Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

An incremental ant colony optimization based approach to task assignment to processors for multiprocessor scheduling*

Hamid Reza BOVEIRI

(*Sama Technical and Vocational Training College, Islamic Azad University, Shoushtar Branch, Shoushtar, Iran*)

E-mail: boveiri@shoushtar-samacollege.ir

Received Nov. 10, 2015; Revision accepted Feb. 16, 2016; Crosschecked Mar. 28, 2017

Abstract: Optimized task scheduling is one of the most important challenges to achieve high performance in multiprocessor environments such as parallel and distributed systems. Most introduced task-scheduling algorithms are based on the so-called list scheduling technique. The basic idea behind list scheduling is to prepare a sequence of nodes in the form of a list for scheduling by assigning them some priority measurements, and then repeatedly removing the node with the highest priority from the list and allocating it to the processor providing the earliest start time (EST). Therefore, it can be inferred that the makespans obtained are dominated by two major factors: (1) which order of tasks should be selected (sequence subproblem); (2) how the selected order should be assigned to the processors (assignment subproblem). A number of good approaches for overcoming the task sequence dilemma have been proposed in the literature, while the task assignment problem has not been studied much. The results of this study prove that assigning tasks to the processors using the traditional EST method is not optimum; in addition, a novel approach based on the ant colony optimization algorithm is introduced, which can find far better solutions.

Key words: Ant colony optimization; List scheduling; Multiprocessor task graph scheduling; Parallel and distributed systems
http://dx.doi.org/10.1631/FITEE.1500394 **CLC number:** TP301

1 Introduction

Today, due to increase in time complexity of software and decrease in hardware costs, the utilization and development of multiprocessor environments such as parallel and distributed systems have been raised a lot. In such systems, each program to be executed is decomposed into smaller segments named tasks. Tasks of a program are not necessarily independent. Some need the data generated by other tasks; hence, there are precedence constraints among them. To formulate, the problem is modeled using a directed acyclic graph (DAG), the so-called task graph. In a task graph, nodes are tasks and edges indicate the precedence constraints among them. In static scheduling, required execution times of the tasks, communication costs, and precedence constraints are specified during the program's compiling step. Tasks must be mapped into a predefined number of identical (homogeneous) processors with respect to their precedence constraints in such a way that the overall finish time of the program, termed 'makespan', is minimized.

The multiprocessor task scheduling problem is NP-hard in terms of time complexity (Chrétienne *et al.*, 1995), so achieving the best possible solution is generally very time consuming, and impossible for large-scale samples. Therefore, applying heuristic and meta-heuristic approaches to solve this problem is a rational idea. Most of the task scheduling approaches introduced in the literature, such as Highest Level First with Estimated Time (HLFET) (Adam *et al.*, 1974), Insertion Scheduling Heuristic (ISH) (Kruatrachue and Lewis, 1987), CLANS (which uses the cluster-like CLANs to partition the given task graph) (McCreary and Gill, 1989), Localized Allocation of

* Project supported by Sama Technical and Vocational Training College, Islamic Azad University, Shoushtar Branch, Shoushtar, Iran
ORCID: Hamid Reza BOVEIRI, http://www.orcid.org/0000-0002-0278-3649

Static Tasks (LAST) (Baxter and Patel, 1989), Earliest Time First (ETF) (Hwang *et al*., 1989), Dynamic Level Scheduling (DLS) (Sih and Lee, 1993), and Modified Critical Path (MCP) (Wu and Gajski, 1990), are based on the list-scheduling technique. That is, these approaches make a list of ready tasks at each stage and assign them some priorities. The ready tasks are either those without any parents or any unscheduled ones. Then, the task with the highest priority in the ready list is assigned to the processor that allows the earliest start time (EST), until all the tasks in the task graph are scheduled.

The makespans achieved by such methods are dominated by two major factors: (1) which order of tasks should be selected (sequence subproblem); (2) how the selected order should be assigned to the processors (assignment subproblem). There are a number of good approaches for overcoming the task sequence dilemma, such as the one introduced by Boveiri (2010), while the task assignment problem has not been studied much. Although a number of priority measurements have been proposed as heuristic values to tackle the task sequence dilemma, there is no definite measure to solve the task assignment problem. Therefore, the only way remaining is to use meta-heuristic algorithms. Boveiri (2014; 2015) was the first to explicitly present the problem and introduce an approach based on cellular learning automata (CLA), though with some drawbacks; the results achieved were slightly poor for some definite types of input graphs. In this study, in addition to proving that assigning tasks to the processors using the traditional EST method is not optimum, a novel approach based on ant colony optimization (ACO) is proposed, which can find far better solutions.

ACO is a meta-heuristic approach simulating the social behavior of real ants. Ants always search for the shortest path from the nest to food, and vice versa. In the same way, their artificial counterparts try to search for the shortest solution of the given problem. Dorigo *et al*. (1991) was the first to apply the ant system, as a multiagent parallel algorithm, to solve the traveling salesman problem, and so far it has been successfully used to solve a large number of difficult discrete optimization problems and dilemmas (Dorigo *et al*., 1999). We believe that ACO is one of the best methods to cope with such types of problems presented in the form of a graph because of the following reason: to find the shortest path, ants have an indirect local communication called stigmergy. The stigmergy, as shown in the following sections, makes ACO fast and efficient.

2 Multiprocessor task scheduling

The multiprocessor task-scheduling problem is formulated using a directed acyclic graph (DAG) such as $G=(N, E, W, C)$, also the so-called task graph, where $N=\{n_1, n_2, \ldots, n_n\}$, $E=\{(n_i, n_j)|n_i, n_j\in N\}$, $W=\{w_1, w_2, \ldots, w_n\}$, $C=\{c(n_i, n_j)|(n_i, n_j)\in E\}$, and n are the set of nodes, set of edges, set of weights of nodes, set of weights of edges, and number of nodes, respectively.

Fig. 1 shows a task graph of a program comprising nine tasks. In such a graph, nodes are tasks, and edges define precedence constraints among them. Each edge such as $(n_i, n_j)\in E$ shows that task n_i has to be finished before task n_j will be able to begin. In this case, n_i is called a parent, and n_j is called a child. Nodes without any parents and nodes without any children are called 'entry nodes' and 'exit nodes', respectively. Each node weight, e.g., w_i, specifies the execution time necessary to complete task n_i, and each weight of edge, e.g., $c(n_i, n_j)$, denotes the required time to transmit data from task n_i to task n_j, identified as the communication cost. Note that if two dependent tasks (both a parent and a child) are executed on the same processor, the communication cost would be zero between them because the data is already available in the processor's memory. In static scheduling, execution times of tasks and their precedence constraints are generated during the program's compiling stage. Tasks must be mapped into the given m processor elements according to their precedence in such a way that the overall finish time of the tasks would be minimized.

