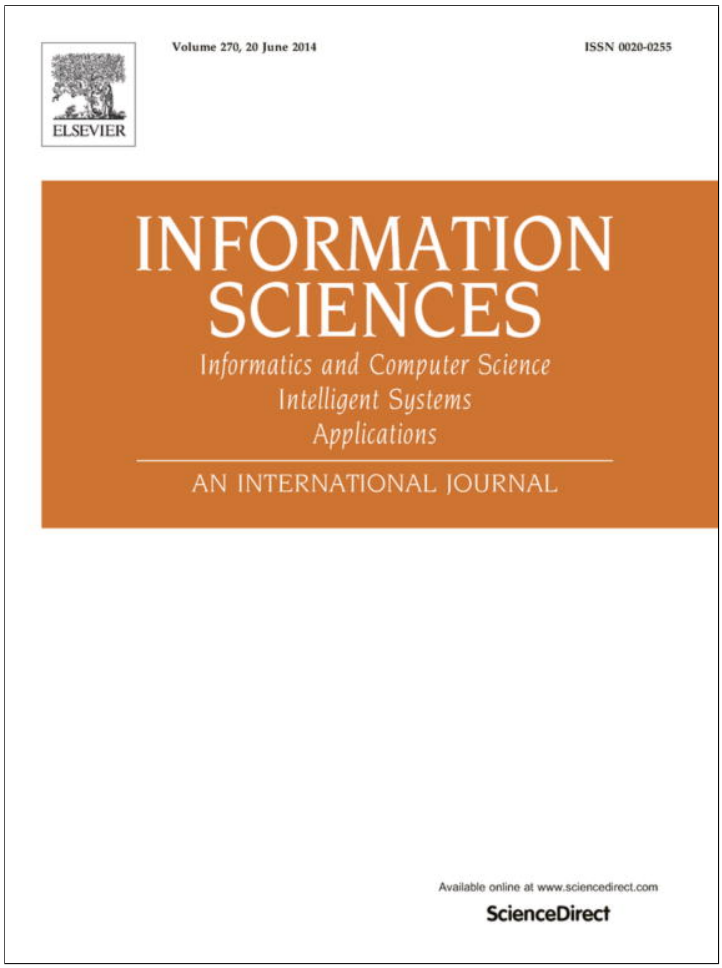


Information Sciences 270 (2014) 255–287

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues

Yuming Xu [a], Kenli Li [a,*], Jingtong Hu [b], Keqin Li [a,c]

[a] College of Information Science and Engineering, Hunan University, Changsha 410082, China
[b] School of Electrical and Computer Engineering, Oklahoma State University, Stillwater, OK 74078, USA
[c] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

Article history:
Received 7 June 2013
Received in revised form 19 January 2014
Accepted 11 February 2014
Available online 27 February 2014

Keywords:
Directed acyclic graph
Genetic algorithm
Heuristic algorithm
Makespan
Multiple priority queue
Task scheduling

ABSTRACT

On parallel and distributed heterogeneous computing systems, a heuristic-based task scheduling algorithm typically consists of two phases: task prioritization and processor selection. In a heuristic based task scheduling algorithm, different prioritization will produce different makespan on a heterogeneous computing system. Therefore, a good scheduling algorithm should be able to efficiently assign a priority to each subtask depending on the resources needed to minimize makespan. In this paper, a task scheduling scheme on heterogeneous computing systems using a *multiple priority queues genetic algorithm* (MPQGA) is proposed. The basic idea of our approach is to exploit the advantages of both evolutionary-based and heuristic-based algorithms while avoiding their drawbacks. The proposedalgorithm incorporates a *genetic algorithm* (GA) approach to assign a priority to each subtask while using a heuristic-based *earliest finish time* (EFT) approach to search for a solution for the task-to-processor mapping. The MPQGA method also designs crossover, mutation, and fitness function suitable for the scenario of *directed acyclic graph* (DAG) scheduling. The experimental results for large-sized problems from a large set of randomly generated graphs as well as graphs of real-world problems with various characteristics show that the proposed MPQGA algorithm outperforms two non-evolutionary heuristics and a random search method in terms of schedule quality.

1. Introduction

Task scheduling on heterogeneous computing systems interconnected by high-speed networks has been extensively studied. Such systems are promising for fast processing of computationally intensive applications with diverse computation needs. In general, an originally large application can be decomposed into a set of smaller subtasks prior to parallel processing. These smaller subtasks almost always have dependencies representing the precedence constraints, in which the results of other subtasks are required before a particular subtask can be executed. By decomposing a computation into smaller subtasks and executing the subtasks on multiple processors, the total execution time of the computation, namely, the makespan, can potentially be reduced. Hence, the goal of a task scheduling algorithm is typically to schedule all the subtasks on a given number of available processors in order to minimize makespan without violating precedence constraints.

* Corresponding author.
E-mail addresses: xxl1205@163.com (Y. Xu), lkl@hnu.edu.cn (K. Li), jthu@okstate.edu (J. Hu), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.ins.2014.02.122
0020-0255/

It is a challenge on heterogeneous computing systems to develop task scheduling algorithms that assign the subtasks of an application to processors. Therefore, numerous algorithms have been proposed to minimize makespan for parallelizing the subtasks with precedence relationships. The precedence relationships are represented as a *directed acyclic graph* (DAG) consisting of vertices that represent computations and directed edges that represent the dependencies between those vertices. DAGs have been shown to be expressive for a large number of and a variety of applications.

Traditional task scheduling research has focused on heuristic-based algorithms, an important class of which is the so-called list scheduling algorithms, such as *heterogeneous earliest finish time* (HEFT) [1] and *critical path on a processor* (CPOP) [1]. The basic idea of list scheduling consists of maintaining an ordered list of subtasks by assigning priority to each subtask according to greedy heuristics. The subtasks are selected in order of priority and the ready subtask with the highest priority is removed from the list to be assigned to a processor which allows the earliest start time. The performance of these algorithms is heavily dependent on the effectiveness of the heuristics. They are not likely to produce consistent results for a wide range of problems, especially when the complexity of the task scheduling problem increases.

Contrary to the heuristic-based algorithms, a guided-random-search-based algorithm incorporates a combinatoric process in the search for solutions, which is less efficient and generates much higher computational cost than the heuristic-based algorithms.

For this reason, balance between makespan and speed of convergence is required. In this paper, we develop a hybrid approach by integrating GA with heuristic algorithms. We find that hybrid approach can achieve similar performance in terms of makespan for DAG scheduling while reducing the scheduling overhead, when compared with the overhead of guided-random-search-based algorithms. It alsocan achieve better performance than heuristic algorithms. Hence, we have published the paper entitled "A multiple priority queueing genetic algorithm for task scheduling on heterogeneous computing systems" in the proceeding of the 14th IEEE International Conference on High Performance Computing and Communications (HPCC-2012) [2]. In the paper, we addressed the task scheduling problem and proposed a task scheduling scheme using *a multiple priority queues genetic algorithm* (MPQGA) on heterogeneous computing systems. In later study, we found more factors which affect the efficiency of the MPQGA algorithm, such as the initial population's size and the method by which its individuals are chosen, and the particular crossover and mutation operators.

This paper significantly extends the conference paper, and definitely contains more than 30% new contents, including new algorithms, discussions, and solid experimental results that were not shown in the conference version. Moreover, in this paper, a new and different way to encode scheduling solutions of the initial population is designed. We also need to develop different operations to be performed on the encoded solutions and generate increasingly better solutions. This paper tackles these challenges. The four major contributions of this study are listed below.

- We develop the MPQGA algorithm, which can be adapted for a wide range of DAG applications. The proposed MPQGA algorithm generates various priority queues using a heuristic-based crossover operator and a heuristic-based mutation operator for dependent task applications. The MPQGA algorithm effectively avoids the drawbacks of existing heuristic-based scheduling algorithms that are only effective for specific types of applications.
- We propose a heuristic-based task-to-processor mapping technique, i.e., the *heterogeneous earliest finish time* (HEFT) approach, to search for a solution in order to minimize makespan without violating precedence constraints. The approach effectively avoids the inefficient task-to-processor mapping and accelerates convergence speed of the MPQGA algorithm.
- We adopt a new approach to generating the initial population using three evaluation criterion: a good "seeding", a good uniform coverage, and genetic diversity. A high-quality solution, obtained from a heuristic technique, can help MPQGA find better solutions faster than it can from a random start. With a good uniform coverage, the individuals are well spread out to cover the whole feasible solution space. Genetic diversity of the initial population can help the GA be able to reach part of the feasible solution space as large as possible.
- We demonstrate through experimental results over a large set of randomly generated graphs as well as graphs of real-world problems with various characteristics that our proposed MPQGA algorithm outperforms several related heuristic-based list scheduling algorithms and guided-random-search-based algorithms in terms of schedule quality.

The remainder of this paper is organized as follows. In Section 2, related work on different scheduling algorithms on heterogeneous systems is reviewed. In Section 3, the system model, the application model, and the scheduling scheme are described. In Section 4, MPQGA is outlined for task scheduling to minimize the makespan on heterogeneous computing systems. In Section 5, we analyze the time and space complexities. In Section 6, results and analyses of software simulations which have been conducted to validate the algorithm are given. Finally, in Section 7, conclusions and suggestions on future work are provided.

2. Related work

The task scheduling problem can be formulated as the search for an optimal assignment of a set of subtasks onto a set of processors, such that the completion of the last subtask being executed is minimized. The task scheduling problem has been proven to be NP-hard [3]. For this reason, scheduling is usually handled by heuristic methods which provide reasonable solutions for restricted instances of the problem. A heuristic-based algorithm normally finds a near-optimal solution in

polynomial time. It searches a path in the solution space at the expense of ignoring some possible paths [4,5]. The heuristic-based scheduling algorithms can be classified as list scheduling [1,5], cluster scheduling [6–8], and task duplication-based scheduling [9–12].

Most existing scheduling heuristics belong to the list scheduling class. The search in list scheduling algorithms is divided into two phases. In the first phase, each subtask is assigned a priority and then added to a list of waiting subtasks in order of decreasing priority according to certain criteria. In the second phase, as processors become available, the subtask with the highest priority is selected and assigned to the processor. Therefore, list scheduling algorithms are more practical and efficient, since the search is narrowed down to a very small portion of the solution space, and a better makespan than others can be obtained. A weakness of the heuristic-based list scheduling algorithms is that the performance of these algorithms is heavily dependent on the effectiveness of the heuristics. Therefore, they are not likely to produce consistent results for task scheduling.

Another group of deterministic algorithms are the clustering algorithms [6–8]. These algorithms assume that there is enough number of processors available to the subtask execution. The clustering algorithms use as many processors as possible in order to reduce the makespan of the schedule.

The duplication scheduling [9–12] heuristic attempts to reduce communication delays by executing a key subtask on more than one processor.

Contrary to the heuristic-based algorithms, a guided-random-search-based algorithm incorporates a combinatoric process in the search for solutions. A guided-random-search-based algorithm typically requires sufficient sampling of candidate solutions in the search space and has shown robust performance on a variety of scheduling problems. It is well known that *particle swarm optimization* (PSO) [13–17], *ant colony optimization* (ACO) [18,19], *artificial bee colony algorithm* (ABC) [20], *simulated annealing* (SA) [21,22], *cuckoo search algorithm* (CS) [23], *tabu search* (TS) [24,25], *evolution algorithm* [26,27], and *genetic algorithms* (GA) [24,28–42] have been successfully applied to the scheduling. GAs have been widely used to evolve solutions for many task scheduling problems. These GA approaches vary in their encoding schemes, the implementation of genetic operators, as well as the methods for solution evaluation. However, due to the large solution space that a GA is required to cover, the search generally incurs considerably high computational cost, thesame as other guided-random-search-based algorithms which are less efficient and generate much higher computational cost than the heuristic-based algorithms.

Table 1
Definitions of notations.

Notation	Definition
T	A set of tasks in an application
E	A set of edges for precedence constraints among the tasks
P	A set of heterogeneous processors
n	Number of tasks
e	Number of edges
m	Number of processors in a heterogeneous computing system
T_i	The ith task in the application
$edge(T_i, T_j)$	The edge from subtask T_i to subtask T_j
P_k	The kth processor in the system
T_{entry}	The entry subtask with no predecessor
T_{exit}	The exit subtask with no successor
$Succ(T_i)$	The set of immediate successors of subtask T_i
$Pred(T_i)$	The set of immediate predecessors of subtask T_i
$W_d(T_i)$	The amount of computation of subtask T_i
$S(T_i, P_k)$	The execution speed of subtask T_i on processor P_k
$W(T_i, P_k)$	The computational cost of subtask T_i on processor P_k
$\overline{W(T_i)}$	The average computational cost of subtask T_i
$C_d(T_i, T_j)$	The amount of communication between subtasks T_i and T_j
$C_s(P_k)$	The communication startup cost of processor P_k
B	A two-dimensional matrix of communication bandwidths
$B(P_k, P_l)$	The communication bandwidth between P_k to P_l
$C(T_i, T_j)$	The communication cost from subtask T_i to subtask T_j
$\overline{C(T_i, T_j)}$	The average communication cost of the $edge(T_i, T_j)$
$EST(T_i, P_k)$	The earliest start time of subtask T_i on processor P_k
$EFT(T_i, P_k)$	The earliest finish time of subtask T_i on processor P_k
$Avail(P_k)$	The earliest time when processor P_k is ready for execution
$AST(T_i, P_k)$	The actual start time of subtask T_i on processor P_k
$AFT(T_i, P_k)$	The actual finish time of subtask T_i on the fittest processor P_k
$Rank_b(T_i)$	The upward rank of subtask T_i
$Rank_t(T_i)$	The downward rank of subtask T_i
CCR	The communication to computation ratio
SLR	Scheduling length ratio of a task graph
α	The shape parameter of a task graph
β	The range percentage of computation costs on processors

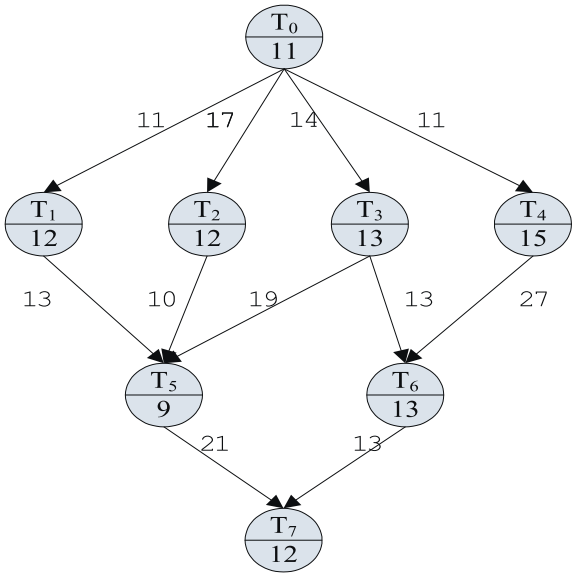
In this paper, we take the trade-off between makespan and convergence speed into consideration and develop a hybrid approach by integrating GA with heuristic algorithms.

3. Models

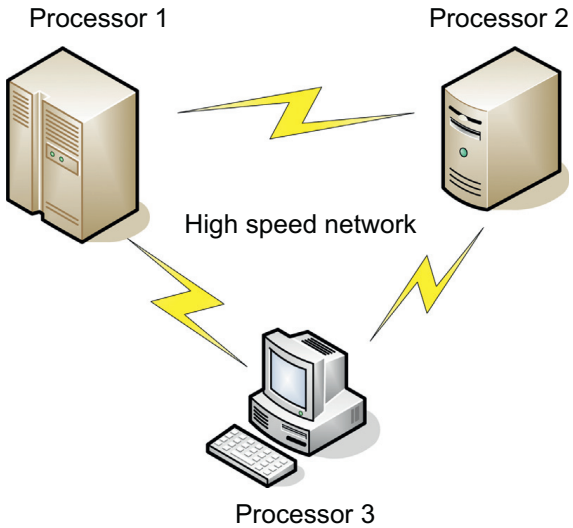
In this section, the system, application, and heterogeneity models used in the study are described. Table 1 provides a list of notations and their definitions used in the paper.

3.1. System model

In this paper, the target system consists of a set P of m heterogeneous processors that are fully interconnected with a high-speed network. Each subtask in a DAG application can only be executed on one processor. The communication time between two dependent subtasks should be taken into account if they are assigned to different processors.



(a): A simple DAG containing 8 subtasks.



(b): A fully connected parallel system with 3 heterogeneous processors.

Fig. 1. A simple DAG containing 8 subtasks and a fully connected parallel system with 3 heterogeneous processors.

