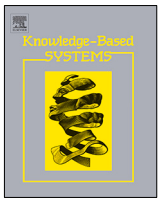
Knowledge-Based Systems 265 (2023) 110369

Contents lists available at ScienceDirect

Knowledge-Based Systems

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



A scheduling algorithm for heterogeneous computing systems by edge cover queue

Yu-meng Chen, Song-lin Liu, Yan-jun Chen, Xiang Ling *

National Key Laboratory of Science and Technology on Communications, University of Electronic Science and Technology of China, Chengdu, 611731, Sichuan, China

ARTICLE INFO

Article history:
Received 13 December 2022
Received in revised form 2 February 2023
Accepted 3 February 2023
Available online 10 February 2023

Keywords:
Heterogeneous computing system
Edge cover queue
Estimation of distribution algorithm
Graph random walk algorithm

ABSTRACT

In heterogeneous computing systems, excellent task scheduling algorithms can shorten the task completion time and improve system parallelism. With the large-scale deployment of edge computing, the task scheduling algorithm in heterogeneous edge computing servers has become a critical factor in improving the overall system performance. This paper proposes a new task scheduling algorithm called the edge cover scheduling algorithm (ECSA), which schedules tasks based on the edge cover queue of the directed acyclic graph (DAG) for heterogeneous computing systems. Based on the estimation of distribution algorithm (EDA) and the graph random walk algorithm, the ECSA generates an edge cover queue from DAG. Then, the ECSA uses the heuristics greedy method with low time and computational complexity to allocate the edge cover queue to processors. Theoretical analysis and simulation results on random DAGs and real-world DAGs show that the ECSA can achieve better scheduling results in terms of makespan, the schedule length ratio (SLR), efficiency, and frequency of best results with low time and computational complexity.

1. Introduction

A heterogeneous computing system (HCS) refers to a group of interconnected processors with different computing and storage capabilities [1–3]. Due to the diversity of tasks and the difference in processor architecture, HCS widely exists in various computing scenarios [1–3]. Applications are usually modeled as DAGs for scheduling in practice. An efficient DAG task scheduling algorithm can improve the heterogeneous computing system's performance and user experience quality. With a suitable scheduling algorithm, an excellent computing system can fully exert its computing capability [4–7]. Because of the heterogeneity of the HCS, the priority constraints between tasks, and the NP-H characteristics of DAGs, it is challenging to achieve efficient scheduling [8].

The existing scheduling algorithms are divided into list scheduling algorithms and evolutionary algorithms [3]. List scheduling algorithms have low time and computational complexity, but the scheduling results are not ideal in many scenarios, and they easily fall into the local optimal solution [9,10]. Evolutionary algorithms are global optimization algorithms [11,12]. Due to high time and computational complexity, it can obtain excellent scheduling results only with sufficient computation [11,12].

The list scheduling algorithms have two phases: the prioritizing phase for giving priority to each task and the processor selection phase for selecting a suitable processor that minimizes the heuristic greedy cost function [13,14]. The task scheduling queue is obtained by sorting the tasks according to the task priority weight. Early scheduling algorithms were designed for homogeneous computing systems, such as the earliest start time (EST) and earliest finish time (EFT) [1], which are the simplest greedy algorithms. To adapt to heterogeneous environments, two well-known heuristic methods, heterogeneous earliest finish time (HEFT) and critical path on a processor (CPOP), are proposed in [1]. In the HEFT algorithm, tasks are sorted by the length of the longest path from the task node to the bottom of the DAG. The task on top of the scheduling queue is greedily assigned to the processor, which allows for the earliest finish time. Furthermore, the HEFT algorithm uses an insertion policy that tries to insert a task in the earliest idle time between two already scheduled tasks on a processor if the slot is large enough to accommodate the task [1]. In the CPOP algorithm, the critical path of the DAG is calculated first, and then the critical path is allocated to the same processor that has the shortest completion time, and the other tasks are allocated in turn [1]. The predictive list scheduling algorithm performs scheduling by estimating the impact of task allocation on its subsequent tasks. The most famous predictive list scheduling algorithm, predict earliest finish time (PEFT), constructs an optimistic cost table (OCT) by listing the shortest path from its child node to the exit node for each combination of tasks and processors [2]. PEFT uses the average OCT value of each task to sort tasks from high to low. The task is

* Corresponding author.
E-mail address: xiangling@uestc.edu.cn (X. Ling).

https://doi.org/10.1016/j.knosys.2023.110369

assigned to the processor with the shortest EFT+OCT or inserting the available idle time slot of the processor [2]. The lookahead scheduling algorithm (LO) is a special list scheduling algorithm. When scheduling, each task is assigned to the processor that has the shortest completion time for all its subtasks [10]. It has a good effect in medium-scale DAG scheduling, Gaussian elimination (GE) programs, and Fast Fourier Transform (FFT) programs, but it is also a type of list scheduling algorithm with the highest time and computational complexity [10].

Evolutionary algorithms are global optimization algorithms that can provide satisfactory solutions to complex problems [12,15]. For the DAG scheduling problem (DAG-SP), evolutionary algorithms have been attempted, including the genetic algorithm (GA) [9], ant colony optimization (ACO) [16], moth search algorithm (MSA) [17], differential evolution (DE) [17], chemical reaction optimization (CRO) [18], and particle swarm optimization (PSO) [19,20]. In addition to giving the task-processor scheduling scheme, the evolutionary algorithm for the priority of the scheduling queue also has research work. In the multiple priority queues genetic algorithm (MPQGA), an evolutionary algorithm is proposed to guide the iteration of the scheduling queue to solve DAG-SP [9]. After initialization, crossover and mutation operators evolve the scheduling queue. Better scheduling results can be obtained by obtaining a better scheduling queue [9]. However, the evolutionary algorithm's large number of iterations also limits its application scope.

In recent research, the scheduling algorithm collects and memorizes the information from the DAG-SP and further explores better scheduling within the acceptable time and computational complexity to achieve a better solution [21,22]. For this point, the estimation of distribution algorithm (EDA) is suitable. The EDA establishes a probability model and extracts information in the iterative process to guide the algorithm to explore new scheduling solutions further [23]. To date, the EDA has performed well in combinatorial optimization problems, including scheduling problems [21–27]. However, the EDA has high time and computational complexity similar to the evolutionary algorithm.

In this paper, to improve the current DAG-SP scheduling results with less complexity and fewer iterations, we present a new task scheduling algorithm called ECSA. Unlike previous work using a vertex queue in the prioritizing phase and processor selection phase, the ECSA uses an edge cover queue for task scheduling. Furthermore, we propose a method using a simplified EDA and graph random walk algorithm to generate the edge cover queue. The tasks in an edge as a whole use the heuristic greedy cost function for processors' task assignments. Theoretical analysis and experimental results show that the ECSA can effectively improve the performance of task scheduling solutions and be closer to the optimal solution in fewer iterations and with low time complexity.

The major contributions of this paper are as follows:

1. This paper first proposes the idea of using an edge cover queue instead of a vertex queue for scheduling. In the scheduling process, the edge cover queue explores a hybrid scheduling strategy of vertex greed and edge greed.
2. Based on the EDA and the graph random walk algorithm, a method of generating an edge cover queue is designed. We adopted the strategy of a single population to simplify the EDA, avoiding the significant time and computational complexity of the EDA.

The content of this paper is organized as follows. Section 2 describes the relevant theoretical basis and preliminaries of the article. Section 3 introduces the relevant work. Section 4 introduces the proposed algorithm and analyzes its advantages. Section 5 introduces the experimental results and discusses them. Section 6 summarizes the article and introduces the content that can be expanded upon in the future.

2. Theoretical basis and preliminaries

In this section, we first introduce both the hardware and software DAG-SP system model and related definitions. Then, we introduce the EDA and graph random walk algorithm, which will be used later.

2.1. DAG-SP

DAG-SP means DAG scheduling problem. An application program can be modeled as a directed acyclic graph, $DAG = (V, E)$, where V is the set of v nodes and each node $n_i \in V$ represents an independent application task that must execute on the same processor. E is the set of edges between tasks. Each $e_{ij} \in E$ represents the task-dependence constraint such that task n_j can be started after task n_i completes its computation. The DAG is complemented by a matrix W, which is a $v \times p$ computation cost matrix, where v is the number of tasks and p is the number of processors in the system. Each $\omega_{ij} \in W$ means the computation cost for task n_i on processor p_j. A matrix C consists of c_{ij} called the communication cost matrix. c_{ij} is the communication cost for the corresponding edge e_{ij}. Note that when tasks n_i and n_j are assigned to the same processor, the actual communication cost c_{ij} can be considered to be zero because it is negligible compared with inter-processor communication costs.

In most models, processors connect in a fully connected topology [1–3]. Moreover, the execution of tasks and communications with other processors can be achieved for each processor without contention. Additionally, the execution of any task is considered non-preemptive. These models are standard in this scheduling problem because these simplifications correspond to natural systems [2].

Then, some terms related to DAG are introduced to characterize the topology of DAG. $pred(n_i)$ refers to the set of immediate predecessors of task n_i. $succ(n_i)$ refers to the set of immediate successors of task n_i. n_{entry} is a task with no immediate predecessors. If a DAG has more than one n_{entry}, a dummy entry node without influence on actual computation will be added to the graph. n_{exit} is a task with no immediate successors. Similarly, if a DAG has more than one n_{exit}, a dummy entry node will also be added to the graph. *makespan* is the maximum finish time of n_{exit}, so *makespan* is the scheduling length of the application program represented by the DAG.