Most scheduling algorithms work based on the so-called list scheduling technique. The basic concept behind this technique is to prepare a sequence of tasks in the form of a ready list by assigning them some priority measurements, and then repeatedly removing the node (task) with the highest priority from the ready list, and allocating it to the processor that provides the EST, until all the nodes in the task graph are scheduled and the final scheduling is produced.

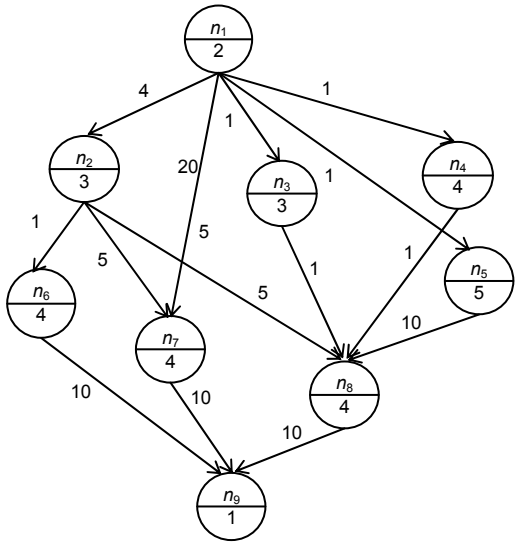

Fig. 1 A program task graph with nine tasks (Hwang *et al*., 2008)

If all the parents of a task such as n_i are scheduled on the same processor such as p_j, EST(n_i, p_j) would be Avail(p_j), that is, the earliest time when p_j is ready to execute the next task. Otherwise, the EST of task n_i on processor p_j is calculated using

$$\begin{aligned}&\mathrm{EST}(n_i, p_j)\\&=\max\left(\mathrm{Avail}(p_j), \max_{j\in\mathrm{Parents}(n_i)}(\mathrm{FT}(n_m)+c(n_m,n_i))\right),\end{aligned} \tag{1}$$

where FT(n_m)=EST(n_m)+w_m is the actual finish time of task n_m, and Parents(n_i) is the set of all the parents of n_i. Ultimately, the total finish time of the whole parallel program (or makespan) is computed using

$$\text{makespan}=\max_{i=1,2,\ldots,n}(\mathrm{FT}(n_i)). \tag{2}$$

For a given task graph with n tasks, using its adjacency matrix, an efficient implementation of assigning all the tasks in the task graph to the given m identical processors has a time complexity of $O(mn^2)$ (Kwok and Ahmad, 1998).

Some measures frequently applicable as task priority are TLevel (top level), BLevel (bottom level), SLevel (static level), ALAP (as late as possible), and NOO (number of offspring). The TLevel or ASAP (as soon as possible) of a node such as n_i is the length of the longest path from an entry node to n_i, excluding n_i itself, where the length of a path is the sum of weights of all the nodes and edges along the path. The TLevel of each node in a task graph can be computed by traversing the graph in topological order and using

$$\mathrm{TLevel}(n_i)=\max_{j\in\mathrm{Parents}(n_i)}(\mathrm{TLevel}(n_j)+c(n_j,n_i)+w_j). \tag{3}$$

BLevel of a node such as n_i is the length of the longest path from n_i to an exit node. An exit node is defined as one to which no children belong. BLevel is calculated for each task by traversing the input graph in reverse topological order, as follows:

$$\mathrm{BLevel}(n_i)=\max_{j\in\mathrm{Children}(n_i)}(\mathrm{BLevel}(n_j)+c(n_i,n_j))+w_i, \tag{4}$$

where Children(n_i) is the set of all the children of n_i.

If the edges' weights are excluded during the calculation of the BLevel of a node, a new attribute named the static level, or simply SLevel, will be generated, which can be computed using

$$\mathrm{SLevel}(n_i)=\max_{j\in\mathrm{Children}(n_i)}(\mathrm{SLevel}(n_j))+w_i. \tag{5}$$

The ALAP start time of a node is a measure that defines how long a node's start time can be delayed without any increase in the overall schedule length of the whole graph. For node n_i, it can be computed as follows:

$$\mathrm{ALAP}(n_i)=\min_{j\in\mathrm{Children}(n_i)}(\mathrm{CPL},\mathrm{ALAP}(n_j)-c(n_i,n_j))-w_i, \tag{6}$$

where CPL, or the critical path length, is the length of the longest path from an entry node to an exit node existing in the given task graph.

Finally, the NOO of node n_i is simply the number of all its descendants (or offspring), which can be computed for each task in a task graph by

$$\mathrm{NOO}(n_i)=1+\mathrm{NOO}(n_j),\ \ \forall n_j\in\mathrm{Children}(n_i). \tag{7}$$

Note that each node would be tagged after visiting, to prevent recounting. Table 1 lists the above-mentioned measures for each node of the task graph in Fig. 1. In addition, a comprehensive list of the notations applied in this section is reviewed in Table 2.

Table 1 TLevel, BLevel, SLevel, ALAP, and NOO of each node in the task graph of Fig. 1

Node	TLevel	BLevel	SLevel	ALAP	NOO
n_1	0	37	12	0	8
n_2	6	23	8	14	4
n_3	3	23	8	14	3
n_4	3	20	9	17	2
n_5	3	30	10	7	2
n_6	10	15	5	22	1
n_7	22	15	5	22	1
n_8	18	15	5	22	1
n_9	36	1	1	36	0

Table 2 A comprehensive list of the notations applied to formulate the multiprocessor task graph-scheduling problem

Symbol	Description
G=(N, E, W, C)	A given task graph
N=$\{n_1, n_2, \ldots, n_n\}$	Set of tasks in the task graph
E=$\{(n_i, n_j)\|n_i, n_j \in N\}$	Set of edges (precedence constraints) among tasks in the task graph
W=$\{w_1, w_2, \ldots, w_n\}$	Set of the required execution times of the tasks
C=$\{c(n_i, n_j)\|(n_i, n_j) \in E\}$	Set of the communication costs (delays) among tasks in the task graph
n	Number of tasks in the task graph
Entry node	A node without any parents
Exit node	A node without any children
m	Number of available processors
ReadyList[ ]	Current set of the tasks ready to be scheduled considering precedence constraints among tasks
Avail(p_j)	Earliest time when p_j is ready to execute the next task
FT(n_m)	Actual finish time of task n_m
EST(n_m)	Earliest start time of task n_m
EST(n_i, p_j)	Earliest start time of task n_i on processor p_j
Parents(n_i)	Set of all the parents of n_i
Children(n_i)	Set of all the children of n_i
Makespan	Total finish time of a parallel program, or scheduling length

3 Cellular learning automata based task assigner

A cellular automata (CA) machine is a mathematical model for simulating systems consisting of a large number of simple agents with local interactions. These simple agents can act together to produce complex emergent global behaviors. Each cell of the CA machine can assume an available state from a finite set of valid states. All cells in a CA machine update their states synchronously at discrete steps according to a predefined number of local rules. The current state of each cell would depend on the previous state of the cell itself and its neighborhood (if any exist) (Wolfram, 1983), where different methods of neighboring can be defined.