We also assume a static computing model in which the dependency relations and the execution order of subtasks are known *a priori* and do not change over the course of the scheduling and subtask execution. In addition, all processors are fully available to the computation on the time slots they are assigned to.

3.2. Application model

In this work, an application is represented by a DAG, with the vertices representing subtasks and edges between vertices representing execution precedence between subtasks. For a pair of dependent subtasks T_i and T_j, if the execution of T_j depends on the output from the execution of T_i, then T_i is the predecessor of T_j, and T_j is the successor of T_i. We use $Pred(T_i)$ and $Succ(T_i)$ to denote the set of predecessor subtasks and the set of successor subtasks of subtask T_i, respectively. There is an entry subtask and an exit subtask in a DAG. The entry subtask T_{entry} is the starting subtask of the application without any predecessor, while the exit subtask T_{exit} is the final subtask with no successor [43]. A weight is associated with each vertex and edge. The vertex weight, denoted as $W_d(T_i)$, represents the amount of computation to be performed by the subtask T_i, while the edge weight, denoted as $C_d(T_i, T_j)$, represents the amount of communication between subtask T_i and subtask T_j. If a DAG has multiple start nodes, a dummy start node with a zero node weight is added. Zero weight communication edges are then added from the dummy start node to the multiple start nodes. Likewise, if a DAG has multiple exit nodes, a dummy exit node is added. The DAG topology of an example application and an example system are shown in Fig. 1(a) and (b), respectively.

3.3. Heterogeneity model

The heterogeneity model of a computing system can be divided into two categories, i.e., *processor-based heterogeneity model* (PHM) and *task-based heterogeneity model* (THM). In a PHM computing system, a processor executes the tasks at the same speed, regardless of the type of the tasks. On the contrary, in a THM computing system, how fast a processor executes a task depends on how well the heterogeneous processor architecture matches the task requirements and features. The work in this paper assumes THM computing systems, where the main characteristics are given in Table 2.

For example, we can set the value of h to be 0.5, so that the level of heterogeneity can be calculated as 3. Note that when the level of heterogeneity is 1 ($h = 0$), the computing system is homogeneous. As we change the value of h, i.e., the level of heterogeneity, the average computing speed remains unchanged. When the level of heterogeneity increases, some heterogeneous computing nodes become increasingly powerful and therefore the subtasks can run faster in those heterogeneous computing nodes.

The execution speeds of the processors in a heterogeneous computing system are represented by a two-dimensional matrix S, in which an element $S(T_i, P_k)$ represents the speed at which processor P_k executes subtask T_i. The computation cost of subtask T_i running on processor P_k, denoted as $W(T_i, P_k)$, can be calculated by Eq. (1):

$$W(T_i, P_k) = \frac{W_d(T_i)}{S(T_i, P_k)}. \tag{1}$$

The average computation cost of subtask T_i, denoted as $\overline{W(T_i)}$, can be calculated by Eq. (2):

Table 2
THM Heterogeneous computing system.

Characterization	Value
The amount of computing power available at each node	0.1–2.0
The maximum number of processors	32
The level (degree) of heterogeneity of the system	$\frac{1+h}{1-h}$, $h \in [0, 1)$

Table 3
Processor heterogeneity and computation costs.

T_i	Speed			Cost			Average cost
	P_0	P_1	P_2	P_0	P_1	P_2	$\overline{W(T_i)}$
0	1.00	0.85	1.22	11	13	9	11.00
1	1.20	0.80	1.09	10	15	11	12.00
2	1.33	1.00	0.86	9	12	14	11.67
3	1.18	0.81	1.30	11	16	10	12.33
4	1.00	1.37	0.79	15	11	19	15.00
5	0.75	1.00	1.79	12	9	5	8.67
6	1.30	0.93	1.00	10	14	13	12.33
7	1.09	0.80	1.20	11	15	10	12.00

$$\overline{W(T_i)} = \frac{1}{m}\sum_{k=1}^{m} W(T_i, P_k). \tag{2}$$

Assume that a DAG has the topology as shown in Fig. 1(a) and there are 3 heterogeneous processors in the computing system as shown in Fig. 1(b). An example of the processor heterogeneity and the computation costs of the subtasks are shown in Table 3. Note that there are two numbers in each vertex in Fig. 1(a). The number at the top is the task name and the one at the bottom is the average computation cost as calculated in Table 3.

The communication bandwidths between any two heterogeneous processors are represented by a two-dimensional matrix B of size $m \times m$, where $B(P_k, P_l)$ represents the communication cost of sending a unit of data from P_k to P_j. The communication startup costs of the processors are represented by an array C_s, in which the element $C_s(P_k)$ is the startup cost of processor P_k. The communication cost $C(T_i, T_j)$ of $edge(T_i, T_j)$, which is the time spent in transferring data from subtask T_i (scheduled on P_k) to subtask T_j (scheduled on P_l), can be calculated by Eq. (3):

$$C(T_i, T_j) = C_s(P_k) + \frac{C_d(T_i, T_j)}{B(P_k, P_l)}. \tag{3}$$

$\overline{C(T_i, T_j)}$ is the average communication cost of the $edge(T_i, T_j)$, which is defined in Eq. (4):

$$\overline{C(T_i, T_j)} = \overline{C_s} + \frac{C_d(T_i, T_j)}{\overline{B}}, \tag{4}$$

where $\overline{C_s} = \frac{1}{m}\sum_{k=1}^{m} C_s(P_k)$ is the average communication startup cost over all processors, and $\overline{B} = \frac{1}{m^2}\sum_{k=1}^{m}\sum_{l=1}^{m} B(P_k, P_l)$ is the average communication cost per transferred unit over all processors.

It is assumed that the inter-processor communications are performed at the same speed (i.e., with the same bandwidth) on all links, and we assume that $B(P_k, P_l) = 1$ and $C_s(P_k) = 0$ to simplify our task scheduling model. In this paper, there is communication cost only when two subtasks are assigned to different heterogeneous processors. In other words, the communication cost can be ignored when the subtasks are assigned to the same processor. For example, if T_i and T_j are scheduled on the same processor, the communication cost is regarded as 0.

The *communication to computation ratio* (CCR) can be used to indicate whether a task graph is communication-intensive or computation-intensive. For a given task graph, it is computed by the average communication cost divided by the average computation cost on a target computing system. The computation can be formulated as in Eq. (5):

$$CCR = \frac{\frac{1}{e}\sum_{edge(T_i,T_j)\in E}\overline{C(T_i, T_j)}}{\frac{1}{n}\sum_{T_i \in T}\overline{W(T_i)}}. \tag{5}$$

3.4. An example DAG application

Figs. 2–6 show five example solutions of scheduling the DAG in Fig. 1(a) on the computing system in Fig. 1(b), using the HEFT-B algorithm, the HEFT-T algorithm, the CPOP algorithm, the MPQGA task scheduling algorithm (a heuristic-based tasks-to-processor mapping approach), and the BGA task scheduling algorithm [44] (a random-based tasks-to-processor mapping approach), respectively. The task scheduling priority queue is generated by upward rank for HEFT-B, and downward rank for HEFT-T [1], and a combination of upward-downward rank.

In Fig. 7, we can observe that the convergence speed of MPQGA algorithm is faster than BGA algorithm in the makespan of the best individual.

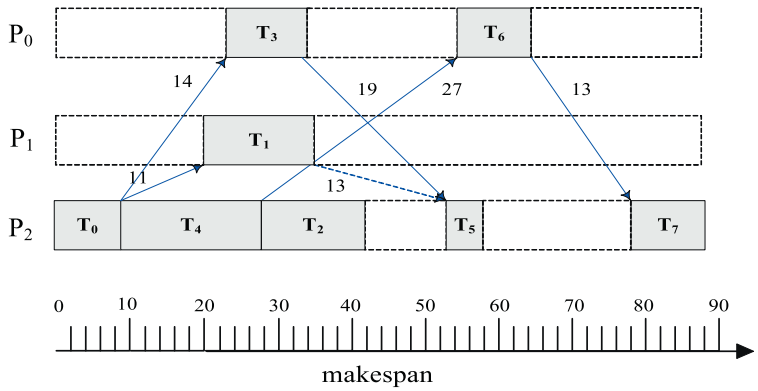


Fig. 2. A schedule for the simple DAG in Fig. 1(a) on 3 processors in Fig. 1(b) using the HEFT-B task scheduling algorithm. The makespan of the schedule is 88.

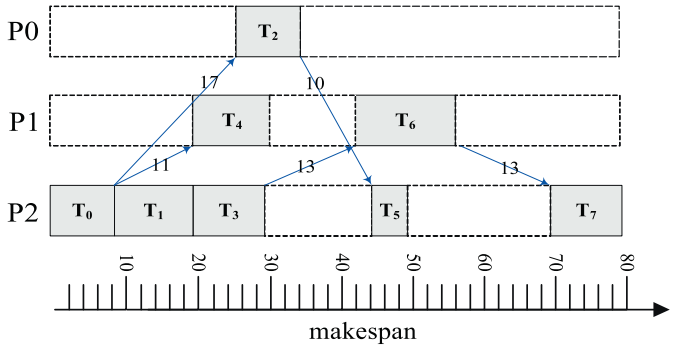


Fig. 3. A schedule for the simple DAG in Fig. 1(a) on 3 processors in Fig. 1(b) using the HEFT-T task scheduling algorithm. The makespan of the schedule is 80.

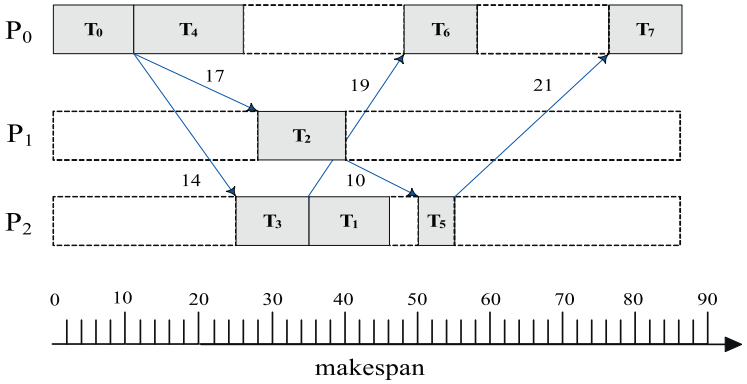


Fig. 4. A schedule for the simple DAG in Fig. 1(a) on 3 processors in Fig. 1(b) using the CPOP task scheduling algorithm. The makespan of the schedule is 87.

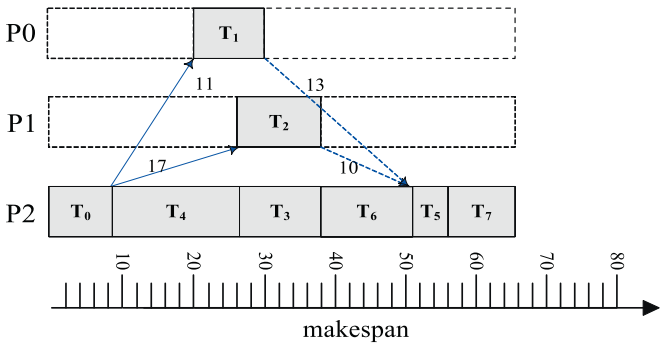


Fig. 5. A schedule for the simple DAG in Fig. 1(a) on 3 processors in Fig. 1(b) using the MPQGA task scheduling algorithm. The makespan of the schedule is 66.

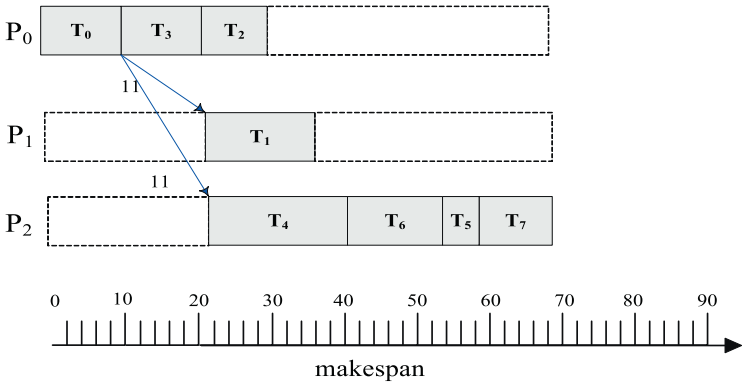


Fig. 6. A schedule for the simple DAG in Fig. 1(a) on 3 processors in Fig. 1(b) using the BGA task scheduling algorithm. The makespan of the schedule is 69.

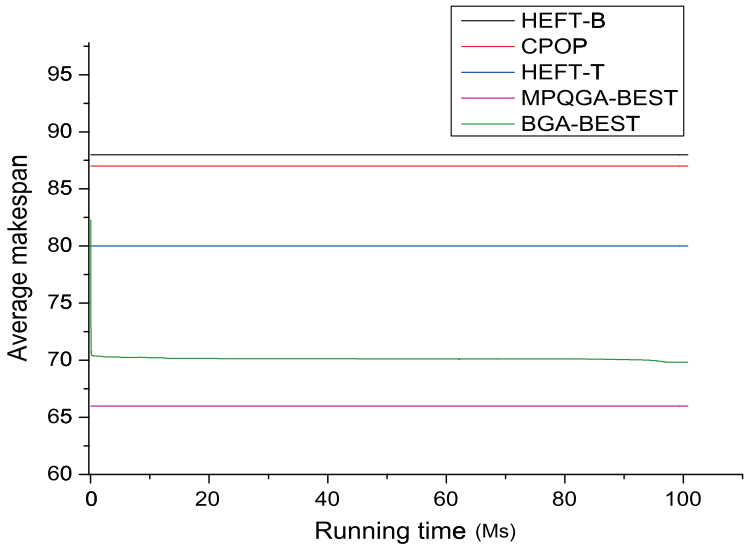


Fig. 7. The convergence trace of the makespan of the best individual for the simple DAG task graph in Fig. 1(a) (the number of processors = 3, 50 independent runs).

4. Algorithm design

In this section, we will first present the background of genetic algorithm. Then, the details of proposed algorithms will be presented. A *genetic algorithm*, introduced by Holland (1975) [60], is an iterative stochastic algorithm in which natural evolution is used to model a search method. GAs may be used to solve optimization problems by imitating the genetic process of biological organisms [4,45–47]. As the name suggests, GAs emulate the evolutionary process in nature to solve optimization problems.

A simple genetic algorithm includes three basic genetic operations: selection, crossover, and mutation. In selection, some solutions from the population are selected as parents; in crossover, the parents are crossbred to produce offspring; and in mutation, the offspring may be altered according to mutation rules. In genetic algorithms, solutions x are called *individuals* and iterations of an algorithm are called *generations*. Many genetic algorithms also employ elitism, which means that a number of the best individuals are copied to the next population.

Our proposed algorithm has the following parameters. *Crossover rate* and *mutation rate* are probabilities of which the parents are crossbred and offspring are mutated, respectively. *Steps* and *tolerance* are parameters for the stopping criteria. The algorithm is terminated if there is no change (within the tolerance) in the best objective function value during the last generation.

Unlike other traditional search techniques, GAs use multiple search nodes simultaneously. Each of the search nodes corresponds to one of the current solutions and is represented by a sequence of symbols. Such a sequence is called a *chromosome*, while the symbols composing the sequence are called *genes*. Each chromosome has an associated value called a *fitness value*, which is evaluated using the objective function (fitness function) value $f(x)$. In a GA, only good chromosomes that have high fitness values are likely to survive and generate offsprings transmitting their biological heredity to new generations. By evolving the chromosomes continuously, the solutions corresponding to the search nodes are improved gradually. A set of chromosomes at a given stage of a GA is called a *Pop*. The number of chromosomes (individuals) in a population is called the *PopSize*. *ElitismSize* is the number of fittest individuals that are copied directly to the next generation.

In the following, we will explain the details of our design and implementation of the genetic algorithm for the task scheduling problem. In Section 4.1, we describe the outline of MPQGA for task scheduling on heterogeneous computing systems. In Section 4.2, we discuss the chromosome representation, initial population, and an algorithm to generate it. In Section 4.3, we develop the heuristic transformational approach and two algorithms for the crossover operator and the mutation operator using the heuristic transformational approach. In Section 4.4, we devise the heuristic task-to-processor mapping approach for DAG task graphs on heterogeneous computing systems, an equation for fitness value computing, and an algorithm to realize the task-to-processor mapping and fitness value computing. In Section 4.5, we present the selection operator and the roulette-wheel selection algorithm, which is an important part of a genetic algorithm, since it affects its convergence significantly. The basic strategy follows the following rule, i.e., the better fitted an individual, the larger the probability of its survival and mating.

