J Supercomput (2015) 71:1357–1377
DOI 10.1007/s11227-014-1368-6

Task scheduling for grid computing systems using a genetic algorithm

Yi-Syuan Jiang · Wei-Mei Chen

Published online: 25 December 2014

Abstract A grid computing environment is a parallel and distributed system that brings together various computing capacities to solve large computational problems. Task scheduling is a critical issue for grid computing; in task scheduling, tasks are mapped onto system processors with the aim of achieving good performance in terms of minimizing the overall execution time. In previous studies, there have been several approaches to solving the task-scheduling problem by genetic algorithms, which is a random search technique that is inspired by natural biological evolution. This study presents a genetic algorithm for solving the problem of task scheduling with two main ideas: a new initialization strategy to generate the first population and new genetic operators based on task–processor assignments to preserve the good characteristics of the found solutions. Our proposed algorithm is implemented and evaluated using a set of well-known applications in our specifically defined system environment. The experimental results show that the proposed algorithm outperforms other popular algorithms in a variety of scenarios with several parameter settings.

Keywords Task scheduling · Genetic algorithms · Directed acyclic graph scheduling

This paper is an extended version of work published in [18].
This work is partially supported by National Science Council under the Grant NSC 103-2221-E-011-113.

Y.-S. Jiang · W.-M. Chen (✉)
Department of Electronic and Computer Engineering,
National Taiwan University of Science and Technology, Taipei 106, Taiwan
e-mail: wmchen@mail.ntust.edu.tw

Y.-S. Jiang
e-mail: M9902144@mail.ntust.edu.tw

1 Introduction

In the past few years, grid computing systems and applications have become popular [8] due to the rapid development of many-core systems. A grid computing environment is a parallel and distributed system that brings together various computing capacities to solve large computational problems. In grid environments, task scheduling, which plays an important role, divides a larger job into smaller tasks and maps tasks onto a parallel and distributed system [5,11]. The goal of task scheduling is typically to schedule all of the tasks on a given number of available processors in such a way as to minimize the overall length of the time required to execute the whole program.

There are deterministic approaches and non-deterministic approaches to solving the task-scheduling problem. An algorithm is referred to as a deterministic algorithm if given a specific input it will always produce the same output and the underlying machine always passes through the same sequence of states. Most of the deterministic algorithms are based on a greedy strategy, which merely attempts to minimize the finishing time of the tasks in such a way that tasks are allocated to the parallel processors without backtracking [20]. Several popular algorithms in this class have been proposed, such as a list scheduling algorithm [25], clustering algorithm [16], and task duplication algorithm [23]. However, these deterministic algorithms can only solve certain cases efficiently and cannot preserve the consistent performance on a broad range of problems due to the greedy property.

A non-deterministic algorithm differs from a deterministic algorithm in its capability to produce various outcomes depending on the choices that it makes during execution. Currently, non-deterministic algorithms have been used to solve a wide range of combinatorial problems, which take more time to explore the solution space for a high-quality solution. To solve the scheduling problem, several research studies have been performed based on non-deterministic algorithms, such as genetic algorithms [20,21,26], ant colony optimization algorithms [13,19], and particle swarm optimization algorithms [17,24], which apply randomized search techniques in the search process.

A parallel and distributed computing system could be a homogeneous [4,20] or heterogeneous system [6,9]. A homogeneous system means that the processors have the same performance in terms of processing capabilities. On the other hand, heterogeneous systems have different processing capabilities in the target system. In general, the processors are connected by a network, which is either fully connected [25,26] or partially connected [7]. For more related topologies and applications, please see [2,3,15]. In the fully connected network, every processor can communicate with every other processor, whereas data can be transferred to some specified processors in a partially connected network. To reduce the communication time, the task duplication issue [20] was discussed by duplicating some tasks on more than one processor. However, to avoid increasing energy consumption, we consider here the target system that is a fully connected heterogeneous system without task duplication.

The genetic algorithm (GA), which was first proposed by Holland [10], provides a popular solution for application problems. GAs have been shown to outperform several algorithms in the task-scheduling problem [8,12,14,27], which simply define the search space to be the solution space in which each point is denoted by a number string, called a chromosome. Based on these solutions, three operators, which are selection, crossover, and mutation, are employed to transform a population of chromosomes to better solutions iteratively. To retain the good features from the previous generation, the crossover operator exchanges the information from two randomly chosen chromosomes, and the mutation operator alters one bit of a chromosome at random.

In this study, we proposed a genetic algorithm for task scheduling on a grid computing system, called TSGA. In general, GA approaches directly initialize the first population by a uniform random process. TSGA develops a new initialization policy, which divides the search space into specific patterns to accelerate the convergence of the solutions. To solve the task-scheduling problem, a chromosome usually contains a mapping part and an order part to indicate the corresponding computer and the executing order. In the standard GA, when crossover and mutation operators are applied, the parents' good characteristics cannot be kept in the next generation. Based on the task–processor assignments of chromosome encoding, TSGA exploits new operators for crossover and mutation to preserve good features from the previous generation.

The remainder of this study is organized as follows. In the next section, we provide the problem definition. Section 3 introduces previous studies for the task-scheduling problem. The proposed genetic-scheduling algorithm is presented in Sect. 4. We describe our experimental results in Sect. 5. Lastly, conclusions are drawn in Sect. 6.

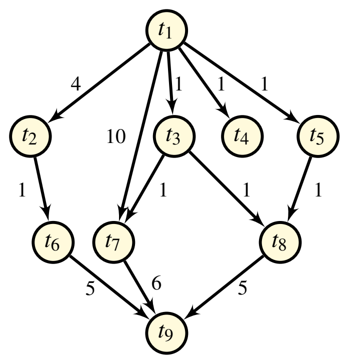
2 Problem definition

Task scheduling is the process of mapping tasks to system processors and arranging the execution order for each processor. Tasks with data precedence are modeled by a directed acyclic graph (DAG) [26]. The main idea of DAG scheduling is to minimize the makespan, which is the overall execution time for all of the tasks.

2.1 DAG modeling

A DAG $G = (V, E)$ is depicted in Fig. 1a, where V is a set of N nodes and E is a set of $M4$ directed edges. For the problem of task scheduling, V represents the set of tasks, and each task contains a sequence of instructions that should be completed in a specific order. Let $w_{i,j}$ be the computation time to finish a specific task $t_i \in V$ on a processor P_j , which is detailed in Fig. 1b. Each edge $e_{i,j} \in E$ in the DAG indicates the precedence constraint that task t_i should be completed before task t_j starts. Let $c_{i,j}$ denote the communication cost that is required to transport the data between task t_i and task t_j, which is the weight on an edge $e_{i,j}$. If t_i and t_j are assigned to the same processor, the communication cost $c_{i,j}$ is zero.

