Hindawi
Applied Computational Intelligence and Soft Computing
Volume 2019, Article ID 6543957, 15 pages
https://doi.org/10.1155/2019/6543957

Research Article

Learning Based Genetic Algorithm for Task Graph Scheduling

Habib Izadkhah

Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran

Correspondence should be addressed to Habib Izadkhah; izadkhah@tabrizu.ac.ir

Received 20 May 2018; Revised 29 October 2018; Accepted 20 November 2018; Published 3 February 2019

Academic Editor: Yangming Li

Nowadays, parallel and distributed based environments are used extensively; hence, for using these environments effectively, scheduling techniques are employed. The scheduling algorithm aims to minimize the makespan (i.e., completion time) of a parallel program. Due to the NP-hardness of the scheduling problem, in the literature, several genetic algorithms have been proposed to solve this problem, which are effective but are not efficient enough. An effective scheduling algorithm attempts to minimize the makespan and an efficient algorithm, in addition to that, tries to reduce the complexity of the optimization process. The majority of the existing scheduling algorithms utilize the effective scheduling algorithm, to search the solution space without considering how to reduce the complexity of the optimization process. This paper presents a learner genetic algorithm (denoted by LAGA) to address static scheduling for processors in homogenous computing systems. For this purpose, we proposed two learning criteria named Steepest Ascent Learning Criterion and Next Ascent Learning Criterion where we use the concepts of penalty and reward for learning. Hence, we can reach an efficient search method for solving scheduling problem, so that the speed of finding a scheduling improves sensibly and is prevented from trapping in local optimal. It also takes into consideration the reuse idle time criterion during the scheduling process to reduce the makespan. The results on some benchmarks demonstrate that the LAGA provides always better scheduling against existing well-known scheduling approaches.

1. Introduction

The parallel processor revolution is underway. Move toward the use of parallel structures is one of the greatest challenges for software industries. In 2005, Justin Rattner, chief technology officer of Intel Corporation, said "We are at the cusp of a transition to multicore, multithreaded architectures, and we still have not demonstrated the ease of programming the move will require. . ." [1].

A good scheduling of a sequential program onto the processors (or computers) is critical to effective utilization of the computing power of a multiprocessor (or multicomputer) system [2]. To perform scheduling, a sequential program is represented by a Directed Acyclic Graph (DAG), which is called a task graph. In this graph, a node represents a task which is a set of instructions that get a set of inputs to produce a set of outputs and must be executed serially in the same processor. Computation cost associated with each node indicates the amount of computation. The edges in the DAG correspond to the communication messages and precedence constraints among the tasks as they show the communication time from one task to another. This number is called the communication cost and is denoted by c_{ij} [3]. A sample DAG is shown in Figure 1. In a task graph, a node is not ready to run until all its required data are received from its parent nodes. The communication cost between two nodes which are allocated to the same processor is considered to be zero.

Scheduling has been used in two different areas in performance evaluation in the literature: multiprocessors distributed systems and cloud computing. The purpose of the scheduling on multiprocessors systems is to minimize the completion time of DAG on a set of homogeneous or heterogeneous processors. This helps to speed up a sequential program. Cloud computing also utilizes scheduling techniques, but with different goals such as response time, energy consummation, throughput, and RAM efficiency. Some papers that use scheduling in this regard are [4–7]. This paper aims to present an algorithm for scheduling a sequential onto multiprocessors systems, not cloud computing.

In the scheduling problem, the completion time is the time required for a program to be completed. Figure 2 shows a sample timeline (scheduling) in the form of a Gantt chart

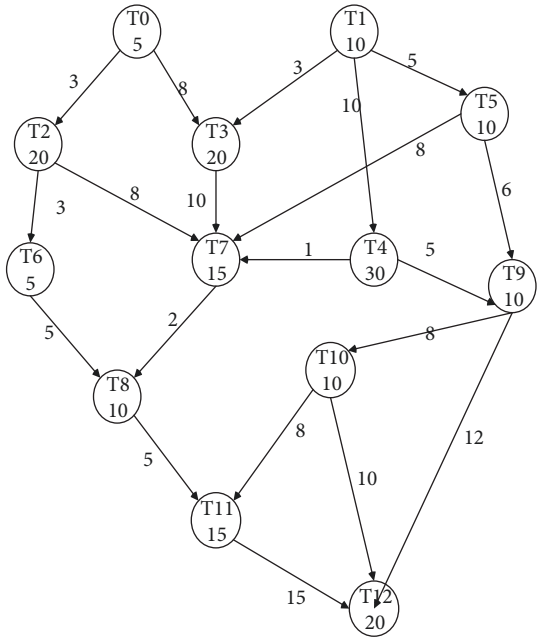

Figure 1: A sample task graph.

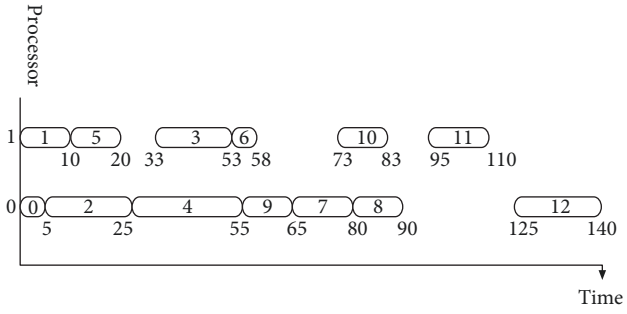

Figure 2: A sample task graph.

on two processors for task graph shown in Figure 1. As can be seen in this figure, the completion time for this task graph is 140.

The performance of a scheduling algorithm is usually measured in terms of completion time and the running time of the algorithm. There is, usually, a trade-off between these two performance parameters; that is, efforts to obtain a minimum completion time often incur a higher time complexity.

Due to the NP-hardness of the scheduling problem, the use of search-based and evolutionary approaches such as genetic algorithm is to be more reasonable than other approaches [2]. The searching procedure in search-based algorithms can be performed in two ways: global search and local search. Global search-based methods usually consider the whole search space to find a good solution. These methods have an operator to explore new regions in search space. The genetic algorithm is one of the main search methods that use the global search strategy. This algorithm has no exploitation ability, and also the speed of convergence is not sufficient while searching for global optima. Local search algorithms, such as Hill Climbing and Learning Automata, utilize exploitation ability to improve the quality of the already existing solutions as well as the rate of convergence.

Hill climbing algorithm starts from an initial solution and then iteratively moves from the current solution to a neighbor solution (to be defined) in the search space by applying local changes. There are no learning concepts in this algorithm. Learning is an ability of a system to improve its responses based on past experience. Reinforcement learning techniques are used to choose the best response based on the *rewards* or *punishments* taken from the environment. Learning automata is an automaton-based approach for learning. Learning automata learn the optimal action through past experiences and repeated interactions with its stochastic environment. The actions are chosen according to a specific probability distribution which is updated based on the environmental response.

The problem addressed in this paper is the scheduling of a task graph on homogeneous processors for minimizing the completion time to achieve a higher performance in the execution of parallel programs. To achieve this aim, this paper presents a new learning schema inspired by the behaviors of both hill climbing algorithm and learning automata. We, then, propose an efficient search approach which adds a new learning function to the evolutionary process of the genetic algorithm for scheduling. This new algorithm combines global search (genetic algorithm) and local search (using the concepts of penalty, reward and neighbors) strategies for scheduling of a task graph. This causes the evolutionary process speed, i.e., convergence rate, to increase considerably and be prevented from trapping in local optimal. Hence, the algorithm, at the same time with other algorithms, can search more search-space of the problem and find the solution with higher quality. Also, the proposed algorithm can be improved by employing the optimal reuse of idle time on processors and the use of the critical path of the task graph for constructing the initial population. This technique enables the GA to schedule the task graph better than previous algorithms. Results of the conducted experimental demonstrate that LAGA is superior to the Parsa, CPGA, TDGA, CGL, BCGA, DSC, MCP, LC, and LAST algorithms.

The contributions of this paper are summarized as below:

(1) Proposing a new learning schema inspired by the behaviors two local search-based algorithms namely hill climbing algorithms and learning automata;

(2) Improving the genetic algorithm evolutionary process using the new learning schema;

(3) Proposing optimal reuse idle time heuristic, aiming to reuse idle times on the processors;

(4) Proposing new reward and penalty function which is based on the earliest start time for task graph scheduling.

