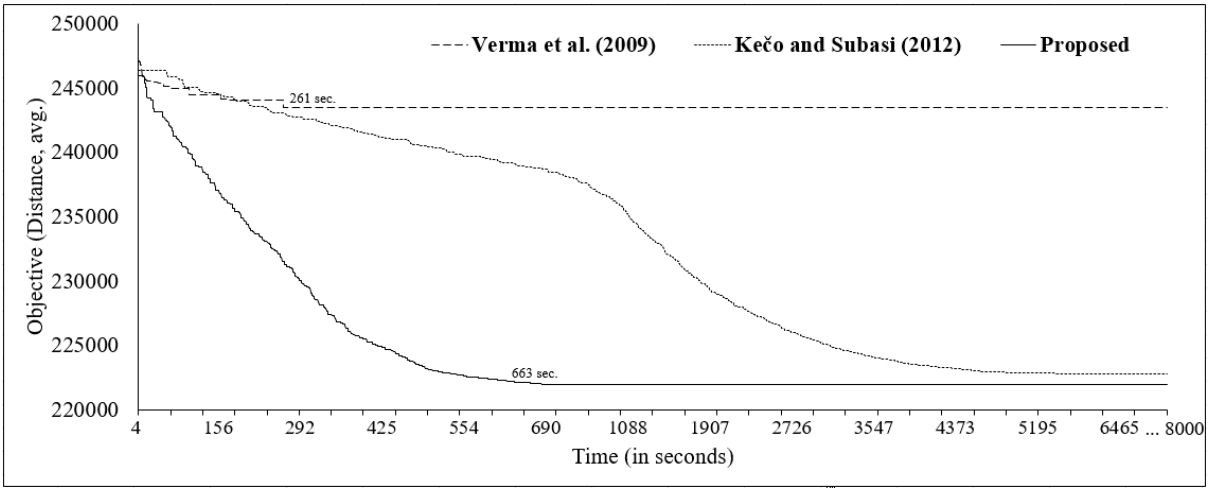


Figure 17. The evolution diagrams of Small-VTSP on Apache Spark

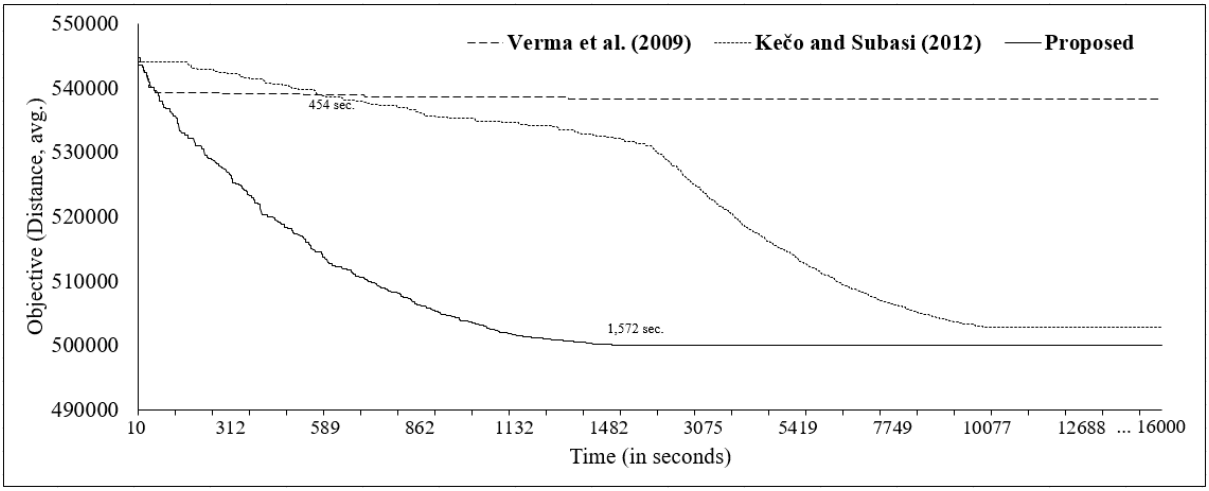


Figure 18. The evolution diagrams of Medium-VTSP on Apache Spark

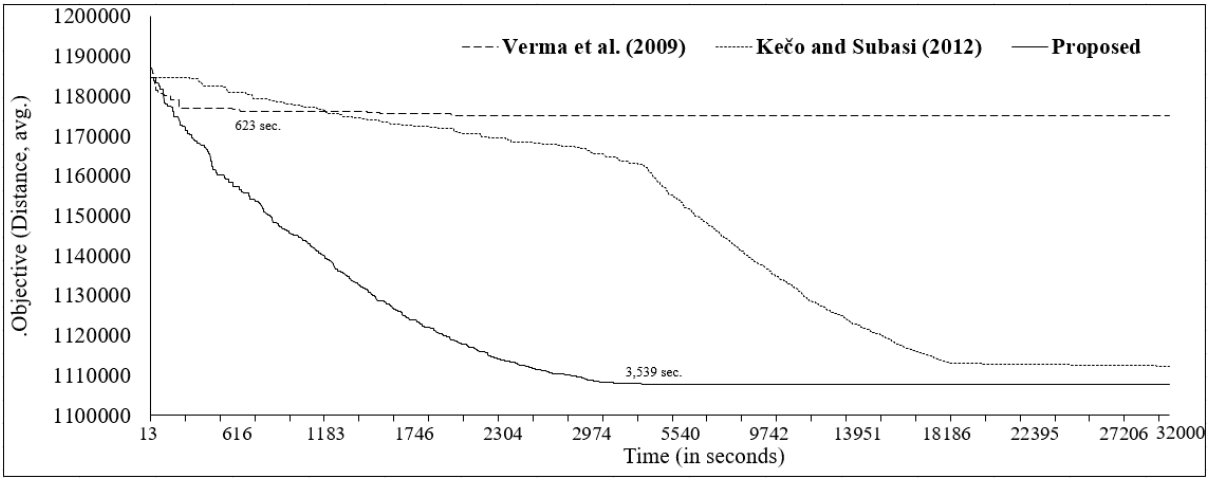


Figure 19. The evolution diagrams of Large-VTSP on Apache Spark

6. Conclusion

This study has developed an enhanced GA parallel and distributed architecture for Apache Hadoop and Spark. We have designed an effective approach to maintain the "survival of the fittest" principle in the selection operator. This sophisticated mechanism makes the proposed architecture fairly select the whole parents for crossover and mutation operations in the parallel and distributed architecture. Our numerical experiments have shown that the proposed architecture on Apache Hadoop or Spark runs significantly more efficiently than the two reference architectures, especially for the large-size optimization problems.

REFERENCES

Almeer, M.H. (2012). Cloud hadoop map reduce for remote sensing image analysis. *Journal of Emerging Trends in Computing and Information Sciences*, 3(4), 634-644.

Apache Hadoop. (2006). The Apache Software Foundation, http://hadoop.apache.org/.

Apache Spark. (2017). The Apache Software Foundation, https://spark.apache.org/.

Arora, S., & Chana, I. (2014). A survey of clustering techniques for big data analysis. *2014 5th International Conference-Confluence The Next Generation Information Technology Summit (Confluence)*, 59-65.

Camacho, D. 2015. Bio-inspired clustering: basic features and future trends in the era of big data. *2015 IEEE 2nd International Conference on Cybernetics (CYBCONF)*, 1-6.

Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107-113.