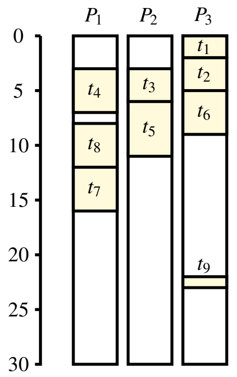
Consider an edge $e_{i,j}$ that starts from t_i and ends at t_j . The source node t_i is called a predecessor of t_j, and the destination node t_j is called a successor of t_i. In Fig. 1a, t_1 is the predecessor of t_2, t_3, t_4, and t_5; t_2, t_3, t_4, and t_5 are the successors of t_1. In a graph, a node with no parent is called an entry node, and a node with no child is called an exit node. If a node t_i is scheduled to a processor P_j, then the start time and the finishing time of t_i are denoted by $\mathrm{ST}(t_i, P_j)$ and $\mathrm{FT}(t_i, P_j)$, respectively.



(a)

	P_1	P_2	P_3
t_1	3	1	2
t_2	3	3	3
t_3	4	3	2
t_4	4	2	6
t_5	7	5	3
t_6	2	6	4
t_7	4	6	2
t_8	4	3	5
t_9	1	1	1

(b)



(c)

Fig. 1 Illustraion example: **a** DAG, **b** the computation cost matrix, and **c** feasibility of scheduling

2.2 Makespan

After all of the tasks are scheduled onto parallel processors, considering a specific task t_i on a processor P_j, the start time $\mathrm{ST}(t_i, P_j)$ can be defined as

$$\mathrm{ST}(t_i, P_j) = \max\{\mathrm{RT}_j, \mathrm{DAT}(t_i, P_j)\},$$

where RT_j is the ready time of P_j and $\mathrm{DAT}(t_i, P_j)$ is the data arrival time of t_i at P_j. Because $\mathrm{DAT}(t_i, P_j)$ is the time for all of the required data to be received, it is computed by

$$\mathrm{DAT}(t_i, P_j) = \max_{t_k \in \mathrm{pred}(t_i)} \{(\mathrm{FT}(t_k, P_j) + c_{k,i})\},$$

where $\mathrm{pred}(t_i)$ denotes the set of immediate predecessors of t_i. Because

$$\mathrm{FT}(t_k, P_j) = w_{k,j} + \mathrm{ST}(t_k, P_j)$$

and

$$\mathrm{RT}_j = \max_{t_k \in \mathrm{exe}(P_j)} \{\mathrm{FT}(t_k, P_j)\},$$

where $\mathrm{exe}(P_j)$ is the set that contains the tasks that are assigned to be executed on P_j, the makespan, i.e., the overall schedule length of the entire program, is the latest finishing time of all of the tasks and can be obtained by

$$\text{makespan} = \max_{t_i \in V} \{\mathrm{FT}(t_i, P_j)\}.$$

Figure 1c demonstrates a scheduling for the graph described in Fig. 1a. The makespan of this scheduling is 23.

3 Related work

Task scheduling has been proven to be an NP-complete problem. There are several algorithms, including deterministic approaches [25] or non-deterministic approaches [20,26], which were proposed to solve the problem of dependent tasks scheduling for real applications.

Topcuoglu [25] classified deterministic algorithms into three main types: list-scheduling algorithms, clustering algorithms, and task duplication algorithms. Among these deterministic approaches, list-scheduling algorithms are practical and provide better performance results with less scheduling time than the others. The heterogeneous earliest-finish-time (HEFT) algorithm [25] is most efficient in list-based algorithms; this algorithm takes advantage of an upward-ranking method and an insertion-based policy to reach the goal of minimizing the makespan. However, HEFT scheduling does not perform very well for irregular DAGs and for higher heterogeneities [1].

On the other hand, since non-deterministic algorithms are popular for a wide range of combinatorial problems, this class of algorithms are successfully applied to solve the task-scheduling problem, such as GAs, ant colony optimization, particle swarm optimization, and hybrid heuristic techniques [6,26]. This kind of algorithm takes more time to explore the solution space for a high-quality solution, compared to deterministic algorithms. For more heuristic task-scheduling algorithms, please see [6,22,29].

GAs have been shown to outperform several algorithms in task-scheduling problems [8,27], in which each solution is represented as a chromosome. Three genetic operators, selection, crossover, and mutation, are employed to transform a population of chromosomes to another better population. To retain the good features from the previous generation, the crossover operator exchanges the information from two chromosomes, which are chosen randomly. The mutation operator alters one bit of a chromosome to increase the diversity of the chromosomes and prevent premature convergence. This genetic process is repeated until the stopping criterion is satisfied, and the best chromosome from all of the generations is reported as the best solution.

Omara [20] proposed the standard genetic algorithm (SGA), which gives a standard pattern to demonstrate this process. Additionally, Omara proposed the critical path genetic algorithm (CPGA), which assigns the tasks in the critical path to the same processor, to reduce the longest communication time. However, the performance of CPGA is not very efficient for graphs that are composed of task sets containing more than one critical path.

The genetic variable neighborhood search (GVNS) algorithm [26] combines a GA with a variable neighborhood search (VNS) to obtain a better load balance between the global exploration and the local exploitation of the search space. Two neighborhood structures, the load balancing neighborhood structure and the communication reduction neighborhood structure, are used by VNS. After the crossover operator and the mutation operator, the current population and the new population are merged and sorted by makespan in an increasing order.

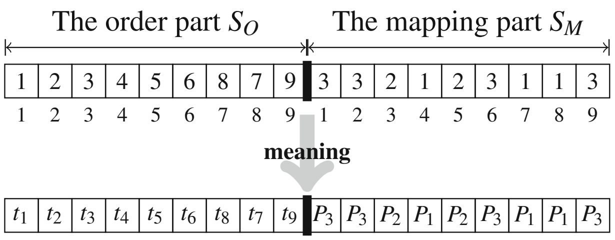


Fig. 2 A representation of a chromosome S

4 Proposed method

In this section, we introduce the TSGA algorithm in detail, including the encoded and decoded representations and genetic operators.

4.1 The representation of solutions

The representation of a chromosome is given in Fig. 2. A chromosome S is composed of two parts: an order part (S_O) and a mapping part (S_M). We use integer arrays to store S_O and S_M, and the size of the arrays equals the number of tasks. If $S_O[i]$ is j and $S_M[i]$ is k, then a task t_j is executed on processor P_k.

According to the chromosome represented in Fig. 2, the solution of a DAG in Fig. 1a can be scheduled in Fig. 1c. We first assign tasks into the mapping processor according to the index of S_M. Tasks t_4, t_7, and t_8 are scheduled on processor P_1. Tasks t_3, and t_5 are executed on processor P_2. Tasks t_1, t_2, t_6, and t_9 are assigned to processor P_3. Following the order in S_O, we schedule t_4, t_7, and t_8 in the order of t_4, t_8, t_7 in P_1. For P_2, t_3 is executed before t_5. Tasks t_1, t_2, t_6, and t_9 are taken in the order of t_1, t_2, t_6, t_9 in P_3. We lastly compute the makespan by including the commutation time for each task.

4.2 Outline

The algorithmic outline of TSGA is given in Algorithm 1, and its flowchart in Fig. 3. In Line 2, a variable g counts the number in the past generation. If g is not less than G, TSGA will output the best solution. Lines 4 to 7 are the main processes that produce the next generation. Line 8 sieves out the best C chromosomes from both the old and the new generation according to the fitness function, where C is the number of chromosomes.