The remainder of this dissertation is structured as follows. In Section 2, we provide an overview of related work. In Section 3, a new algorithm for scheduling a task graph is

proposed. In Section 4, the results of the proposed algorithm are evaluated. Finally, Section 5 concludes the paper.

2. Background and Previous Works

The need for accurate task graph scheduling has been widely recognized by the research community. In the literature, there are many works about the task graph scheduling algorithms. The Scheduling techniques are typically categorized into two classes; the first class is homogeneous [12] and the second one is heterogeneous [13, 14]. In homogeneous systems, the processing power of all processors is considered similar. In heterogeneous type, processors have different processing power; therefore, for each task, time on every processor should be given separately. In this paper, scheduling is assumed as homogeneous.

Generally, all task graph scheduling algorithms can be grouped into two categories: static and dynamic. Static scheduling is performed off-line; in other words, desirable parallel program characteristics such as computation costs, communication cost, and data dependencies are known before execution [11]. Dynamic scheduling algorithms can be further grouped into two categories: accomplished quickly mode and batch processing mode algorithms. In the accomplished quickly mode, a task is mapped as soon as it arrives. In the batch mode, tasks are not mapped as they arrive; instead, they are collected into a list for scheduling and mapping [12]. The work in this paper is solely concerned with the static scheduling.

Static task graph scheduling algorithms are classified into two categories: Heuristic-based algorithms and guided random search-based algorithms [14]. Heuristic-based algorithms have acceptable performance with a low complexity. These algorithms are further classified into three classes: list-based algorithms, clustering-based algorithms, and task duplication algorithms.

List-based scheduling approaches, such as HEFT [15] and CPOP [14], consist of two phases of tasks prioritization and processor selection (to assign the task to the processor). Task duplication approaches aim, such as STDS [16], HCNF [17], is to run a task on more than one processor to reduce the waiting time for the dependent tasks. The main idea behind this approaches is to use the processors time gap.

In the literature, there are several heuristic-based (nonevolutionary) scheduling algorithms such as PEFT [13], ETF [18], DSC [19], DLS [20], HLFET [20], LAST [21], ISH [22], MCP [23], LC [24], EZ [25], and MTSB [26]. Most of these algorithms consider several simple assumptions about the structure of a task graph [2, 12]. For example, ignoring task communication cost, or considering the same computational cost for all tasks [2], or assuming a tree structure for a task graph [8] is some of these simple assumptions. In general, nowadays many recent algorithms are designed to deal with any graphs. Moreover, scheduling on a limited number of processors, called Bounded Number of Processors (BNP), or the availability of unlimited number of processors, called Unbounded Number of Clusters (UNC), is another assumption of scheduling algorithms.

Since the task graph scheduling problem is an NP-Hard problem [2, 12]; even simplified versions of the task scheduling problem are NP-Hard. Let P and T indicate the number of processors and the number of tasks, respectively, thus, the possible number of task to processor allocation is P^T and the possible number of task orderings is $T!$; then, the overall search space is $T!P^T$. Considering this huge search space, obtaining an optimal schedule by a comprehensive search of all possible schedules is not achievable; hence, an evolutionary approach such as genetic algorithms may be applied to solve such problems, effectively [8, 27].

Using genetic algorithms to solve task graph scheduling have received much attention. The main difference among genetic algorithms is the encoding scheme used to represent a schedule. The structure of encoding significantly impacts the complexity of the genetic process to the convergence of an optimal schedule. For example, Wang and Korfhage [10] presented a matrix representation of schedules which records the execution order of the tasks on each processor. In this encoding, the crossover and mutation operators do not prevent the production of invalid solutions. Although exist the repair operations to correct these solutions. However, the operations consume additional time. Parsa et al. [8] proposed a new representation of schedules based on the string representation. In this new encoding scheme, a scheduling solution is represented by an array of pairs (t_i, p_j), where t_i indicates the task number assigned to the processor p_j. Using this representation, both the execution order of the tasks on each processor and the global order of the tasks executions on the processors are determined in each chromosome. Hou et al. [9] proposed a variable-length representation which orders the tasks on each processor. Their proposed string representation prevents the production of invalid solutions. Whereas this representation cannot span the complete space of possible schedules, it may be impossible for the genetic algorithm to converge to an optimal solution.

Omara and Arafa [11] proposed two genetic-based algorithms with a new encoding scheme, namely, Critical Path Genetic Algorithm (CPGA) and Task-Duplication Genetic Algorithm (TDGA). The CPGA algorithm tries to schedule the tasks placed on the critical path on the same processor aiming to reduce the completion time. The second algorithm, TDGA, is based on the principle of task duplication to minimize the communication overhead. One of the weaknesses of this method is to generate invalid chromosomes. The runtime of these two algorithms is very high for finding a scheduling because it uses some internal algorithms for scheduling such as critical path and task duplication. Also, the performance of the algorithm is not clear on benchmarked DAGs.

Daovod et al. [28] proposed a new scheduling algorithm, which is called the Longest Dynamic Critical Path (LDCP), to support scheduling in heterogeneous computing systems. The LDCP is a list-based scheduling algorithm, which effectively selects the tasks for scheduling in heterogeneous computing systems. The accurate and better selection of tasks enables the LDCP to provide a high quality in scheduling the heterogeneous systems. The function of this algorithm is compared with two well-known HEFT and DLS algorithms

in this context. One of the weaknesses of this algorithm is its greediness, which will not respond well in the case of large-scale DAGs.

Bahnasawy et al. [29] proposed a new algorithm, named Scalable Task Duplication Based Scheduling for static scheduling in computing systems with heterogeneous multiprocessors. Considering the priority between the tasks, this algorithm divides the graph into levels so that the tasks at each level are arranged in a list based on their size. The tasks based on their priority are assigned to the first processor found in the list. One of the weaknesses of this algorithm is its greediness, which will not function well in solving large-scale DAGs.

Naser et al. [30] proposed a scheduling algorithm, called Communication Leveled DAG with Duplication (CLDD), in the heterogeneous multiprocessor systems. This algorithm consisted of three phases: (1) level arrangement, (2) tasks prioritization, and (3) processor selection.

Wen [31] proposed a hybrid algorithm of genetics and variable neighborhood search (VNS) [32] to minimize the tasks scheduling in the heterogeneous multiprocessor systems. This algorithm has derived a variety of features from GA and VNS.

NSGA-II [33] is a genetic-based algorithm for task graph scheduling. In this algorithm does not exist the dependency among the tasks, and elitism operator is used to protect the good solutions in the evolutionary process of the genetic algorithm. To provide diversity in solution, this algorithm uses crowding distance technique.

TSB algorithm [34] belongs to a class of scheduling algorithms called BNP. TSB utilizes two queues, namely, ready task queue and not ready task queue to perform scheduling. Then, it uses breath first search traversal algorithm for selecting tasks. The MTSB algorithm [26], also, belongs to the BNP class, with the difference that, unlike the algorithm TSB, it does not consider communication time between tasks and processors.

Akbari et al. [35] proposed a genetic-based algorithm for static task graph scheduling in heterogeneous computing systems. Their proposed algorithm presents a new heuristic to generate an initial population and also proposed new operators aiming to guarantee the diversity and convergence. Akbari and Rashidi [36] employed cuckoo optimization algorithm for static task graph scheduling in heterogeneous computing systems. Because this algorithm works on continuous space, hence, they proposed a discretization of this algorithm to solve the task graph scheduling problem.

Moti Ghader et al. [37] proposed a learning automata-based algorithm for static task graph scheduling. This algorithm uses learning automata to represent different possible scheduling. Since this algorithm is a local-based search algorithm, thus, it traps in the local optima.

A significant performance assessment and comparison of the addressed algorithms is a complex task and it is necessary to consider a number of issues. First, there are several scheduling algorithms based on various assumptions. Second, the performance of the most existing algorithm is evaluated on small sized task graphs, and, hence, it is not clear their performance on large-scale problem size [38]. Due to the NP-hardness of the task graph scheduling, the rate of convergence when searching for global optima is still not sufficient. For this reason, there is a need for methods that can improve convergence speed, to achieve a good scheduling.

Table 1: Classic chromosome structure for a sample DAG.