Dittrich, J., & Quiané-Ruiz, J.A. (2012). Efficient big data processing in Hadoop MapReduce. *Proceedings of the VLDB Endowment*, 5(12), 2014-2015.

Dyckhoff, H. (1990). A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2), 145-159.

Ferrucci, F., Kechadi, M.T., Salza, P., & Sarro, F. (2013). A framework for genetic algorithms based on Hadoop. *ariXiv*, 30.

Gallagher, K., & Sambridge, M. (1994). Genetic algorithms: a powerful tool for large-scale nonlinear optimization problems. *Computers & Geosciences,* 20(7), 1229-1236.

Gen, M., & Cheng, R. 2000. Genetic algorithms and engineering optimization. *John Wiley &*

Sons Inc.

Geronimo, L.D., Ferrucci, F., Murolo, A., & Sarro, V. (2012). A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 785-793.

Goldberg, D.E. (1989). Genetic algorithms in search optimization and machine learning. *Addison-Wesley Professional*.

Goldberg, D.E. (2013). The design of innovation: Lessons from and for competent genetic algorithms. *Springer Science & Business Media*, 7.

Gong, Y.J., Chen, W.N., Zhan, Z.H., Zhang, J., Li, Y., Zhang, Q., & Li, J.J. (2015). Distributed evolutionary algorithm and their models: A survey of state-of-the-art. *Applied Soft Computing*, 34, 286-300.

Grefenstette, J., Gopal, R., Rosmaita, B., Van Gucht, D. (1985). Genetic algorithms for the traveling salesman problem. In Grefenstette, J., *Proceedings of the first International Conference on Genetic Algorithms and their Applications:* 160-168.

Gu, R., Yang, X., Yan, J., Sun, Y., Wang, B., Yuan, C., & Huang, Y. (2014). SHadoop: improving mapreduce performance by optimizing job execution mechanism in Hadoop clusters. *Journal of parallel and distributed computing*, 74(3), 2166-2179.