The crossover operator exchanges the information from two chromosomes to hold some good features from the previous generation. Because we can obtain better or worse chromosomes than the parent chromosomes, the crossover operator is applied with a predestined probability called the crossover rate, which is denoted by p_c. On the other hand, the mutation operator alters one bit of a chromosome, to increase the diversity of the chromosomes and prevent premature convergence. The mutation rate is p_m.

Algorithm 1 TSGA

```
Require: $G$: number of generation
    $C$: population
    DAG: a graph with $N$ tasks and $M$ communication edges
    $n$: the number of processors
    $p_c$: the probability of the crossover operator
    $p_m$: the probability of the mutation operator
Ensure: Best scheduling length
1: Initialization_operator ($C$, $n$, DAG)
2: $g \leftarrow 0$
3: while $g < G$ do
4:    for each chromosome in $C$ do
5:       Crossover_operator($p_c$)
6:       Mutation_operator($p_m$)
7:    end for
8:    Select chromosomes according to the fitness function
9:    $g$++
10: end while
11: return the best solution
```

4.3 Initialization

TSGA initializes the first population that consists of encoded chromosomes, as shown in Algorithm 2. Instead of using a random strategy to give the processor assignment, we devise a new method that divides the search space into specific patterns and a new rank value that is used to build the executed order. Because the best scheduling solution can occupy a number of processors in different cases, we provide some patterns with a different number of processors to explore the solution space in different aspects. In Line 1, the search space is divided into $\log_2 n$ subspaces, where n is the number of processors. For each group d, the processor assignment S_M is given by an integer random function in the range of $\{1 \ldots 2^d\}$, and the execution order S_O follows Rank_order(DAG), which is depicted in Algorithm 3. Note that the notation $\oplus$ is the concatenation of two strings in Line 7.

Because a task should be executed earlier if the task has more successors, we use the Rank_order(DAG) procedure to help the implementation of this idea. Lines 2 to 8 compute the r value by traversing the task graph upward, starting from the exit task. If a task is the exit task, the r value of the task is 1. Otherwise, the r value of a task is the summation of its successors' r values and 1. In Line 6, the notation $\text{succ}(t_i)$ is the set of immediate successors of t_i. An example of calculation for r is given in Table 1.

4.4 Crossover

The crossover operator exchanges the information in two chromosomes to retain the good features from the parent generation. Each part of a chromosome is applied to a specific crossover operator: the crossover map operator or the crossover order operator. These two operators have the same probability to be taken. The pseudo code of the crossover operator is given in Algorithm 4.

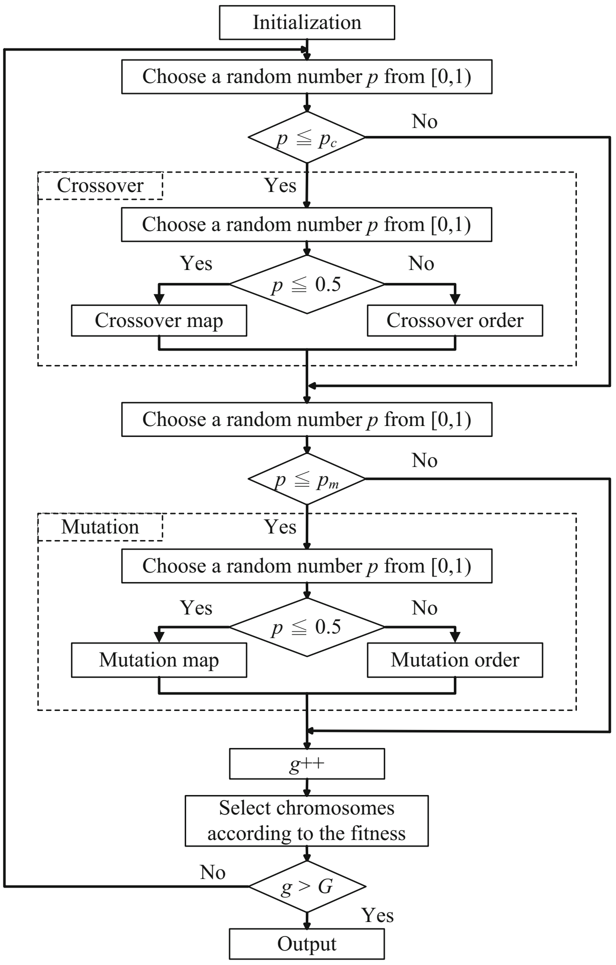

Initialization
Choose a random number p from [0,1)
$p \leqq p_c$
No
Crossover
Yes
Choose a random number p from [0,1)
Yes
$p \leqq 0.5$
No
Crossover map
Crossover order
Choose a random number p from [0,1)
$p \leqq p_m$
No
Mutation
Yes
Choose a random number p from [0,1)
Yes
$p \leqq 0.5$
No
Mutation map
Mutation order
g++
Select chromosomes according to the fitness
No
$g > G$
Yes
Output

Fig. 3 TSGA flowchart

Algorithm 2 Initialization_operator(C, n, DAG)

Require: C: population set; n: the number of processors
Ensure: The population C

```
1: $d \leftarrow \log_2 n$    {Divide chromosomes into $d$ groups}
2: for group = 1 to $d$ do
3:     for $i = 1$ to population_ size/$d$ do
4:         $S_M \leftarrow$ choose a processor for each task from $\{1, 2, \ldots, 2^d\}$
5:         $S_O \leftarrow$ an executed order according to the Rank_order(DAG)
6:         $S \leftarrow S_O \oplus S_M$
7:         $C \leftarrow C \bigcup \{S\}$
8:     end for
9: end for
10: return $C$
```

Algorithm 3 Rank_order(DAG)

```
Require: DAG: a graph with $N$ tasks and $M$ communication edges
Ensure: The executed order list $L$
1: Set the executed order list $L$ as empty
2: for $i = N$ down to 1 do
3:     if task $t_i$ is the exit node then
4:         $r(t_i) = 1$
5:     else
6:         $r(t_i) = \sum_{t_j \in \mathrm{succ}(t_i)} r(t_j) + 1$
7:     end if
8: end for
9: $L \leftarrow$Sort tasks according to the $r$ value in a non-increasing order
10: return $L$
```

Table 1 An example of calculation for r

Task	r	Calculation
t_1	15	$r(t_2) + r(t_3) + r(t_4) + r(t_5) + r(t_7) + 1$
t_2	3	$r(t_6) + 1$
t_3	5	$r(t_7) + r(t_8) + 1$
t_4	1	1
t_5	3	$r(t_8) + 1$
t_6	2	$r(t_9) + 1$
t_7	2	$r(t_9) + 1$
t_8	2	$r(t_9) + 1$
t_9	1	1

The relationship between the tasks and processors is the most important information encoded in the chromosomes for the scheduling problem. However, crossover operators in most previous studies use some simple cut-and-paste processes to produce offspring. These are just random steps and cannot keep the valuable features of the task–processor assignments from the parents. Our crossover operator is based on the task–processor relationship in such a way that parents can pass good features to their children.