On the other hand, a learning automata (LA) machine is an abstract model capable of interacting with any stochastic environment. Each cell in the LA machine is able to select and perform a finite set of actions on the environment (Narendra and Thathachar, 1974). The election of an action depends on the internal state of the cell, which is represented by an action probability vector. The stochastic environment is used to evaluate each chosen action of the LA machine and responds to it. The LA machine uses this feedback to reform itself for the subsequent situations. This interactive behavior is repeated until the LA machine learns to analyze and select the best action in each situation, where the best action is the one that has the highest probability of obtaining a reward from the environment.

The full potential of the LA machine is revealed when there are a large number of automata interacting with each other. The structure may be assumed to be in different forms, such as tree, mesh, and array. In Meybodi *et al.* (2004), CA and LA were combined, and a novel model named the CLA model was proposed, opening up a new unsupervised learning paradigm. CLA is better than a single CA because of its capability to learn; it is also superior to a single LA because it is a collection of LAs with an interaction behavior.

A variable-structure CLA model is represented by a tuple CLA=(Z^d, ϕ, A, N, F), where Z^d is a lattice of d-tuple of integers, ϕ=$\{1, 2, \ldots, m\}$ is a set of finite states, A is the set of LAs (each of which is assigned to one cell of CLA), $N=\{\overline{x}_1,\overline{x}_2,...,\overline{x}_{\overline{m}}\}$, $\overline{x} \in Z^d$ is a finite subset of Z^d, called the neighborhood vector, and $F:\varphi_{\overline{m}} \to \beta$ is the local rule of CLA, where β is the set of values that the reinforcement signal can take, or environmental feedback. Here, β can be a two-member set, with β=0 denoting a favorable

response and β=1 an unfavorable response from the environment.

In addition, each LA in CLA is represented by a 4-tuple like (α, β, p, T), where $\alpha=\{\alpha_1, \alpha_2, \ldots, \alpha_r\}$, $\beta=\{\beta_1, \beta_2, \ldots, \beta_m\}$, $\boldsymbol{p}=[p_1, p_2, \ldots, p_r]$, T, r, and m are the action set, input set (environment response), action probability vector, learning algorithm, number of permitted actions of each automaton, and number of inputs (reactions of the environment), respectively. The learning algorithm, in a recurrent manner, tries to accommodate the action probability vector $\boldsymbol{p}$ of each automaton existing in the LA based on the environmental feedback. Although various learning algorithms have been introduced in the literature, a fast reliable one called the linear learning algorithm is given as follows.

Let $\alpha_i(n)$ be the action selected by automaton LA^k at time instant n using its current action probability vector $\boldsymbol{p}^k(n)$. In the linear learning algorithm, if the environmental feedback is favorable, the action selected by LA^k will obtain a reward:

$$p_j^k(n+1)=\begin{cases} p_j^k(n)+a(1-p_j^k(n)), & j=i, \\ (1-a)p_j^k, & j\neq i. \end{cases} \tag{8}$$

Else, if the environmental response is unfavorable, the selected action will obtain a penalty by

$$p_j^k(n+1)=\begin{cases} (1-b)\ p_j^k(n), & j=i, \\ \dfrac{b}{r-1}+(1-b)p_j^k(n), & j\neq i, \end{cases} \tag{9}$$

where a and b are reward and penalty parameters, respectively, and should be tuned experimentally. If $a=b$, the algorithm is called $L_{R\text{-}P}$; if $b<a$, the algorithm will be $L_{R\text{-}\varepsilon P}$; if $b=0$, the algorithm will be named $L_{R\text{-}I}$.

Boveiri (2014; 2015) was the first to propose a CLA-based approach to solve the task assignment dilemma. In his proposed approach, each task in the task graph is represented by a CLA, so a graph of CLAs is generated, which is isomorphic to the given task graph. Then, an iterative algorithm is executed so that each iteration such as t results in a solution (a complete scheduling). Each iteration consists of n stages, where n is the number of tasks in the given task graph. In each stage such as k, CLAs with no parents (or without any unscheduled ones) are activated to be scheduled. Among them, the activated automaton with the highest priority selects one of the available processors to be executed based on its action probability vector; that is, the action permitted by any automaton is to choose a processor among the available ones. Finally, after activating all the CLAs, a solution that means a complete scheduling is obtained for iteration k. If the solution obtained during iteration k is better than the previous ones (or equal to the best previous solution), CLA will obtain a reward using Eq. (8), else a penalty based on Eq. (9).

Moreover, there is a local rule to enhance the results. If an automaton selects the processor on which all of its parents are scheduled, the automaton will obtain an additional reward, else a penalty. This rule is inspired from the fact that if all the parents of a task have already been executed on the same processor, this task can be started instantly without wasting any time; that is, no latency for the communication costs needs to be considered. The iteration continues until the solutions achieved converge to a unique scheduling, which is of course the best one. Fig. 2 shows the flowchart of the described approach.

The $L_{R\text{-}\varepsilon P}$ learning algorithm guarantees a convergence, and it has been used to educate CLA. The reward and penalty parameters were a=0.05 and b=0.01a, respectively, and the quit condition of the algorithm was 50 iterations with the same solution, and the maximum number of iterations (K) was set to 5000. The implementations reveal that the overall time complexity of the CLA-based approach is $O(K(mn+n^2))$, where n is the number of tasks in the task graph and m is the total number of available processors. Because K is a constant limited to 5000, for sufficiently large numbers of n and m, we can assume that the overall time complexity of the proposed approach is $O(mn+n^2)$.

4 Ant colony optimization

ACO is a parallel meta-heuristic approach, in which a colony of artificial ants acts together in a cooperative manner to search for the optimized solutions of a given problem. The ant system was first introduced by Dorigo *et al.* (1999) as a multiagent approach to solve the traveling salesman problem,

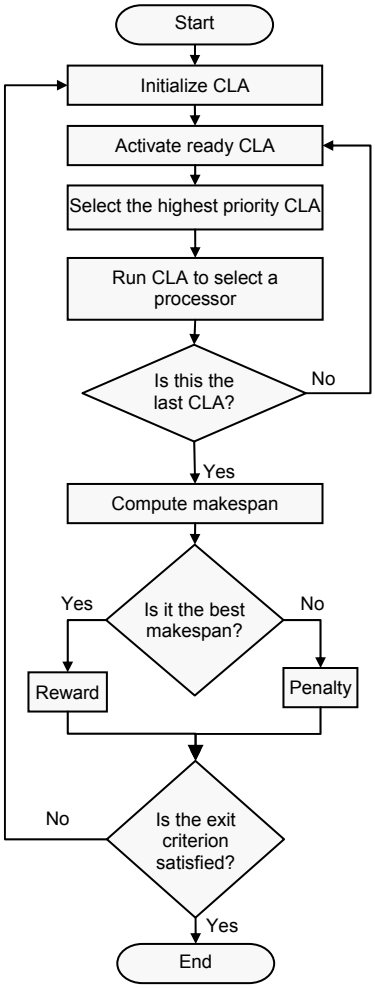

Fig. 2 Flowchart of the cellular learning automata based approach

and so far it has been successfully applied to a large number of difficult discrete optimization problems, such as the quadratic assignment problem, job shop and flow shop scheduling, vehicle routing, graph coloring and partitioning, sequential ordering, and network routing.