Task No.	1	2	3	4	5	6
Processor No.	1	1	2	3	1	2

3. The New Algorithm

In this section, a new genetic algorithm is presented for the task graph scheduling problem which is based on learning. In this algorithm, each solution is represented by a chromosome. Each chromosome's cell is called a gene. The number of genes depends on the number of tasks. We use two different chromosome structures in our algorithm named classic and extended chromosome structure. In classic chromosome, the structure of each chromosome includes two rows where the first row is task number and the second row is processor number assigned to the task (see Table 1).

The extended chromosome structure is utilized for the learning process. This representation has four rows where the first row is task number, the second row is processor number assigned to the task, the third row is the depth of each gene, and the fourth row is selection probability of each task needed to perform learning. The total probability of them, initially, is equal (Table 2).

In extended chromosome, structure a chromosome is defined as a tuple $\{a, v, \beta, \varphi, F, P, T\}$ as follow:

(i) $a = \{a_1, ..., a_r\}$: is a set of genes (r is number of the tasks).

(ii) $v = \{v_1, ..., v_r\}$: is a set of processors in the chromosome.

(iii) $\beta = \{\beta_1, ..., \beta_r\}$: This is the outcome of evaluation of a gene considering interactions with its stochastic environment and indicates that a gene must be fined or rewarded.

(iv) $\varphi_1, \varphi_2, ..., \varphi_{RN}$: Let N and R denote the depth allocated to a gene and the number of genes, respectively, this shows a set of internal states for a gene.

(v) $F : \varphi \times \beta \longrightarrow \varphi$: This is a mapping function. This function based on evaluating a gene determines the next action for a gene.

(vi) $P = \{p_1, ..., p_r\}$: is a chromosome's genes probability. This vector (fourth row in Table 2) shows the selection probability of each gene to perform reward or penalty operations. Based on the fact that the penalty is taken or rewarded, these probabilities are updated. If no prior information is available, there is no basis in which the different genes a_i can be distinguished. In such a case, all gene probabilities would be equal - a

Table 2: Extended chromosome structure for a sample DAG.

Task No.	1	2	3	4	5	6
Processor No.	1	1	2	3	1	2
Depth	0	0	0	0	0	0
Probability	0.16	0.16	0.16	0.16	0.16	0.16

Step 1:
- Create the initial population (classic chromosomes) as described in Section 3.1.

Step 2:
- **While** termination condition has not been satisfied **Do**
  - **For** each classic chromosome in the current population **Do**
    - Convert it to the extended structure chromosome (such as Table 2).
    - Enhance chromosome by applying learning operators (reward and penalty) on the chromosome until the makespan could be less, described in Section 3.4.
  - **Convert** extended chromosome to classic chromosome by removing rows related depth and probability
  - **Apply** the standard roulette wheel selection strategy for selecting potentially useful individuals for recombination
  - **Apply** the standard two-point crossover operator on processors in two enhanced chromosomes
  - **Apply** the standard mutation operator on processors in two enhanced chromosomes
  - **Apply** the reuse idle time heuristic on 30% of best chromosomes, described in Section 3.3.

Algorithm 1: A new task graph scheduling algorithm.

- Calculate the t-level + b-level values for all the tasks;
- Considering t-level + b-level, identify all the tasks placed on the critical path;
- **Repeat**
  - Select a task amongst the tasks ready to schedule
  - **If** the selected task placed on the critical path **Then**
    Assign the task to a processor that has assigned other tasks on the critical path;
    Else Find a task with highest communication cost with this task and locate both of them on the same processor;
  - **Until** the list of tasks is empty.
- At first, each gene on the chromosome is in the boundary state or zero depth and its probability is: $P_i(t) = 1/r$

Algorithm 2

"pure chance" situation. For an r-gene chromosome, the gene probability is given by:

$$P_i(t) = \frac{1}{r} \quad i = 1, 2, \ldots, r \text{ (r is number of tasks)} \quad (1)$$

(vii) $T = T[a(n), \beta(n), p(n)]$ indicates the parameters required for a learning.

In Algorithm 1, the steps of our proposed task graph scheduling algorithm are given. Different parts of this algorithm are further described in the following sections.

The objective of the new task scheduling algorithm is to minimize the completion time. Some terms are used; the proposed algorithm is defined below.

Definition 1. In a task graph, the t-level of node i is the length of the longest path from an entry node to i (excluding i). Here, the length of a path is the sum of all the node and edge weights along the path [39].

Definition 2. The b-level of a node is the length of the longest path from the node to an exit node [39].

Definition 3. A Critical Path (CP) of a task graph is a path from an entry node to an exit node as the sum of execution time and communication cost is the maximum. The nodes located on the CP can be identified with their values of (t-level + b-level); i.e., t-level + b-level for that node should be maximized [39].

Definition 4. EST_{ij} — *Earliest Start Time of* task i and processor j: The EST refers to the earliest execution start time of the node n_i on processor p_j.

3.1. Initial Population. The initial population is constructed as shown in Algorithm 2.

The initial population is generated in such a way that the overall communication time is minimized. To do this, the algorithm identifies all tasks located on the critical path

(1) **For** each task in a chromosome **Do**
    Set its reference count equal to the number of immediate parents of the task, in the task graph corresponding to the chromosome.
(2) **For** each processor P_i **Do**
    Set its local timer s_i to zero.
    Add execution time of processor's first task to s_i.
(3) Set the global timer S to one.
(4) Repeat
    For each processor P_i **Do** {
        Read the value of timer s_i of P_i.
        If $S = s_i$ **Then**
            Reduce one from the *reference count* of each child of the task.
            Set *EST* of each child of the task that are not assigned to P_i, to:
            Max (Current *EST* of the child, Sum of s_i and communication cost between the tasks)
            Take next task of P_i from the chromosome.
            Add the sum of S and execution time of the task to s_i.
        If *EST* of the task is bigger than S **or** *reference count* of task is not zero **Then**
            add one unit of time to s_i.
        Add one unit of time to the global timer S.
    } **Until all the tasks are scheduled**.
(5) makespan = the maximum value of the timers, s_i, of all processors, P_i, as the fitness of the chromosome.

Algorithm 3: Schedule length.

and then tries to shorten the critical path in the DAG by assigning the tasks on the critical path into the same processor. By reducing the length of the critical path, the minimum completion time of the scheduling is reduced.

3.2. Computing Fitness of Chromosome. Quality of each the chromosome depends on the completion time of the scheduling. The completion time of a scheduling represents its fitness. The objective is to minimize the completion time. In the proposed algorithm, first, the EST for all tasks are calculated. The value of an entry EST_{ij}, referring to task i and processor j, will be given by

$$EST_{ij} = \max\left(FT_j, ETT_{ij}\right) \tag{2}$$

where FT_j is the time that machine j finishes the execution of all previous tasks of task i and ETT_{ij} indicates the time that all the data required to run task i on processor j is ready. To calculate this time, the finish time and time required to transfer data between processors of all immediate predecessors of task i in the task graph are considered.

The fitness function used to guide the evolutionary process of the genetic algorithm for finding a scheduling is based on the total finish time of the schedule, which is obtained by

$$\text{Makespan} = \max\left\{\text{FT}\left(\text{P}_\text{j}\right)\right\}; \quad \text{for j} = 1, 2, \ldots, \text{N}_\text{p} \tag{3}$$

where N_p and $FT(P_j)$ denote the number of processor and the finish time of the last task in the processor P_j, respectively.

$$\begin{aligned} \max\_\text{makespan} &= \max\{\text{makes-spans}\}; \\ \text{Fitness value} &= \max\_\text{makes-pan} - \text{makespan} \end{aligned} \tag{4}$$

In our method the length of an obtained schedule and makespan is computed as Algorithm 3.

In the LAGA, we employed the roulette wheel selection operator to select potentially useful chromosomes for the reproduction process. The fitness value is used to associate a selection probability for each chromosome. Let f_i and N denote the fitness value of the chromosome i (which is calculated by (4)) and the number of chromosomes in the population, respectively. The selection probability of this chromosome, P_i, is calculated as follows:

$$P_i = \frac{f_i}{\sum_{j=1}^{N} f_i} \tag{5}$$

Using (5), a proportion of the wheel is assigned to each chromosome. The higher the fitness value of the chromosome, the higher its opportunities of being elected for the reproduction process.