Algorithm 4 Crossover_operator()

```
1: $p \leftarrow$ random[0, 1)
2: if $p \le 0.5$ then
3:     Crossover_map()
4: else
5:     Crossover_order()
6: end if
```

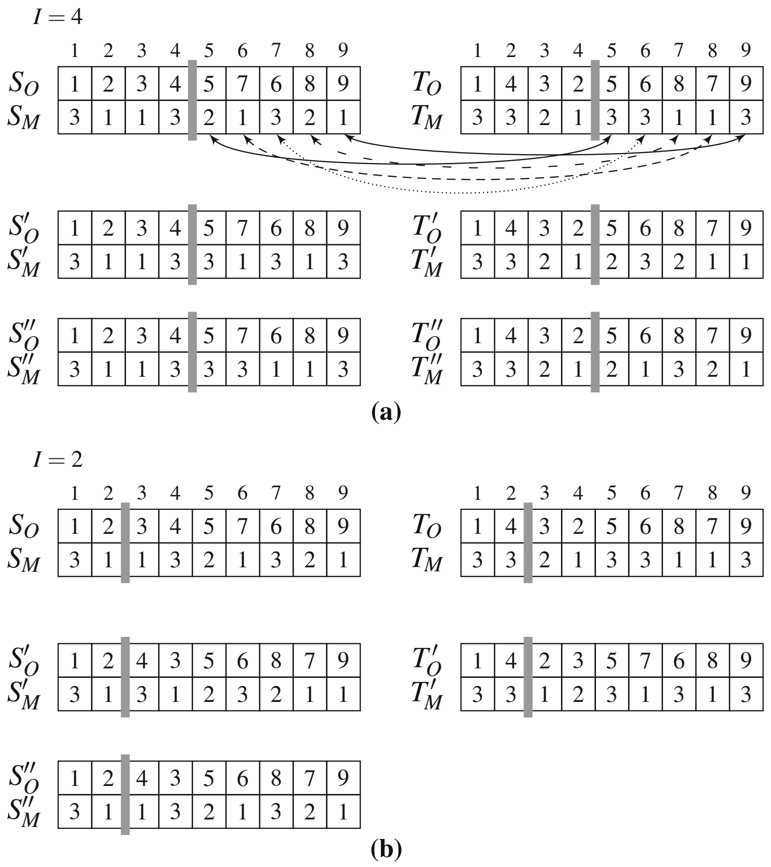


Fig. 4 An example of crossover: **a** the crossover map operator and **b** the crossover order operator

4.4.1 Crossover map operator

The crossover map operator is used to interchange the mapping assignments of two chromosomes, as shown in Fig. 4a. The description of the crossover map operator is given in Algorithm 5. The crossover map operator chooses two chromosomes, S and T, from the population and picks a random number I from the set $\{1, 2, \ldots, N\}$. TSGA keeps the processor assignment, which is located on the left of I (including I). For the processors on the right of I, our crossover map swaps the processor numbers of S_M and T_M, which are assigned to execute the same task according to the task places in S_O and T_O. For example, because t_5 is assigned to P_2 in S_M and to P_3 in T_M, the crossover map swaps the processor assignments; in other words, it maps t_5 to P_3 in S'_M and to P_2 in T'_M. Note that the order part of the two chromosomes does not need to be changed.

The chromosomes S''_M and T''_M are generated by the crossover map operator in SGA [20] by a cut-and-paste process, which exchanges $S_M[5..9]$ and $T_M[5..9]$ directly. Task t_7 is executed on P_1 in both parents. After the crossover map, t_7 is executed on processor

P_1 and P_2 in S''_M and T''_M, respectively, while our crossover map assigns t_7 to P_1 in both S'_M and T'_M. The result shows that the operator can inherit good features from the previous chromosomes successfully.

Algorithm 5 Crossover_map()

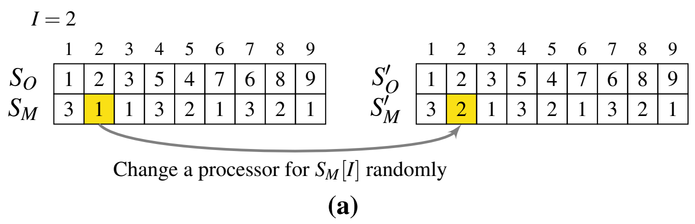
```
1: Select two chromosomes $S$ and $T$
2: $I \leftarrow$ select a number between 1 and $N$ randomly
3: $S'_O \leftarrow S_O$
4: $T'_O \leftarrow T_O$
5: $S'_M[1..I] \leftarrow S_M[1..I]$
6: $T'_M[1..I] \leftarrow T_M[1..I]$
7: for $i = I+1$ to $N$ do
8:     $S'_M \leftarrow S'_M \oplus$ the processor in which task $S_O[i]$ is assigned in $T_M$
9:     $T'_M \leftarrow T'_M \oplus$ the processor in which task $T_O[i]$ is assigned in $S_M$
10: end for
11: $S' \leftarrow S'_O \oplus S'_M$
12: $T' \leftarrow T'_O \oplus T'_M$
```

Algorithm 6 Crossover_order()

```
1: Select two chromosomes $S$ and $T$
2: $I \leftarrow$ Select a number between 1 and $N$ randomly
3: $S'_O \leftarrow S_O[1..I]$
4: for $i = 1$ to $N$ do
5:     if $T_O[i]$ is not in $S'_O$ then
6:         $S'_O \leftarrow S'_O \oplus T_O[i]$
7:     end if
8: end for
9: Build $S'_M$ according to the task–processor assignments in $S_M$
10: $S' \leftarrow S'_O \oplus S'_M$
11: $T'_O \leftarrow T_O[1..I]$
12: for $i = 1$ to $N$ do
13:     if $S_O[i]$ is not in $T'_O$ then
14:         $T'_O \leftarrow T'_O \oplus S_O[i]$
15:     end if
16: end for
17: Build $T'_M$ according to the task–processor assignments in $T_M$
18: $T' \leftarrow T'_O \oplus T'_M$
```

4.4.2 Crossover order operator

The description of the crossover order operator is listed in Algorithm 6. In Fig. 4b, after selecting two chromosomes S and T, we choose a crossover point I from the set $\{1, 2, \ldots, N\}$ and copy the left portion of I (including I) in S_O and T_O to the new order parts S'_O and T'_O, respectively. For the right segment of I in S'_O, our crossover order operator follows the task order in T_O, except for the tasks in the left segment on I of S'_O. We then use the same process to generate T'_O. Because we only produce new chromosomes by the order part, the task–processor assignment should be maintained



	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
DAT	0	6	3	2	3	10	12	17	26

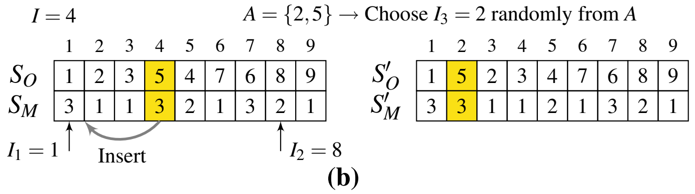


Fig. 5 An example of mutation: **a** the mutation map operator and **b** the mutation order operator

the same as the parents. Thus, we adjust the map part according to the original processor assignment.