Leaving the nest, real ants have a completely stochastic behavior. As soon as they encounter a food source, while walking from the food to the nest, they deposit on the ground a chemical substance named a pheromone, forming in this way a pheromone trail. Ants are able to smell pheromone. Other ants are attracted by the pheromone existing in the environment, and eventually they discover the food source too. The more the pheromone deposited, the more the ants attracted, and the more the ants that will find the food. This cycle is a type of autocatalytic behavior. In this way (by the pheromone trails), ants have an indirect communication, which is locally accessible by the ants, so-called stigmergy, a powerful tool enabling them to be very fast and efficient. Last but not the least, note that the deposited pheromone is constantly evaporated by sunshine and environmental heat, a powerful mechanism to clear undesirable pheromone paths to the food sources without any remains left.

If an obstacle whose one side is longer than the other cuts a pheromone trail, at first, ants execute random motions to circle around the obstacle. Nevertheless, as the pheromone of the longer side is evaporated faster, little by little, the ants will converge to the shorter side, and thereby they always find the shortest path from the food to the nest, and vice versa.

ACO tries to simulate this foraging behavior. In the beginning, each state of the problem takes a numerical variable named pheromone trail, or simply pheromone. Initially, these variables have an identical and very small value. ACO is an iterative algorithm. In each iteration, one or more ants are generated. In fact, each artificial ant is just a list (or tabu list) showing the states visited by the ant. The generated ant is placed on the start state and then it selects the next state using a probabilistic decision-making based on the value of pheromone trails of the adjacent states. The ant repeats these operations until it reaches a final state. At this time, the values of the pheromone variables of the visited states are increased based on the desirability of the solution achieved (depositing pheromone). Finally, all the variables are decreased, simulating pheromone evaporation. Through this mechanism, ants converge to the more optimal solutions.

One factor that renders ACO more efficiently as compared to stochastic methods such as the genetic algorithm is, as mentioned earlier, stigmergy, which is an indirect local interaction among ants using the pheromone variable. In contrast to the genetic algorithm, in which decisions are often random and based on the mutation and crossover (much experience will also be eliminated by putting the weaker chromosomes aside), in ACO, all the decisions are purposeful and are based on the experience gathered by all the previous walking ants.

5 Proposed approach

At first, an $n \times m$ matrix named $\boldsymbol{\tau}$ is considered to represent the pheromone variables, where n is the number of tasks in the given task graph and m is the number of existing processors. Actually, τ_{ij} is the desirability of assigning task n_i to processor p_j. All the matrix elements initiate with the same and very small value. In addition, a valid task order of the given task graph is selected based on the priority measurements introduced in Section 2. Actually, each ant is a list of length n and has a novel encoding as follows. Each element in the ant list, such as ant[6]=2, demonstrates that task n_6 is assigned to processor p_2. To clarify the issue, we give a typical ant list filled by the proposed approach, along with its task order and corresponding scheduling on two processors, demonstrated by a Gantt chart (Fig. 3). The gaps between the tasks are due to the latencies inspired from the communication costs. Then, the iterative ant colony algorithm is executed. Each individual iteration has the following steps:

1. Generate an ant (or ants).
2. Loop for each ant (until the complete scheduling of all tasks in the selected task order): Assign the next task to the processors using a probabilistic decision-making based on the pheromone variables.
3. Deposit pheromone on the visited states.
4. Evaporate pheromone.
5. Daemon activities (to improve the results).

The flowchart of these operations with more details and an implementation in pseudo code are provided in Fig. 4 and Algorithm 1, respectively.

In the first stage, a list of length n is constructed as one ant and filled with the selected task order. Different task orders, using TLevel, BLevel, SLevel, ALAP, and NOO as priority measurements of nodes, need to be selected for a comprehensive evaluation and unbiased judgment.

In the second stage, there is a loop for each ant. In each iteration such as t, the active ant assigns the next node in the selected task order to the supposed suitable processor using a probabilistic decision making based on the current values of the pheromone variables. The desirability of assigning task n_i to processor p_j at time instant t is obtained as follows:

$$a_{ij}(t) = \tau_{ij}(t) \Big/ \sum_{l \in N(t)} \tau_{il}(t), \quad \forall j \in N(t), \tag{10}$$

where $\tau_{ij}(t)$ is the amount of pheromone on edge (n_i, p_j) at time instant t and $N(t)$ is the set of m existing processors. For ant k at time instant t, the probability of assigning task n_i to processor p_j is computed by

$$p_{ij}^{k}(t) = a_{ij}(t) \Big/ \sum_{l \in N(t)} a_{il}, \quad \forall j \in N(t). \tag{11}$$

At this time, $p_{ij}^{k}(t)$ for all the existing processors should be computed. Then a random number in the range of [0, 1) is generated, and the processor will be selected according to the generated number—for each processor, the larger the pheromone value, the larger the chance of being selected. These operations are repeated until a complete scheduling of all the tasks is achieved, which means the completion of the ant list.

In the third stage, based on each selected pair of (n_i, p_j), using Eq. (2), the maximum finish time is calculated as 'makespan', which is also the desirability of the obtained scheduling for this ant, and according to this desirability, the quantity of pheromone should be deposited on the visited states, which is calculated by

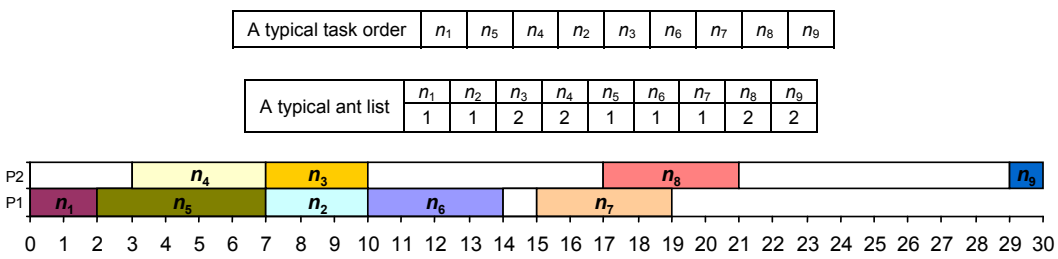

Fig. 3 A typical ant list filled by the proposed approach, along with its task order and the corresponding scheduling on two processors, demonstrated by a Gantt chart