4.1. The outline of MPQGA

In this paper, the MPQGA algorithm for task scheduling on heterogeneous computing systems is developed to exploit the advantages of both evolutionary-based and heuristic-based algorithms while avoiding their disadvantages. The MPQGA task

scheduling algorithm performs task scheduling by incorporating a GA-based approach to assigning the priority of task execution while using a heuristic-based approach (namely HEFT) to completing task-to-processor mapping, and we use an integer-string-coded genetic algorithm that employs roulette-wheel selection, heuristic crossover, heuristic mutation, and elitism.

The outline of MPQGA for DAG task scheduling on heterogeneous computing systems is given in Algorithm 1.

Algorithm 1. MPQGA

Input:
Parameters for the genetic algorithm;
Parameters for task scheduling.
Output:
A task schedule.
1: Call Algorithm 2 to create an initial population;
2: **repeat**
3: Call Algorithm 5 to perform task-to-processor mapping and evaluate the fitness (makespan);
4: Copy the elitism directly to the next new population;
5: **repeat**
6: Call Algorithm 6 to select candidates as survivals according to their fitness values using roulette-wheel selection;
7: Call Algorithm 3 to perform the heuristic crossover operator;
8: Call Algorithm 4 to perform the heuristic mutation operator;
9: **until** the new population is complete;
10: Replace the old population by the new population;
11: **until** the termination condition is satisfied;
12: **return** a near-optimal schedule.

In the algorithm, the MPQGA first initializes the population (Step 1) by calling Algorithm 2. In Algorithm 2, which is presented in Section 4.2.3, the chromosomes are generated by three heuristic rank policies in order to generate multiple priority queues for a DAG application respectively and the priorities of the rest of the chromosomes are generated by random perturbation in the priority queues for these chromosomes. Then, the process enters an outer loop (Steps 2–11). In Step 3, the task-to-processor mapping is performed by calling Algorithm 5. In Algorithm 5, which is presented in Section 4.4.1, the heuristic-based *Heterogeneous Earliest Finish Time* (HEFT) is adopted to search for a solution in order to minimize the makespan without violating precedence constraints, and the fitness value of each chromosome is evaluated. Then, the process enters an inner loop (Steps 5–9). In Step 6, we call Algorithm 6, which is presented in Section 4.5. Based on the fitness values, a selection mechanism chooses candidates for survivals for the next generation using roulette-wheel selection. The selection policy is that weaker chromosomes perish and stronger chromosomes survive to produce better offspring. In Step 7, we call Algorithm 3 to perform the crossover operator. Algorithm 3 is presented in Section 4.3.2. The crossover operator takes a pair of chromosomes selected at the previous step. After that, it chooses a crossover point at random for each chromosome, and then exchanges portions of their genes based on the crossover point. Crossover is not always performed and is associated with a predetermined probability (crossover probability). In Step 8, we call Algorithm 4 to perform the mutation operator. Algorithm 4 is presented in Section 4.3.3. The mutation operator changes each gene with a certain probability (mutation probability). This operator can produce a chromosome that cannot be generated only by the crossover operator. Thus it helps the search algorithm escape from local optimal solutions. As a result of reproduction and genetic manipulations, a new generation is created (Step 9). The inner loop is completed when the size of a new generation reaches a pre-defined size. The outer loop is completed when certain termination condition is satisfied.

4.2. Chromosome representation and initial population

Genetic algorithms are Markov chains [48,49]. Such chains are expected to converge to an equilibrium distribution independent of the initial state. However, in many applications the state space of the chain (possible point configurations in the feasible region) is extremely large and convergence can be very slow. This poses the question whether the number of the generations used is sufficient in order to achieve the equilibrium. The initial configuration is one factor in the speed of convergence. Hence, the chromosome representation and the initial population are the main elements of genetic algorithms, and the genetic operations like crossover and mutation are just instruments for manipulating the population so that it evolves towards the final population that isclose to optimal solution.

In this section, firstly, in Section 4.2.1 we address the generation policy of priority queues for DAG applications. In many task scheduling algorithms, the subtask with the highest priority is selected for processor allocation, and a processor which can ensure the *earliest finish time* is selected to run the subtask. Secondly, in Section 4.2.2, the encoding mechanism of our

task scheduling algorithm is presented for representing search nodes as chromosomes. It is desirable that any chromosome can determine a schedule uniquely. Thirdly, in Section 4.2.3, Algorithm 2 is developed which takes advantage of three heuristic rank policies (upward rank, downward rank, and a combination of upward-downward rank) in order to initialize the initial population.

4.2.1. Policy of priority queues

A good "seeding", i.e., the possibility of seeding the initial population with known good solutions, is very important. [50,51] report that a high-quality solution obtained from another heuristic technique can help a GA find better solutions more quickly than it can from a random start.

In this study, the task scheduling problem is the process of mapping a set T of n subtasks in a DAG application to a set P of m processors, aiming at minimizing the makespan. The subtask with the highest priority is selected for processor allocation, and a processor which can ensure the *earliest finish time* is selected to run the subtask. In many task scheduling algorithms, a subtask which has a higher rank is more important, and is therefore preferred for processor allocation to other subtasks. In our paper, we adopted a heuristic approach to obtained the good "seeding". Intuitively, the upward rank of a subtask reflects the average remaining cost to finish all subtasks after that subtask starts up. The upward rank of subtask T_i, denoted as $Rank_b(T_i)$, can be computed by Eq. (6):

$$Rank_b(T_i) = \overline{W(T_i)} + \max_{T_j \in Succ(T_i)} (\overline{C(T_i, T_j)} + Rank_b(T_j)), \tag{6}$$

where $Succ(T_i)$ is the set of immediate successors of subtask T_i. The upward rank is computed recursively by traversing the task graph upward, starting from the exit subtask T_{exit}.

Similarly, the downward rank of subtask T_i, denoted as $Rank_t(T_i)$, can be calculated by Eq. (7):

$$Rank_t(T_i) = \max_{T_j \in Pred(T_i)} (\overline{W(T_i)} + \overline{C(T_j, T_i)} + Rank_t(T_j)), \tag{7}$$

where $Pred(T_i)$ is the set of immediate predecessors of subtask T_i. The downward ranks are computed recursively by traversing the task graph downward starting from the entry task of the graph. For the entry task T_{entry}, the downward rank value is equal to zero. Basically, $Rank_t(T_i)$ is the longest distance from the entry subtask to subtask T_i, excluding the computation cost of the subtask itself.

Basically, for a DAG application, the task schedule represented by a chromosome can be constructed by allocating tasks as early as possible considering the imposed constraints, that is, precedence constraints, communication delays, and execution order specified by the chromosome. In this paper, we uses upward rank values, downward rank values, and a combination of upward-downward values to set priorities of subtasks, respectively. The set of priority queues will be used as the seeds of initial population to generate more individual, as shown in Algorithm 2. Note that both computation and communication costs are the costs averaged over all vertices and links.

The level of a task is defined as:

$$Level(T_i) = \begin{cases} 0, & \text{if } T_i = T_{entry}; \\ \max\limits_{T_j \in Pred(T_i)} (Level(T_j)) + 1, & \text{otherwise.} \end{cases} \tag{8}$$

Table 4 shows the upward rank, downward rank, and a combination of upward-downward rank values of each subtask in the DAG in Fig. 1(a).

4.2.2. Chromosome representation

One of the most fundamental and important tasks in the design of a GA is devising an encoding mechanism for representing search nodes as chromosomes. Since we solve a task scheduling problem, each search node corresponds to a schedule. Therefore, it is desirable that any chromosome can determine a schedule uniquely. For this purpose, in this paper, MPQGA uses a chromosome structure with n genes, which represents a solution of the task scheduling problem. The chromosome structure contains a permutation of integers 0, 1, 2, ..., $n-1$, representing a priority queue of the n tasks, as shown in Fig. 8.

A $Queue_{Priority} = (T_0, T_1, T_2, \ldots, T_i, \ldots, T_{n-1})$ represents a priority queue of the set of subtasks. Each gene of the chromosome as shown in Fig. 8 corresponds to one of these subtasks in a DAG application, and the order of genes in the priority queue represents their execution order. Moreover, the priority order of the subtasks in the chromosome should be a valid topological order, where the entry node should be placed at the beginning of the molecule, while the exit node should be placed at the end. To assure that a queue of integers should be feasible, all the subtasks in a DAG should be scheduled and the schedule should satisfy the precedence relations.

4.2.3. Initial population

The initial population consists of *PopSize* individuals, where *PopSize* is the population size to be kept constant through the generations. The size of the initial population has been investigated from several theoretical points of view, although the underlying idea is always of a trade-off between efficiency and effectiveness. Intuitively, it would seem that there should be some some individuals with the "good" fitness values for a given chromosome length, on the grounds that a too small

population would not allow sufficient room for exploring the search space effectively, while a too large population would so impair the efficiency of the method that no solution could be expected in a reasonable amount of time. In our study, we adopted three approaches to generating a good "seeding", good uniform coverage, and genetic diversity for the initial population.

In this paper, in order to achieve a good "seeding" for the task scheduling problem, we take advantage of three heuristic rank policies (upward rank, downward rank, and a combination of level and upward-downward rank), which are the mostly used by traditional list scheduling approaches to estimating the priority of each subtask. Tabel 5 shows the task priority queues according to Table 4.

A desirable property for an initial population is a *good uniform coverage*. By a good uniform coverage, the individuals are well spread out to cover the whole feasible solution space. The individuals have a good uniform coverage if they do not form clusters and leave relatively large areas of the feasible solution space unexplored. A good uniform coverage is desired, because information is obtained throughout the whole feasible solution space. However, there is also the possibility of inducing premature convergence [52,53].

In order to achieve a good uniform coverage, initially, the chromosomes are generated by three heuristic rank policies in order to generate multiple priority queues for a DAG application, respectively. Then, the priority queues are generated by selecting a gene in the priority queues for these chromosomes from left to right, where a gene is selected and inserted into a right position without violating the precedence constraints in order to make the largest hamming distance of two chromosomes as shown in Fig. 10. In other words, this approach uses the criterion of maximizing the hamming distance between the solution (new chromosome) under consideration and the solution (seeding chromosome) already generated before. In information theory, the hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. Essentially, a gene T_i is placed at position l, which is chosen between $j+1$ and $k-1$ whose distance is longer from the gene T_i, where T_j is the last predecessor $Pred(i)$, and T_k is the first successor $Succ(i)$. Depending on the position of l, genes in a sub-interval are shifted to generate an empty slot for T_i. The approach can effectively improve uniform coverage of the population. Fig. 9 illuminates the uniform coverage of the population of task graph in Fig. 1(a).

Genetic diversity of the initial population can help the GA be able to reach part of the feasible solution space as large as possible. Genetically more diversified initial populations are preferable. Finally, the rest priority queues are generated by selecting randomly a chromosome in the priority queues for these chromosomes, where a gene is

Table 4
Task priorities.

T_i	$Rank_b$	$Rank_t$	$Rank_{b+t}$	$Level$
0	101.33	0.00	101.33	0
1	66.67	22.00	88.67	1
2	63.33	28.00	91.33	1
3	73.00	25.00	98.00	1
4	79.33	22.00	101.33	1
5	41.67	56.33	98.00	2
6	37.33	64.00	101.33	2
7	12.00	89.33	101.33	3

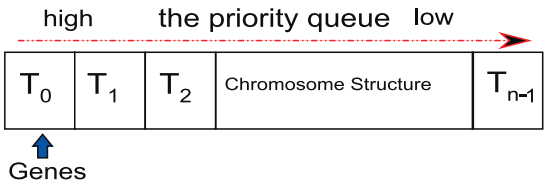


Fig. 8. Illustration of chromosome structure that encodes a solution.

Table 5
Task priority queues.

	Position of genes							
	0	1	2	3	4	5	6	7
Blevel	0	4	3	1	2	5	6	7
Tlevel	0	1	4	3	2	5	6	7
Llevel	0	4	3	2	1	6	5	7

randomly selected in the selected chromosome and inserted into a right position without violating the precedence constraints in order to make the largest hamming distance of two chromosomes. The new generated chromosome is added into the population according to tabu list in order to effectively improve genetic diversity of the population.

A detailed description of the initial population generation process is given in Algorithm 2.

Algorithm 2. IniPopulation

Input:
Seed chromosomes generated by three heuristic rank policies.
Output:
The initial population.

PopulationN = 3;
for each of the initial three individuals **do**
  for $i = 1$ to $n - 1$ **do**
    Copy a temporary individual from the initial three individuals;
    Find the last predecessor $T_j \in Pred(i)$ in the temporary individual;
    Find the first successor $T_k \in Succ(i)$ in the temporary individual;
    temp $\leftarrow T_i$;
    if $(k - i) > (i - j)$ **then**
      $l = k - 1$;
      for $q = i$ to $k - 1$ **do**
        $T_q \leftarrow T_{q+1}$;
      end for
    else
      $l = j + 1$;
      for $q = i$ to $j + 1$ **do**
        $T_q \leftarrow T_{q-1}$;
      end for
    end if
    $T_l \leftarrow$ temp;
    Add the temporary individual to the Population;
    PopulationN = *PopulationN* + 1;
  end for
end for;
while *PopulationN* $<$ *PopSize* **do**
  Select a gene $i \in (1, n - 1)$ randomly;
  Generate a new individual using the maximum code distance approach and tabu list;
  Add the temporary individual to the Population;
  PopulationN = *PopulationN* + 1;
end while;
return Population.

4.3. Reproduction

In this section, firstly, we discuss the heuristic transformational approach without violating precedence constraints of DAG applications in Section 4.3.1. Secondly, in Section 4.3.2, Algorithm 3 is presented for the crossover operator of the task scheduling algorithm. The crossover operator combines two valid parents, whose subtasks are ordered topologically, to generate two offsprings which will also be valid. Thirdly, in Section 4.3.3, Algorithm 4 is presented for the mutation operator of the task scheduling algorithm. The mutation operator changes each gene with certain mutation probability, which focuses on maintaining the diversity of the population so as to enlarge the search space, thus helps the search algorithm escape from local optimal solutions.

4.3.1. Heuristic transformational approach

In this paper, an application is represented by a DAG, with the graph nodes representing subtasks and edges between nodes representing execution precedence between subtasks. An important factor in selecting the chromosome representation for the priority queues of task scheduling is that all of them in a search space are represented and that the representation is unique. Note that not every permutation of n tasks corresponds to a legal schedule because of the precedence relations.

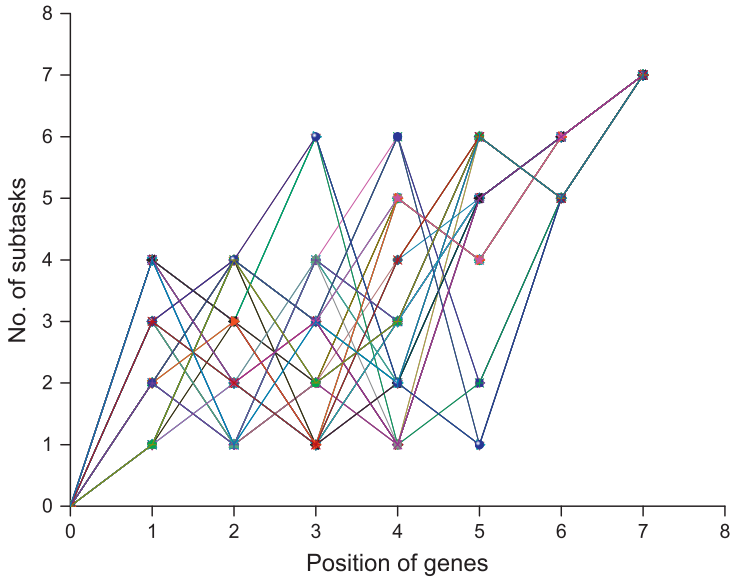


Fig. 9. An example of uniform coverage of the population for the task graph in Fig. 1(a).

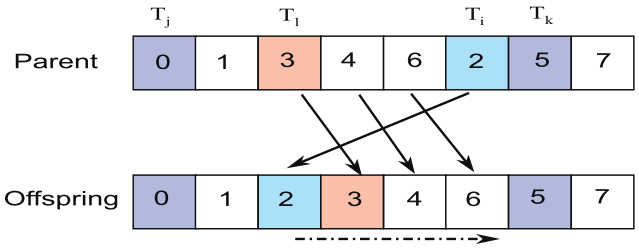


Fig. 10. Random perturbation.

For the task scheduling problem, an intermediate encoded representation of the schedules and a decoder is used that would always yield legal solutions to the problem. Therefore, a legal priority queue (a schedule) is one that satisfies the following conditions.

- The precedence relations among the tasks are satisfied (correctness).
- Every task is present and appears only once in the priority queue (completeness and uniqueness).

In this paper, the representation needs to consider the precedence relations between the subtasks of a DAG application. The precedence relations within a processor, however, must still be maintained. We must bear this in mind when we design GA operators.