The crossover order operator in SGA does not adjust the processor assignment after completing the crossover order operator. As shown in Fig. 4b, the chromosome S'', created by the crossover order operator in SGA, lost the task–processor assignments of its parents, since the order part is changed while the map part is not adjusted correspondingly.

4.5 Mutation

The mutation operator is typically implemented by altering one element of a chromosome at random. Because we have order and map parts in a chromosome, each part has a specific mutation operator: the mutation map operator or the mutation order operator. These two operators have the same probability to be taken, similar to our crossover operator.

4.5.1 Mutation map operator

The mutation map operator is used for the map part of the chromosome, which is shown in Fig. 5a. Similar to the crossover operator, we choose a random number I from the set $\{1, 2, \ldots, N\}$. Then, the mutation map operator changes the processor in which the chosen task $S_O[I]$ is executed to another processor that is chosen randomly.

4.5.2 Mutation order operator

The mutation order operator is applied to the order part of the chromosome, which is given in Algorithm 7. We select an index I from 1 to N randomly, which decides a task $S_O[I]$ to be mutated. For the task $S_O[I]$, let I_1 and I_2 be the indices of its closest predecessor on the left segment of I and the closest successor on the right segment of I, respectively. For the index i between $I_1 + 1$ and I, if DAT of $S_O[i]$ is larger than DAT of $S_O[I]$, the task $S_O[I]$ should be executed before the task $S_O[i]$, because the task $S_O[I]$ can be executed early. For the index i between I and $I_2 - 1$, if DAT of $S_O[i]$ is smaller than DAT of $S_O[I]$, the task $S_O[i]$ should be executed before the task $S_O[I]$, since the task $S_O[i]$ can be executed early. We collect the tasks that can be executed early into a set called A. If A is empty, then we put all of the indices between $I_1 + 1$ and $I_2 - 1$ into A. We select an index I_3 from A randomly and insert the task $S_O[I]$ before the index I_3. We lastly adjust the processor assignment to correspond with the original processor assignment, for the same reason mentioned in the crossover order operator.

Algorithm 7 Mutation_order()

```
1: Select two chromosomes $S$ and $T$
2: $S' \leftarrow S$
3: $I \leftarrow$ Select a number between 1 and $N$ randomly
4: $A \leftarrow$ an empty set
5: $I_1 \leftarrow$ the index of the first predecessor of task $S_O[I_0]$ on the left of $I_0$
6: $I_2 \leftarrow$ the index of the first successor of task $S_O[I_0]$ on the right of $I_0$
7: for $i$ from $I_1 + 1$ to $I_0$ do
8:   if DAT of $S_O[i]$ > DAT of $S_O[I_0]$ then
9:     $A \leftarrow A \bigcup \{i\}$
10:  end if
11: end for
12: for $i$ from $I_0$ to $I_2 - 1$ do
13:   if DAT of $S_O[i]$ < DAT of $S_O[I_0]$ then
14:     $A \leftarrow A \bigcup \{i\}$
15:   end if
16: end for
17: if $A$ is empty then
18:   $A \leftarrow$ a set contains indexes between $I_1 + 1$ and $I_2 - 1$
19: end if
20: $I_3 \leftarrow$ Select an index from $A$ randomly
21: Move $S_O[I_0]$ to $I_3$
22: Adjust the processor assignment to correspond to the original assignment
```

Figure 5b demonstrates the mutation order operator for $I = 4$. Since $S_O[4]$ is t_5, the mutation order operator changes the execution order of t_5. According to the graph shown in Fig. 1a, we have the table with the DAT value and find $I_1 = 1$ and $I_2 = 8$. We then have that the elements in the set A are the indices 2 and 5, because $S_O[2]$ is t_2 and $6 = \text{DAT}(t_2) > \text{DAT}(t_5) = 3$, and $S_O[5]$ is t_4 and $2 = \text{DAT}(t_4) < \text{DAT}(t_5) = 3$. If we choose the index $I_3 = 2$ randomly from A, the task t_5 is inserted in the front of t_2 (i.e., $S_O[2]$). Additionally, the processor P_1 (i.e., $S_M[4]$) should be moved correspondingly.

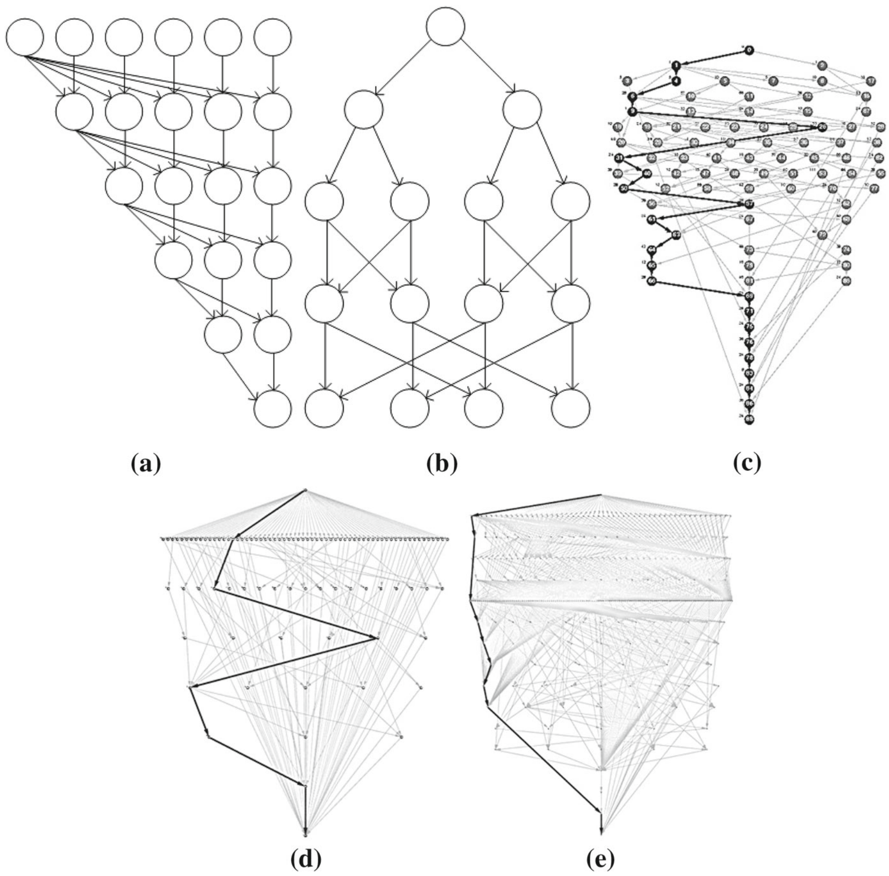


Fig. 6 The graph structure. **a** 21-task GJ, **b** 15-task FFT, **c** 88-task Robot control, **d** 96-task Sparse matrix solver, and **e** 334-task SPEC fpppp

5 Experimental results

At the beginning of this section, we describe the simulation environment for realistic problems. We next demonstrate five DAGs of well-known parallel applications and explain the related parameters. We lastly show the performance of TSGA compared with SGA, CPGA, GVNS, and HEFT.