3.3. Reuse Idle Time. A reuse idle time criterion can improve the quality of a schedule [40]. In this section, we present a reuse idle time greedy algorithm to control the original system and improve the quality of returned results. For a randomly selected chromosome in the population, in each generation, we assign some tasks to idle time slots. For example, Figure 3 shows a sample of the Gantt scheduling chart related to Figure 1. According to Figure 3 (left), the time gap in the P1 processor is as large as 10 units and the completion time for the processor P1 is equal to 148. Also, the weight of the task t10 is equivalent to 10. By applying changes, the t10 can start from the time 73, and the completion time reduces from 148 to 138 units with this reuse, as can be seen in Figure 3 (left).

```
Let STT be Start Time of Task
      SIT be Idle Start Time
      EIT be Idle End Time
(t_j) denotes task j
For P=1 to the number of processors
   For i=1 to the number of slots on processor P
      For each slot i
            Create ready task list; for doing this task, select the tasks that for those
            STT >= EIT, i.e., select the tasks that is scheduled after a slot.
            Sort tasks acceding based on w(t_j)
               If (((EIT_i - SIT_i) >= w(t_j)) && DAT(t_j, P_i) <= SIT_i)
               Allocate task j to slot i
               Update slot start and end time
               Update the ready task list
               After rescheduling, newly created slot is added to the slot ready list.
```

Algorithm 4: Reuse idle time.

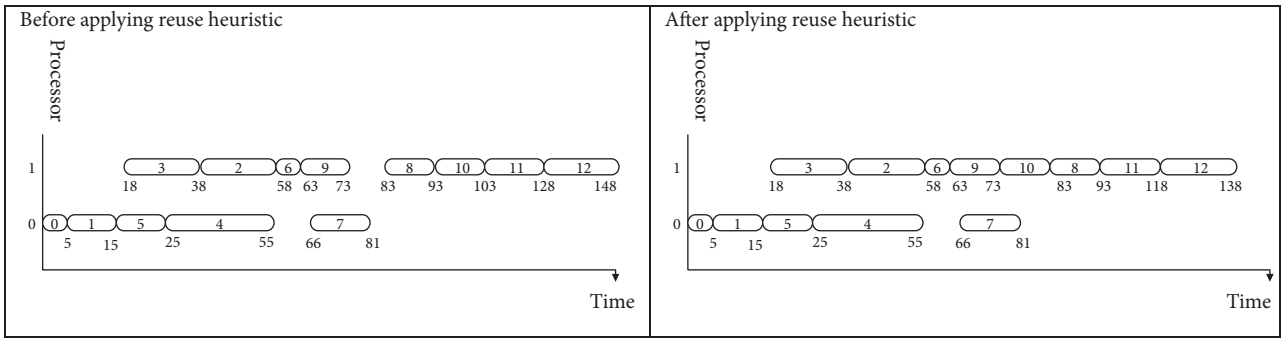

Figure 3: Reusing the standby resources.

The algorithm of this heuristic is shown in Algorithm 4; while all tasks and all slots are not addressed, this algorithm will be repeated. Let idle start time and idle end time of slot i denoted by SIT_i and EIT_i, respectively. Also assume $DAT(t_i, P_i)$ and $w(t_i)$ denote data arrival time of task t_i on processor P_i and execution time of task t_i, respectively. Task t_i allocated to slot s_i when

$$\text{If } (((EIT_i - SIT_i) >= w(t_i)) \;\&\&\; DAT(t_i, P_i) <= SIT_i)$$

For example, assume the processor P_1 has an idle slot; and the SIT_1 =73 and EIT_1 =83. On the other hand, $w(t_{10})$= 10 and DAT (t_{10}, P_1)=59. In our algorithm t_{10} is scheduled to start time 93. By applying the reuse idle algorithm, t_{10} can be rescheduled to start at time 83.

We need a list that maintains all of SIT and EIT for each slot. The algorithm tries to fill the slots as much as possible. Our aim is to select the maximum tasks that meet the above requirement. For doing this task, first, we sort the tasks ascending considering their execution time, i.e., $w(t_i)$; then, using Algorithm 4, we assign the tasks in slots.

The reuse idle time heuristic more minimizes the makespan; however, this algorithm for large-scale task graphs makes an overhead cost in terms of time. Therefore, we limit reuse idle time heuristic just to 30% of best chromosomes. This means that, among the chromosomes of a generation, 30% of them with the shortest completion time are selected to apply reuse idle time heuristic.

3.4. Learning Operators. The evolutionary process of the LAGA is enhanced using reward and penalty operators. These operators are utilized for learning. In the LAGA, in addition to evaluating the chromosome, the genes of a chromosome are assessed based on their outcome on chromosome's fitness. Thus, the most fitting location for genes within chromosomes and the most proper chromosome inside the population is regularly discovered during the evolutionary process of the genetic algorithm. Intuitively, the algorithm selects a gene a chromosome and evaluates it based on certain criteria. If the result of the evaluation is good, this gene will be rewarded. This means that the gene in the chromosome is in the right place and should be protected from future changes. But if the evaluation result is not good, the gene must be penalized. In this case, if the gene has not already been rewarded or the amount of penalties and rewards are the same, the whole possible neighbors (to be defined) of the gene are searched for a penalty action. The neighbor with the highest quality will be the replacement of this gene. But if the number of rewards received by the gene is greater than the number of received penalties, one of the numbers of rewards is reduced.

Formally, the penalty and reward operators are employed in the LAGA as follows.

during the scheduling process, the LAGA selects gene a_i in a chromosome and assesses it, if it gets a promising feedback, the probability, $P_i(n)$, of this gene increases and the probability of other genes decreases. If it gets an inappropriate feedback, the probability, $P_i(n)$, of this gene decreases and probability of other genes increases. In this paper, we utilize a linear learning scheme for learning. This scheme was considered first in mathematical psychology [41] but was later independently conceived and introduced into the engineering literature by Shapiro and Narendra [41]. Letting Γ_j and Ψ_j be two functions, for a linear learning scheme with multiple genes, we have

$$\Psi_j\left[p_j(n)\right] = \alpha p_j(n) \quad 0 < \alpha < 1 \tag{6}$$

$$\Gamma_j\left[p_j(n)\right] = \frac{\beta}{m-1} - \beta p_j(n) \tag{7}$$

In (6) and (7), m, α, and β are the number of genes of a chromosome, reward, and penalty parameters, respectively. The parameters α and β are used to control the rate of convergence of a chromosome. According to (6) and (7), if α and β are equal, the learning algorithm will be known as linear reward penalty. If $\beta \ll \alpha$, the learning algorithm will be known as linear reward epsilon penalty and if b=0, the learning algorithm will be a linear reward inaction. Depending on whether penalty or reward has been made, the probability of genes is updated as follows.

A promising feedback for gene i

$$P_j(n+1) = P_j(n) - \Psi_j\left[P_j(n)\right]; \quad \forall j : j \neq i \tag{8}$$

but

$$\Psi_j\left[p_j(n)\right] = \alpha p_j(n) \tag{9}$$

so

$$p_j(n+1) = p_j(n) - \alpha p_j(n) = (1-\alpha)\, p_j(n) \tag{10}$$

and

$$\begin{aligned} p_i(n+1) &= p_i(n) + \sum_{j=1, j\neq 1}^{r} \Psi_j\left[p_j(n)\right] \\ &= p_i(n) + \sum_{j=1, j\neq i} \alpha p_j(n) \\ &= p_i(n) + \alpha \sum p_j(n) \\ &\qquad \text{//all values except j = i so} \ldots \\ &= p_i(n) + \alpha\left(1 - p_i(n)\right) \end{aligned} \tag{11}$$

An inappropriate feedback for gene i

$$p_j(n+1) = p_j(n) + \Gamma_j\left[p_j(n)\right] \tag{12}$$

but

$$\Gamma_j\left(p_j(n)\right) = \frac{\beta}{m-1} - b p_j(n) \tag{13}$$

so

$$\begin{aligned} p_j(n+1) &= p_j(n) + \frac{\beta}{m-1} - \beta p_j(n) \\ &= \frac{\beta}{m-1} + (1-\beta)\, p_j(n) \end{aligned} \tag{14}$$

and

$$\begin{aligned} p_i(n+1) &= p_i(n) - \sum_{j=1, j\neq i}^{r} \Gamma_j\left[p_j(n)\right] \\ &= p_i(n) - \sum\left(\frac{\beta}{r-1} - \beta p_j(n)\right) \\ &= (1-\beta)\, p_i(n) \quad 0 \leq \beta < 1 \end{aligned} \tag{15}$$

These probabilities cause good genes to find their place in the chromosome over time. The main intention of these probabilities is to exploit the previous behavior of the system in order to make decisions for the future, hence, learning occurs. To perform learning, the number of internal states of each gene must be determined. These internal states for each gene maintain the number of failures and successes. How to select a gene from a chromosome is based on calculating the cumulative probability of each gene as follows (p_i is gene probability):

$$\begin{aligned} q_1 &= p_1, \\ q_2 &= p_1 + p_2, \\ &\vdots \\ q_n &= p_1 + p_2 + \ldots + p_n \end{aligned} \tag{16}$$

A random number is generated between 0 and 1. If generated random number is higher than q_i and lower than q_j, the corresponding gene to q_i is selected to evaluate for reward and penalty. Obviously, genes with higher probabilities are preferred for reward or penalty.