- For GA crossover operation, which aims to take the best features of each parent and mix the remaining features in forming the offspring, a subtask (gene) is deleted from a priority queue (chromosome) of a DAG application that can generate a new offspring without violating precedence constraints. The detail is shown in Fig. 11 and Algorithm 3.
- For GA mutation operation, which aims to introduce variations into the individuals, a subtask (gene) can move to any position between the nearest predecessor and the nearest successor in a priority queue (chromosome) of a DAG application. The approach can generate a new offspring without violating precedence constraints. The detail is shown in Fig. 12 and Algorithm 4.

4.3.2. Crossover operator

The population of a genetic algorithm evolves largely by crossovers and mutations. In our experiments, the most dominant genetic operator is crossover, since it usually changes the solutions most. A crossover is a procedure of replacing some of the genes in one parent by corresponding genes of the other. In the task scheduling problem, the crossover operator is combining two valid parents, whose subtasks are ordered topologically, to generate two offsprings which will also be valid.

Theorem 1. *For a DAG graph, a solution of task scheduling is an execution order which is a topological order of tasks based on their dependencies. It is still a topological order without violating precedence constraints when a task T_i is deleted from the topological order (i.e., the priority queue).*

Proof. When task T_i is deleted from a topological order, the remaining priority queue is actually a topological order of the new DAG obtained by removing T_i from the original DAG. Hence, all precedence constraints of the new DAG are preserved in the remaining priority queue, which is certainly a topological order of the new DAG. □

Theorem 2. *For a DAG graph, a task T_i can be inserted into any position between $Pred(T_i)$ and $Succ(T_i)$, which can generate a new topological order (i.e., priority queue) without violating precedence constraints.*

Proof. Notice that all tasks between $Pred(T_i)$ and $Succ(T_i)$ are independent of T_i. In other words, if task T_j is between $Pred(T_i)$ and $Succ(T_i)$, then there is no precedence constraint between T_i and T_j. Hence, the relative order between T_i and T_j in any topological order can be arbitrary. This implies that T_i can be inserted into any position between $Pred(T_i)$ and $Succ(T_i)$. □

In order to keep the genetic diversity, we adopt a novel approach to reducing the chance of cloning before offspring are generated. [54] suggested that before applying crossover, we should examine the selected parents to find suitable crossover points. This entails computing an exclusive-OR (XOR) between the parents, so that only positions between the outermost 1s of the XOR string (the reduced surrogate) should be considered as crossover points. For instance, the XOR result of the two task priority queues, Blevel and Llevel in Table 5, is shown in Table 6.

They will generate only clones if the crossover point is any of the first three positions, so that, as previously stated, only from 3 to 6 crossover points will give rise to a different string.

In this paper, a single point crossover is applied. First, the suitable crossover point i is chosen randomly between the outermost 1s of the XOR string and n, and the crossover point cuts the priority queue of the pair of parents (father and mother) into left and right segments.

For example, suppose there are 8 subtasks, and the chromosome are coded as an integer list shown in Fig. 11. When $i = 4$, two parents swap some genes to generate two offsprings by means of single-point crossover operator. The left segment of the son or daughter is inherited from the corresponding segment of the father (0, 1, 2, 3) or mother (0, 3, 4, 6), respectively. Then according to Theorem 1, we can delete the subtasks from the topological order, and it will still be a topological order (total order) without violating precedence constraints. For example, the left segment of mother is still a topological order (4, 6, 5, 7), when we delete the subtasks which are in the left segment of father (0, 1, 2, 3). Similarly, the left segment of father is still a topological order (1, 2, 5, 7), when we delete the subtasks which are in the left segment of father (0, 3, 4, 6). Finally, according to Theorem 2, the genes of right segment of the son is copied from the mother (4, 6, 5, 7) or father (1, 2, 5, 7) by inserting them from right to left one by one in the order given by the mother (father, respectively) that do not appear in the left segment of the father (mother, respectively). A detailed description of the crossover operator is given in Algorithm 3.

Algorithm 3. Crossover operator

Input:
Two parents from the current population.
Output:
Two new offsprings.
1: Choose randomly a suitable crossover point i;
2: Cut the father's chromosome and the mother's chromosome into left and right segments;
3: Generate a new offspring, namely the son;
4: Inherit the left segment of the father's chromosome to the left segment of the son's chromosome;
5: Copy genes in mother's chromosome that do not appear in the left segment of father's chromosome to the right segment of son's chromosome;
6: Generate a new offspring, namely the daughter;
7: Inherit the left segment of the mother's chromosome to the left segment of the daughter's chromosome;
8: Copy genes in father's chromosome that do not appear in the left segment of mother's chromosome to the right segment of daughter's chromosome;
9: **return** the two new offsprings.

4.3.3. Mutation operator

The mutation operator changes a gene with certain probability (mutation probability). The mutation operator focuses on maintaining the diversity of the population so as to enlarge the search space, and helps the search algorithm escape from local optimal solutions or cooperates with crossover operator to achieve a "better" solution. According to Theorems 1 and 2, we can generate new individuals by exchange two genes without violating the precedence constraints. First, it randomly selects one gene (subtasks) T_i. Then, it finds the first successor T_j of T_i from the mutation position to the end of priority queue. If $\exists k \in [i+1, j-1]$ and the predecessors of T_k are all ahead of T_i, T_i and T_k can be interchanged, as shown in

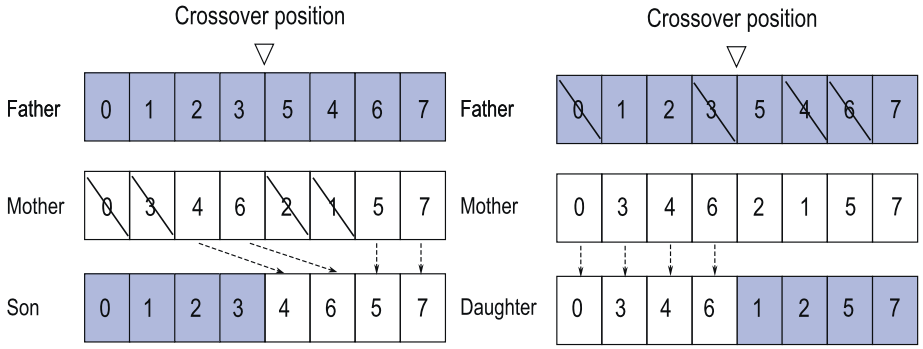


Fig. 11. Single-point crossover operator.

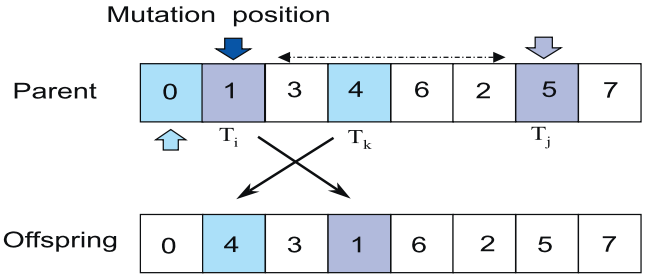


Fig. 12. Mutation operator.

Table 6
XOR string operator.

	Position of genes							
	0	1	2	3	4	5	6	7
Blevel	0	4	3	1	2	5	6	7
Llevel	0	4	3	2	1	6	5	7
Result	0	0	0	1	1	1	1	0

Fig. 12. Final, in order to maintain the genetic diversity of the population, the new chromosome can only be added to the offspring if it is not in tabu list. A detailed description of the mutation operator is given in Algorithm 4.

Algorithm 4. Mutation operator

Input:
A randomly chosen chromosome.
Output:
A new chromosome.
1: Choose randomly a gene T_i;
2: Find the first successor $T_j \in Succ(i)$;
3: Choose randomly a gene T_k in the interval $[i+1, j-1]$;
4: **if** $l < i$ for all $T_l \in Pred(k)$ **then**
5: Generate a new offspring by interchanging gene T_i and gene T_k;
6: **return** the new offspring;
7: **else**
8: Go to Step 1;
9: **end if.**

4.4. Task-to-processor mapping and fitness value computing

In this section, we first address the heuristic task-to-processor mapping approach based on the HEFT algorithm for DAG applications with precedence constraints in Section 4.4.1. In Section 4.4.2, the method of computing fitness value, which plays an important role in deciding which individuals would be used to generate the next-generation population while the genetic operators realize the concrete evolution, is presented. In the end of this section, Algorithm 5 is developed which performs the task-to-processor mapping and the fitness value computing.

4.4.1. Heuristic mapping approach

For task-to-processor mapping, the heuristic-based HEFT algorithm was proposed to search for a solution in order to minimize makespan without violating precedence constraints. The HEFT algorithm selects the subtask with the highest upward rank value at each step and assigns the selected subtask to the processor which minimizes its *earliest finish time* with an insertion-based approach [1].

The *earliest start time* (EST) of the node/subtask T_i on processor P_k is represented as $EST(T_i, P_k)$, shown in Eq. (9):

$$EST(T_i, P_k) = \begin{cases} 0, & \text{if } T_i = T_{entry}; \\ \max\limits_{T_j \in Pred(T_i)} AFT(T_j, P_l), & \text{if } P_k = P_l; \\ \max\limits_{T_j \in Pred(T_i)} (AFT(T_j, P_l) + C(T_j, T_i)), & \text{if } P_k \neq P_l. \end{cases} \tag{9}$$

The *actual start time* (AST) of node T_i on processor P_k is represented as $AST(T_i, P_k)$, shown in Eq. (10).

$$AST(T_i, P_k) = \max(EST(T_i, P_k), Avail(P_k)), \tag{10}$$

where $Avail(P_k)$ is defined as the earliest time at which the processor P_k is ready for the task execution.

The *earliest finish time* (EFT) of node T_i on processor P_k is represented as $EFT(T_i, P_k)$, shown in Eq. (11).

$$EFT(T_i, P_k) = AST(T_i, P_k) + W(T_i, P_k). \tag{11}$$

The *actual finish time* (AFT) of a subtask T_i over all processors is represented as $AFT(T_i, P_k)$, P_k is the fittest processor for the subtask T_i, shown in Eq. (12):

$$AFT(T_i, P_k) = \min_{1 \leqslant l \leqslant m} EFT(T_i, P_l). \tag{12}$$

The HEFT algorithm is used to map the subtasks to the processors without modifying it at all. The subtasks have been assigned to the processors in order of their priority. At each step of the assignment, the selected processor provides the earliest finish time for the subtask under consideration, taking into account all the communications from the subtask's parents. A detailed description is given in Algorithm 5.

Algorithm 5. Task-to-processor mapping

Input:
A task priority queue.
Output:
The makespan.
1: Fill the priority queue with subtasks;
2: **while** the priority queue is not empty **do**
3: Select the first subtask T_i from the priority queue;
4: **for** each processor P_k in the processor set **do**
5: Compute $EFT(T_i, P_k)$ value using the insertion-based HEFT scheduling policy;
6: Assign subtask T_i to the processor P_k that minimizes $EFT(T_i, P_k)$;
7: **end for**;
8: Remove T_i from the priority queue;
9: **end while**;
10: **return** makespan=$AFT(T_{exit})$.

HEFT is a performance-effective and low-complexity task scheduling algorithm on heterogeneous computing systems. The time complexity of the HEFT algorithm is $O(em)$ for e edges and m processors. For a dense graph where the number of edges is proportional to $O(n^2)$ (n is the number of subtasks), the time complexity is in the order of $O(n^2m)$ [1]. In this paper, the HEFT algorithm is adopted to realize performance-effective and low-complexity task-to-processor mapping.

4.4.2. Fitness value computing

Fitness value plays an important role in deciding which individuals would be used to generate the next-generation population while the genetic operators realize the concrete evolution.

In this paper, the overall schedule length of the entire DAG, namely makespan, is the latest finish time among all subtasks, which is equivalent to the actual finish time of the exit subtask T_{exit}. For the task scheduling problem, the goal is to obtain a subtask assignment that ensures the minimum makespan and ensures that the precedence of the subtasks are not violated. The makespan, or the scheduling length, is defined as:

$$makespan = AFT(T_{exit}), \tag{13}$$

where $AFT(T_{exit})$ is the actual finishing time of the exit subtask T_{exit}.

Hence, the fitness function value is defined as

$$fitness_i = \max_{j \in Pop}(makespan_j) - makespan_i + 1. \tag{14}$$

The above definition means that the shorter the makespan, the greater the fitness.

4.5. Selection operator

Selection is an important part of a genetic algorithm, since it affects its convergence significantly. The basic strategy follows the following rule, i.e., the better fitted an individual, the higher the probability of its survival and mating. The most straightforward implementation of this rule is the so-called *roulette-wheel selection* [46]. This method assumes that the probability of selection is proportional to the fitness of an individual. It can be briefly described as follows. Some chromosomes will be selected to undergo genetic operations for reproduction according to their fitness values. A chromosome having a higher fitness value should therefore have a higher chance to be selected. Let us consider *PopSize* individuals, each characterized by its fitness $fitness_i > 0 (i = 1, 2, \ldots, PopSize)$. The probability p_i of each chromosome to be selected can be calculated according to the probability defined by the equations:

$$p_i = \frac{fitness_i}{\sum_{j=1}^{PopSize} fitness_j}, \tag{15}$$

and

$$S_i = \sum_{j=1}^{i} p_j, \tag{16}$$

where S_i is the sum of p_j from 1 to i. Consequently, a more fitted chromosome will be selected with higher probability and it will get more offsprings. The average will stay and the worst will die off. A detailed description of roulette-wheel selection is given in Algorithm 6.

Algorithm 6. Roulette-wheel selection

Input:
A current population *Pop*.
Output:
A selected chromosome.
1: Generate a random number $R \in [0, 1]$;
2: **for** $i = 1$ to *PopSize* **do**
3: **if** $S_i > R$ **then**
4: Select the ith individual;
5: **return** the ith individual;
6: **end if**
7: **end for.**

4.6. Termination criterion

GAs are stochastic search methods that could in principle run for ever. In practice, a termination criterion is needed. Common approaches are to set a limit on the number of fitness evaluations or the computer clock time, or to trace the population's diversity and stop when this falls below a preset threshold. In this paper, the termination criterion is that the makespan stays unchanged for 1,000 consecutive iterations in the search loop.

5. Time and space complexity analysis

The time complexity of MPQGA is analyzed as follows. According to Algorithm 1, the time is mainly spent in running the searching loop (Steps 2–11) in the proposed MPQGA. In each iteration of the loop, the algorithm needs to execute fitness evaluation function, selection operator, crossover operator, and mutation operator. The time complexity of the fitness evaluation function (Step 3) is $O(e \times m)$ [1], where e is the number of edges in the DAG and m is the number of heterogeneous processors. The time complexity of the selection operator (Step 6) is $O(n)$. The time complexity of the crossover operator (Step 7) is $O(n^2)$. The time complexity of the mutation operator (Step 8) is $O(n^2)$. Therefore, the time complexity of MPQGA is $O(generes \times (e \times m + n + n^2 + n^2))$, where *geners* is the number of iterations (generations) performed by MPQGA. For a dense graph where the number of edges is proportional to $O(n^2)$ (n is the number of subtasks), the time complexity is in the order of $O(generes \times n^2 \times m)$.

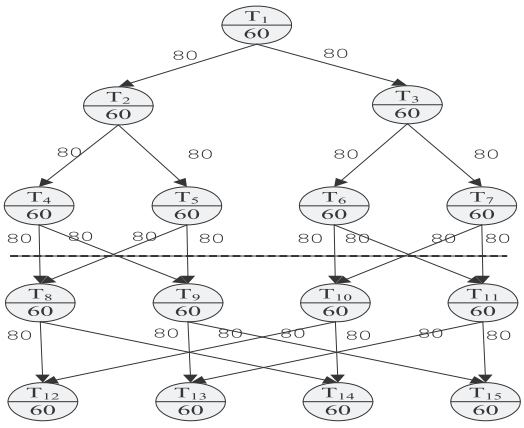
The space complexity of MPQGA can be analyzed as follows. In MPQGA, for each chromosome we need an array of size n to store it. There are *PopSize* chromosomes in the initial population. Therefore, the space complexity of MPQGA is $O(PopSize \times n)$.

Table 7
Experimental parameters.

Experimental parameters	Comment
The population size	Ten times the number of subtasks
The crossover probability	0.7
The mutation probability	0.35
The elitism size	The number of the fittest individuals in the current population
The stopping criteria	The makespan stays unchanged for 1,000 consecutive iterations in the search loop

```
FFT(A,ω ) function
1:n=length(A)
2:if (n=1) return(A)
3:Y(0)=FFT((A[0],A[2]… A[n-2]),ω 2)        recurse
4:Y(1)=FFT((A[1],A[3]… A[n-1]),ω 2)
5:for i=1 to n/2-1 do
6:   Y[i]=Y(0)[i]+ω i*Y(1)[i]              butterfly
7:   Y[i+k/2]=Y(0)[i]-ω i*Y(1)[i]
8:endfor
9:return(Y)
```

(a): FFT Algorithm.