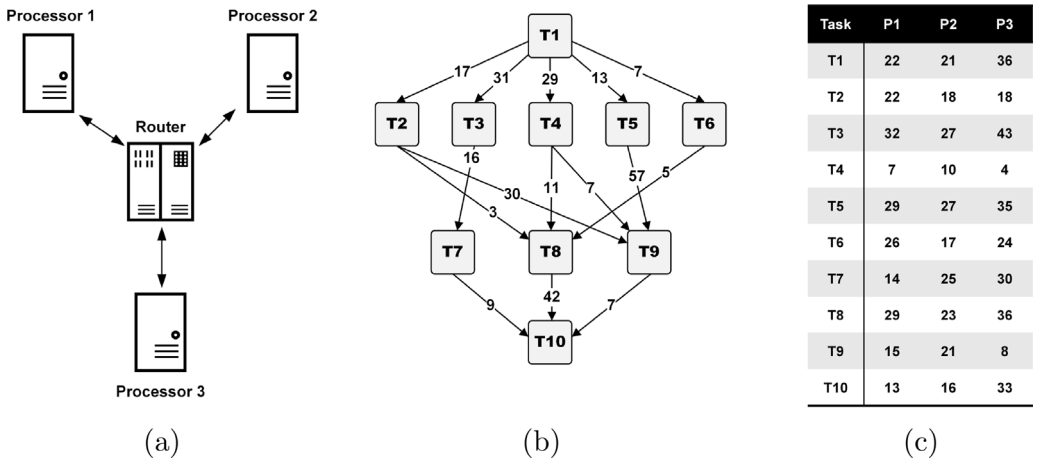
The critical path (CP) of a DAG is the longest path from the entry node to the exit node in the graph. The lower bound of *makespan* is the minimum critical path length (CP_{min}), which is computed by considering the minimum computational costs of each node in the critical path. CCR is the communication-to-computation ratio that is designed to measure the impact of communication latency on computing performance. For an application program, high CCR refers to communication-intensive, and low CCR refers to computation-intensive. CCR can be computed by Eq. (1) as follows:

$$CCR = \frac{\sum_{ij} c_{ij}/|E|}{\sum_i \bar{\omega}_i/|V|} \tag{1}$$

$\bar{\omega}_i$ denotes the average computation cost which can be computed by Eq. (2) as follows:

$$\bar{\omega}_i = (\sum_{j \in P} \omega_{ij})/p \tag{2}$$

The range percentage of computation costs β on processors refers to the heterogeneity factor for processor speeds. A high β value implies higher heterogeneity and different computation



Task	P1	P2	P3
T1	22	21	36
T2	22	18	18
T3	32	27	43
T4	7	10	4
T5	29	27	35
T6	26	17	24
T7	14	25	30
T8	29	23	36
T9	15	21	8
T10	13	16	33

(a) (b) (c)

Fig. 1. DAG-SP (a) a HCS with three processors (b) a DAG with ten tasks (c) computation cost matrix W.

costs among processors. The computation cost ω_{ij} of each task n_i on each processor p_j is randomly set from the range of Eq. (3):

$$\bar{\omega}_i(1-\beta/2) \le \omega_{ij} \le \bar{\omega}_i(1+\beta/2) \tag{3}$$

Fig. 1(a) shows a fully connected computing system with three heterogeneous processors. The DAG task diagram shown in Fig. 1(b) must be assigned to the HCS. The computation time of every task on different processors is different in the computation cost matrix W in Fig. 1(c) [2].

2.2. Scheduling queue

In the list scheduling algorithm [1–3] or evolutionary algorithms [9], it is necessary to generate a scheduling queue. Different algorithms will give different scheduling orders and obtain different vertex scheduling queues. For example, in PEFT, vertex scheduling queues are sorted according to the average OCT value.

Definition 1 (*Topology Feasible Queue*)**.** A topology feasible queue means that sequential scheduling will not result in violating the topology of the DAG, that is, the succ of every task is behind them in the topology feasible queue.

Definition 2 (*Edge Cover Queue*)**.** An edge cover queue is a queue that arranges edge cover in topological order. In graph theory, an edge cover is a subset of the edges of a graph, so that each node in the graph is associated with an edge in the subset of edges. The queue of all edges in the DAG graph arranged in topological order is called the max edge cover queue.

Taking Fig. 1(b) as an example, the number of edges is fifteen, and the max edge cover queue is [(1,2), (1,3), (1,4), (1,5), (1,6), (3,7), (2,8), (2,9), (4,8), (4,9), (6,8), (5,9), (7,10), (8,10), (9,10)]. An example of an edge cover queue is [(1,2), (1,3), (1,4), (1,5), (1,6), (3,7), (2,8), (2,9), (9,10)].

2.3. Estimation of distribution algorithm

The main idea of the EDA is to combine the evolutionary algorithm with the constructive mathematical analysis method to guide the effective search of the problem space [24–26]. Essentially, the EDA is an algorithm based on a probability model. It guides the following search range of the algorithm by updating the probability model from the currently found excellent individuals and generates new individuals from the probability distribution function of the obtained better solution for iteration. Combining genetic algorithms with statistical learning allows search iteration to be completed faster.

2.4. Graph random walk algorithm

The graph random walk algorithm traverses a graph from one or a series of vertices. At any vertex, the traverser of the random walk will walk to the neighbor vertex of this vertex with probability $(1-a)$ and jump to a non-neighbor vertex in the graph with probability a. After each walk, a probability distribution is obtained. The probability distribution depicts the probability that each vertex in the graph is visited. This probability distribution is used as the input of the next walk, and the process is iterated repeatedly. Over time, the probability distribution will tend to converge.

3. Related work

In this section, we present a concise survey of frequently used task scheduling algorithms including list scheduling algorithms and the EDA. We introduce the main ideas of these algorithms. Over the past few years, research on DAG-SP has focused on finding suboptimal solutions to obtain a good solution in an acceptable time. List scheduling algorithms can generate high-quality suboptimal solutions at a reasonable cost without a high processor workload.

3.1. List scheduling algorithm

Research on the list scheduling algorithm for DAGs focuses on obtaining an acceptable solution in a short time. Compared with the evolutionary algorithm, the list scheduling algorithm has a low time complexity [1–3]. Therefore, this kind of algorithm is widely used in various HCSs. Here, we introduce several common list scheduling algorithms, most importantly HEFT and PEFT.

3.1.1. Earliest finish time

The EFT algorithm has two phases: a task prioritizing phase and a processor selection phase [2]. In the first phase, task priorities are defined as $rank_u$, as shown in Eq. (4).

$$rank_u(n_i) = \bar{\omega}_i + \max_{n_j \in succ(n_i)} (\bar{c}_{ij} + rank_u(n_j)) \tag{4}$$

$rank_u$ represents the length of the longest path from task n_i to the exit node, including the computational cost of n_i. The task list is ordered by decreasing value of $rank_u$.

In the processor selection phase, the task on top of the list is assigned to the processor p_j that allows for the earliest finish time of task n_i: $p_{n_i} = \arg\min_{p_j}(EFT(n_i, p_j))$. $EFT(n_i, p_j)$ denotes the finish time of task n_i on processor p_j.

3.1.2. Heterogeneous earliest finish time

The HEFT algorithm is highly competitive and widely used since it was proposed [1]. In the task prioritizing phase, it is the same as the EFT algorithm with $rank_u$. In the processor selection phase, the task on top of the task list is assigned to processor p_j, which allows for the EFT of task n_i. However, the HEFT algorithm uses an insertion policy that tries to insert a task in the earliest idle time between two already scheduled tasks on a processor if the slot is large enough to accommodate the task.

3.1.3. Predict earliest finish time

The novelty of the PEFT algorithm is its ability to forecast by computing OCT [2]. The OCT is a matrix in which the rows indicate the number of tasks and the columns indicate the number of processors, where each element $OCT(n_i, p_j)$ indicates the maximum of the shortest paths of n_i children tasks to the exit node considering that processor p_j is selected for task n_i.

In the task prioritizing phase, PEFT computes the average OCT for each task that is defined by Eq. (5) as follows:

$$rank_{OCT}(n_i) = \frac{\sum_{j=1}^{p} OCT(n_i, p_j)}{p} \tag{5}$$

In the processor selection phase, with an insertion policy, the processor that gives the minimum O_{EFT} for a task is selected to execute that task. $O_{EFT}(n_i, p_j)$ equals $EFT(n_i, p_j) + OCT(n_i, p_j)$.

In this way, PEFT is looking forward in the processor selection phase. Perhaps PEFT is not selecting the processor that achieves the EFT for the current task, but it expects to achieve a shorter finish time for the tasks in the next steps. In general, PEFT is still an effective heuristic greedy algorithm. Unlike EFT or HEFT, it is not greedy for time but greedy for O_{EFT}.

3.1.4. Lookahead scheduling algorithm

The LO algorithm is based on the HEFT algorithm, whose main feature is its processor selection policy [10]. To select a processor for the current task, it iterates over all available processors and computes the *EFT* for the child tasks on all processors. The processor selected for the current task is the one that minimizes the maximum *EFT* from all children of it. This procedure can be repeated for each child of task by increasing the number of levels analyzed. However, the LO algorithm has high time and computational complexity, even if only one level of children's tasks is considered. The LO algorithm performs well in some DAGs with regular structures, such as FFT and GE.