5.1 Parameter Setting

This study considers five well-known applications: Gauss–Jordan elimination (GJ) [28], the fast Fourier transformation (FFT) [25], Robot control, Sparse matrix solver, and SPEC fpppp. Robot control, sparse matrix solver, and SPEC fpppp are taken from a Standard Task Graph (STG) archive [30]. Their graph structures are drawn in Fig. 6 and the detailed information is given in Table 2.

Table 2 Parameter settings used in the experiment

	GJ	FFT	Robot control	Sparse matrix solver	SPEC fpppp
Number of tasks	300	223	88	96	334
Number of edges	552	382	131	67	1,145

Each application has various features that depend on different settings, as described below:

1. Number of processors (P): P is the number of processors in the system. The target heterogeneous system is a fully connected computing network in which various processors are connected to each other. Note that the number of processors and the performance speedup are not linearly proportional due to the communication delay.
2. Heterogeneity factor (β): This value indicates the range percentage of the computation cost on the processors. The variable β is defined by the ratio of the most efficient processing capability and the least efficient processing capability in a system. We set the most efficient processing capability to be 1. Thus, we have $0 < \beta \leq 1$. For practical computing environments, heterogeneous systems could contain some processors that have different processing capabilities. Here, we set four types of processing capability, and there is the same number of processors in each type.
3. Communication-to-computation ratio (CCR): For a graph, the CCR is defined as the average communication time of the edges divided by the average computation time of the nodes in the system. A graph is represented as a computation-intensive application if its CCR is low. On the other hand, if the CCR value is high, then it is a communication-intensive application. The communication time of DAGs is generated by uniform distribution.

The values of the corresponding sets shown below are the settings for the experimental parameters.

- $\text{SET}_P = \{4, 8, 16\}$
- $\text{SET}_\beta = \{0.1, 0.25, 0.5, 0.75, 1.0\}$
- $\text{SET}_{\text{DAG}} = \{$GJ, FFT, Robot control, Sparse matrix solver, SPEC fpppp$\}$
- $\text{SET}_{\text{CCR}} = \{0.25, 0.5, 1.0, 2.0, 4.0\}$

15 distinct system models were created by the combination of SET_P and the set SET_β. The different settings of SET_{DAG} and SET_{CCR} generate 25 different types of DAGs, and 10 graphs are produced for each type of DAG; thus the total number of DAGs is 250. The combinations of all of these parameters give 3,750 various scenarios. Based on these DAGs with different characteristics, the experimental result will not be biased toward any specific algorithm.

5.2 Performance results

To evaluate our proposed algorithms, we have implemented them using an AMD FX(tm)-8120 eight-core processor (3.10 GHz) using the C^{++} language. Usually, an

Table 3 Parameter settings used in the experiment

Population size	Generation size	Crossover rate	Mutation rate	Selection operator
400	1,000	0.8	0.2	Binary tournament

excellent solution to various problems would be generated by a variety of parameters for a specific algorithm. However, we use the same parameter values that were listed in Table 3, to show the performance in terms of makespan in this study.

The performance of TSGA is compared with four algorithms, CPGA, SGA, GVNS, and HEFT. For each data configuration of the five DAGs, we record the average makespan obtained over ten runs for CPGA, SGA, GVNS, and TSGA. On the other hand, HEFT, a deterministic algorithm, is run only once. Please note that the time complexity of HEFT is $O(n^2p)$, where n is the number of tasks and p is the number of processors, and the time complexity of these GAs are $O(n^2p)$ for fixed population size and number of generations. Thus, it is evident that HEFT is fastest among these five algorithm.

The experimental results of the five DAGs with P fixed to 16 and varying the β value are given in Fig. 7, which presents the histograms for the makespan. Each application is shown in its row and was tested with a different CCR value and a varying β value. Each column contains five applications with a fixed CCR value. Note that the vertical coordinate shows the makespan. To demonstrate that the stability of TSGA is accepted, we add the standard deviation for each TSGA's result in Fig. 7.

The experiment reveals that TSGA outperforms other algorithms in terms of makespan in our test cases with different characteristics. Because CPGA schedules the nodes of the critical path to the same processor, it has good outcomes in Robot control. However, CPGA produces unfavorable scheduling results in other applications. GVNS and SGA show their good scheduling ability only in certain cases. When the CCR value increases, the performance of SGA decreases. On the other hand, although HEFT shows efficient scheduling results, TSGA is better than HEFT in terms of makespan in most cases.

For all of the algorithms, the makespan in the tests with a larger β value is always smaller than those with a smaller β value. The system with a higher β value has a larger number of processors with good processing capability. If a task is assigned to a processor that has high processing capability, then the task will be completed early. In general, the makespan decreases when β increases.

When we consider a smaller β value ($\beta = 0.1$), there are more possible scheduling solutions. For a small value of CCR, we can produce good scheduling easily because the application has less communication delay and less idle time. We therefore focus on a large CCR value (CCR $= 8$). The results, which are provided by testing five applications, are given in Fig. 8.

For the concept of parallelism, when the number of processors increases, the makespan will decrease. However, most of the algorithms will produce an increased makespan, with an increasing number of processors in certain cases, especially when the CCR value is large, as shown in Fig. 8. Having a larger CCR value implies that the communication time is larger than the computation time. In these scenarios, good