In each iteration step of the algorithm, a gene of each chromosome is selected based on its probability ((8), (11), (12), and (15)) and this gene is evaluated using Algorithm 5.

Generally, a gene is penalized or rewarded as follows.

Suppose that a chromosome includes R genes ($\{\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_R\}$) and has RN internal states ($\varphi_1, \varphi_2, \ldots, \varphi_{RN}$). Internal states of $\varphi_1, \varphi_2, \ldots, \varphi_N$ are related to $\alpha_1, \varphi_{N+1}, \varphi_{N+2}, \ldots, \varphi_{2N}$

(1) Select a gene of a chromosome according to its probability (Eqs. (8), (11), (12), and (15))
// a gene of a chromosome indicates a vertex in the task graph
(2) Compute gene's EST (indicated by CEST)
(3) Compute EST of the previous task of the gene (indicated by PEST)
(4) If (PEST is lesser than CEST)
(4.1) The gene will be rewarded
(5) else
(5.1) The gene will be penalized

Algorithm 5: Evaluating a selected gene from a chromosome for performing reward and penalty.

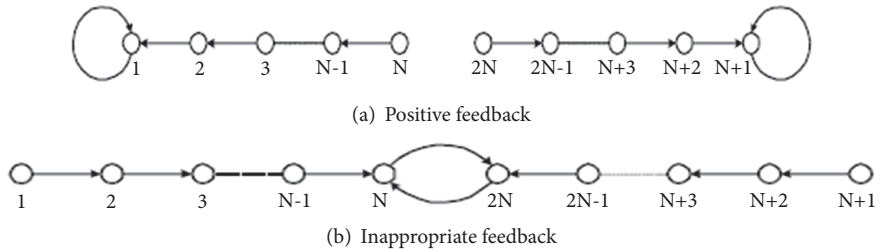

Figure 4: State transition for reward and penalty.

are related to α_2, and $\varphi_{(R-1)N+1}, \varphi_{(R-1)N+2}, \ldots, \varphi_{RN}$ are related to α_R. Therefore,

$$G[\varphi_i] = \begin{cases} a_1 & i = 1, 2, \ldots, N \\ a_2 & i = N+1, \ldots, 2N \\ \cdots & \\ a_{R-1} & \\ a_R & i = (R-1)N+1, \ldots, RN \end{cases} \quad (17)$$

so that G is a mapping function.

If gene α_1 is in the internal state φ_i, $(1 \le i \le N)$ on the chromosome and this gene gets a penalty ($\beta = 1$) then the internal state changes as follows:

$$\varphi_i \longrightarrow \varphi_{i+1} \quad (i = 1, 2, \ldots, N-1)$$
$$\varphi_N = \varphi_{2N} \quad (18)$$

If the gene gets the reward ($\beta = 0$) then the internal state changes as follows:

$$\varphi_i \longrightarrow \varphi_{i-1} \quad (i = 1, 2, \ldots, N)$$
$$\varphi_1 = \varphi_1 \quad (19)$$

If gene α_2 is in internal state φ_i, $(N+1 \le i \le 2N)$ and it gets a penalty then the internal state changes as follows:

$$\varphi_i \longrightarrow \varphi_{i+1} \quad (i = N+1, N+2, \ldots, 2N-1)$$
$$\varphi_{2N} = \varphi_N \quad (20)$$

If the gene gets the reward then transferring of state performs as follows:

$$\varphi_i \longrightarrow \varphi_{i-1} \quad (i = N+2, N+3, \ldots, 2N)$$
$$\varphi_{N+1} = \varphi_{N+1} \quad (21)$$

In Figure 4, the internal states of each gene are considered to be N. In this figure, N and 2N indicate the boundary states of the first gene and second gene, respectively. After evaluating a gene, if it gets a positive feedback, it moves to internal states, and if an inappropriate feedback is taken, it moves to a boundary state. If a gene reaches boundary state and gets an undesirable feedback, then it is displaced with another gene in the chromosome, so that the overall quality of the chromosome is improved. To perform this, the neighbors of a gene are generated and then from these neighbors the algorithm searches for a processor in the chromosome for displacement so that makespan value in that chromosome is lower than others. If the makespan value of new chromosomes generated is more than the initial chromosome, it remains the same initial chromosome. Indeed, the algorithm continues until there no better solution is obtained. The concept of neighbor scheduling is illustrated in Figure 5. In this figure, A shows the current scheduling and B, C, and D are three neighbor scheduling for that. For example, consider the task *a*, so that, in B, it moved into processor 4, in C, it moved into processor 2, and in C, it moved into processor 3.

In the learning process, several percents of the genes from a selected chromosome are chosen and each chromosome selected is regularly improved by this algorithm (Algorithm 6). The proposed algorithm considers a threshold β for determining the number of neighbors for searching. Such threshold could be any percentage in the range of $0\% \le \beta \le 100\%$. Using a low value for β reduces the algorithm

Input: getting a current scheduling
Output: the best neighbor of a scheduling
Step 1- Compute the total number of neighbors for current scheduling using Eq. (22), and determine the threshold t. This threshold determines how many percents of the current scheduling neighbors must be assessed to find the next scheduling.
Total number of neighbors = total number of processors in scheduling × the total number of tasks in the scheduling × t (22)
Step 2- **Searching for the best neighbor for current scheduling:**
- In the NAHC: algorithm generates a random neighbor scheduling for a selected gene and examine it. If the quality of the neighbor scheduling is lower than the current scheduling, another neighbor scheduling is generated for the current scheduling, and it repeats this action until a higher-quality neighbor is found.
- In the SAHC: algorithm generates all neighbors for a selected gene and examine all of them, then, the best neighbor scheduling is returned.

Algorithm 6: Next Ascent and Steepest Ascent Algorithms.

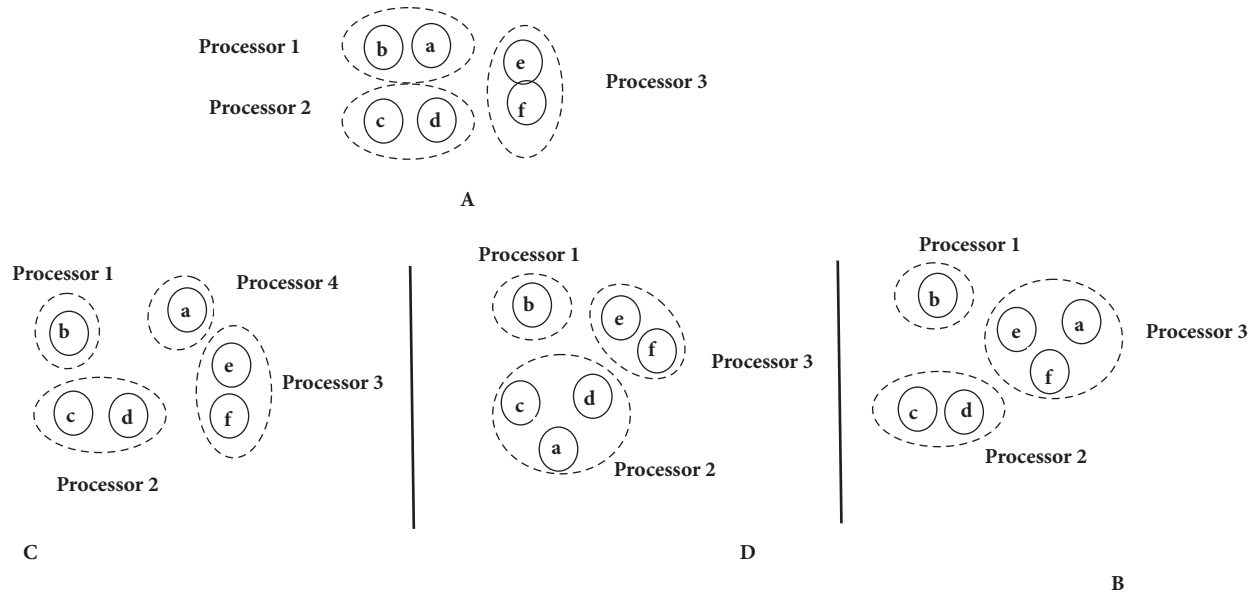

Figure 5: Neighborhood for task a.