3.2. Estimation of distribution algorithm

The EDA is an appropriate algorithm to solve scheduling problems since its probability model can represent the precedence constraints between tasks and simulate the distribution of search space [24–26].

In multi-objective flexible job-shop scheduling problem with sequence-dependent setups problem, to minimize the completion time and installation cost, the relevant work uses the EDA to guide the machine learning strategy, learn valuable information from the nondominated solution of the main population, and establish a probability model to obtain better scheduling [26]. In resource scheduling for energy-efficient fog-enhanced Internet of Things, some papers use EDA and partition operators to partition the graph to determine task processing arrangement and processor allocation. It has similar performance to other algorithms in terms of fog node energy consumption and is superior to other algorithms in terms of maximum completion time [21,22]. Furthermore, there are articles that apply some mathematical skills to speed up the EDA. The path relinking enhanced in the control field is used to speed up the EDA iterative solution process [3]. The scheduling solution obtained is better than the list scheduling algorithm in terms of completion time.

4. The proposed algorithm

In this section, we introduce a new scheduling algorithm for HCS, called the ECSA, which aims to achieve a better solution within acceptable time and computational complexity. The ECSA uses a designed structure's edge cover queue for task scheduling. The ECSA mainly includes two programs: Init-S and Iteration-of-S. Init-S completes the heuristic generation of the initial edge cover queue and probability model matrix S. Iteration-of-S completes the iteration of S and the edge cover queue. The Init-S and Iteration-of-S programs use more basic methods, including the generation for edge cover queue, scheduling for edge cover queue, graph random walk algorithm, and simplified EDA. This section starts with introducing these basic methods in Sections 4.1, 4.2, 4.3, and 4.4, then two programs based on these basic methods are constructed in Sections 4.5 and 4.6, and the ECSA is constructed in Section 4.7. Finally, we introduce the overall process and theoretical advantages of the algorithm.

4.1. Generation for edge cover queue

In a DAG, the number of edges is generally considered to be at the n^2 level, so the edge cover queue has a huge number of permutations and combinations. Generating the edge cover queue through traversal will lead to massive time and computational complexity and may not have a good scheduling performance. This section introduces how to generate an edge cover queue based on a vertex queue and vertex sets with low time and computational complexity. Specifically, it includes two steps, the vertex-to-line step and loss-continuity step. Below, we will elaborate on this processing.

In the vertex-to-line step, the algorithm converts the input vertex queue π and vertex set V_1 to an edge cover queue. The vertices in vertex queue π are successively converted into corresponding edges under the guidance of vertex set V_1, and an edge cover queue $line_1$ is obtained. In the loss-continuity step, the algorithm deletes some edges of $line_1$ under the guidance of vertex set V_2 on the premise that the edge cover queue without topology errors. The pseudo code is as follows in Method 1. It is evident that the scheduling result of $line_1$ generated by the vertex-to-line step is consistent with the vertex queue π. The scheduling result of $line_2$ may be different from the original vertex queue. How to obtain vertex set V_1 and vertex set V_2 will be introduced in Sections 4.3 and 4.4.

Taking the DAG in Fig. 1(b) as an example, if the vertex queue is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], the vertex sets V_1 and V_2 are vertex 1. Vertices 0 and 11 are dummy entry nodes without influence on the actual computation introduced in Section 2. The edge cover queue $line_1$ obtained by the vertex-to-line step is [(0,1), (1,2), (1,3), (1,4), (1,5), (1,6), (3,7), (4,8), (5,9), (8,10), (10,11)]. The output edge cover queue $line_2$ obtained by the loss-continuity step is [(1,2), (1,3), (1,4), (1,5), (1,6), (3,7), (4,8), (5,9), (8,10), (10,11)].

4.2. Scheduling for edge cover queue

Unlike the vertex task queue, the edge cover queue takes the edge as the minimum unit in the scheduling process. In e_{ij}, if task n_i has been scheduled, task n_j will be scheduled according to a greedy cost function. Under this circumstance, task n_j is vertex greedy scheduled. We call task n_j a continuous vertex. If task n_i has not been scheduled, task n_i and task n_j will be scheduled together to minimize the overall greedy cost function. Under this circumstance, task n_i and task n_j are edge greedy scheduled. We call task n_i and task n_j an un-continuous vertex. *EFT* and O_{EFT} with

Method 1 Generate edge cover queue

Input: vertex task queue π, vertex set V_1, V_2

vertex-to-line step:
for each task π_i in π **do**
  if $pred(\pi_i) \cap V_1 = Y \neq \emptyset$ **then**
    for each task Y_j in Y **do**
      Put (Y_j, π_i) in edge cover queue $line_1$
    end for
  else
    Put one of $(pred(\pi_i), \pi_i)$ in $line_1$
  end if
end for
loss-continuity step:
for each task n_i in V_2 **do**
  Delete $(pred(n_i), n_i)$ in $line_1$
  if $line_1$ get topology error **then**
    Cancel deletion
  end if
end for

Output: $line_2$

Method 2 Schedule edge cover queue

Input: edge cover queue $line_3$, *label*

if label=0 **then**
  for each edge $(pred(n_i), n_i)$ in $line_3$ **do**
    if task $pred(n_i)$ has been scheduled **then**
      Schedule n_i by O_{EFT} with insert policy
    else
      Schedule $pred(n_i)$ and n_i with minimum O_{EFT} for n_i with insert policy
    end if
  end for
end if
if label=1 **then**
  for each edge $(pred(n_i), n_i)$ in $line_3$ **do**
    if task $pred(n_i)$ has been scheduled **then**
      Schedule n_i by *EFT* with insert policy
    else
      Schedule $pred(n_i)$ and n_i with *EFT* for n_i with insert policy
    end if
  end for
end if

Output: *makespan* of $line_3$

the insert policy are chosen as greedy methods. The pseudo code is as follows in Method 2.

In this method, *label* indicates which greedy cost function to be used. Edge cover queue $line_3$ is obtained from the after method. As mentioned earlier, each max edge cover queue may be very long. However, the scheduling of the edge cover queue $line_2$ relative to the vertex queue does not have additional redundancy. This means that the edge cover queue's scheduling process has a similar time complexity as the scheduling process of the vertex queue.

4.3. Graph random walk algorithm

In the ECSA, the graph random walk algorithm generates vertex set V_1 under the guidance of the imaginary part of the diagonal element of probability model matrix S. How to obtain complex matrix S will be introduced in Section 4.5. In the graph random walk algorithm, first, $random(0, 1/n)$, the vertex whose imaginary part of the diagonal element $Im(S_{ii})$ is larger than this value, is put into vertex set V_1. The imaginary part of the diagonal element is updated according to the scheduling result Δt. The specific pseudo code is as follows in Method 3.

Method 3 Graph random walk

Input: complex matrix S, Δt

random step:
e=Random(0,1/n)
if $Im(S_{ii}) \geqslant e$ **then**
  Put task i to V_1
end if
update step:
for each task n_i in V_1 **do**
  $Im(S_{n_i,n_i}) = kIm(S_{n_i,n_i}) + (1-k)\Delta t$
end for
$Normalization(Im(S_{ii}))$

Output: V_1

4.4. Local EDA in the ECSA

EDA establishes a probability model to simulate the distribution of the solution space. The standard EDA steps are as follows: First, the probability model matrix S is established. The population in the subsequent iteration is generated by S. Then, under the guidance of S, a new population is generated, and S is updated according to the results of the new population. The above steps are repeated until the stop criteria are met [24–26]. However, when the standard EDA is used to generate a vertex queue in the scheduling algorithm, it leads to a high time complexity $O(GNn^3)$, where n is the total number of tasks, N is the size of the population, and G is the number of iterations [3].

In some research, it is feasible to simplify the EDA to improve the solution speed according to the scenario requirements [28]. In the ECSA, to reduce complexity, we use a single individual instead of a population to update S and generate new solutions. The updated formula of S is also changed to adapt to this change, and the specific form will be introduced later. The edges $(n_i, succ(n_i))$ with the same front vertex (n_i) in the edge cover queue have high parallelism between each other. Then, the real part between $succ(n_i)$ in the matrix S is updated by taking advantage of the influence of the topological structure change of these edges. In this way, the EDA plays a role only on local edges with the same front vertex, so we call this the local EDA in the ECSA. A single individual updating strategy and local EDA strategy will significantly reduce the complexity of the EDA in the ECSA. In this section, we divide the local EDA in the ECSA into the EDA-sample step, EDA-produce step and EDA-update step, for a total of three steps.

The EDA-produce step uses EDA to obtain a subset of the input vertex set V_1. The output vertex subset V_2 guides the loss-continuity step in Method 1 to generate un-continuous vertices in the edge cover queue.