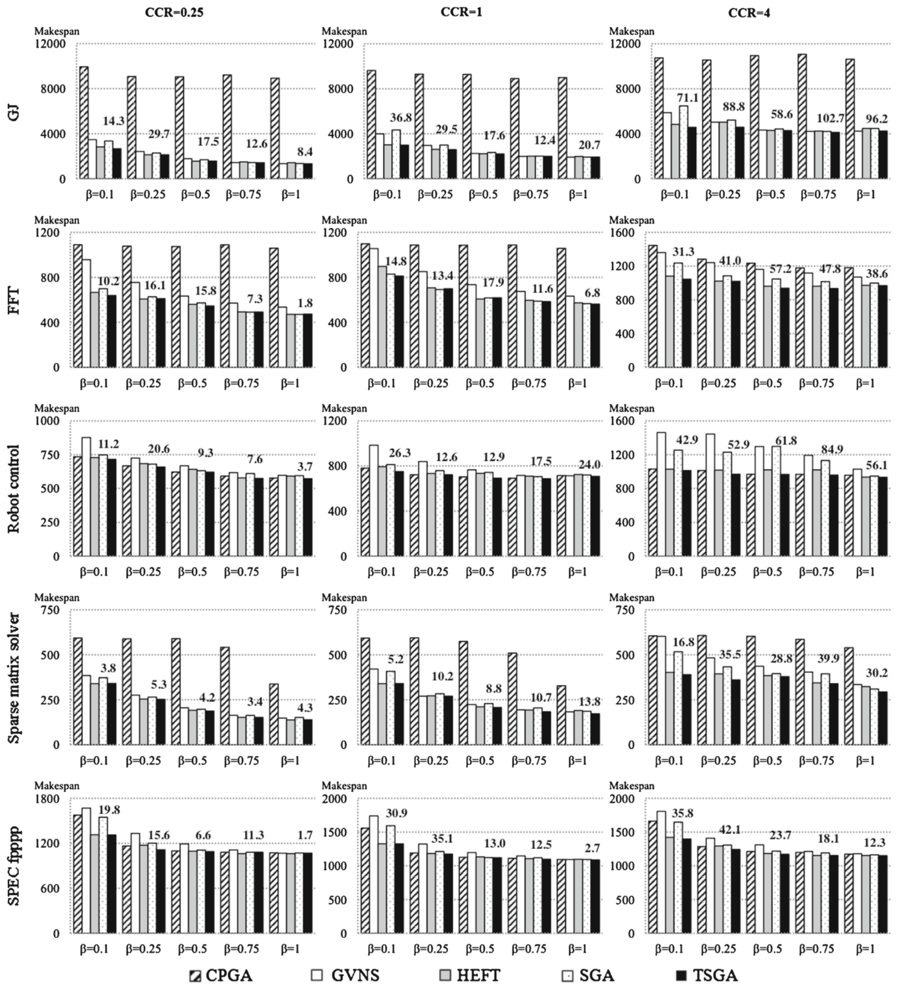


Fig. 7 Makespan comparison with varying parameters for $P = 16$. Each number represents the standard deviation of the corresponding TSGA's result

scheduling should not occupy most of the processors. Most of the genetic algorithms cannot catch this factor and initialize with random strategies. On the other hand, TSGA with the initialization operator can reduce the probability of this phenomenon and produce better scheduling results in all of the cases.

To demonstrate that the proposed processor assignment strategy is efficient at the initial step, we give the comparison of genetic algorithms that employ the original initial operator or our proposed processor assignment technique. Figure 9 is the result of testing with $\beta = 0.1$ and $\text{CCR} = 8$, where the three new algorithms CPGA′, GVNS′, and SGA′ are the three algorithms CPGA, GVNS, and SGA that employ our proposed initialization operator, respectively. Obviously, the new algorithms always outperform the original algorithms by using the random assignment technique. Although the qual-

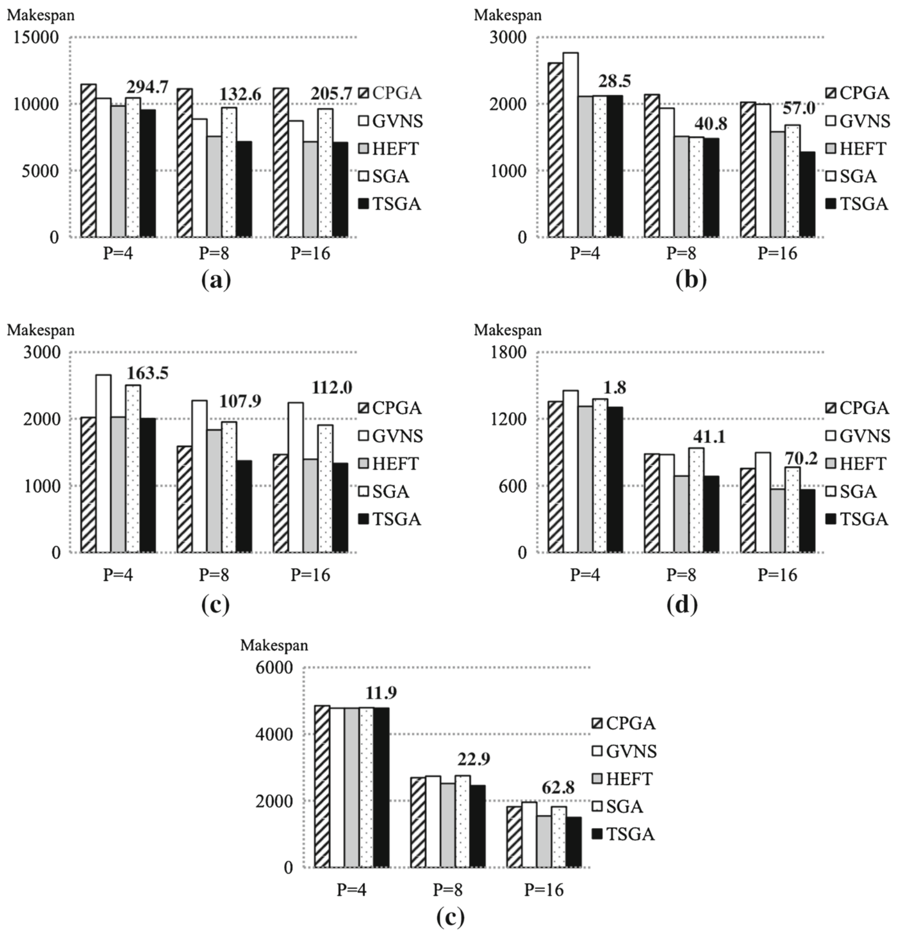


Fig. 8 Makespan comparison with varying P. **a** GJ, **b** FFT, **c** robot control, **d** sparse matrix solver, and **e** SPEC fpppp for CCR $= 8$ and $\beta = 0.1$. Each number represents the standard deviation of the corresponding TSGA's result

ities of those algorithms have been improved, our proposed method TSGA always has the best scheduling results among all of the test cases.

6 Conclusions

In this study, we presented a genetic algorithm for task scheduling, referred to as TSGA, to solve the problem of task scheduling on parallel and distributed computing systems to improve the standard genetic algorithm by increasing the convergence speed and preserving the good features of the previous generation. To demonstrate the TSGA performance, we introduce new genetic operators in TSGA. The initialization operator that was proposed divides the search space into specific patterns to allow TSGA to explore the search space extensively. The crossover map operator and the