steps prior to converging, and using a high value for β increases the algorithm steps prior to converging. However, it is reasonable that a high threshold can often produce a good result. Based on β value, we propose to types of learning schema named Steepest Ascent Learning Criterion (SALC) and Next Ascent Learning Criterion (NALC). In Next Ascent Learning Criterion β is equal to zero. This means the algorithm stops searching a scheduling when the first scheduling found with a lower makespan (i.e., high-quality solution); and in the Steepest Ascent Learning Criterion, β is equal to 100%. This causes the algorithm to generate all the neighbors scheduling and examine all of them and choose the one with the lowest makespan.

Theorem 5. *The time complexity of LAGA algorithm is O(MaxGen×PopSize× N×V).*

Proof. To calculate the time complexity of the algorithm, first, we consider the following assumptions:

PopSize: the number of individuals in the population

MaxGen: The number of generation

I: Number of iterations for reward and penalty operators

V: the number of tasks in the task graph

N: the number neighbors for a selected gene

#P: the number of processors

#S: the number of idle slots in a processor

v': the number of tasks allocated into a processor

To calculate time complexity for a chromosome, it is necessary to calculate the completion time for all processors and select the maximum time from them as time completion of the chromosome. Thus, the time complexity to calculate the completion time for a chromosome is O(V). The rewarding operation time complexity is O(1), because only the internal state of a gene changes. But, the penalizing operation time complexity is O(N×V), because it is necessary to find its neighbors and to compute them completion times and then select a neighbor with minimum completion time.

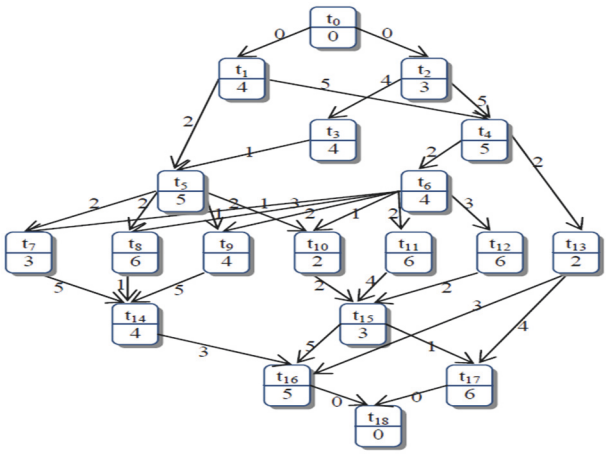

FIGURE 6: A sample task graph (from [8]).

Creating the initial population can be accomplished in linear time, thus, the main computational cost lies in Step 2 (Algorithm 1). The computational complexity of learning schema is O(N×V). The computational complexity of extending a chromosome to classic chromosome is O(PopSize×V). The computational complexity of learning is O(PopSize× N×V). The computational complexity of the selection operator is O(V). The computational complexity of crossover is O(PopSize×V), and that of mutation is O(PopSize×V). The computational complexity of reuse idle time heuristic is O(PopSize × #P × #S × v'logv'). Thus, the overall time complexity is O(V) + O(N×V) + O(N×V) + O(PopSize × #P × #S × v'logv') + O(V) + O(PopSize×V) + O(PopSize× N×V) + O(PopSize×V) + O(PopSize×V). Therefore, considering the number of generations, the time complexity of LAGA can be simplified as O(MaxGen×PopSize× N×V). □

4. Experimental Results

The task graphs used to support the findings of this study have been deposited in the [8, 21] and http://www.kasahara.cs.waseda.ac.jp/schedule/optim_pe.html. In the following, we compare the proposed algorithm with a number of other algorithms on these task graphs.

We, first, compare the LAGA with Standard Genetic Algorithm (denoted by SGA). In SGA, there is no depth and tasks are arranged based on the topological order in each chromosome. The result is obtained by applying LAGA to the task graph offered in [8, 21], where this graph is redrawn in Figure 6. The result indicates considerably improvement LAGA compared to SGA. The parameter setting for the experiments are as follows:

(i) Population size: 30 chromosomes,

(ii) Number of generations: 500,

(iii) Number of processors: 3,

(iv) Crossover probability: 80%,

(v) mutation probability: 5%,

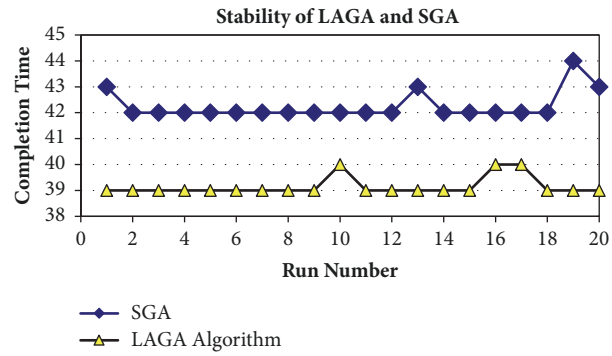

FIGURE 7: Comparison of the stability proposed algorithm and standard genetic algorithm.

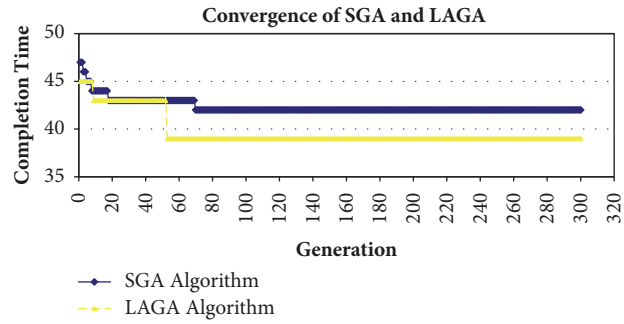

FIGURE 8: A Comparison of the convergence of the proposed and genetic scheduling algorithm.

(vi) Depth: 3,

(vii) β = 100%

(viii) Reward and penalize probability: 50%.

Table 3 displays the scheduling and completion time obtained for the task graph shown in Figure 6 for both LAGA and SGA algorithms. Also, the stability of the proposed algorithm is evaluated. Stability is a critical issue in evolutionary and heuristic algorithms. It means that the algorithm gives the same or close solutions in different runs. To demonstrate the stability of the LAGA, the results obtained by twenty runs of the algorithm to schedule the task graph shown in Figure 7.

The progression of the chromosomes, while applying LAGA and SGA algorithms, is shown in Figure 8.

Comparison of the Proposed Algorithm with Other Algorithms. Most existing genetic algorithms and other heuristic algorithms have been evaluated on graphs which are randomly produced and do not communication cost. Hence, we select the algorithms for comparison for which their results on the certain graph are known. Therefore, in this section, the LAGA is compared with nine different algorithms on the graphs which are used by them.

In Figure 9, LAGA is compared against Parsa algorithm and six other scheduling algorithms. The comparison is performed by applying LAGA to the task graph drawn in Figure 6 [8, 21] and is used by Parsa and other six algorithms. The results demonstrate that the scheduling obtained by

Table 3: The completion times for task graph shown in Figure 4 with LAGA and SGA algorithms.