The EDA-sample step changes topology between $(n_i, succ(n_i))$ in $line_2$ guided by V_1, S. To produce individuals, a sampling method is designed for the relative position probability model between $(n_i, succ(n_i))$. For task n_i, find all the positions of topological sorting $(n_i, succ(n_i))$ and place them in a backfill queue B. The order of the edges in B is determined by the roulette method with reference to the probability distribution value between the corresponding real part of $succ(n_i)$ in the matrix S. Then $(n_i, succ(n_i))$ is backfilled to $line_2$ in order. Because the EDA is performed only for the topological relationship of $(n_i, succ(n_i))$, the backfilling process only needs to check the topology of the partial edge cover

Method 4 local-EDA

Input: vertex set V_1, matrix S, edge cover queue $line_2$, Δt

EDA-produce step:
for each task n_i in V_1 **do**
    Put n_i to V_2 with roulette method by $Re(S_{n_i,n_i})$
end for
EDA-sample step:
for each task n_i in V_1 **do**
    Put all $(n_i, succ(n_i))$ in backfill queue B and topological sort B with roulette method by S
end for
for each edge $(n_i, succ(n_i))$ in B **do**
    Backfill $(n_i, succ(n_i))$ to $line_2$ in order
    if topology error in edge cover queue **then**
        Place $(n_i, succ(n_i))$ at the top of B
    end if
end for
if B not empty **then**
    Put them back, re-backfilling
end if
Get edge cover queue $line_3$
EDA-update step:
for each task n_i in V_1 **do**
    Update S_{n_i,n_i} by Eq. (6)
    for each task in $succ(n_i)$ **do**
        if task i is before task j **then**
            Update S_{ij} by Eq. (6)
            Update S_{ji} by Eq. (7)
        end if
    end for
end for

Output: vertex set V_2, edge cover queue $line_3$, matrix S

queue containing $(n_i, succ(n_i))$. Suppose a topology error occurs after backfilling, placing it at the beginning of B and selecting the next edge. Using the described method, the permutations produced by the EDA probability model can be guaranteed to be valid and satisfy the topological order of the DAG. If B is not empty after backfilling, ECSA backfills the remaining edges in B to the original position and then backfills the other edges. Finally, the EDA-sample step outputs the edge cover queue $line_3$.

The EDA-update step updates S according to the edge cover queue $line_3$ and vertex set V_1, V_2. In ECSA, S is designed as a $n \times n$ matrix to reduce the complexity of operation and storage. In the edge cover queue, the existing topological relationship between two vertices in the directed edge makes it impossible to update the probability model matrix S directly by using the edge topological relationship. Thanks to the idea of local in the EDA-sample step, we only need to use the topological relationship between $(n_i, succ(n_i))$ to update the real part between $succ(n_i)$ in S. Furthermore, the probability model matrix S also avoids being designed as a vast $n^2 \times n^2$ matrix.

Moreover, the EDA-sample step generates a single individual. We need to introduce a weighting term into the updated formula to adapt to this change. In ECSA, the weighted item is designed as the difference between the scheduling length *makespan* of the old and new queues: Δt. If task i is before task j in $succ(n_i)$, the off-diagonal element $Re(S_{ij})$ and $Re(S_{ji})$ update formulas are as shown in Eqs. (6) and (7).

$$Re(S_{ij}) = kRe(S_{ij}) + (1-k)\Delta t \tag{6}$$

$$Re(S_{ji}) = 1 - Re(S_{ij}) \tag{7}$$

Program 1 Init-S

Schedule DAG with HEFT and PEFT
if $makespan_{HEFT} \leqslant makespan_{PEFT}$ **then**
    $\pi = \pi_{HEFT}, t = makespan_{HEFT}, label = 1$
else
    $\pi = \pi_{PEFT}, t = makespan_{PEFT}, label = 0$
end if
for each task n_i in V **do**
    Calculate PL of n_i, put max to $\lfloor\sqrt{n}\rfloor$ PL of n_i to PLS
end for
for i=1:n **do**
    for j=1:n **do**
        if i=j **then** $Re(S_{ij}) = 0.5$, $Im(S_{ij}) = 1/n$
        else
            if $P_{ij} = 0$ **then** $Re(S_{ij}) = 0.5$
            else
                $Re(S_{ij}) = 0$
            end if
        end if
    end for
end for
for each task n_i in PLS **do**
    V_1=V_2=n_i
    Get $line_2$ by Method 1
    Get $line_3$ by EDA-sample step and schedule $line_3$
    Calculate $\Delta t = t - t_{line_3}$
    Update S by EDA-update step and update step in Method 3
end for

4.5. Heuristic updating of S

In this section, we design the initialization and heuristic update method of the probability model matrix S. HEFT and PEFT are the most commonly used task scheduling algorithms due to their simplicity and performance [1,2]. Therefore, ECSA will heuristically update S based on HEFT and PEFT.

First, HEFT and PEFT are performed on DAG to obtain the vertex queue of the HEFT, PEFT, OCT table and the label of which heuristic greedy cost function is better. Then, the prediction lookahead value (PL) of tasks is sorted. PL is calculated by $PL = \sum_{n_j \in succ(n_i)} \sigma(\omega_j)$. $\sigma(\omega_j)$ is variance. The PL is considered as the potential of a vertex. Selecting vertices by decreasing the value of PL total $\lfloor\sqrt{n}\rfloor$ nodes to form vertex set PLS.

Next, the ECSA initializes the complex matrix S and assigns values to the elements in matrix S. The real and imaginary parts of the matrix S are given initial values under the guidance of the matrix P, which is the topological matrix of the DAG graph. The initialized matrix S is an equal probability matrix. The assignment method can be found in the pseudo code.

It is worth remembering that the real part of the matrix S is used for the EDA, and the imaginary part is used for the graph random walk algorithm. In this part of the program, the algorithm puts the first vertex in PLS into vertex set V_1, V_2, and then moves it out from the top of the PLS. V_1 and a better vertex queue from HEFT or PEFT are transferred to Method 1 to obtain $line_2$. The EDA-sample step further samples $line_2$ with probability matrix S and V_1. Calling the EDA-update step and graph random walk update algorithm update S. Thus far, the initialization of the matrix S has been completed. The specific pseudo code is as shown in Program 1.

4.6. Iteration of S

In the previous Section 4.5, we used the heuristic queue *PLS* to heuristically update the probability matrix. This section introduces the iterative process of generating a new edge cover queue guided by S. Using the graph random walk algorithm, $random(0, 1/(n))$, the vertices whose imaginary part of the diagonal is greater than this value are placed in V_1. If the selected vertices can form an edge, the vertex with the larger *PL* value is selected. The length of V_1 shall not exceed $\lfloor\sqrt{n}\rfloor$. The following steps are similar to the Init-S algorithm, except that V_1 replaces a single vertex in the *PLS*, iterating $\lfloor\sqrt{n}\rfloor$ times and outputting the last edge coverage queue, that is, the ECSA output. The specific steps are as shown in Program 2.

Program 2 Iteration-of-S

1: **for** num=1:$\lfloor\sqrt{n}\rfloor$ **do**
2: $Num = num + +$
3: Get V_1 by Graph random walk
4: Get V_2 by EDA-produce step
5: Get $Line_1$ by vertex-to-line step
6: Get $Line_2$ by loss-continuity step
7: Get $Line_3$ by EDA-sample step and schedule $line_3$
8: Calculate $\Delta t = t - t_{line_3}$
9: Update S by EDA-update step and update step in Method 3
10: **end for**

4.7. ECSA

Program 3 ECSA

1: Convert E to 01-matrix A
2: Compute longest path s in A
3: Compute $P = A^1 + A^2 + \cdots + A^s$
4: Init-S
5: Iteration-of-S

With the knowledge of graph theory, the power of the adjacency matrix of a graph contains the path information of the graph. ECSA can quickly obtain the topological matrix P by using this property. First, ECSA converts E to 01-matrix A and finds the longest path s in matrix A through dynamic programming. Then,topological matrix P calculate by $P = A^1 + A^2 + \cdots + A^s$. $P_{ij} > 0$ that is, task i is the pre-topological node of task j. $P_{ij} = 0$, that is, task i and task j can be exchanged freely without topological relationship. Afterward, the ECSA uses the Program 1 to obtain the heuristic matrix S. Finally, the ECSA, as shown in Program 3 calls the Program 2 to iterate and output the final result.

4.8. Analysis for ECSA

4.8.1. Time complexity

The time complexity of our proposed algorithm is analyzed as follows. For Method 1, the generation for the edge cover queue is $O(n^2)$. For Method 2, scheduling for the edge cover queue method is near $O(n^2p)$. In the EDA-sample step, not all vertices are involved in sampling. Second, the EDA-sample step is sampling the topological relationship between $(n_i, succ(n_i))$ with the same front vertex. Moreover, the use of a single population further reduces the time complexity. Therefore, the EDA-sample step has a complexity near $O(n^{2.5})$, and other operators in the local-EDA are $O(n^2)$ instead of the total $O(GNn^3)$ in the standard EDA.

For program Init-S, the time complexity comes from single vertex sampling and scheduling of the edge cover with $\lfloor\sqrt{n}\rfloor$ times iterations. The total time complexity is $O(n^{2.5} + n^{2.5}p)$.

For program Iteration-of-S, the calculation for the topological matrix P is $O(n^2)$. In one iteration, the time complexity is $O(n^{2.5} + n^2p)$. There are $\lfloor\sqrt{n}\rfloor$ times iterations as well. Thus, the complexity of the proposed ECSA is $O(n^3 + n^{2.5}p)$. This is an acceptable time complexity compared to $HEFT(n^2p)$, $PEFT(n^2p)$, and $LO(n^4p^3)$.

4.8.2. Summary

The ECSA proposes using an edge cover queue for scheduling and proposes a method to generate an edge cover queue. It can be seen from the generation process of the edge cover queue that it is still similar in length to the initial vertex queue, avoiding additional redundancy. In addition, the edge cover queue can include the scheduling results of the vertex queue and reasonably explore more solution spaces on the basis of the excellent vertex queue to jump out of the greedy trap.