(b): The generated task graph of fast Fourier transformation with four points.

Fig. 13. Fast Fourier transformation.

6. Simulation and results

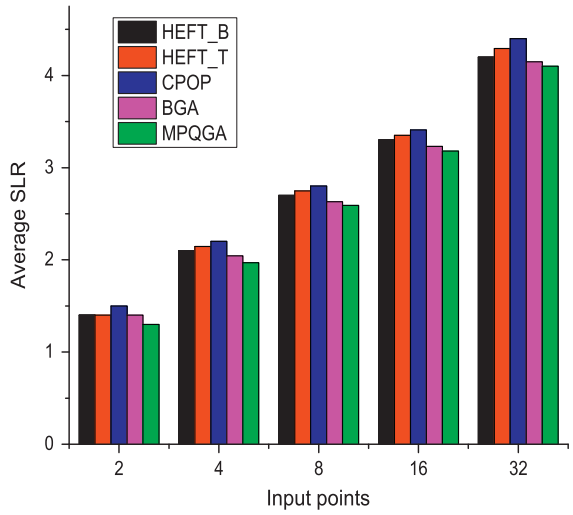
To demonstrate the power of MPQGA for the task scheduling problem, we compare this algorithm with previously proposed heuristics algorithms, i.e., HEFT-B, HEFT-T, CPOP [1], and BGA [44]. For this purpose, we consider two sets of graphs as the workload, two kinds of real-world application graphs, and some randomly generated application graphs. Several metrics are used to evaluate the performance.

MPQGA is programmed in C#. A DAG in the program is represented by a class, whose members include an array of subtasks, a matrix of the speed at which each subtask runs on each processor, and a matrix of communication data between every pair of subtasks. A subtask in the program is also represented by a class, whose members include an array of predecessors of the subtask, an array of successors of the subtask, the indegree of the subtask, the outdegree of the subtask, and the computational data of the subtask. The simulations are performed on the same PC with an Intel Core 2 Duo-E6700 @ 2.66 GHz CPU and 2 GB RAM.

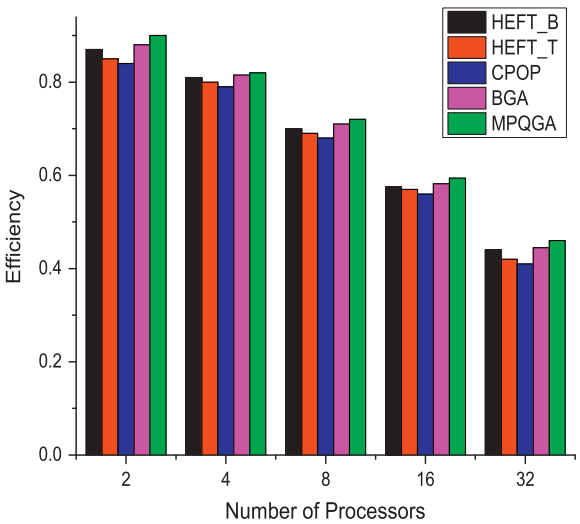
In the experiments, based on the results of preliminary studies, the values of parameters to be used in the MPQGA are described in Table 7.

6.1. Performance comparison metrics

The following two metrics are used to compare the performance of the MPQGA task scheduling algorithm developed in this paper with related work.



(a): Average SLRs of the algorithms vs. the size of FFT graph.



(b): Efficiency of the algorithms vs. the number of processors.

Fig. 14. Average SLR and the efficiency of algorithms for FFT.

6.1.1. Scheduling length ratio

The main performance measure of a scheduling algorithm on a DAG is the schedule length (makespan). Since a large set of application graphs with different properties is used, it is necessary to normalize the schedule length of each graph to a lower bound. *Schedule Length Ratio* (SLR) is defined as the normalized schedule length to the lower bound of the schedule length. The SLR value of an algorithm for a task graph is defined as

$$SLR = \frac{makespan}{\sum_{T_i \in CP_{min}} \min_{P_k \in P}(W(T_i, P_k))}. \tag{17}$$

The CP_{min} is the critical path of the unscheduled application DAG based on the computation cost of tasks on the fastest processor P_k. The denominator is equal to the sum of computation costs of tasks located on CP_{min} when they are scheduled on P_k.

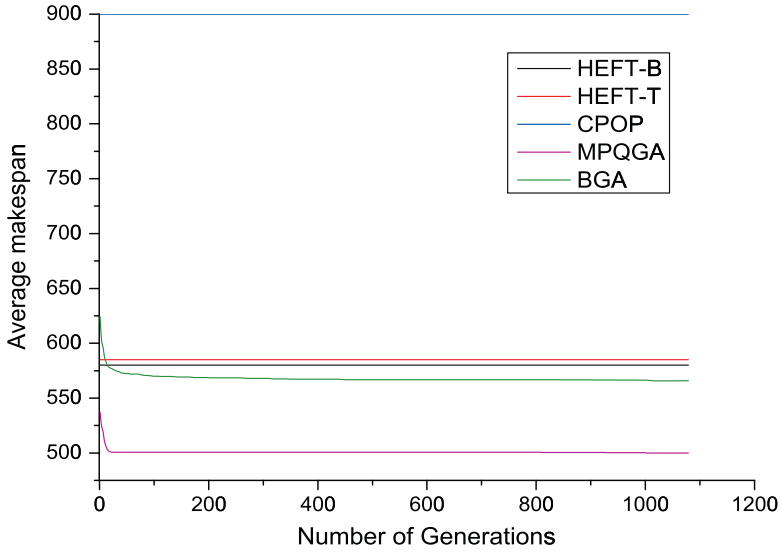


Fig. 15. The convergence trace of the makespan of the best individual for FFT (the number of processors = 4, CCR = 1.0, 4 data points, 50 independent runs).

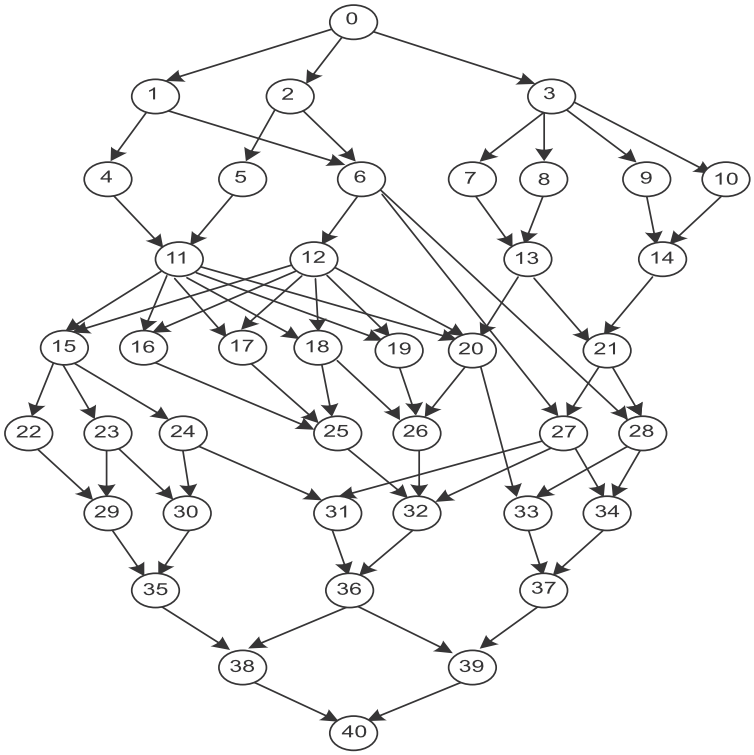


Fig. 16. A molecular dynamics code.

The SLR value of any algorithm for a DAG cannot be less than one, since the denominator in the equation is a lower bound for the completion time of the graph. For comparing the performance of scheduling algorithms, the average SLR values over several task graphs were used in the experiments, and the smaller the SLR the better.

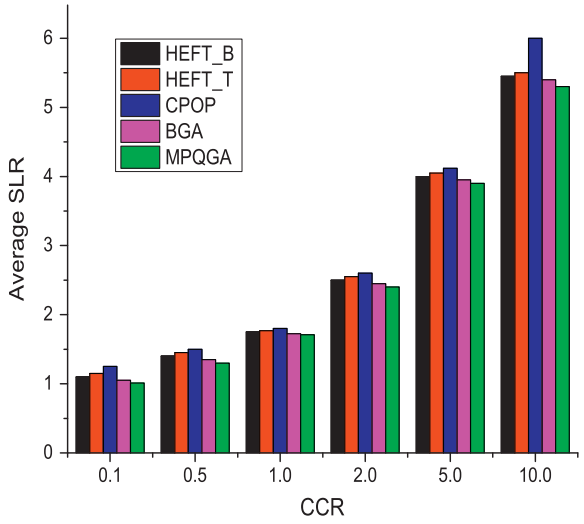
6.1.2. Speedup and efficiency

The speedup value is defined as the ratio of the sequential execution time obtained by assigning all tasks to the fastest processor (cumulative computation cost of the subtasks in a graph), to the parallel execution time (the makespan of the output schedule). The sequential execution time is computed by assigning all tasks to a single processor that minimizes the cumulative computation cost of the subtasks, namely $speedup_{min}$. It is defined as:

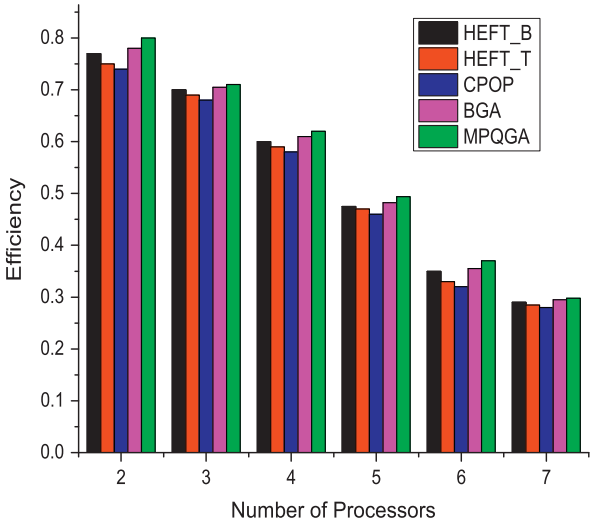
$$Speedup_{min} = \frac{\min_{P_k \in P}\left(\sum_{T_i \in T} W(T_i, P_k)\right)}{makespan}. \tag{18}$$

Efficiency is the ratio of the speedup value to the number of processors used. In addition to providing minimum SLR values, the task scheduling algorithms target the maximization of speedup and efficiency.

In our experiments, every experimental result is the average value for 50 separate runs.



(a): Average SLR soft heal gorithms vs. CCR.



(b): Efficiency of the algorithms vs.the number of processors.

Fig. 17. Average SLR and the efficiency of algorithms for a molecular dynamics code.

6.2. Real-world application graphs

The first test set includes task graphs of two real-world problems, i.e., *fast Fourier transformation* (FFT) [55] and *molecular dynamics code* [56], to evaluate the performance of MPQGA.

6.2.1. Fast Fourier transformation

The recursive, one-dimensional FFT algorithm [55] and its task graph (when there are four data points) are given in Fig. 13. In this figure, A is an array of size k which holds the coefficients of the polynomial and array Y is the output of the algorithm. The algorithm consists of two parts, i.e., recursive calls (lines 3–4) and the butterfly operation (lines 6–7). The task graph in Fig. 13(b) can be divided into two parts, i.e., the subtasks above the dashed line are the recursive call subtasks and the ones below the line are butterfly operation subtasks. For an input vector of size k, there are $2k-1$ recursive call subtasks and $k\log_2 k$ butterfly operation subtasks. (We assume that $k = 2^i$ of some integer i.) Each path from the entry subtask to any of the exit subtasks in an FFT task graph is a critical path, since the computation costs of subtasks in any level are identical and the communication costs of all edges between two consecutive levels are identical.

Fig. 14(a) shows the average SLR values of the scheduling algorithms for FFT task graphs of various sizes. The MPQGA algorithm outperforms the other algorithms in terms of the average SLR. Fig. 14(b) presents the efficiency of the algorithms when there are 64 data points. The MPQGA algorithm outperforms the other algorithms in terms of efficiency.

Fig. 15 shows the final makespan achieved by MPQGA and BGA after the stopping criteria are satisfied. In this case, the fact that MPQGA converges faster than BGA means that the makespan obtained by MPQGA could be much better than that by BGA when the algorithms stop.

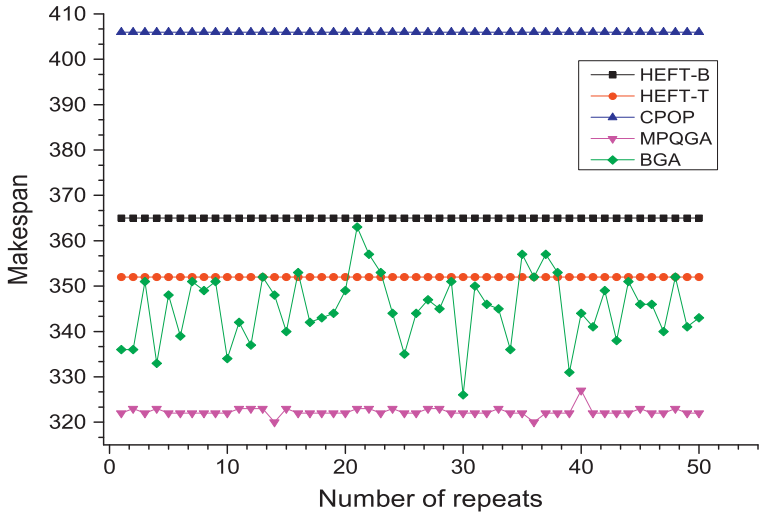


Fig. 18. The makespan of molecular dynamics code vs. the number of running repeats (the number of processors = 8, CCR = 1.0, 41 subtask nodes, 50 independent runs).

Table 8
Experimental statistical analysis.

	Fast Fourier transformation		Molecular dynamics code	
	MPQGA	BGA	MPQGA	BGA
Average makespan	66	70.2	322.3	345.22
Maximum makespan	66	79	327	363
Minimum makespan	66	69	320	326
Standard deviation	0.00	2.07	0.87	7.55

Table 9
The different parameters of random task graphs.

Tasks	Levels	Maximum		Average
		In-degree	Out-degree	In-degree
10	3–11	11	11	3.63
20	4–21	21	21	5.46
50	6–44	42	44	5.86
100	7–57	47	51	7.81
200	7–77	181	188	30.72

6.2.2. Molecular dynamics code

Because a fast Fourier transformation has a regular task graph, we did analytical work to set approximate computation costs of the tasks and communication costs of the edges (see Section 6.2.1).

The molecular dynamics code has an irregular task graph. For a molecular dynamics application, the computation cost of each subtask is randomly set from a normal distribution with mean equal to an assigned average computation cost, and each communication cost is set randomly from a normal distribution with mean equal to the multiplication of CCR and average computation cost. Fig. 16 represents the DAG of a molecular dynamics code as given in [56]. Again, since the graph has a fixed structure and a fixed number of processors, the only parameters that could be varied are the number of heterogeneous processors and the CCR values. The CCR values that were used in our experiments are 0.1, 0.5, 1.0, 2.0, 5.0, and 10.0.

Fig. 17(a) and (b) show the average SLR and the efficiency of algorithms under different CCR values and different number of heterogeneous processors respectively. Fig. 17(a) shows that the average SLR values of all algorithms increase when the CCR value is increased. Since there are at most seven tasks in any level in Fig. 16, the number of processors in the experiment is bounded from above by seven processors. Fig. 17(b) plots the efficiency of the algorithms vs. the different numbers of processors. Again, we observe that the MPQGA algorithm outperforms the other algorithms in terms of both SLR and efficiency.