(a) Scheduling produced by LAGA

Task Number	**0**	**1**	**2**	**4**	**3**	**6**	**5**	**11**	**9**	**12**	**7**	**8**	**10**	**15**	**13**	**14**	**17**	**16**	**18**
Processor Number	2	2	2	0	2	1	2	1	2	1	0	0	2	0	1	0	1	0	1
Depth	0	0	1	2	2	2	2	1	1	2	1	2	2	2	1	1	1	1	0

(b) Scheduling produced by SGA

Task Number	**0**	**1**	**2**	**4**	**3**	**6**	**5**	**11**	**9**	**12**	**7**	**8**	**10**	**15**	**13**	**14**	**17**	**16**	**18**
Processor Number	2	1	0	2	1	2	1	0	0	2	0	1	2	0	2	1	1	0	2

(c) Scheduling results

Algorithm	LAGA	SGA
Completion time	39	42

Table 4: Fast Fourier Transform Task Graphs (FFT).

Task or node#	**FFT1**		**FFT2**		**FFT4**	
1-8	1	25	60	25	20	5
9-12	20	25	50	25	20	600
13-16	30	25	5	25	30	5
17-20	20	25	5	25	20	5
21-28	1	25	5	25	5	5

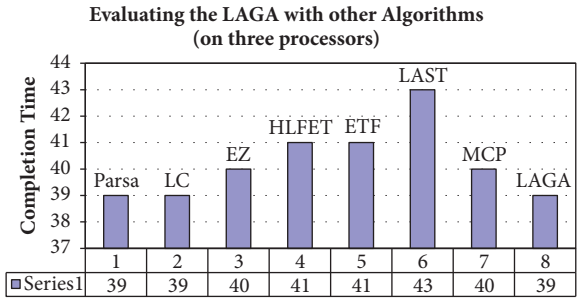

Figure 9: Comparison of the completion times on task graph Figure 6.

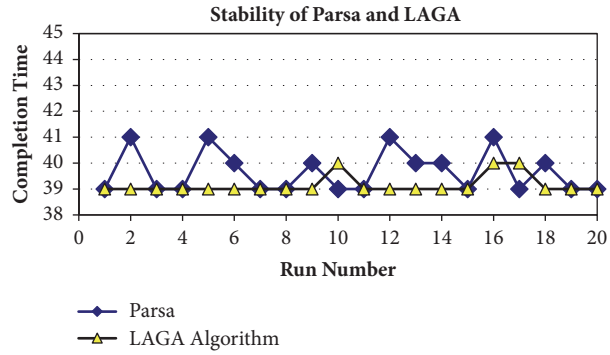

Figure 10: Comparison of the stability proposed algorithm and Parsa algorithm.

LAGA outperforms the algorithms 3–7 and is the same with Parsa and LC algorithms. However, LAGA obtains the result very swift and stable. In addition, LAGA runs faster when the number of tasks grows. Furthermore, Parsa algorithm is evaluated using small problem sizes, and it is not clear their performance on large-scale problem size. In Figure 10, the stability of LAGA is compared with Parsa algorithm.

Fast Fourier Transform Task Graphs (FFT), depicted in Figure 11, are good benchmarks to assess the performance of the scheduling algorithms. Table 4 shows three versions of these graphs (i.e., FFT1, FFT2, and FFT4). The first column in each of these cases shows the computation time of each task and the second column shows the communication time of each task with others.

According to Tables 5 and 6, it has been found that LAGA with $\beta = 100\%$ always outperforms the genetic and nongenetic based algorithms. Because, considering high-speed of LAGA, at the same time with other algorithms, it can search more state-space of the problem and, as a result, can find the answers with higher quality.

Usually, the efficiency of most algorithms presented for task's graph scheduling problem is studied on graphs with small size and low communications and their results on graphs with real applications which are graphs with large size and high communications are not clear. The LAGA ($\beta = 100\%$) is applied to specific benchmark application programs which are taken from *a Standard Task Graph (STG)* archive (http://www.kasahara.cs.waseda.ac.jp/schedule/optim_pe.html) (see Table 7). The first program of this STG set, Task Graph 1, is a randomly generated task graph, the second program is a robot control program (Task Graph 2) which is an actual application program, and the last program is a sparse matrix solver (Task Graph 3). Also, we considered the task graphs with random

Table 5: Comparison of LAGA with well-known genetic-based algorithms on FFT benchmarks.

Graph	Sequential Time	CGL [9]	BCGA [10]	Parsa [8]	LAGA $\beta = 0\%$	LAGA $\beta = 100\%$	Speedup*	improvement%**
FFT 1	296	152	124	124	124	124	2.38	0%
FFT 2	760	270	240	240	240	200	3.80	16%
FFT 4	480	260	255	185	255	170	2.82	8%

*Speedup = sequential time / LAGA ($\beta = 100\%$) completion time.
**improvement = ((Parsa completion time - LAGA ($\beta = 100\%$) completion time)/Parsa completion time) × 100.

Table 6: Comparison of LAGA with well-known nongenetic algorithms considering the number of processors on FFT benchmarks (completion time/#processor used).

Graph	Sequential Time	DSC	MCP	LC	LAST	LAGA $\beta = 100\%$
FFT 1	296 / 1	124 / 4	148 / 8	172 / 8	146 / 4	124 / 4
						128 / 8
FFT 2	760 / 1	205 / 8	205 / 8	225 / 8	240 / 8	190 / 8
						240 / 2
FFT 4	480 / 1	710 / 12	710 / 8	710 / 8	710 / 8	185 / 4
						185 / 8
						160 / 12

Table 7: Three selected task graphs.

Task graphs	Number of tasks
Task Graph 1	100
Task Graph 2	90
Task Graph 3	98

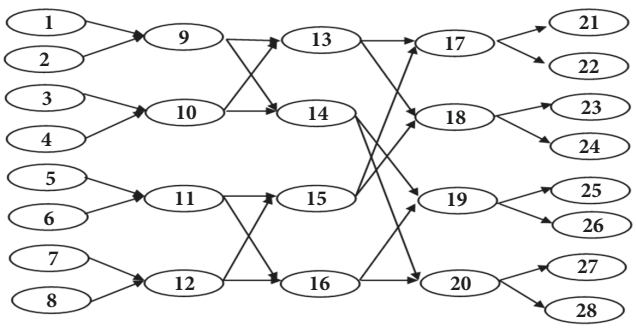

Figure 11: Fast Fourier Transform (FFT) Task Graph (from [8]).

communication costs. These communication costs are distributed uniformly between 1 and a specified maximum communication delay.

Table 8 shows the results of comparing among SGA and LAGA (on depth 3 and 5), Parsa, CPGA, and TDGA algorithms on 4 processors for programs listed in Table 7. According to Table 8, the proposed algorithm (LAGA) with depth=3 produces schedule length lower than other algorithms.

Table 9 shows a comparison of three genetic-based algorithms with LAGA on a number of randomly generated graphs. The task execution time and communication time between processors are considered random. The task execution time and communication time are distributed uniformly within (10, 50) and (1, 10) second, respectively. To evaluate the efficiency of the algorithms, we set the running time of the algorithms equally. As can be seen in this table, the LAGA, in all cases and at the same time with them, outperforms better than the rest. This demonstrates that the LAGA is able to search more space than the rest at the same time, and as a result it is efficient.

5. Conclusions

The task graph scheduling problem is an NP-Hard problem. Hence, to solve this problem, evolutionary approaches are pretty effective. But, their convergence rate when searching for global optima is still not sufficient. For this reason, there is a need for methods that able to improve the convergence speed, to achieve a good scheduling. In this paper, we combined genetic algorithm with an accelerated convergence rate strategy, aiming to achieve higher performance in parallel environments. We have proposed a learning-based genetic algorithm for solving the problem which is to better utilize the processors. Considering high-speed of the proposed approach, the proposed approach can search more state-space of the problem at the same time with other algorithms, as a result, it can find the solution with higher quality.

5.1. Future Works. A number of directions can be explored to extend and improve this work.

(1) Finding a useful and effective a reward and penalty function;

(2) Using clustering in the scheduling of task graph, instead of, using the critical path;

Table 8: Comparison of the genetic-based algorithms.

Benchmarks programs	SGA	LAGA (depth=3)	LAGA (depth=5)	Parsa	CPGA [11]	TDGA [11]
Task Graph 1	256	194	231	230	210	201
Task Graph 2	995	635	650	800	690	650
Task Graph 3	332	190	221	322	322	221

Table 9: Comparison of the three genetic-based algorithms with LAGA on four processors.