Furthermore, in the edge cover queue, if n_i in e_{ij} is not scheduled, the directed edge will be scheduled as a whole. Therefore, the loss-continuity step brings local greed to the scheduling process. For the ECSA, compared with the vertex queue, the final edge cover queue is a mixture of vertex greed and local greed in scheduling.

The ECSA is a memory algorithm. The complex matrix S guides the EDA and graph random walk algorithm to generate the edge coverage queue. The single individual updating strategy and the local EDA strategy not only greatly reduce the complexity of the EDA but also make the ECSA have fewer iterations. Compared with the exponential iteration number of the genetic algorithm, the ECSA iteration number is lower, only $2\lfloor\sqrt{n}\rfloor$ times.

Taking the DAG in Fig. 1 as an example, the edge cover queue given by ECSA is [(1,3), (1,2), (1,4), (1,6), (1,5), (4,8), (3,7), (5,9), (8,10)]. As shown in Fig. 2, compared with the PEFT, LO, and HEFT algorithms, the ECSA has a shorter scheduling length.

5. Experimental results

In this section, we compare the performance of the ECSA with that of the list scheduling algorithms presented above. We consider random DAGs and graphs that represent some real-world applications. We first introduce the evaluation metrics used for the scheduling algorithm. Then we show the performance of different algorithms for different types of DAGs.

5.1. Scheduling algorithm evaluation metrics

The evaluation metrics for the scheduling algorithm include *makespan*, efficiency, the ratio of better solutions and the scheduling length ratio(SLR). It is worth mentioning that the SLR is a parameter that considers the lower bound of the algorithm. It reflects the improvement by the algorithm better than *makespan*. Efficiency can reflect the improvement of HCS parallelism by the algorithm.

Makespan as shown in Eq. (8) is the most intuitive parameter. The smaller *makespan* is, the faster the execution of programs on the HCS is.

$$makespan = maxFT(v_{exit}) \tag{8}$$

Efficiency is defined as $Speedup/m$. *Speedup* as shown in Eq. (9) is defined as the ratio of the sequential execution time to the *makespan* of the algorithm. The calculation of sequential execution time assigns all tasks to a single processor to minimize the calculation time of DAG.

$$Speedup = \frac{min_{p_j \in P}[\sum_{n_i \in V} \omega_{ij}]}{makespan} \tag{9}$$

The ratio of better solutions is well understood. This comparison is presented as a pairwise table, where the percentage of

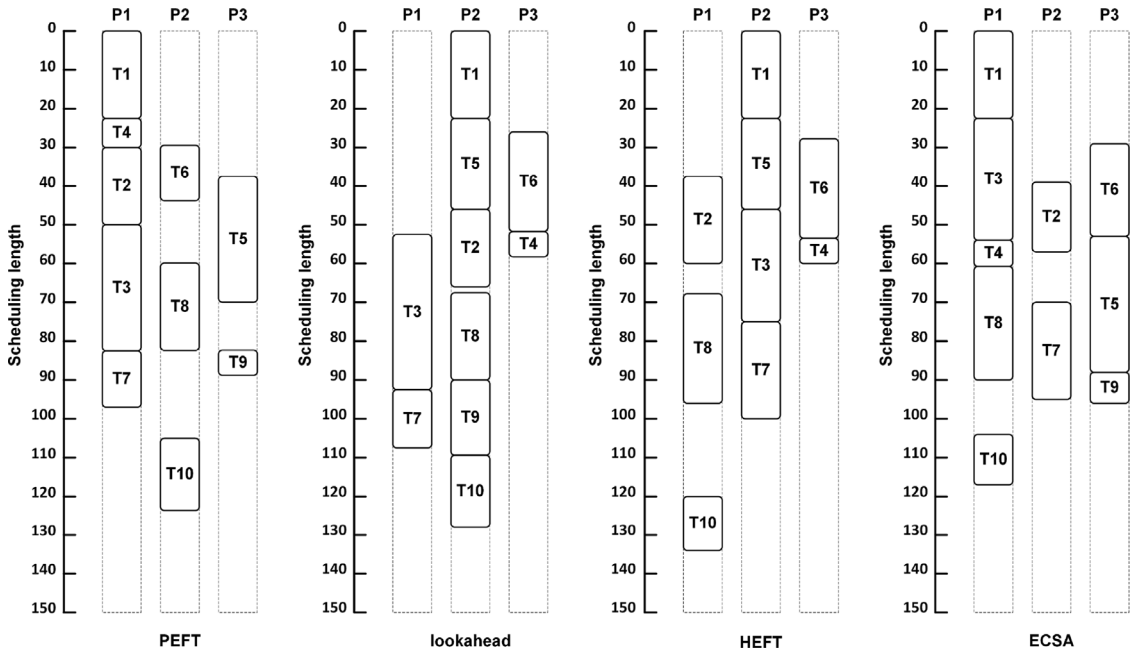


Fig. 2. ECSA and other algorithm (PEFT, lookahead and HEFT) scheduling results.

better, equal, and worse solutions produced by ECSA is compared to other algorithms.

The scheduling length ratio is the ratio of the scheduling solution to an impossible lower bound. Here, we want to use a metric that compares DAGs with very different topologies. The metric most commonly used to do so is the normalized schedule length (NSL). It is also called SLR. For a given DAG, both represent the makespan normalized to the lower bound. SLR is defined in Eq. (10) as follows.

$$SLR = \frac{makespan}{\sum_{n_i \in CP_{MIN}} min_{p_j \in P}[\omega_{ij}]} \tag{10}$$

The denominator in SLR is the minimum computation cost of the critical path tasks (CP_{min}). There is no *makespan* less than the denominator of the *SLR* equation. Therefore, the algorithm with the lowest *SLR* is the best algorithm.

5.2. Random DAGs

To evaluate the relative performance of the ECSA, we randomly generated DAGs. For this purpose, we used a DAG generation program TGFF [29]. The TGFF, as a general DAG generation tool, ensures the universality of the original data. For TGFF, there are a number of parameters relevant to the task graph structure: the average (x) and multiplier (y) for the lower bound on the number of nodes in a graph and the maximum in-degree (id) and out-degree (od) of graph nodes. A value for the lower bound of nodes is selected at random from the uniform range $[x - y, x + y]$. The upper bound of id and od for a DAG will relate to α. Every node will obtain id and od randomly from the uniform range $[0, \alpha\lfloor\sqrt{n}\rfloor]$. For random DAG generation, we considered the following parameters:

$n = [20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500]$

$\alpha = [0.4, 0.6, 0.8, 1]$

$CCR = [0.1, 0.5, 1, 5]$

$p = [2, 4, 8, 16]$

$\beta = [0.8, 1]$

Generally, we think that 20 nodes' DAGs are small-scale and 500 nodes are large-scale, so we set n from 20 to 500. α determines the upper bound of the in-degree and out-degree and further affects the number of DAG layers. Therefore, DAGs with large α look fat and DAGs with small α look thin, so we set the above α to make the test data evenly contain thin and fat DAGs. The *CCR* of 0.1 is very low communication traffic and 5.0 is very high. Therefore, we set *CCR* from 0.1 to 5 so that the test data will gradually change from computation-intensive to communication-intensive. In the HCS, the number of computing cores varies greatly. Those with strong performance have 16 or more cores, while those with low performance have fewer cores (2-4) due to power consumption and cost considerations. Therefore, we choose 2 to 16 processors. Because our algorithm is for the HCS, our default system has strong heterogeneity. Therefore, we select β as 0.8 and 1 to represent medium and high heterogeneity, respectively. These combinations produce 1920 groups of different DAGs. In each DAG, 20 different random graphs were generated for different communication cost edge and computation cost tasks. Thus, 38400 random DAGs were used in the research.

In the ECSA, in local-EDA and graph random walk update, the learning rate k is 0.995. To evaluate the performance of the ECSA, we compare it with four algorithms: EFT, HEFT, PEFT, and LO. For a fair comparison, the algorithms are all coded in C++ and run on the same computer with AMD 5800x 3.8 GHz and 32 GB RAM.