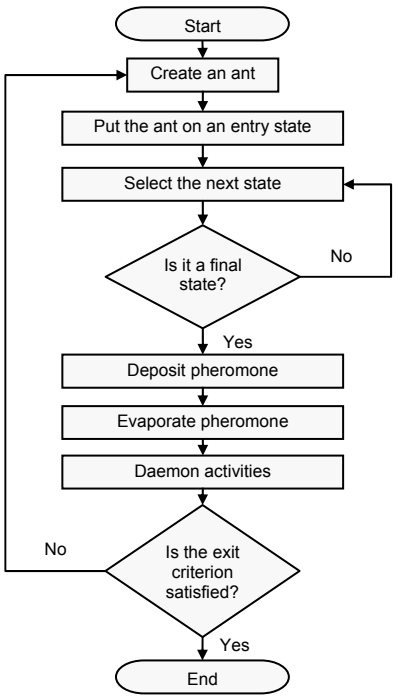

Fig. 4 The flowchart of the proposed approach

$$\Delta\tau_{ij}^{k}=\frac{1}{L^{k}},\quad \text{if } (n_i,p_j)\in T^{k}, \tag{12}$$

where L^k is the overall finish time for ant k and T^k is the set of all (n_i, p_j)'s executed by this ant. Accordingly, $\Delta\tau_{ij}^{k}$ should be deposited on every τ_{ij} if (n_i, p_j) exists in T^k (task n_j has been scheduled on to processor p_j). Otherwise, τ_{ij} will remain unchanged.

In the fourth stage, using Eq. (13), all the pheromone variables are decreased to simulate pheromone evaporation in real environments. Of course, this stage is very important to prevent premature convergence and stagnation caused by local minima:

$$\tau_{ij}=(1-\rho)\tau_{ij}, \tag{13}$$

where ρ is the evaporation rate in the range of [0, 1) and should be determined experimentally.

In the last stage, we have daemon activity (any other activities, such as local search and extra pheromone deposition, to boost ACO). In this stage, to enhance the performance of the proposed approach and specifically, to avoid removing good solutions, the best-ant-until-now ($\text{Ant}^{\min}$) is selected as the best solution, and some extra pheromone is deposited on the states visited by this ant:

$$\Delta\tau_{ij}^{\min}=\frac{1}{3L^{\min}},\quad \text{if } (n_i,p_j)\in T^{\min}. \tag{14}$$

Algorithm 1 The proposed approach

```
00: int n←number of nodes in the task order;
01: int m←number of existing processor elements;
02: int task_order[n]←a selected topological task order,
    extracted based on the task priority measurements intro-
    duced in Section 2, to be assigned to the processors;
03: int w[1..n]←required execution times of all the tasks
04: int τ[1..n, 1..m]←ε; // Initialize global pheromone matrix
05: int a[1..m]←0;  // Desirability of assigning a task to
                    // each of the m existing processors
06: int p[1..m]←0;  // Probability of assigning a task to
                    // each of the m existing processors
07: int Avail[1..m]←0; // The earliest time for the processor
                    // to be available for running the next task
08: int ant^[1..x][1..n], ant^min[1..n]; // x is the total number of ants
09: int makespan;
10: for k=1 to the total number of ants
11:    ant^k[1..n]←0;   // Initialize ant^k
12:    Avail[1..m]←0;
13:    for t=1 to n   // For each task in the task graph
14:       for i=1 to m
15:          Compute the desirability vector (a^t[i]);
                    // Use Eq. (10)
16:          Compute the probability vector (p^t[i]);
                    // Use Eq. (11)
17:       next i
18:       r←a randomized number in [0, 1);
19:       ant^k[t]←for the task n_t in ant^k, select one of the m
             processors based on r and p^t[1..m];
20:       for j=1 to n   // For each parent of task n_t
21:          if n_j∈Parents(n_t) then update EST(n_t, ant^k[t]);
22:       next i
23:       Avail[ant^k[t]]←EST(n_t, ant^k[t])+w[n_t];
24:    next t
25:    makespan←Max(Avail[1..m]);
26:    for i=1 to n     // Each task–processor pair in ant^k
27:       update τ[i, ant^k[i]] based on the makespan;
             // Use Eq. (12)
28:    next i
29:    for i=1 to n
30:       for j=1 to m
31:          τ[i, j]←τ[i, j]×(1–ρ);
             // Pheromone evaporation using Eq. (13)
32:    next i, j
33:    if ant^k<ant^min then ant^min=ant^k;
             // Start daemon activities
34:    for i=1 to n
             // For each task–processor pair in ant^min
35:       update τ[i, ant^min[i]] based on the makespan of ant^min;
             // Use Eq. (14)
36:    next i
37: next k
38: print ant^min;
```

6 Implementation and experimental results

The proposed approach was implemented on a Pentium IV (8-core 3.9 GHz i7-3770K processor) desktop computer with a Microsoft Windows 7 (X64) platform using Microsoft Visual Basic 6.0 programming language. All initial values of the pheromone variables were identically set to 0.1. The evaporation rate was considered as 0.997. The algorithm was terminated after 2500 iterations, that is, after generating 2500 ants.

The implementation of the proposed approach in Algorithm 1 reveals that there are four main iterations in the algorithm (lines 10, 13, 14, and 20); hence, the overall time complexity of the proposed approach is $O(K(mn+n^2))$, where n is the number of tasks in the task graph and m is the total number of available processors. Because K is a constant limited to 2500, for sufficiently large numbers of n and m, we can assume that the overall time complexity of the proposed approach is $O(mn+n^2)$, slightly better than that of the traditional EST method, which is $O(mn^2)$. That is, for large-scale samples, the actual performance of the proposed approach will be slightly better than the results presented in the following experiments.

6.1 Utilized data set

Table 3 lists six task graphs of real-world applications (and their comments) considered to evaluate the proposed approach. These graphs are the standard ones in the literature and have been used for evaluation in a number of related works. Hence, we compare the proposed approach against its traditional counterparts. All these six graphs are used to compare the proposed approach versus the traditional EST and the pre-introduced CLA-based approach.

Table 3 Selected task graphs for evaluating the proposed approach

Graph	Comment	Number of nodes	Communication cost
G1	Kwok and Ahmad (1998)	9	Variable
G2	Al-Mouhamed (1990)	17	Variable
G3	Wu and Gajski (1990)	18	60 and 40
G4	Al-Maasarani (1993)	16	Variable
G5	Fig. 1 (Hwang *et al*., 2008)	9	Variable
G6	Hwang *et al.* (2008)	18	120 and 80

In addition, for a rational judgment, a set of 45 random task graphs is used to evaluate the proposed approach. These graphs have different shapes for the following three parameters:

Size (n): number of tasks in the task graph. Five different values, 32, 64, 128, 256, and 512, were considered.