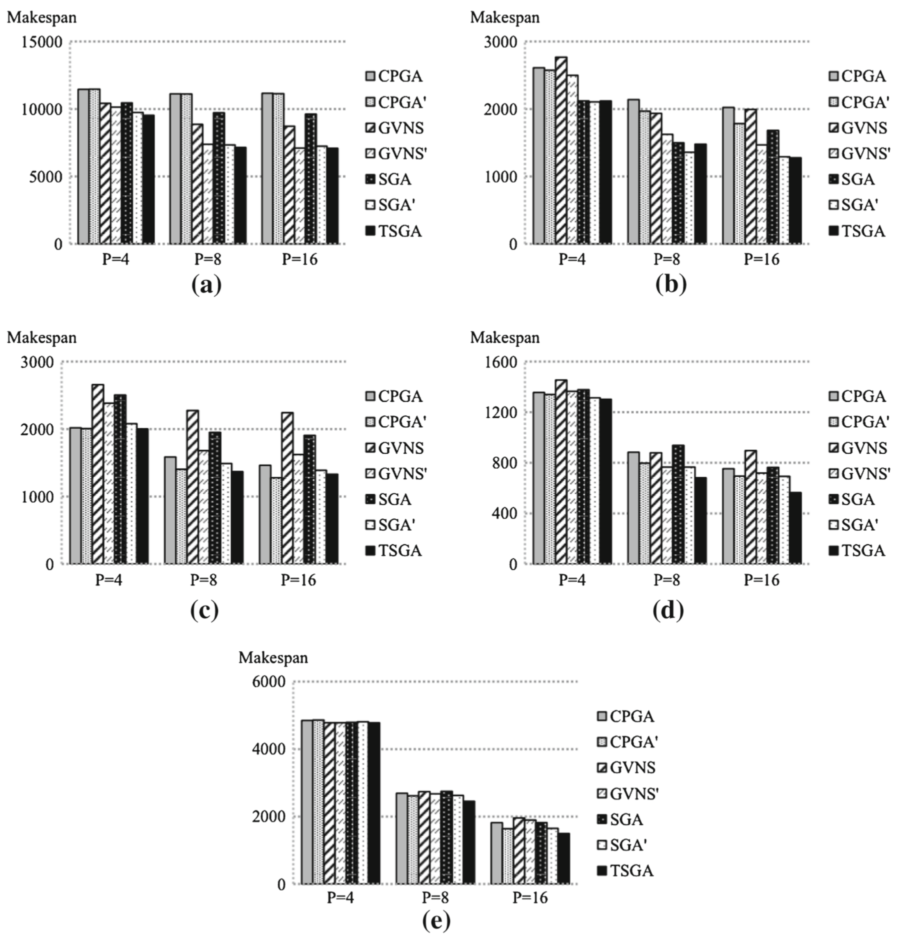


Fig. 9 Makespan comparison of algorithms with our processor assignment strategy for **a** GJ, **b** FFT, **c** robot control, **d** sparse matrix solver, and **e** SPEC fpppp

mutation map operator help us to search more suitable processor assignments, and the crossover order operator and the mutation order operator provide us with a more efficient execution order. Compared with the four related algorithms, SGA, CPGA, GVNS, and HEFT, the experimental results show that our proposed method TSGA outperforms other algorithms in a variety of scenarios in terms of makespan. We also give the experimental results to show the relationship between TSGA and various parameters. For cases with high CCR values, TSGA schedules tasks that occupy fewer processors to reduce the communication time. For cases with small β values, TSGA assigns tasks that occupy the processors having higher processing capability to minimize the execution time. Furthermore, we use other genetic algorithms to show the advantage of our initialization operator and the satisfactory results.

References

1. Arabnejad H, Barbosa JG (2014) List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Trans Parallel Distrib Syst 25(3):682–694
2. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. J Parallel Distrib Comput 10(2):188–193
3. Arabnia HR, Oliver MA (1989) A transputer network for fast operations on digitised images. Comput Graph Forum 8(1):3–12
4. Bhandarkar SM, Arabnia HR (1995) The REFINE multiprocessor: theoretical properties and algorithms. Parallel Comput 21(11):1783–1806
5. Culler D, Singh J, Gupta A (1998) Parallel computer architecture: a hardware/software approach. Morgan Kaufmann Publisher, San Francisco
6. Chitra P, Rajaram R, Venkatesh P (2011) Application and comparison of hybrid evolutionary multiobjective optimization algorithms for solving task scheduling problem on heterogeneous systems. Appl Soft Comput 11(2):2725–2734
7. Choudhury P, Chakrabarti PP, Kumar R (2012) Online scheduling of dynamic task graphs with communication and contention for multiprocessors. IEEE Trans Parallel Distrib Syst 23(1):126–133
8. Falzon G, Li M (2012) Enhancing genetic algorithms for dependent job scheduling in grid computing environments. J Supercomput 62(1):290–314
9. Freund RF, Siegel HJ (1993) Guest editor's introduction: heterogeneous processing. Computer 26(6):13–17
10. Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor
11. Hwang K (1993) Advanced computer architecture: parallelism, scalability, programmability. McGraw-Hill Inc, New York
12. Hou ESH, Ansari N, Ren H (1994) A genetic algorithm for multiprocessor scheduling. IEEE Trans Parallel Distrib Syst 5(2):113–120
13. Hyunjin K, Sungho K (2011) Communication-aware Task scheduling and voltage selection for total energy minimization in a multiprocessor system using ant colony optimization. Inf Sci 181(18):3995–4008
14. Kwok YK, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput Surv 31(4):406–471
15. Leighton FT (1992) Introduction to parallel algorithms and architectures: arrays, trees, hypercubes. Morgan Kaufmann, San Mateo
16. Liou J, Palis MA (1996) An efficient task clustering heuristic for scheduling DAGs on multiprocessors. In: Proceeding of workshop on resource management, symposium of parallel and distributed processing, pp 152–156
17. Liu H, Abraham A, Snášel V, McLoone S (2012) Swarm scheduling approaches for work-flow applications with security constraints in distributed data-intensive computing environments. Inf Sci 192:228–243
18. Jiang YS, Chen WM (2013) Task scheduling in grid computing environments. In: Proceedings of the Seventh International Conference on Genetic and Evolutionary Computing, pp 23–32
19. Mirabi M (2011) Ant colony optimization technique for the sequence-dependent flowshop scheduling problem. Int J Adv Manufact Technol 55(1–4):317–326
20. Omara FA, Arafa MM (2010) Genetic algorithms for task scheduling problem. J Parallel Distrib Comput 70(1):13–22
21. Rewini HE, Lewis T, Ali H (1994) Task scheduling in parallel and distributed systems. Prentice Hall, New Jersey
22. Rahman M, Hassan R, Ranjan R, Buyya R (2013) Adaptive workflow scheduling for dynamic grid and cloud computing environment. Concurr Comput Pract Exp 25(13):816–1842
23. Tang X, Li K, Liao G, Li R (2010) List scheduling with duplication for heterogeneous computing systems. J Parallel Distrib Comput 70(4):323–329
24. Tao Q, Chang HY, Yi Y, Gu CQ, Li WJ (2011) A rotary chaotic PSO algorithm for trustworthy scheduling of a grid workflow. Comput Oper Res 38(5):824–836
25. Topcuoglu H, Hariri S, Wu M-Y (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst 13(3):260–274

26. Wen Y, Xu H, Yang J (2011) A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system. Inf Sci 181(3):567–581
27. Wu AS, Yu H, Jin S, Lin K, Schiavone G (2004) An incremental genetic algorithm approach to multiprocessor scheduling. IEEE Trans Parallel Distrib Syst 15(9):824–834
28. Yu H (2008) Optimizing task schedules using an artificial immune system approach. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, pp 151–158
29. Zarrabi A, Samsudin K (2014) Task scheduling on computational grids using gravitational search algorithm. Cluster Comput 17(3):1001–1011
30. http://www.Kasahara.Elec.Waseda.ac.jp/schedule/