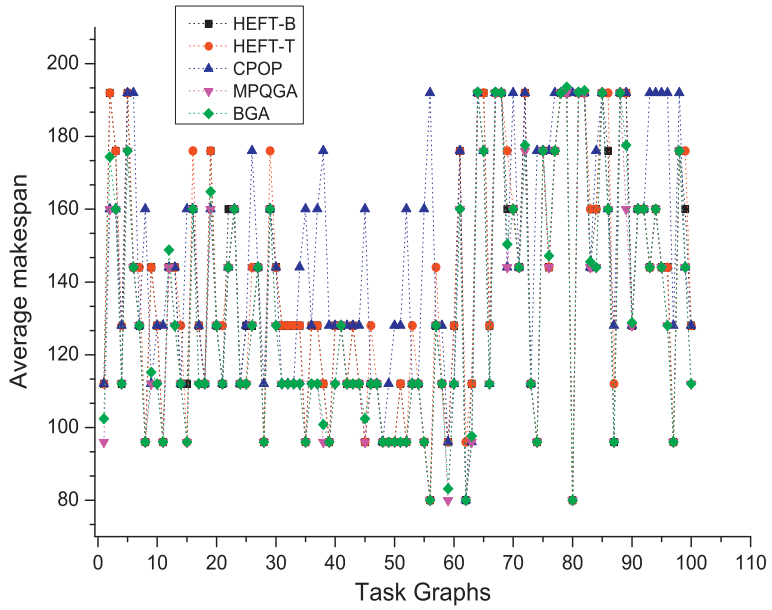


Fig. 19. The average makespan of task graphs with different characteristics (the size of the task graphs = 10, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

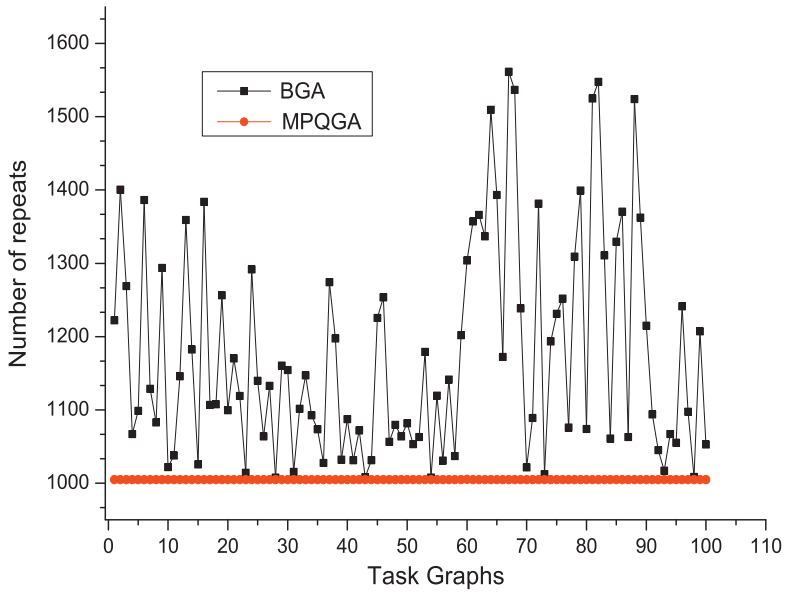


Fig. 20. The average running time of task graphs with different characteristics (the size of the task graphs = 10, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

Fig. 18 shows the final makespan achieved by MPQGA and BGA after the stopping criteria are satisfied, and MPQGA can obtain the better average makespan performance than BGA. This result verify that MPQGA is able to strike a good balance between performance and overhead, and MPQGA algorithm has a better robustness than BGA algorithm in 100 repeated independent experiments.

Table 8 shows that the result of the experimental statistical analysis for fast Fourier transformation task graphs with four data points and molecular dynamics code task graphs with 41 subtask nodes.

6.3. Random generated application graphs

In these experiments, we used randomly generated task graphs to evaluate the performance. In order to generate random graphs, we implemented a random graph generator which allows the user to generate a variety of random graphs with different characteristics. The input parameters of the generator are the number of subtasks in a graph, the number of instructions in a subtask (representing the computational data), the number of successors of a subtask (representing the degree of

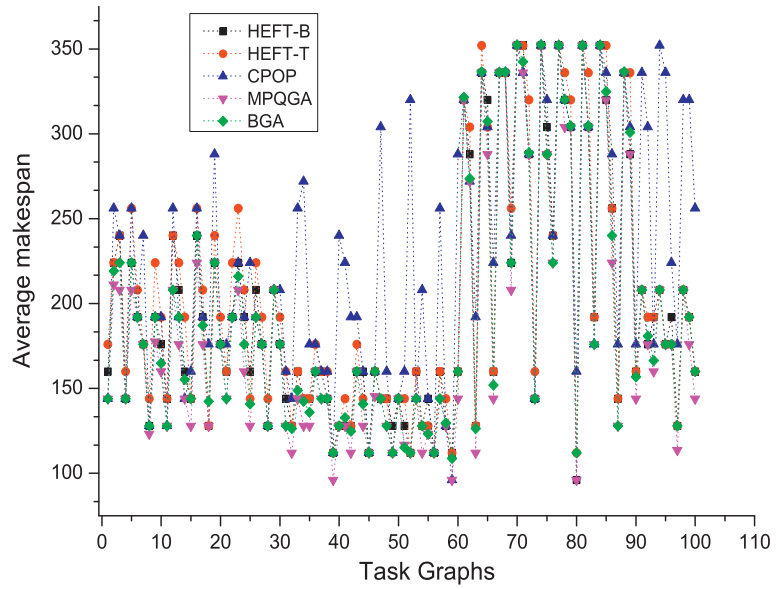


Fig. 21. The average makespan of task graphs with different characteristics (the size of the task graphs = 20, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

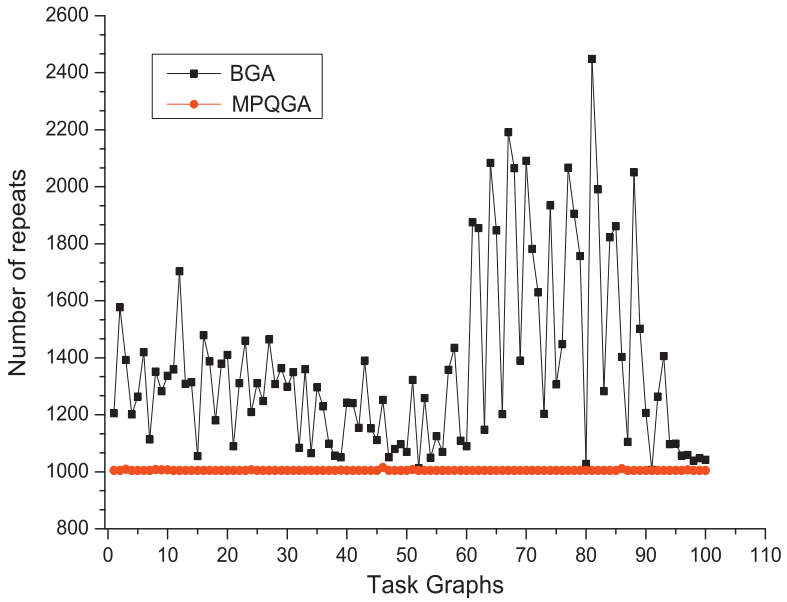


Fig. 22. The average running time of task graphs with different characteristics (the size of the task graphs = 20, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

parallelism), and CCR. We have generated a large set of random task graphs with different characteristics, and scheduled these task graphs on a heterogeneous computing system.

We have evaluated the performance of the algorithms under different parameters, including different numbers of subtasks, different numbers of heterogeneous processors, and different CCR values. MPQGA is compared with other algorithms in terms of makespan. Each value plotted in the graphs is the result averaged over 100 different random DAG graphs.

The following is the values of parameters used in the simulation experiments, unless otherwise stated. The number of tasks generated in a DAG is randomly selected from 10, 20, 50, 100, and 200. The computational costs of the DAG subtasks are generated as follows. The amount of computation of each subtask in the DAG (i.e., $W_d(T_i)$) is randomly selected from a range. Unless otherwise stated, the range is [1, 80]. The computational cost of subtask T_i on processor P_k is then calculated using Eq. (1). The number of successors that a task can have is a random number. The number of processors is 16. The random DAG task graphs have the following characteristics shown in Table 9.

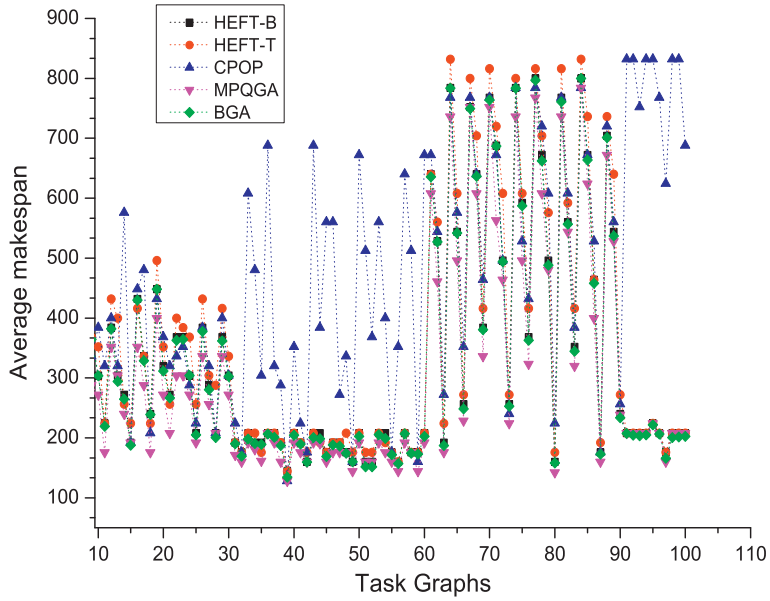


Fig. 23. The average makespan of task graphs with different characteristics (the size of the task graphs = 50, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

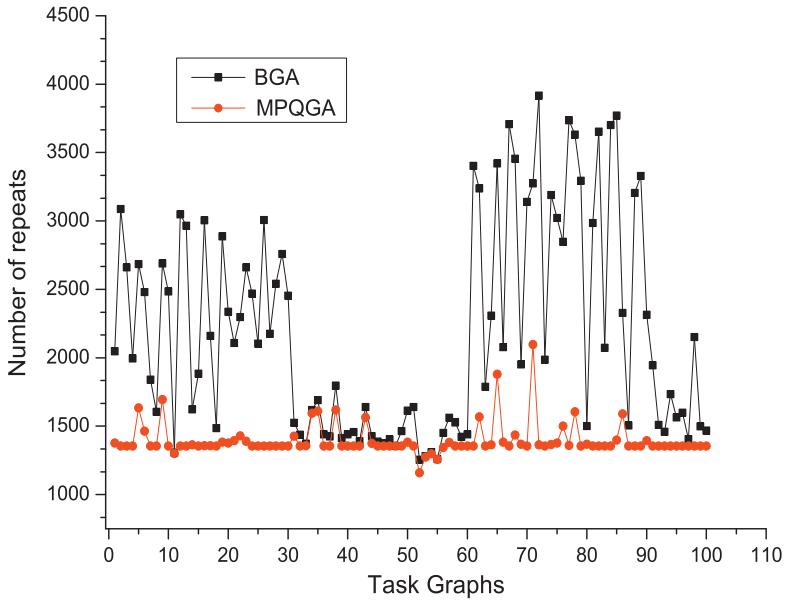


Fig. 24. The average running time of task graphs with different characteristics (the size of the task graphs = 50, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

Fig. 19, Fig. 21, Fig. 23, Fig. 25, and Fig. 27 show that MPQGA outperforms HEFT-B and CPOP, which is to be expected. This may be because when the number of tasks becomes bigger, the problem becomes more complicated and it comes increasing difficult for the heuristic algorithms to find good solutions. As it can also be seen from these figures that MPQGA is able to achieve better average performance than BGA in all cases.

Figs. 20, 22, 24, 26, and 28 show the number of repetitions required by MPQGA and BGA after the stopping criteria are satisfied. Our experiments show that MPQGA incurs much less overhead to find a desirable solution than BGA. The results indicate that although MPQGA costs much less than BGA to find a sub-optimal solution, it can still achieve similar or better performance. Our experimental results show that MPQGA can obtain better average makespan performance than BGA when they stop searching.

As shown in Fig. 29, MPQGA always outperforms the other algorithms as the number of subtasks in a DAG increases.

Fig. 30 shows the average makespan for the increasing values of CCR. It can be observed that the average makespan increases rapidly with the increase of CCR. This is because when CCR increases, an application becomes more communication-intensive, and consequently the processors are in the idle state for more time.

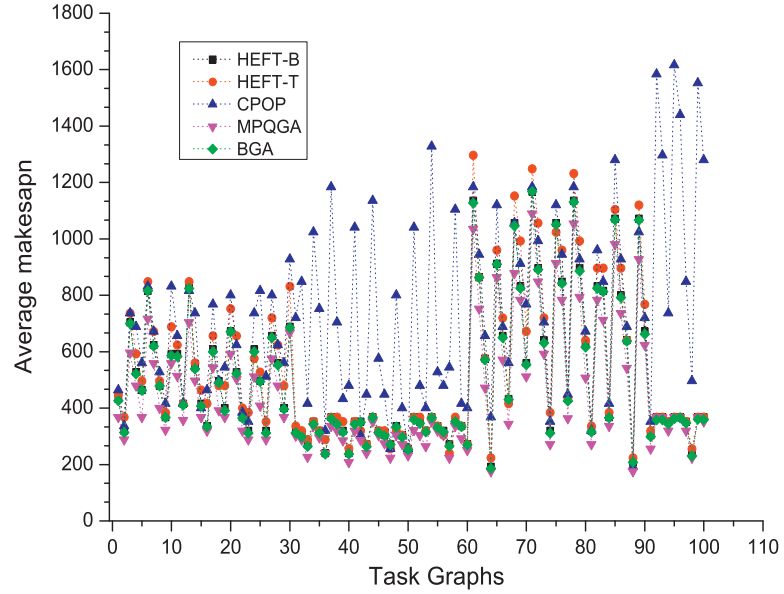


Fig. 25. The average makespan of task graphs with different characteristics (the size of the task graphs = 100, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

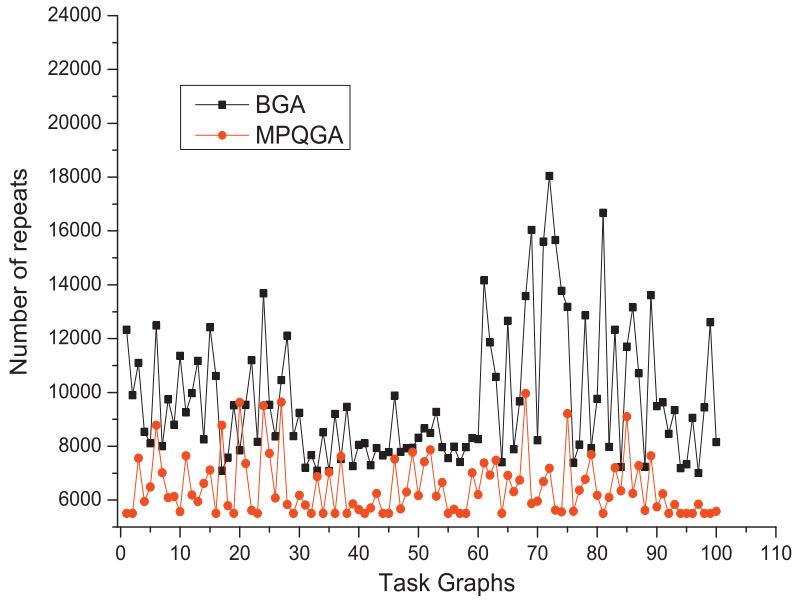


Fig. 26. The average running time of task graphs with different characteristics (the size of the task graphs = 100, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

Fig. 31 compares the efficiency of these algorithms for the increasing number of heterogeneous processors. As can be seen from the figure, MPQGA outperforms the other heuristic algorithms in all cases.

In Fig. 32, the results show that the MPQGA and the BGA algorithms typically outperform other algorithms. The reason that MPQGA and BGA typically outperform HEFT-B, HEFT-T, and CPOP is that MPQGA and BGA search a wider range of the solution space for an optimal scheduling, while HEFT-B, HEFT-T, and CPOP narrow the search down to a very small portion of the solution space by means of their heuristics. Therefore, MPQGA and BGA are more likely to obtain better solutions than HEFT-B, HEFT-T, and CPOP.