DAG	# of tasks	# of edges	Run Time (minute)	SGA	CPGA	Parsa	TDGA	LAGA (depth=3)
Random graph 1	200	420	60	1241	1108	1206	1108	998
Random graph 2	400	850	60	2209	1943	1954	1829	1458
Random graph 3	600	882	90	4592	4320	4832	4001	3765
Random graph 4	800	1232	240	8092	7809	7994	7031	6895
Random graph 5	1000	1891	240	17431	16490	15923	14134	11290

(3) Finding a useful and effective load balancing heuristic for the proposed algorithm.

(4) The parameter tuning is one of the major drawbacks of the genetic algorithm. Recently, for solving NP-Hard problems, using optimization algorithms without parameter tuning, such as Beetle Antennae Search Algorithm [42], has attracted a lot of attention. In the future work, we intend to use this algorithm for task graph scheduling problem.

Also, the proposed method may be extended to solving scheduling in grid computing, cloud computing, and heterogeneous environment; and also it is possible to use our approach to solve other NP-Hard problems.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The author declares that they have no conflicts of interest.

References

[1] M. Hall, D. Padua, and K. Pingali, "Compiler research: The next 50 years," *Communications of the ACM*, vol. 52, no. 2, pp. 60–67, 2009.

[2] Y.-K. Kwok and I. Ahmad, "On multiprocessor task scheduling using efficient state space search approaches," *Journal of Parallel and Distributed Computing*, vol. 65, no. 12, pp. 1515–1532, 2005.

[3] A. S. Wu, H. Yu, Sh. Jin, K. Ch, and G. Schiavone, "An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 9, pp. 824–834, 2004.

[4] X. Lin, Y. Wang, Q. Xie, and M. Pedram, "Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 175–186, 2014.

[5] Z. Tang, L. Qi, Z. Cheng, K. Li, S. U. Khan, and K. Li, "An Energy-Efficient Task Scheduling Algorithm in DVFS-enabled Cloud Environment," *Journal of Grid Computing*, vol. 14, no. 1, pp. 55–74, 2016.

[6] S. K. Panda and P. K. Jana, "Efficient task scheduling algorithms for heterogeneous multi-cloud environment," *The Journal of Supercomputing*, vol. 71, no. 4, pp. 1505–1533, 2015.

[7] B. Keshanchi, A. Souri, and N. J. Navimipour, "An improved genetic algorithm for task scheduling in the cloud environments using the priority queues: Formal verification, simulation, and statistical testing," *The Journal of Systems and Software*, vol. 124, pp. 1–21, 2017.

[8] S. Parsa, S. Lotfi, and N. Lotfi, "An Evolutionary approach to Task Graph Scheduling," *LNCS – Springer Link Journal*, 2007.

[9] E. S. H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 2, pp. 113–120, 1994.

[10] M. Rinehart, V. Kianzad, and S. Bhattacharyya, "A Modular Genetic Algorithm for Scheduling Task Graphs," Tech. Rep., 2003.

[11] F. A. Omara and M. M. Arafa, "Genetic algorithms for task scheduling problem," *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 13–22, 2010.

[12] Y.-K. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381–422, 1999.

[13] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2014.

[14] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[15] H. Topcuoglu, S. Hariri, and M. Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Proceedings of the Eighth Heterogeneous Computing Workshop (HCW'99)*, pp. 3–14, IEEE, 1999.

[16] S. Ranaweera and D. P. Agrawal, "A scalable task duplication based scheduling algorithm for heterogeneous systems," in

Proceedings of the 2000 International Conference on Parallel Processing, IEEE, 2000.

[17] S. Baskiyar and P. C. SaiRanga, "Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length," in *Proceedings of the 2003 International Conference on Parallel Processing Workshops, ICPPW 2003*, pp. 97–103, IEEE, Taiwan, October 2003.

[18] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "Scheduling Precedence Graphs in Systems with Inter-processor Communication Times," *SIAM Journal on Computer*, vol. 18, no. 2, pp. 244–257, 1989.

[19] T. Yang and A. Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951–967, 1994.

[20] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, 1993.

[21] J. Baxter and J. H. Patel, "The LAST algorithm: A heuristic-based static task allocation algorithm," in *Proceedings of the 1989 International Conference on Parallel Processing*, vol. 2, pp. 217–222, August 1989.

[22] B. Kruatrachue and T. G. Lewis, "Duplication Scheduling Heuristics (DSH): A New Precedence Task Scheduler for Parallel Processor Systems," Tech. Rep., Oregon State University, Corvallis, OR, USA, 1987.

[23] M. Wu and D. D. Gajski, "Hypertool: a programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330–343, 1990.

[24] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, USA, 1989.

[25] S. J. Kim and J. C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architecture," in *Proceedings of the International Conference on Parallel Processing*, vol. 2, pp. 1–8, 1988.

[26] R. Rajak, C. P. Katti, and N. Rajak, "A Modified Task Scheduling Algorithm of Task Graph without Communication Time," *International Journal of New Computer Architectures and their Applications (IJNCAA)*, vol. 3, no. 4, pp. 88–93, 2013.

[27] A. Radulescu and A. J. C. Van Gemund, "Low-cost task scheduling for distributed-memory machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 6, pp. 648–658, 2002.

[28] M. I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 399–409, 2008.

[29] N. A. Bahnasawy, M. A. Koutb, M. Mosa, and F. Omara, "A new algorithm for static task scheduling for heterogeneous distributed computing systems," *African Journal of Mathematics and Computer Science Research*, vol. 3, no. 6, pp. 221–234, 2011.

[30] A. A. Nasr, N. A. El-Bahnasawy, and A. El-Sayed, "Task scheduling algorithm for high performance heterogeneous distributed computing systems," *International Journal of Computer Applications*, vol. 110, no. 16, pp. 23–29, 2015.

[31] Y. Wen, H. Xu, and J. Yang, "A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system," *Information Sciences*, vol. 181, no. 3, pp. 567–581, 2011.

[32] P. Hansen and N. Mladenović, "Variable neighborhood search: principles and applications," *European Journal of Operational Research*, vol. 130, no. 3, pp. 449–467, 2001.

[33] P. Chitra, R. Rajaram, and P. Venkatesh, "Application and comparison of hybrid evolutionary multiobjective optimization algorithms for solving task scheduling problem on heterogeneous systems," *Applied Soft Computing*, vol. 11, no. 2, pp. 2725–2734, 2011.

[34] R. Rajak, "A Novel Approach for Task Scheduling in Multiprocessor System," *International Journal of Computer Applications*, vol. 44, no. 11, pp. 12–16, 2012.

[35] M. Akbari, H. Rashidi, and S. H. Alizadeh, "An enhanced genetic algorithm with new operators for task scheduling in heterogeneous computing systems," *Engineering Applications of Artificial Intelligence*, vol. 61, pp. 35–46, 2017.

[36] M. Akbari and H. Rashidi, "A multi-objectives scheduling algorithm based on cuckoo optimization for task allocation problem at compile time in heterogeneous systems," *Expert Systems with Applications*, vol. 60, pp. 234–248, 2016.

[37] H. M. Ghader, D. KeyKhosravi, and A. HosseinAliPour, "DAG scheduling on heterogeneous distributed systems using learning automata," in *Proceedings of the Asian Conference on Intelligent Information and Database Systems*, pp. 247–257, Springer, Berlin, Heidelberg, 2010.

[38] T. Davidović and T. G. Crainic, "Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems," *Computers & Operations Research*, vol. 33, no. 8, pp. 2155–2177, 2006.

[39] K. S. Shin, M. J. Cha, M. S. Jang, J. Jung, W. Yoon, and S. Choi, "Task scheduling algorithm using minimized duplications in homogeneous systems," *Journal of Parallel and Distributed Computing*, vol. 68, no. 8, pp. 1146–1156, 2008.

[40] Z. Jovanovic and S. Maric, "A heuristic algorithm for dynamic task scheduling in highly parallel computing systems," *Future Generation Computer Systems*, vol. 17, no. 6, pp. 721–732, 2001.

[41] I. J. Shapiro, *The Use of Stochastic Automata in Adaptive Control [Ph.D. Thesis]*, Dep. Eng. and Applied Sci., Yale Univ., New Haven, Conn., USA, 1969.

[42] X. Jiang and S. Li, "BAS: beetle antennae search algorithm for optimization problems," 2017, https://arxiv.org/abs/1710.10724.