Communication-to-computation ratio (CCR): CCR demonstrates how much a graph is communication or computation based. The weight of each node in this data set was randomly selected from a uniform distribution, with mean equal to the specified average computation cost, which is 50. The weight of each edge was also randomly selected from a uniform distribution, with mean equal to the product of the average computation cost and CCR. Three different values of CCR, 0.1, 1.0, and 10.0, were selected, with 0.1 giving computation-intensive task graphs and 10.0 communication-intensive ones.

Parallelism: the average number of children for each node in the task graph. Increase in this parameter makes the graph more connected. Three different values of parallelism, 3, 7, and 15, were chosen.

Since the makespans achieved from these random graphs are in a wide range because of their various shapes and parameters, the normalized schedule length (NSL), which is a normalized measure, is used. It can be calculated for every input task graph by dividing the makespan obtained to a lower bound, so-called CPL, defined as the sum of weights of the nodes on the original critical path, using

$$\mathrm{NSL}=\frac{\text{Schedule length}}{\sum_{n_i\in \mathrm{CP}} w_i}, \tag{15}$$

where CP is the set of nodes on the critical path (the longest existing path) in the given graph.

For each graph using the list-scheduling technique, five different task orders were extracted with respect to the different priority measurements, namely TLevel, BLevel, SLevel, ALAP, and NOO. The task orders extracted from TLevel and ALAP were in ascending order, while the others in descending order. Table 4 shows the aforementioned orders extracted from the task graph in Fig. 1. To ascertain the validity, one can use the values in Table 1.

Table 4 Different task orders extracted from the task graph in Fig. 1 with respect to the different priority measurements

Measure	Task order								
TLevel	n_1	n_3	n_4	n_5	n_2	n_6	n_8	n_7	n_9
BLevel	n_1	n_5	n_2	n_3	n_4	n_6	n_7	n_8	n_9
SLevel	n_1	n_5	n_4	n_2	n_3	n_6	n_7	n_8	n_9
ALAP	n_1	n_5	n_2	n_3	n_4	n_6	n_7	n_8	n_9
NOO	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9

6.2 Experiments and results

The first set of experiments was conducted on the different task orders presented in Table 4 using two processors. The bars in Fig. 5 are the final makespans obtained by the traditional EST method, in addition to CLA and the proposed approach. For all the experiments, the mean of results of 10 times of algorithm execution is considered to regulate the results and prevent biased judgment. As can be seen, the proposed approach outperforms EST in all cases and is slightly better than the introduced CLA approach. The results are the most important contributions of the work, demonstrating that the traditional EST method, which is used extensively in the literature to assign tasks to the processors, is not an optimum solution.

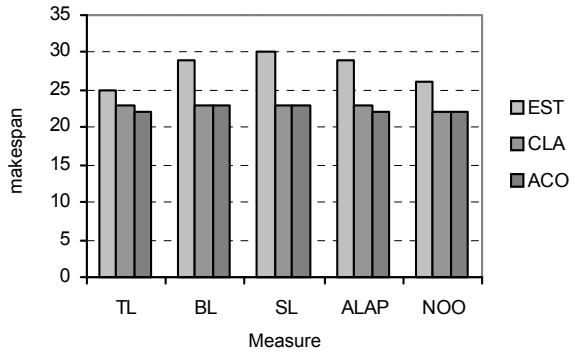

Fig. 5 The final makespans obtained by the traditional EST method, in addition to CLA and the proposed approach, for different task orders presented in Table 4, using only two processors

One of the most challenging issues in utilization of meta-heuristic approaches is their poor performance, encountering problems with huge dimensions, that is, when each state of the problem has a large number of neighborhoods. In such a condition, each decision is almost scholastic, and convergence cannot be achieved but with a large number of iterations. The issue is presented using a sample in Fig. 6. The diagrams show the makespans achieved by EST and the proposed approach for the different task orders in Table 4. There is a problem here: the number of processors, one of the dimensions of the scheduling problem, grows from 2 to 6; surprisingly, for the task orders extracted from TLevel and NOO, the results achieved get worse. This phenomenon is due to the fact that increasing the number of processors makes the dimensions of the problem grow. As a result, the decisions are taken stochastically, with no convergence to a good solution.

To tackle the issue, we revise the proposed approach to an incremental one; that is, we break the whole colony into $m-1$ different subcolonies, each of which uses a different number of available processors in an incremental manner, where m is the total number of existing processors. They sequentially explore the problem space. For example, if $m=6$, then there will be five different subcolonies. The first subcolony explores the problem using only two processors to schedule the tasks, the second subcolony uses three processors to work, the next one uses four, and so on. All the ants' experience, such as pheromone trails and Ant^{min}, will be retained till the next subcolonies. Using this incremental policy, ACO, if needed, tries to use fewer processors, meaning a compact scheduling that leads to better solutions with faster convergence in some cases. Fig. 7 shows the superiority of the revised approach, which is shown as ACO2 in comparison with the basic algorithm shown as ACO, and therefore all the next experiments will be done using the revised version.

In the next set of experiments, the number of processor elements varies from 2 to 6. Six processors are enough for each approach to make its best scheduling. All different task orders of all the graphs presented in Table 3 were used in this set of experiments. These experiments make it possible to comprehensively compare the proposed approach with the traditional EST and the CLA-based task assigner. The results are listed in Fig. 8. Again, in these experiments, the proposed approach outperforms EST in almost all the cases and outperforms CLA for most ones.

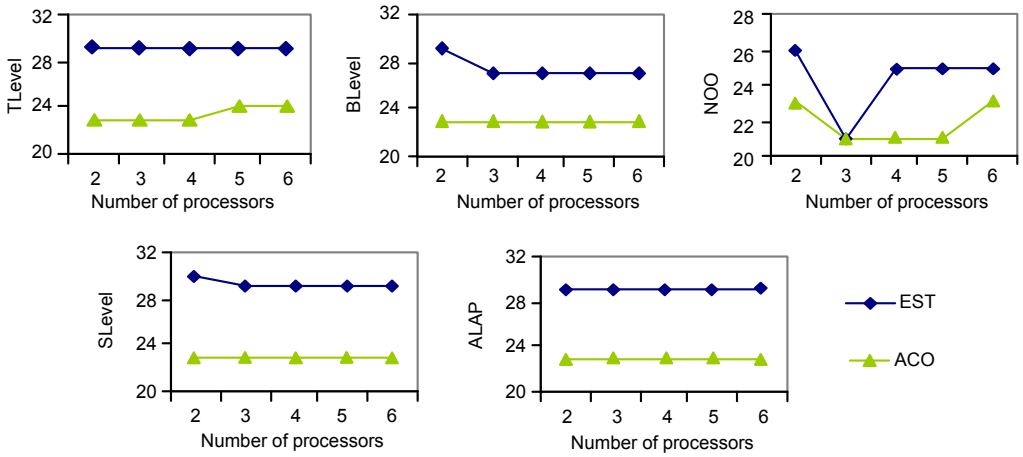

Fig. 6 The makespans achieved by EST and the proposed approach for different task orders of Table 4 using various numbers of processors ranging from 2 to 6