Hartmann, S. (2002). A self-adapting genetic algorithm for project scheduling under resource constraints. *Naval Research Logistics*, 49(5), 433-448.

Herrera, F., & Lozano, M. (2000). Gradual distributed real-Coded genetic algorithms. *IEEE Transactions on Evolutionary Computation* 4(1), 43–63.

Holland J.H. (1975). Adaptation in natural and artificial systems: an introductory analysis. *Control and artificial intelligence.*

Huang Y.H., & Hwang, F.J. (2017). Global optimization for the three-dimensional open-dimension rectangular packing problem. *Engineering Optimization*, DOI: 10.1080/0305215X.2017.1411484.

Huang, Y.H., Hwang, F.J., & Lu, H.C. (2016). An effective placement method for the single container loading problem. *Computers & Industrial Engineering*, 97, 212-221.

Kečo, D., & Subasi, A. (2012). Parallelization of genetic algorithms using Hadoop Map/Reduce. *SouthEast Europe Journal of Soft Computing*, 1(2).

Lu, H., & Huang, Y.H. (2015). An efficient genetic algorithm with a corner space algorithm for a cutting stock problem in the TFT-LCD industry. *European Journal of Operational*

Research, 246(1), 51-65.

Luque, G., & Alba, E. (2011). Parallel models for genetic algorithms. *Parallel Genetic Algorithms: Theory and Real World Applications*, 367.

Maenhout, B., & Vanhoucke, M. (2018). A perturbation matheuristic for the integrated personnel shift and task re-scheduling problem. *European Journal of Operational Research*, 269(3), 806-823.

Martino, S.D., Ferrucci, F., Maggio, V., & Sarro, F. (2012). Chapter 6: Towards migrating genetic algorithms for test data generation to the cloud. *Software Testing in the Cloud: Perspectives on an Emerging Discipline*, 113-135.

Melab, N., & Talbi, E.G. (2010). GPU-based island model for evolutionary algorithms. *Proceedings of the 12 Annual Conference on Genetic and Evolutionary Computation*, 1089-1096.

Murata, T., Ishibuchi, H., & Tanaka, H. (1996). Genetic algorithms for flowshop scheduling problems. *Computers & Industrial Engineering*, 30(4), 1061-1071.

Pongchairerks, P., & Kachitvichyanukul, V. (2009). A particle swarm optimization algorithm on job-shop scheduling problems with multi-purpose machines. *European Journal of Operational Research*, 26(2), 161-184.

Qi, R.Z., Wang, Z.J., & Li, S.Y. (2016). A parallel genetic algorithm based on spark for pairwise test suite generation. *Journal of Computer Science and Technology*, 31(2), 417–427.

Ren, Y., Zhang, C., Zhao, F., Xiao, H., & Tian, G. (2018). An asynchronous parallel disassembly planning based on genetic algorithm. *European Journal of Operational Research,* 269(2), 647-660.

Sachar, P., & Khullar, V. (2016). Genetic algorithm using MapReduce-A critical review. *i-manager's Journal on Cloud Computing*, 2(4), DOI:10.26634/jcc.2.4.4907.

Scheithauer, G. (1992). Algorithms for the container loading problem. *Operations Research Proceedings,* 1991, 445-452.

Teobaldo, B., Minh, H.H., Rafael, M., & Thibaut, V. (2018). The vehicle routing problem with service level constraints. *European Journal of Operational Research*, 265(2), 544-558.

Tsai, J.F., Wang, P.C., & Lin, M.H. (2015). A global optimization approach for solving three-dimensional open dimension rectangular packing problems. *Optimization,* 64, 2601-2618.

Verma, A., Llorà, X., Goldberg, D.E., & Campbell, R.H. (2009). Scaling genetic algorithms using mapreduce. In IEEE, *Intelligent Systems Design and Applications. ISDA'09. Ninth International Conference on:* 13-18.

Victor, P., Michel, G., Christelle, G., & Andrés, L.M. (2013). A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1), 1-11.

Vidal, P., & Alba, E. (2010). Cellular genetic algorithm on graphic processing units. *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, 223-232.

Wuttke, D.A., & Heese, H.S. (2018). Two-dimensional cutting stock problem with sequence dependent setup times. *European Journal of Operational Research*, 265(1), 303-315.

Yu, B., Yang, Z., Sun, X., Yao, B., Zeng, Q., & Jeppesen, E. (2011). Parallel genetic algorithm in bus route headway optimization. *Applied Soft Computing*, 11(8), 5081-5091.

Yu, W., & Zhang, W. (2006). Study on function optimization based on master-slave structure genetic algorithm. *2006 8th International Conference on Signal Processing*, 3.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2.