6.4. Convergence trace

Our experiments show that MPQGA and BGA achieve very similar performance when the size of a DAG application is small, and MPQGA can achieve better performance than BGA when the size of a DAG application is large. The BGA we used

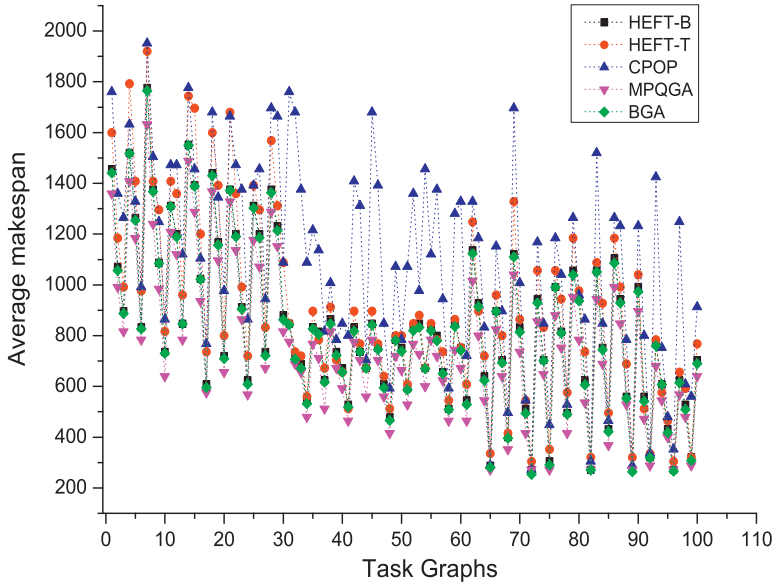


Fig. 27. The average makespan of task graphs with different characteristics (the size of the task graphs = 200, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

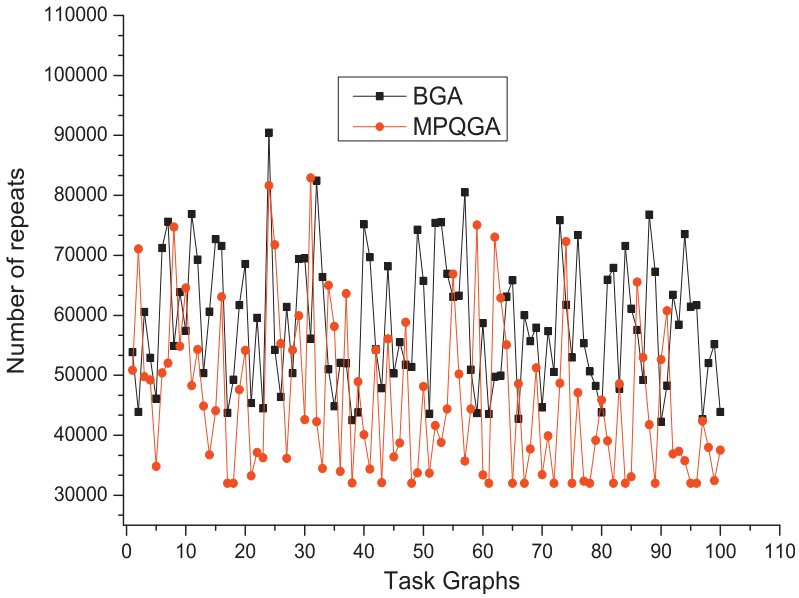


Fig. 28. The average running time of task graphs with different characteristics (the size of the task graphs = 200, the number of task graphs = 100, the number of processors = 16, 50 independent runs).

in the experiments is taken from reference [44], and it has been shown that it is well designed. Therefore the fact that MPQGA presents better average performance than BGA indicates that MPQGA developed in our work is also well designed.

A close observation of the figures shows that MPQGA slightly outperforms BGA in all cases. All meta-heuristic methods that search for optimal solutions are the same in performance when averaged over all possible objective functions, and as long as it runs for long enough, it will gradually approach the optimal solution in theory. The MPQGA adopts a heuristic-based task-to-processor mapping technique to search for a solution in order to minimize makespan without violating precedence constraints. The heuristic-based task-to-processor mapping approach effectively avoids the inefficient task-to-processor mapping and accelerates the convergence speed of the MPQGA algorithm. The reason why MPQGA has better performance than BGA in all cases is because when the stopping criteria set in the experiments is satisfied, the performance

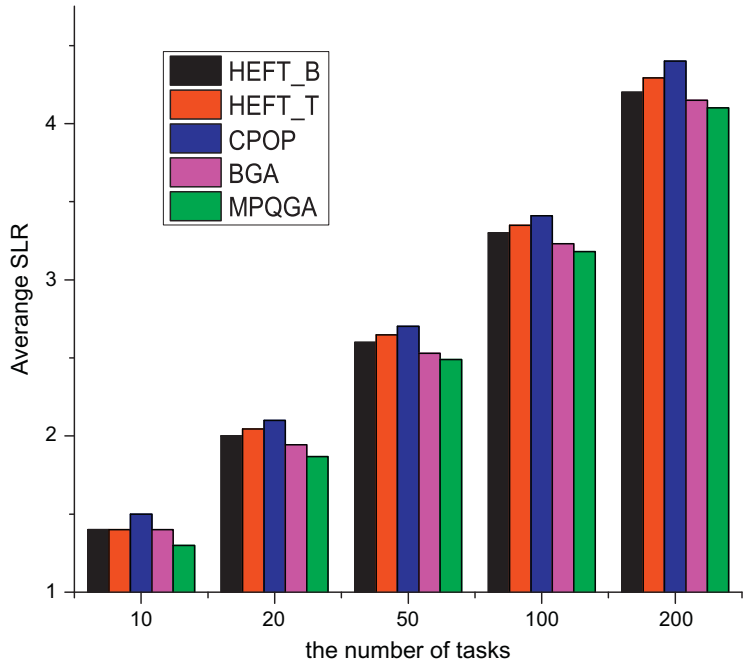


Fig. 29. Average SLR vs. the number of subtasks (CCR = 1, the number of processors = 16).

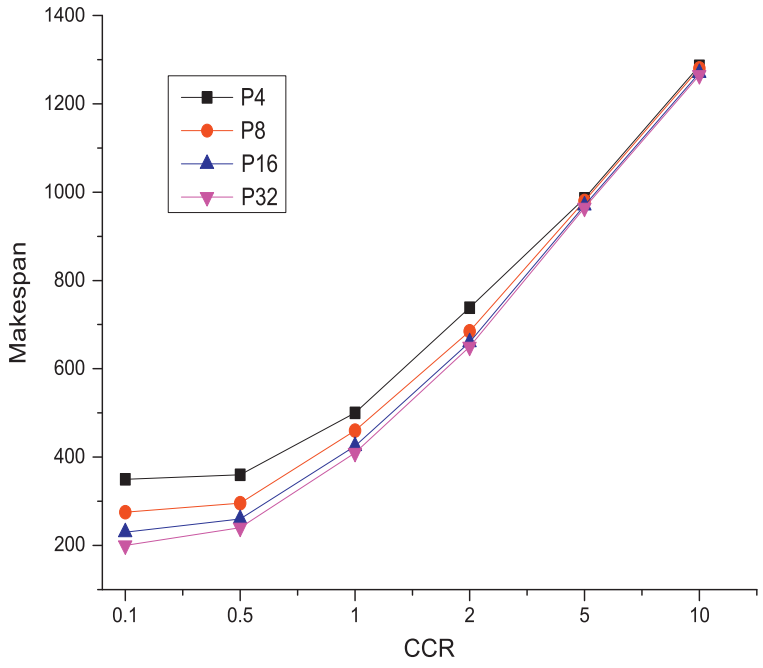


Fig. 30. Average makespan of MPQGA under different values of CCR.

obtained by MPQGA is better than that obtained by BGA. This also shows that MPQGA is more efficient in searching good solutions than BGA.

The last experiments show the final makespan achieved by MPQGA and BGA after the stopping criteria are satisfied. These results show that MPQGA can obtain better makespan than BGA, and that in all cases the final makespan obtained by MPQGA is better than that by BGA when they stop searching.

Figs. 33–37 plot the convergence of makespan for processing a set of randomly generated DAGs with 10, 20, 50, 100, and 200 subtasks. It can be observed from the figures that the makespan decreases quickly as both MPQGA and BGA progress when the size of a DAG application is small, and that the decreasing trends tail off when the size of a DAG application is large. The figures also show that their convergence speeds are rather different, i.e., our proposed MPQGA algorithm converges faster than BGA algorithm, and the final makespan achieved by MPQGA algorithm is better than BGA algorithm when the size of a DAG application is large, although the final makespan achieved by both algorithms are almost the same when the size of a DAG application is small.

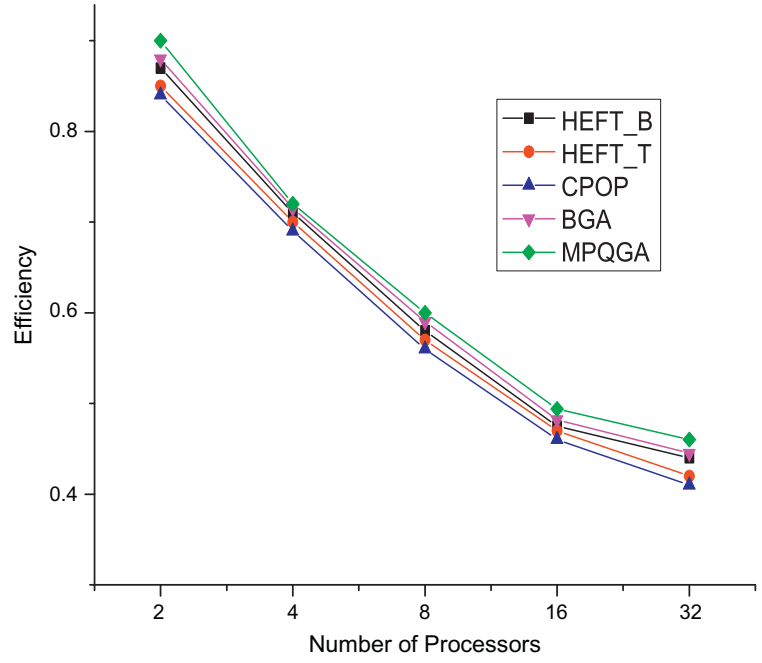


Fig. 31. Efficiency of the algorithms vs. different numbers of processors (CCR = 0.1).

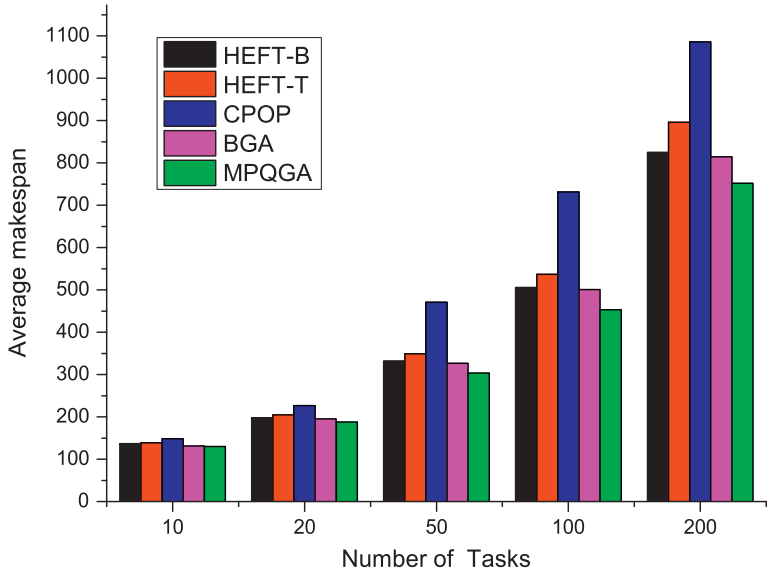


Fig. 32. Average makespan vs. the size of random DAG applications.

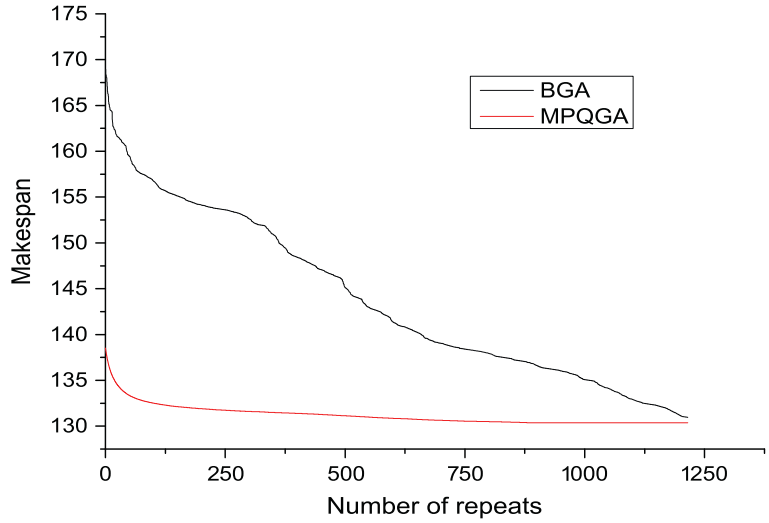


Fig. 33. The convergence of makespan for randomly generated DAGs with 10 subtasks.

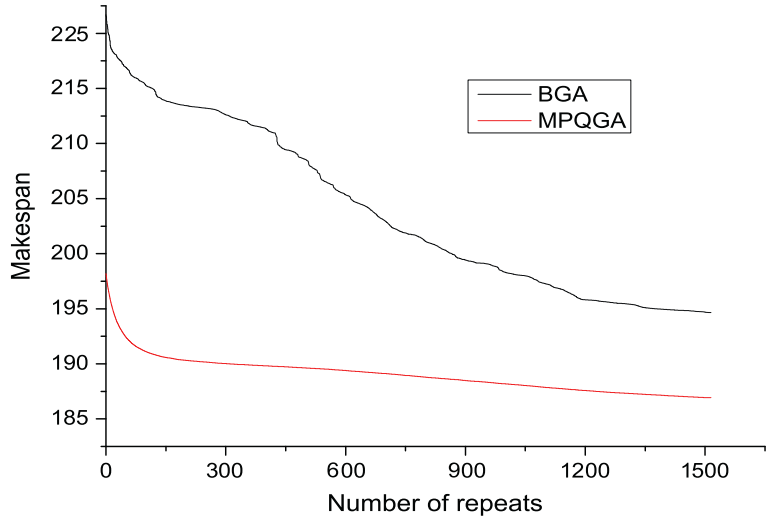


Fig. 34. The convergence of makespan for randomly generated DAGs with 20 subtasks.

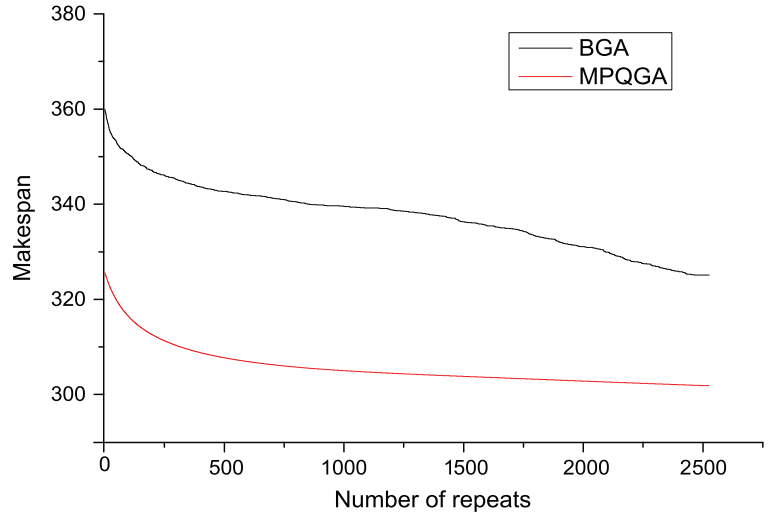


Fig. 35. The convergence of makespan for randomly generated DAGs with 50 subtasks.

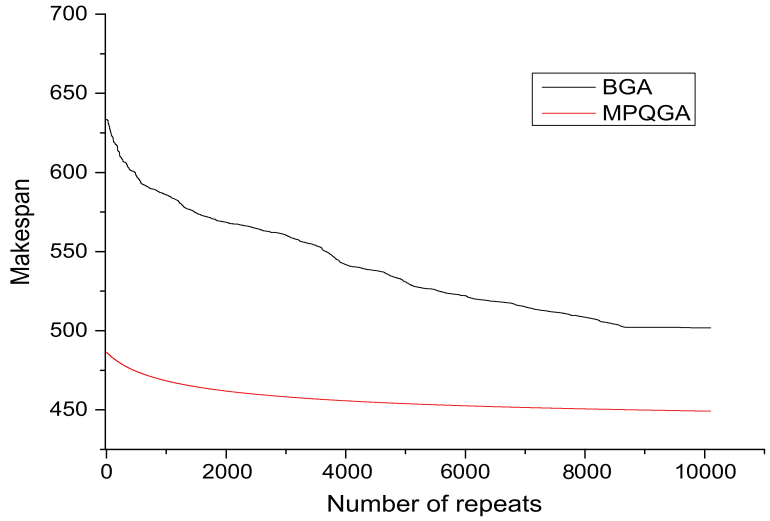


Fig. 36. The convergence of makespan for randomly generated DAGs with 100 subtasks.