Fig. 3 shows the *makespan* as a function of the DAG size n, processor number p and *CCR*. From Fig. 3(a), with the gradual increase of the DAG size, the ECSA can effectively reduce the *makespan* of the DAGs. The greater the task quantity is, the more *makespan* decreases. The HEFT and PEFT algorithms maintain similar performance in random DAGs. EFT is the simplest greedy algorithm and has the worst performance. The LO algorithm is designed and used in DAGs with specific structures. Although it has high complexity, it performs similarly to HEFT and PEFT when the DAG size is small or medium and performs worse than HEFT and PEFT when the DAG size is large, which is consistent with the result of the PEFT algorithm [2]. In general, the ECSA shows good optimization performance on *makespan* for various DAG sizes. More numerically, considering all random graphs as a whole, the reduction percentage of the ECSA compared with

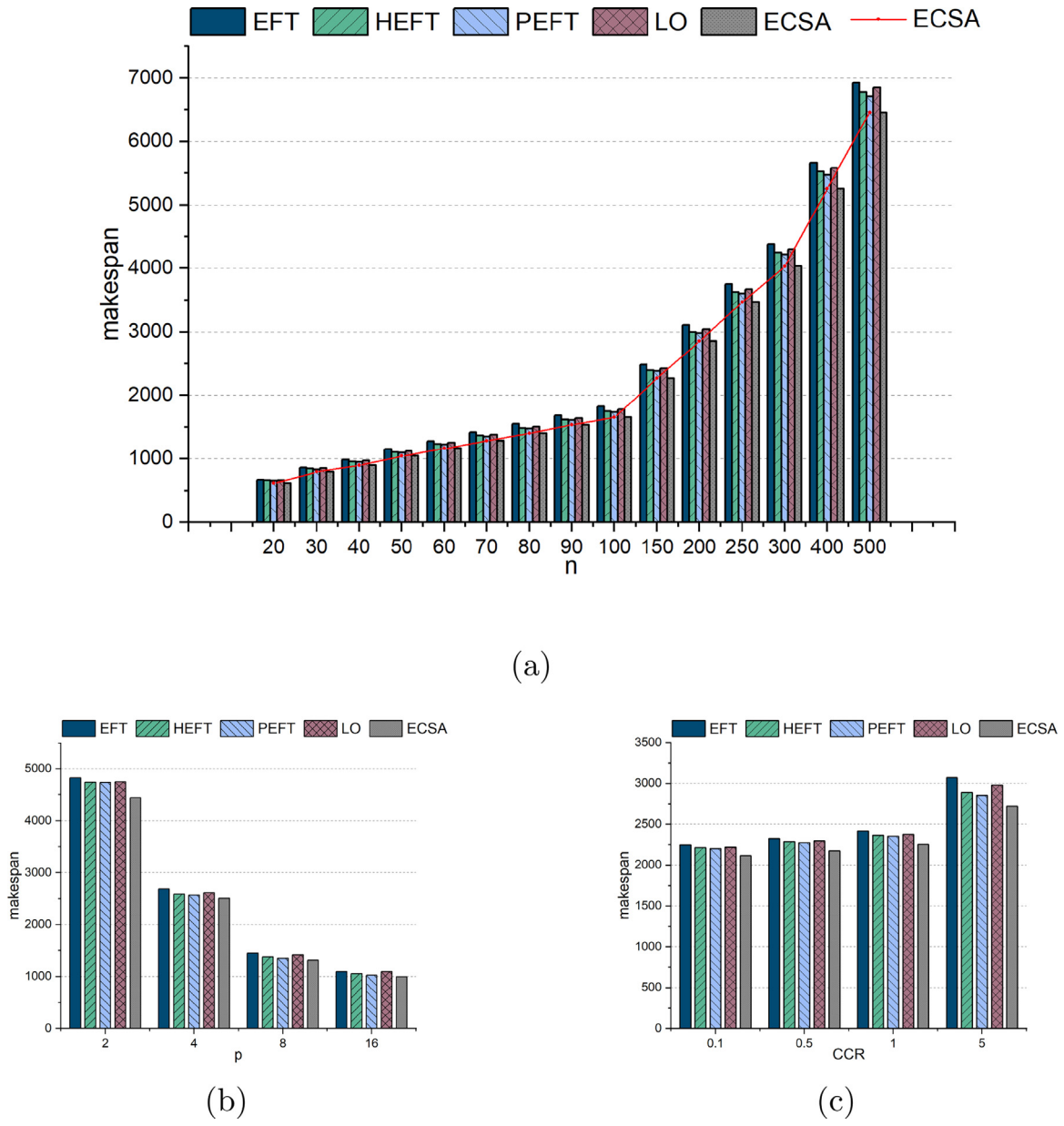


Fig. 3. (a) n-makespan of random DAGs (b) p-makespan of random DAGs (c) CCR-makespan of random DAGs.

that of EFT on makespan is 8%. This number for HEFT is 5%, for PEFT is 4.3%, and for LO is 6.2%. The ECSA brings considerable reduction to the scheduling of random DAGs, and this reduction does not require any upgrading of the HCS. From Fig. 3(b), the ECSA can reduce the *makespan* of the DAGs on the HCS for a different number of processors. It can effectively improve the computing performance in HCSs at various scales. From Fig. 3(c), we have verified the reduction of *makespan* by various algorithms on different *CCR* random DAGs. The ECSA can effectively reduce the *makespan* of the DAG in both communication-intensive and computation-intensive tasks.

Low *makespan* is the most direct target in task scheduling, but the SLR is a parameter that can better reflect the performance of the scheduling algorithm. Because SLR reflects the approximation ratio between the current scheduling scheme and the optimal solution. To further compare ECSA with other algorithms, we then calculate the SLR of the scheduling results of the ECSA and other algorithms for DAGs with different sizes. From Fig. 4, the scheduling solution given by the ECSA has a lower SLR than other scheduling algorithms. We can clearly see that as the DAG size increases, the SLR increases, and the ECSA reduces the SLR more. Under different DAGs, the scheduling results obtained by the ECSA can obtain a lower SLR.

Efficiency measures the utilization of scheduling algorithms for parallel computing systems. In Fig. 5, whether it is a small number of processors or a large number of processors, the ECSA can better improve the system efficiency, that is, enhance the parallelism of the system.

Table 1 shows the percentage of better, equal, and worse results for the ECSA when compared with other algorithms based on makespan. We can conclude that the ECSA can obtain better scheduling results in most situations.

Finally, we compare the execution time and memory usage of the above algorithms. We randomly selected five groups of random DAGs. Each group contains 1920 DAGs corresponding to the 1920 types of random DAGs mentioned above and calculates the average execution time of each DAG as the completion time and average memory usage. From Fig. 6(a), EFT is the simplest

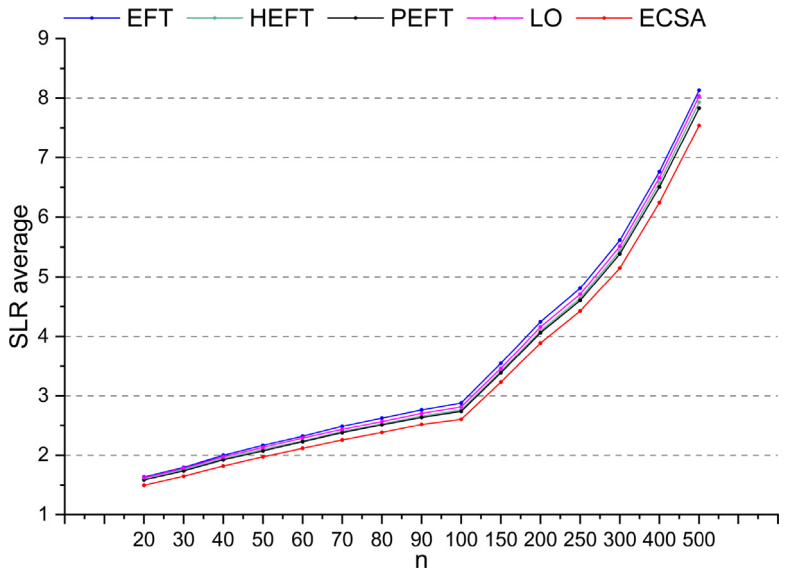


Fig. 4. n-SLR average of random DAGs.

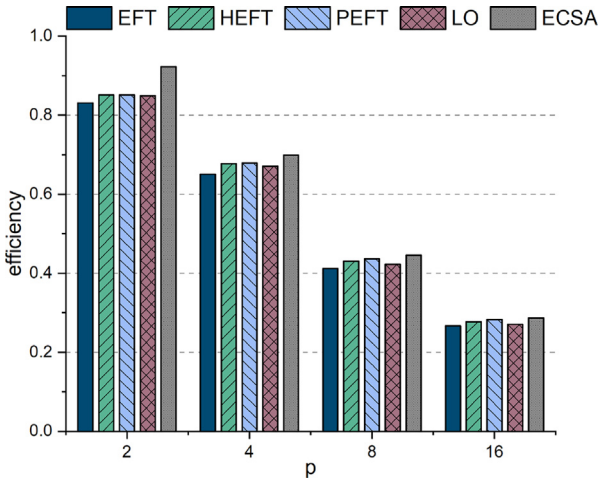


Fig. 5. p-efficiency of random DAGs.

Table 1
Better-equal-worse ratio.

ECSA	EFT	HEFT	PEFT	LO
Better	94.2%	83.2%	83.0%	87.8%
Worse	4.3%	14.3%	6.5%	10.7%
Equal	1.5%	2.5%	10.5%	1.5%

list scheduling algorithm, so it has the lowest completion time. As algorithms with the same time complexity(n^2p), HEFT and PEFT also have low completion times in general, but PEFT is approximately three times higher than HEFT. The LO algorithm is a list scheduling algorithm with high time complexity, but because we use LO with one step to greatly reduce the amount of computation, the completion time is also low. The ECSA is an iterative algorithm, but it also has a low completion time, which is close to the list scheduling algorithm of only 1.3 s. The ECSA is a memory-intensive algorithm that requires additional memory. From Fig. 6(b), however, the ECSA does not require too much memory usage. Therefore, we can conclude that the ECSA finds suboptimal solutions to obtain a good solution in an acceptable time.

From the experimental results, for random DAGs, in general, the ECSA is a reliable algorithm that can bring considerable improvement to the scheduling process. The ECSA improves the efficiency of the system and makes the scheduling result closer to the optimal solution.

5.3. Real-world program

In addition to the random DAGs, we evaluated the performance of the algorithms with respect to the real-world program, Gaussian Elimination(GE) [30], and Fast Fourier Transform (FFT) [1]. All of these applications are well-known and used in real-world problems.