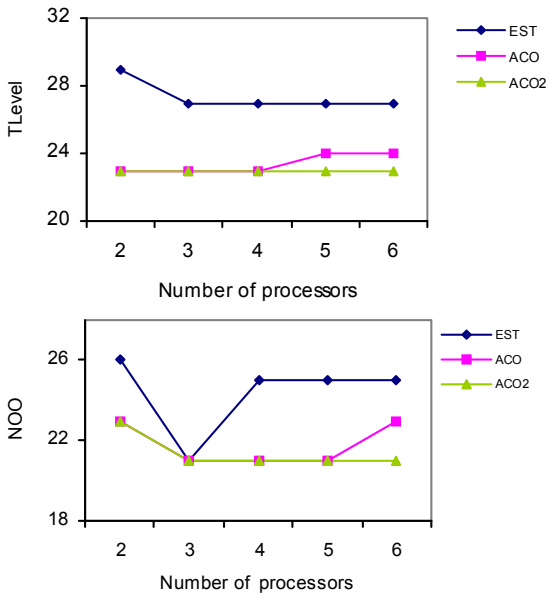

Fig. 7 The results achieved by the proposed approach and the revised algorithm shown as ACO2

The last set of experiments was conducted on the aforementioned large-scale random-task graphs. Because the size of the input task graphs and the number of processors used in the experiments are two major metrics of the problem space, we analyzed the results from these points of view. In addition, because the makespans achieved from these random graphs are in a wide range in relation to their various shapes and parameters, NSL, which is a normalized measure, will be used to compare the methods. Besides, to guarantee the convergence, the number of iterations for ACO and CLA is changed from 2500 and 5000, respectively, to $m \times n \times 100$, where m and n are the numbers of processors and tasks, respectively. Figs. 9 and 10 demonstrate the average NSL achieved by EST, CLA, and ACO for the different task orders of all the 45 random task graphs with respect to their different sizes varying from 32 to 512, and the different numbers of processors used. The larger the size of the graph inputted, the larger the NSL achieved. The reverse is true for the number of processors; i.e., using more processors results in a better NSL. Again, the proposed ACO-based assigner is victorious in its race versus the pre-introduced CLA and the traditional EST in all cases. These results, accompanied with the results achieved from the real task graphs listed in Table 3, indisputably show the superiority of the proposed approach in comparison with EST and the pre-introduced CLA approach.

7 Conclusions

In the list scheduling technique, after extracting a proper sequence of tasks (a task order), most of the related algorithms would assign each task to the processor providing EST. This study proves that the performance of EST is not actually optimum and that the makespans achieved by such algorithms are practically restricted. Subsequently, a novel approach based on ACO, which can find far better solutions, was introduced. To evaluate the proposed approach, six frequently used task graphs inspired from real-world

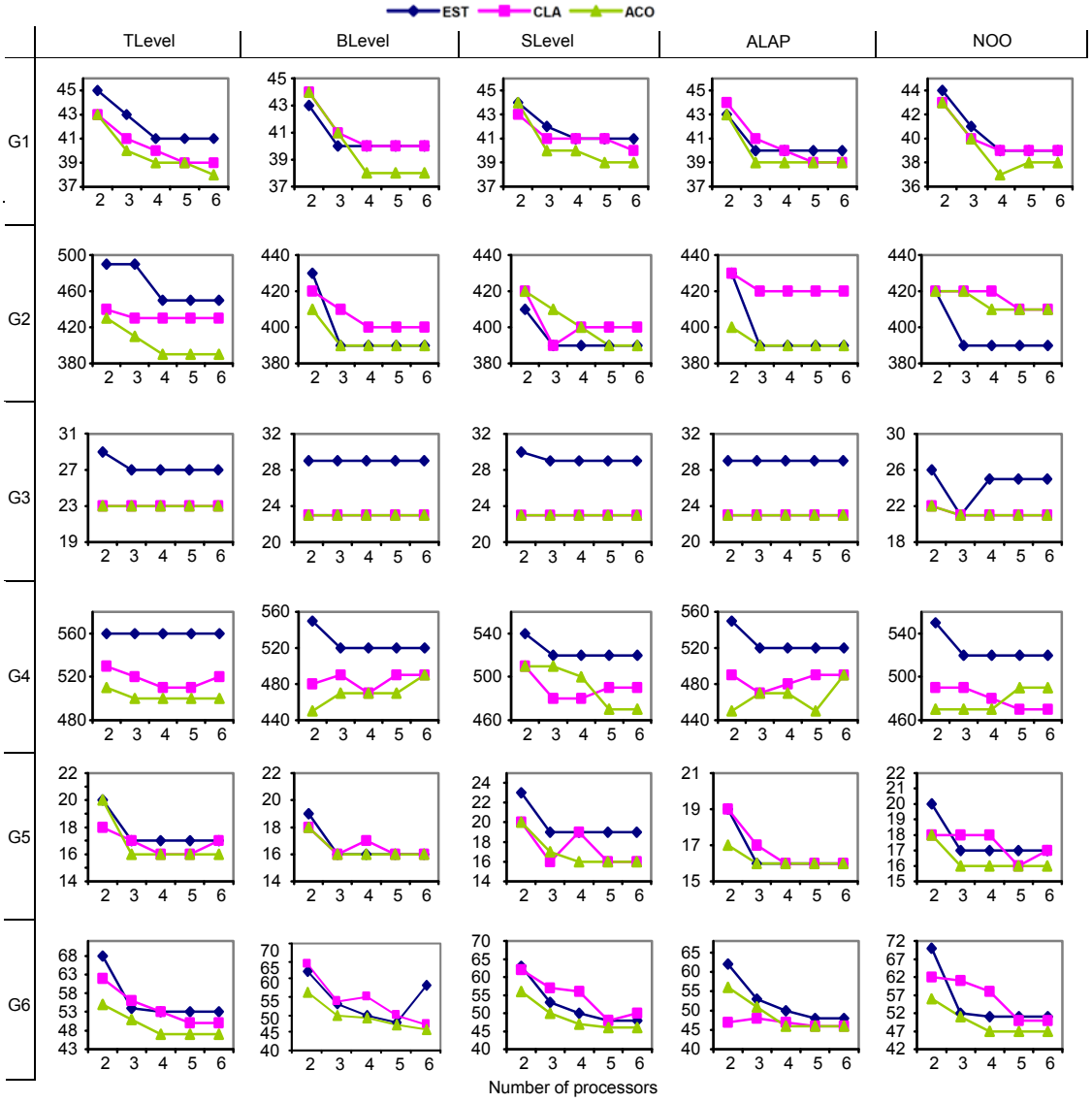

Fig. 8 The results obtained by the proposed ACO approach, in addition to EST and CLA, for all the different task orders of each of the six task graphs in Table 4, using a different number of processors ranging from 2 to 6