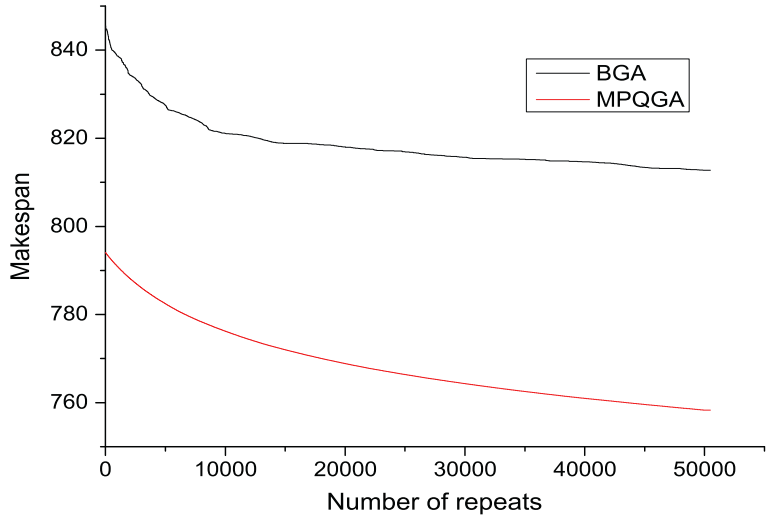


Fig. 37. The convergence of makespan for randomly generated DAGs with 200 subtasks.

7. Conclusions

In this paper, a stochastic search method based on the genetic algorithm approach is proposed and we designed MPQGA for task scheduling on heterogeneous computing systems. This algorithm incorporates a GA-based approach to assigning priority to subtasks while using a heuristic-based HEFT search to map subtasks to processors. Moreover, the crossover and the mutation operators developed take into account the precedence relations of the subtasks and guarantee that the new offsprings generated are legal. As a result, this algorithm can cover a larger search space than deterministic scheduling approaches without incurring high computational cost. The experiments show that our MPQGA algorithm outperforms HEFT-B, HEFT-T, CPOP, and BGA, with a higher speedup of subtask execution.

In future work, we are planning to extend our study using additional experiments to evaluate the performance of our algorithm. We will test the algorithm on larger task graphs and more processors, and variable degrees of heterogeneity among processors and subtasks. Furthermore, we plan to extend MPQGA by investigating the following three challenging directions. First, we plan to use MPI to parallelize the running of MPQGA, so as to further reduce the time needed to find good solutions. Second, we will apply the *dynamic voltage scaling* (DVS) technique [57–59] to our MPQGA task scheduling algorithm to realize bi-objective optimization to minimize makespan and energy. Third, we will apply MPQGA task scheduling algorithm in cloud environments.

Acknowledgments

We are very grateful to the editor and the three anonymous reviewers for their constructive comments which have helped to substantially improve the manuscript. This research was partially funded by the Key Program of National Natural

Science Foundation of China (Grant No. 61133005), the National Natural Science Foundation of China (Grant Nos. 61370095, 90715029, 61070057, 60603053), Project supported by the National Science Foundation for Distinguished Young Scholars of Hunan (12JJ1011) and the Project supported by Hunan Provincial Science and Technology Program of China (No. 2013GK3082).

References

[1] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.
[2] Y. Xu, K. Li, T.T. Khac, M. Qiu, A multiple priority queueing genetic algorithm for task scheduling on heterogeneous computing systems, in: High Performance Computing and Communication 2012 IEEE 14th International Conference on, 2012, pp. 639–646. http://dx.doi.org/10.1109/HPCC.2012.91.
[3] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, New York, 1979.
[4] A. Zomaya, C. Ward, B. Macey, Genetic scheduling for parallel processor systems: comparative studies and performance issues, IEEE Trans. Parallel Distrib. Syst. 10 (8) (1999) 795–812.
[5] Y.-K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. (CSUR) 31 (4) (1999) 406–471.
[6] A. Amini, T.Y. Wah, M. Saybani, S. Yazdi, A study of density-grid based clustering algorithms on data streams, in: 2011 Eighth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), vol. 3, 2011, pp. 1652–1656.
[7] K. Gkoutioudi, H.D. Karatza, Task cluster scheduling in a grid system, Simul. Modell. Practice Theory 18 (9) (2010) 1242–1252. http://dx.doi.org/10.1016/j.simpat.2010.04.011.
[8] P.K. Mishra, A. Mishra, K.S. Mishra, A.K. Tripathi, Benchmarking the clustering algorithms for multiprocessor environments using dynamic priority of modules, Appl. Math. Modell. 36 (12) (2012) 6243. http://dx.doi.org/10.1016/j.apm.2012.02.011.
[9] C.-S. Lin, C.-S. Lin, Y.-S. Lin, P.-A. Hsiung, C. Shih, Multi-objective exploitation of pipeline parallelism using clustering, replication and duplication in embedded multi-core systems, J. Syst. Architect. 59 (10, Part C) (2013) 1083–1094. http://dx.doi.org/10.1016/j.sysarc.2013.05.024.
[10] K. Shin, M. Cha, M. Jang, J. Jung, W. Yoon, S. Choi, Task scheduling algorithm using minimized duplications in homogeneous systems, J. Parallel Distrib. Comput. 68 (8) (2008) 1146–1156. http://dx.doi.org/10.1016/j.jpdc.2008.04.001.
[11] O. Sinnen, A. To, M. Kaur, Contention-aware scheduling with task duplication, J. Parallel Distrib. Comput. 71 (1) (2011) 77–86. http://dx.doi.org/10.1016/j.jpdc.2010.10.004.
[12] X. Tang, K. Li, G. Liao, R. Li, List scheduling with duplication for heterogeneous computing systems, J. Parallel Distrib. Comput. 70 (4) (2010) 323–329. http://dx.doi.org/10.1016/j.jpdc.2010.01.003.
[13] A.A. Badawi, A. Shatnawi, Static scheduling of directed acyclic data flow graphs onto multiprocessors using particle swarm optimization, Comput. Oper. Res. 40 (10) (2013) 2322–2328. http://dx.doi.org/10.1016/j.cor.2013.03.015.
[14] A.H. Kashan, M.H. Kashan, S. Karimiyan, A particle swarm optimizer for grouping problems, Inform. Sci. 252 (0) (2013) 81–95. http://dx.doi.org/10.1016/j.ins.2012.10.036.
[15] F. Zhao, J. Tang, J. Wang, Jonrinaldi, An improved particle swarm optimization with decline disturbance index (ddpso) for multi-objective job-shop scheduling problem, Comput. Oper. Res. 45 (0) (2014) 38–50. http://dx.doi.org/10.1016/j.cor.2013.11.019.
[16] H. Li, L. Wang, J. Liu, Task scheduling of computational grid based on particle swarm algorithm, in: 2010 Third International Joint Conference on Computational Science and Optimization (CSO), vol. 2, 2010, pp. 332–336.
[17] Q. Kang, H. He, A novel discrete particle swarm optimization algorithm for meta-task assignment in heterogeneous computing systems, Microproc. Microsyst. 35 (1) (2011) 10–17. http://dx.doi.org/10.1016/j.micpro.2010.11.001.
[18] F. Ferrandi, P.-L. Lanzi, C. Pilato, D. Sciuto, A. Tumeo, Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems, IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 29 (6) (2010) 911–924. http://dx.doi.org/10.1109/TCAD.2010.2048354.
[19] H. Kim, S. Kang, Communication-aware task scheduling and voltage selection for total energy minimization in a multiprocessor system using ant colony optimization, Inform. Sci. 181 (18) (2011) 3995–4008. http://dx.doi.org/10.1016/j.ins.2011.04.037.
[20] A.R. Yildiz, Optimization of cutting parameters in multi-pass turning using artificial bee colony-based approach, Inform. Sci. 220 (2013) 399–407, http://dx.doi.org/10.1016/j.ins.2012.07.012.
[21] J. Torres-Jimenez, E. Rodriguez-Tello, New bounds for binary covering arrays using simulated annealing, Inform. Sci. 185 (1) (2012) 137–152. http://dx.doi.org/10.1016/j.ins.2011.09.020.
[22] J. Wang, Q. Duan, Y. Jiang, X. Zhu, A new algorithm for grid independent task schedule: genetic simulated annealing, in: World Automation Congress (WAC), 2010, 2010, pp. 165–171.
[23] A. Yildiz, Cuckoo search algorithm for the selection of optimal machining parameters in milling operations, Int. J. Adv. Manuf. Technol. 64 (1–4) (2013) 55–61, http://dx.doi.org/10.1007/s00170-012-4013-7.
[24] G. Garai, B. Chaudhurii, A novel hybrid genetic algorithm with tabu search for optimizing multi-dimensional functions and point pattern recognition, Inform. Sci. 221 (0) (2013) 28–48. http://dx.doi.org/10.1016/j.ins.2012.09.012.
[25] R. Shanmugapriya, S. Padmavathi, S. Shalinie, Contention awareness in task scheduling using tabu search, in: IEEE International Advance Computing Conference, 2009 (IACC 2009), 2009, pp. 272–277.
[26] A.R. Yildiz, A comparative study of population-based optimization algorithms for turning operations, Inform. Sci. 210 (2012) 81–88, http://dx.doi.org/10.1016/j.ins.2012.03.005.
[27] A.R. Yildiz, Comparison of evolutionary-based optimization algorithms for structural design optimization, Eng. Appl. Artif. Intell. 26 (1) (2013) 327–333. http://dx.doi.org/10.1016/j.engappai.2012.05.014.
[28] A. Swiecicka, F. Seredynski, A. Zomaya, Multiprocessor scheduling and rescheduling with use of cellular automata and artificial immune system support, IEEE Trans. Parallel Distrib. Syst. 17 (3) (2006) 253–262, http://dx.doi.org/10.1109/TPDS.2006.38.
[29] A. Yildiz, N. Öztürk, N. Kaya, F. Öztürk, Hybrid multi-objective shape design optimization using taguchis method and genetic algorithm, Struct. Multidisc. Optim. 34 (4) (2007) 317–332. http://dx.doi.org/10.1007/s00158-006-0079-x.
[30] C.-H. Lin, A rough penalty genetic algorithm for constrained optimization, Inform. Sci. 241 (0) (2013) 119–137. http://dx.doi.org/10.1016/j.ins.2013.04.001.
[31] E. Hou, N. Ansari, H. Ren, A genetic algorithm for multiprocessor scheduling, IEEE Trans. Parallel Distrib. Syst. 5 (2) (1994) 113–120.
[32] H. Lu, R. Niu, J. Liu, Z. Zhu, A chaotic non-dominated sorting genetic algorithm for the multi-objective automatic test task scheduling problem, Appl. Soft Comput. 13 (5) (2013) 2790–2802. http://dx.doi.org/10.1016/j.asoc.2012.10.001.
[33] J. Behnamian, S.F. Ghomi, The heterogeneous multi-factory production network scheduling with adaptive communication policy and parallel machine, Inform. Sci. 219 (0) (2013) 181–196, http://dx.doi.org/10.1016/j.ins.2012.07.020.
[34] J. Kołodziej, S.U. Khan, Multi-level hierarchic genetic-based scheduling of independent jobs in dynamic heterogeneous grid environment, Inform. Sci. 214 (0) (2012) 1–19, http://dx.doi.org/10.1016/j.ins.2012.05.016.
[35] K. Li, Z. Tong, D. Liu, T. Tesfazghi, X. Liao, A pts-pgats based approach for data-intensive scheduling in data grids, Front. Comput. Sci. China 5 (4) (2011) 513–525.

[36] M.I. Daoud, N. Kharma, A hybrid heuristic-genetic algorithm for task scheduling in heterogeneous processor networks, J. Parallel Distrib. Comput. 71 (11) (2011) 1518–1531. http://dx.doi.org/10.1016/j.jpdc.2011.05.005.
[37] M. Pedergnana, P. Marpu, M. Mura, J. Benediktsson, L. Bruzzone, A novel technique for optimal feature selection in attribute profiles based on genetic algorithms, IEEE Trans. Geosci. Remote Sens. 51 (6) (2013) 3514–3528. http://dx.doi.org/10.1109/TGRS.2012.2224874.
[38] M.D. Robles-Ortega, L. Ortega, F.R. Feito, A new approach to create textured urban models through genetic algorithms, Inform. Sci. 243 (0) (2013) 1–19. http://dx.doi.org/10.1016/j.ins.2013.03.053.
[39] R. Köker, A genetic algorithm approach to a neural-network-based inverse kinematics solution of robotic manipulators based on error minimization, Inform. Sci. 222 (0) (2013) 528–543. http://dx.doi.org/10.1016/j.ins.2012.07.051.
[40] R. Zhang, C. Wu, Bottleneck machine identification method based on constraint transformation for job shop scheduling with genetic algorithm, Inform. Sci. 188 (0) (2012) 236–252, http://dx.doi.org/10.1016/j.ins.2011.11.013.
[41] S. Balin, Parallel machine scheduling with fuzzy processing times using a robust genetic algorithm and simulation, Inform. Sci. 181 (17) (2011) 3551–3569, http://dx.doi.org/10.1016/j.ins.2011.04.010.
[42] Y. Wen, H. Xu, J. Yang, A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system, Inform. Sci. 181 (3) (2011) 567–581, http://dx.doi.org/10.1016/j.ins.2010.10.001.
[43] A. Lam, V. Li, Chemical-reaction-inspired metaheuristic for optimization, IEEE Trans. Evol. Comput. 14 (3) (2010) 381–399.
[44] S. Gupta, G. Agarwal, V. Kumar, Task scheduling in multiprocessor system using genetic algorithm, in: 2010 Second International Conference on Machine Learning and Computing (ICMLC), 2010, pp. 267–271. http://dx.doi.org/10.1109/ICMLC.2010.50.
[45] D.B. Fogel, Evolutionary algorithms in theory and practice, Complexity 2 (4) (1997) 26–27. http://dx.doi.org/10.1002/(SICI)1099-0526(199703/04).
[46] D. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, Massachusetts, New York, 1989.
[47] S. Song, K. Hwang, Y.-K. Kwok, Risk-resilient heuristics and genetic algorithms for security-assured grid job scheduling, IEEE Trans. Comput. 55 (6) (2006) 703–719.
[48] A. Nix, M. Vose, Modeling genetic algorithms with markov chains, Ann. Math. Artif. Intell. 5 (1) (1992) 79–88, http://dx.doi.org/10.1007/BF01530781.
[49] A.H. Wright, Y. Zhao, Markov chain models of genetic algorithms, in: Proceedings of the Genetic and Evolutionary Computation (GECCO) Conference, Morgan Kaufmann, 1999, pp. 734–741.
[50] C.R. Reeves, A genetic algorithm for flowshop sequencing, Comput. Oper. Res. 22 (1) (1995) 5–13. genetic Algorithms. doi:http://dx.doi.org/10.1016/0305-0548(93)E0014-K.
[51] R.K. Ahuja, J.B. Orlin, Commentary developing fitter genetic algorithms, INFORMS J. Comput. 9 (3) (1997) 251–253, http://dx.doi.org/10.1287/ijoc.9.3.251.
[52] A. Kapsalis, V.J. Rayward-Smith, G.D. Smith, Solving the graphical steiner tree problem using genetic algorithms, J. Oper. Res. Soc. 44 (4) (1993) 397–406.
[53] D. Levine, Commentary genetic algorithms: a practitioner's view, INFORMS J. Comput. 9 (3) (1997) 256–259, http://dx.doi.org/10.1287/ijoc.9.3.256.
[54] L. Davis, Genetic Algorithms and Simulated Annealing, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.
[55] Y.-C. Chung, S. Ranka, Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors, in: Proceedings of Supercomputing '92., 1992, pp. 512–521.
[56] M.-Y. Wu, D. Gajski, Hypertool: a programming aid for message-passing systems, IEEE Trans. Parallel Distrib. Syst. 1 (3) (1990) 330–343.
[57] K. Li, X. Tang, Q. Yin, Energy-aware scheduling algorithm for task execution cycles with normal distribution on heterogeneous computing systems, in: Proceedings of the International Conference on Parallel Processing, Pittsburgh, PA, United States, 2012, pp. 40–47.
[58] K. Li, Performance analysis of power-aware task scheduling algorithms on multiprocessor computers with dynamic voltage and speed, IEEE Trans. Parallel Distrib. Syst. 19 (11) (2008) 1484–1497, http://dx.doi.org/10.1109/TPDS.2008.122.
[59] K. Li, Scheduling precedence constrained tasks with reduced processor energy on multiprocessor computers, IEEE Trans. Comput. 61 (12) (2012) 1668–1681, http://dx.doi.org/10.1109/TC.2012.120.
[60] J. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, USA, 1975.