The FFT algorithm can be separated into two parts: recursive calls and the butterfly operation [1]. The input points x determine the number of tasks [1]. The specific formula is $n = (x+2)2^x - 1$. In our experiment, the values for x are [3, 4, 5, 6].

The GE algorithm is a common method for solving matrix [30]. For GE, the matrix size x determines the number of tasks. The number of tasks n in a Gaussian Elimination DAG is equal to $\frac{x^2+x-2}{2}$. In our experiment, the values for x are [20, 21, 22, 23, 24, 25, 26, 27, 28, 29].

Because of the fixed structure of these applications, we simply used different values for CCR, heterogeneity, and CPU number. The range of values that we used in our simulation was [0.1, 0.5, 1, 5] for CCR, [0.6, 1] for heterogeneity, and [2, 4, 8, 16] for CPU number. Moreover,we will use the boxplot to analyze the SLR of scheduling results of different algorithms instead of the average SLR, which is different from random DAGs.

5.3.1. Fast Fourier Transform

The FFT algorithm is one of the most commonly used algorithms in the communication field and can help researchers analyze the spectrum of signals [1]. Almost every engineering researcher has learned the FFT algorithm. From Fig. 7, compared with the comparison algorithms, the ECSA performs better in terms of efficiency and SLR in FFT DAGs. Similar to the results of in the PEFT algorithm, HEFT seems to be the most appropriate algorithm among these comparison algorithms [2]. However, the ECSA has shown better performance in FFT programs. It can not only improve the efficiency of a variety of systems but also

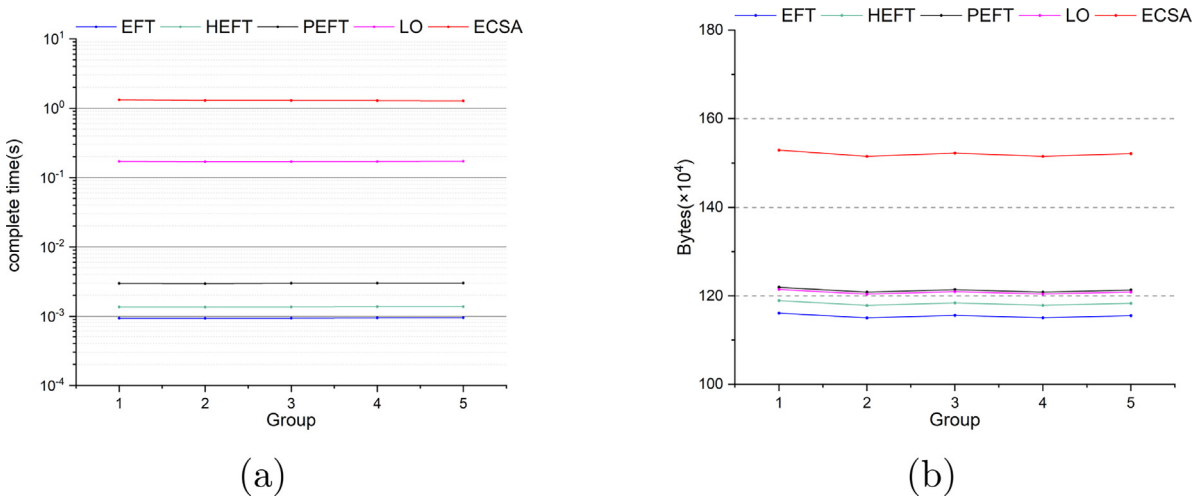


Fig. 6. (a) average completion time (b) average memory usage.

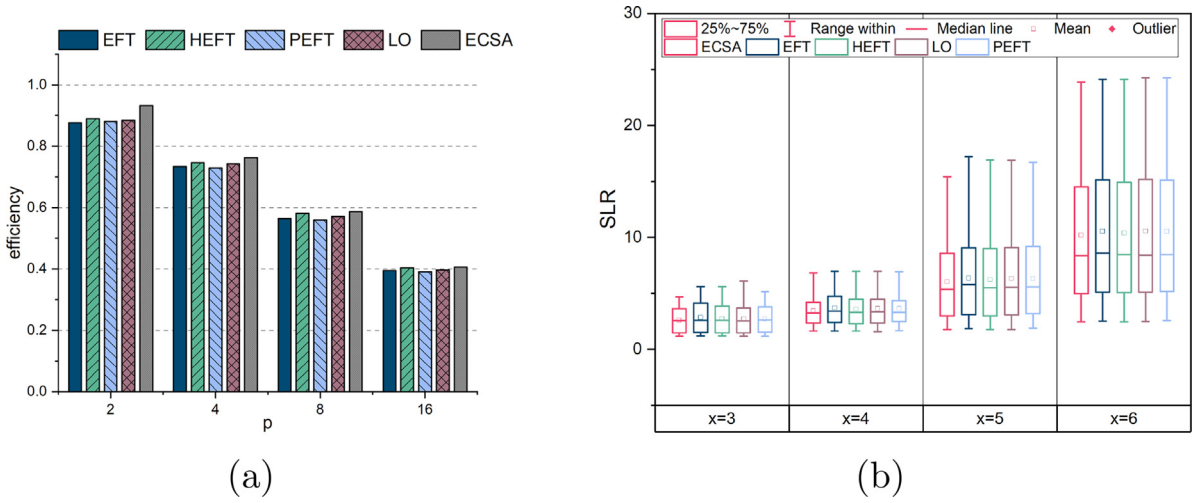


Fig. 7. (a) p-efficiency of FFT (b) n-SLR boxplot of FFT.

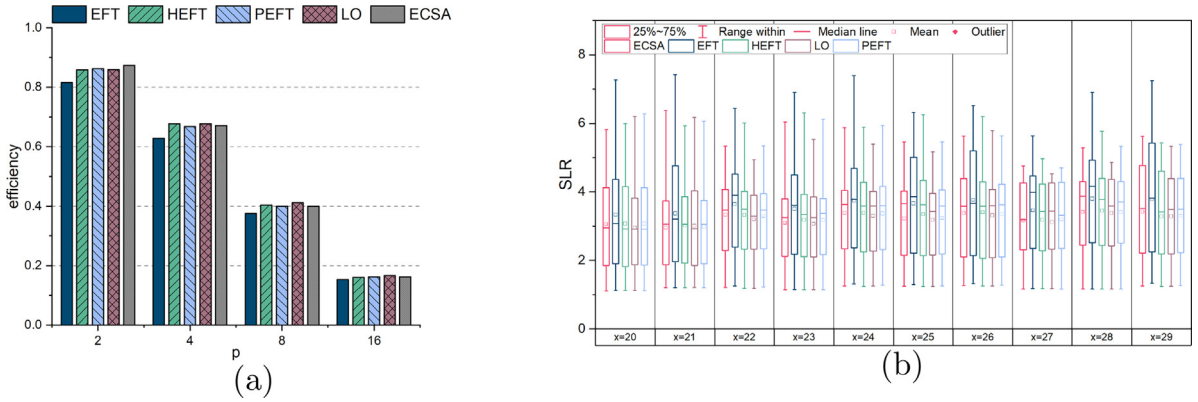


Fig. 8. (a) p-efficiency of GE (b) n-SLR boxplot of GE.

effectively reduce the SLR. In total, numerically, the ECSA brings about a 3.2% reduction in makespan and a 3.1% reduction in average SLR compared to HEFT.

5.3.2. Gaussian elimination

The GE algorithm is an algorithm in linear algebraic programming that can be used to solve linear equations. It is used to find the rank of a matrix and the inverse matrix of a reversible square matrix. From Fig. 8, the results obtained from ECSA were similar to HEFT and PEFT with only minor advantages, and the LO algorithm shows the best performance. This is not surprising. The LO algorithm often performs very well in the Gaussian elimination method DAG with a high time complexity $O(n^4p^3)$. Therefore, the ECSA is also an optional algorithm for GE programs.

6. Conclusions and future work

In this paper, we design a new algorithm, the ECSA, for the DAG-SP. An efficient DAG task scheduling algorithm can benefit the performance of IoT and cloud computing to improve the quality of user experience. In addition, an appropriate task scheduling algorithm can upgrade the resource balancing in a single processor with multiple cores and multiple virtual machines. The experimental results of a vast number of random DAG and real-world DAG data prove the theoretical advantages of the ECSA algorithm. The ECSA can benefit the DAG-SP in terms of makespan, efficiency, and SLR. It can be claimed that the ECSA can obtain a solution closer to the optimal, achieving better performance solutions in an acceptable time.

In the future, we will study two aspects. First is nondeterministic task scheduling model. In this model, the computing resources and completion time required by a single task vertex are no longer known or can accurately predict such as in a D2D network [31,32]. The existing task scheduling algorithm will cause a significant decline in system performance when applied to such scheduling problems. Second, a lighter edge cover queue scheduling algorithm for the DAG-SP model. We want to further reduce the time and computational complexity for edge cover queue generation.

CRediT authorship contribution statement

Yu-meng Chen: Conceptualization, Methodology, Software, Writing – review & editing. **Song-lin Liu:** Data curation, Software. **Yan-jun Chen:** Visualization, Investigation. **Xiang Ling:** Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.knosys.2023.110369.

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, http://dx.doi.org/10.1109/71.993206.

[2] H. Arabnejad, J.G. Barbosa, List scheduling algorithm for heterogeneous systems by an optimistic cost table, IEEE Trans. Parallel Distrib. Syst. 25 (3) (2014) 682–694, http://dx.doi.org/10.1109/TPDS.2013.57.

[3] C. ge Wu, L. Wang, J. jing Wang, A path relinking enhanced estimation of distribution algorithm for direct acyclic graph task scheduling problem, Knowl.-Based Syst. 228 (2021) 107255, http://dx.doi.org/10.1016/j.knosys.2021.107255, URL https://www.sciencedirect.com/science/article/pii/S0950705121005177.

[4] L. Zhang, L. Zhou, A. Salah, Efficient scientific workflow scheduling for deadline-constrained parallel tasks in cloud computing environments, Inform. Sci. 531 (2020) 31–46, http://dx.doi.org/10.1016/j.ins.2020.04.039, URL https://www.sciencedirect.com/science/article/pii/S0020025520303479.

[5] D. Yu, Y. Ying, L. Zhang, C. Liu, X. Sun, H. Zheng, Balanced scheduling of distributed workflow tasks based on clustering, Knowl.-Based Syst. 199 (2020) 105930, http://dx.doi.org/10.1016/j.knosys.2020.105930, URL https://www.sciencedirect.com/science/article/pii/S095070512030263X.

[6] H. Wu, X. Chen, X. Song, C. Zhang, H. Guo, Scheduling large-scale scientific workflow on virtual machines with different numbers of vCPUs, J. Supercomput. 77 (1) (2021) 679–710, http://dx.doi.org/10.1007/s11227-020-03273-3.

[7] B. Hu, Z. Cao, Minimizing resource consumption cost of DAG applications with reliability requirement on heterogeneous processor systems, IEEE Trans. Ind. Inform. 16 (12) (2020) 7437–7447, http://dx.doi.org/10.1109/TII.2019.2959070.

[8] J.D. Ullman, NP-Complete Scheduling Problems, Academic Press, Inc., 1975, http://dx.doi.org/10.1016/S0022-0000(75)80008-0.

[9] Y. Xu, K. Li, J. Hu, K. Li, A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues, Inform. Sci. 270 (2014) 255–287, http://dx.doi.org/10.1016/j.ins.2014.02.122, URL https://www.sciencedirect.com/science/article/pii/S002002551400228X.

[10] L.F. Bittencourt, R. Sakellariou, E.R.M. Madeira, DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm, in: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP '10, IEEE Computer Society, USA, 2010, pp. 27–34, http://dx.doi.org/10.1109/PDP.2010.56.

[11] S.M. Hussain, G.R. Begh, Hybrid heuristic algorithm for cost-efficient QoS aware task scheduling in fog–cloud environment, J. Comput. Sci. 64 (2022) 101828, http://dx.doi.org/10.1016/j.jocs.2022.101828, URL https://www.sciencedirect.com/science/article/pii/S1877750322001909.

[12] F. Abdallah, C. Tanougast, I. Kacem, C. Diou, D. Singer, Genetic algorithms for scheduling in a CPU/FPGA architecture with heterogeneous communication delays, Comput. Ind. Eng. 137 (2019) 106006, http://dx.doi.org/10.1016/j.cie.2019.106006, URL https://www.sciencedirect.com/science/article/pii/S0360835219304644.

[13] G. Sih, E. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Trans. Parallel Distrib. Syst. 4 (2) (1993) 175–187, http://dx.doi.org/10.1109/71.207593.

[14] L. Bouali, K. Oukfif, S. Bouzefrane, F. Oulebsir-Boumghar, A hybrid algorithm for DAG application scheduling on computational grids, in: Selected Papers of the First International Conference on Mobile, Secure, and Programmable Networking - Volume 9395, MSPN 2015, Springer-Verlag, Berlin, Heidelberg, 2015, pp. 63–77.

[15] 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, URL https://www.sciencedirect.com/science/article/pii/S0743731511001018.

[16] 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.-Aided Des. Integr. Circuits Syst. 29 (6) (2010) 911–924, http://dx.doi.org/10.1109/TCAD.2010.2048354.

[17] M.A. Elaziz, S. Xiong, K. Jayasena, L. Li, Task scheduling in cloud computing based on hybrid moth search algorithm and differential evolution, Knowl.-Based Syst. 169 (2019) 39–52, http://dx.doi.org/10.1016/j.knosys.2019.01.023, URL https://www.sciencedirect.com/science/article/pii/S0950705119300322.

[18] Y. Xu, K. Li, L. He, T.K. Truong, A DAG scheduling scheme on heterogeneous computing systems using double molecular structure-based chemical reaction optimization, J. Parallel Distrib. Comput. 73 (9) (2013) 1306–1322, http://dx.doi.org/10.1016/j.jpdc.2013.05.005, URL https://www.sciencedirect.com/science/article/pii/S074373151300110X.

[19] H. Hafsi, H. Gharsellaoui, S. Bouamama, Genetically-modified multi-objective particle swarm optimization approach for high-performance computing workflow scheduling, Appl. Soft Comput. 122 (2022) 108791, http://dx.doi.org/10.1016/j.asoc.2022.108791, URL https://www.sciencedirect.com/science/article/pii/S1568494622002113.

[20] S. Saeedi, R. Khorsand, S. Ghandi Bidgoli, M. Ramezanpour, Improved many-objective particle swarm optimization algorithm for scientific workflow scheduling in cloud computing, Comput. Ind. Eng. 147 (2020) 106649, http://dx.doi.org/10.1016/j.cie.2020.106649, URL https://www.sciencedirect.com/science/article/pii/S0360835220303831.

[21] X. Zhang, X. Liu, A. Cichon, G. Królczyk, Z. Li, Scheduling of energy-efficient distributed blocking flowshop using pareto-based estimation of distribution algorithm, Expert Syst. Appl. 200 (2022) 116910, http://dx.doi.org/10.1016/j.eswa.2022.116910, URL https://www.sciencedirect.com/science/article/pii/S0957417422003475.

[22] X. Hao, L. Lin, M. Gen, K. Ohno, Effective estimation of distribution algorithm for stochastic job shop scheduling problem, Procedia Comput. Sci. 20 (2013) 102–107, http://dx.doi.org/10.1016/j.procs.2013.09.246, Complex Adaptive Systems, URL https://www.sciencedirect.com/science/article/pii/S1877050913010466.

[23] J. Ceberio, E. Irurozki, A. Mendiburu, J.A. Lozano, A review on estimation of distribution algorithms in permutation-based combinatorial optimization problems, Progress Artif. Intell. 1 (1) (2012) 103–117, http://dx.doi.org/10.1007/s13748-011-0005-3.

[24] C.-g. Wu, W. Li, L. Wang, A.Y. Zomaya, Hybrid evolutionary scheduling for energy-efficient fog-enhanced internet of things, IEEE Trans. Cloud Comput. 9 (2) (2021) 641–653, http://dx.doi.org/10.1109/TCC.2018.2889482.

[25] Q. Zhao, Y. Gao, A new algorithm based on the gbest of particle swarm optimization algorithm to improve estimation of distribution algorithm, in: 2018 International Conference on Smart Computing and Electronic Enterprise, ICSCEE, 2018, pp. 1–5, http://dx.doi.org/10.1109/ICSCEE.2018.8538372.

[26] Z. Li, B. Qian, R. Hu, L. Chang, J. Yang, An elitist nondominated sorting hybrid algorithm for multi-objective flexible job-shop scheduling problem with sequence-dependent setups, Knowl.-Based Syst. 173 (2019) 83–112, http://dx.doi.org/10.1016/j.knosys.2019.02.027, URL https://www.sciencedirect.com/science/article/pii/S0950705119300887.

[27] M. Faraji Amiri, J. Behnamian, Multi-objective green flowshop scheduling problem under uncertainty: Estimation of distribution algorithm, J. Clean. Prod. 251 (2020) 119734, http://dx.doi.org/10.1016/j.jclepro.2019.119734, URL https://www.sciencedirect.com/science/article/pii/S0959652619346049.

[28] B. Liu, Y. Fan, Y. Liu, A fast estimation of distribution algorithm for dynamic fuzzy flexible job-shop scheduling problem, Comput. Ind. Eng. 87 (2015) 193–201, http://dx.doi.org/10.1016/j.cie.2015.04.029, URL https://www.sciencedirect.com/science/article/pii/S0360835215002077.

[29] R. Dick, D. Rhodes, W. Wolf, TGFF: Task graphs for free, in: Proceedings of the Sixth International Workshop on Hardware/Software Codesign, CODES/CASHE'98, 1998, pp. 97–101, http://dx.doi.org/10.1109/HSC.1998.666245.

[30] M. Cosnard, M. Marrakchi, Y. Robert, D. Trystram, Gauss elimination algorithms for mimd computers, in: W. Händler, D. Haupt, R. Jeltsch, W. Juling, O. Lange (Eds.), CONPAR 86, Springer Berlin Heidelberg, Berlin, Heidelberg, 1986, pp. 247–254.

[31] N. Eshraghi, B. Liang, Joint offloading decision and resource allocation with uncertain task computing requirement, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 1414–1422, http://dx.doi.org/10.1109/INFOCOM.2019.8737559.

[32] W. Chang, Y. Xiao, W. Lou, G. Shou, Offloading decision in edge computing for continuous applications under uncertainty, IEEE Trans. Wireless Commun. 19 (9) (2020) 6196–6209, http://dx.doi.org/10.1109/TWC.2020.3001012.