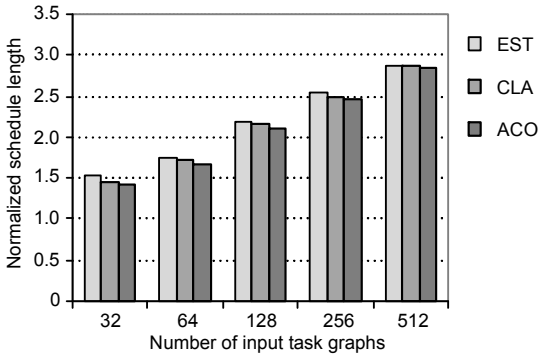

Fig. 9 The average NSL achieved by EST, CLA, and ACO for the different task orders of all the 45 random task graphs, with respect to different sizes of the input task graphs varying from 32 to 512

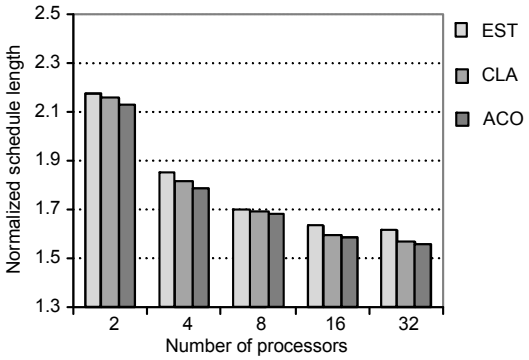

Fig. 10 The average NSL achieved by EST, CLA, and ACO for the different task orders of all the 45 random task graphs with respect to the various numbers of processors ranging from 2 to 32

applications were selected. For each task graph, five different task orders based on different priority measurements of the nodes, namely, TLevel, BLevel, SLevel, ALAP, and NOO, were extracted for experimental use. The proposed approach found far better solutions using a restricted number of processors, but as the number of processors (which is a major dimension of the problem space) was increased, ACO cannot achieve good solutions. To tackle the issue, a revised version of the proposed approach was introduced, using an incremental policy; that is, the whole colony was divided into m–1 different subcolonies, each of which used a different number of available processors in an incremental manner, where m is the total number of processors. Actually, each subcolony sequentially explores a portion of the problem space. Using this incremental policy, ACO, if needed, tries to use fewer processors, meaning a compact scheduling which leads to better solutions, with faster convergence for some cases. To evaluate the validity of the proposed approach, different sets of experiments were conducted from different points of view, and the results showed the performance superiority of the proposed approaches in almost all the cases.

References

Adam, T.L., Chandy, K.M., Dickson, J., 1974. A comparison of list scheduling for parallel processing systems. *Commun. ACM*, **17**(12):685-700. http://dx.doi.org/10.1145/361604.361619

Al-Maasarani, A., 1993. Priority-Based Scheduling and Evaluation of Precedence Graphs with Communication Times. MS Thesis, King Fahd University of Petroleum and Minerals, Saudi Arabia.

Al-Mouhamed, M.A., 1990. Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Trans. Softw. Eng.*, **16**(12):1390-1401. http://dx.doi.org/10.1109/32.62447

Baxter, J., Patel, J.H., 1989. The LAST algorithm: a heuristic-based static task allocation algorithm. Proc. Int. Conf. on Parallel Processing, p.217-222.

Boveiri, H.R., 2010. ACO-MTS: a new approach for multiprocessor task scheduling based on ant colony optimization. Proc. IEEE Int. Conf. on Intelligent and Advanced Systems, p.1-5. http://dx.doi.org/10.1109/ICIAS.2010.5716203

Boveiri, H.R., 2014. Assigning tasks to the processors for task-graph scheduling in parallel systems using learning and cellular learning automata. Proc. 1st National Conf. on Computer Engineering and Information Technology, p.1-8 (in Farsi).

Boveiri, H.R., 2015. Multiprocessor task graph scheduling using a novel graph-like learning automata. *Int. J. Grid Distr. Comput.*, **8**(1):41-54. http://dx.doi.org/10.14257/ijgdc.2015.8.1.05

Chrétienne, P., Coffman, E.G., Lenstra, J.K., *et al*., 1995. Scheduling Theory and Its Application. John Wiley & Sons, New York.

Dorigo, M., Maniezzo, V., Colorni, A., 1991. Positive Feedback as a Search Strategy. Technical Report No. 91-016, Politecnico di Milano, Milan, Italy.

Dorigo, M., di Caro, G., Gambardella, L., 1999. Ant algorithm for discrete optimization. *Artif. Life*, **5**(2):137-172. http://dx.doi.org/10.1162/106454699568728

Hwang, J.J., Chow, Y.C., Anger, F.D., *et al*., 1989. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, **18**(2):244-257. http://dx.doi.org/10.1137/0218016

Hwang, R., Gen, M., Katayama, H., 2008. A comparison of multiprocessor task scheduling algorithms with communication costs. *Comput. Oper. Res.*, **35**(3):976-993. http://dx.doi.org/10.1016/j.cor.2006.05.013

Kruatrachue, B., Lewis, T.G., 1987. Duplication Scheduling Heuristics (DSH): a New Precedence Task Scheduler for Parallel Processor Systems. Technical Report No. OR 97331, Oregon State University, Corvallis.

Kwok, Y., Ahmad, I., 1998. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, **31**(4):406-471. http://dx.doi.org/10.1145/344588.344618

McCreary, C., Gill, H., 1989. Automatic determination of grain size for efficient parallel processing. *Commun. ACM*, **32**(9):1073-1078. http://dx.doi.org/10.1145/66451.66454

Meybodi, M.R., Beigy, H., Taherkhani, M., 2004. Cellular learning automata and its applications. *J. Sci. Technol. Sharif Univ.*, **25**:54-77 (in Farsi).

Narendra, K.S., Thathachar, M.A.L., 1974. Learning automata: a survey. *IEEE Trans. Syst. Man Cybern.*, **SMC-4**(4): 323-334. http://dx.doi.org/10.1109/TSMC.1974.5408453

Sih, G.C., Lee, E.A., 1993. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parall. Distr. Syst.*, **4**(2):175-187. http://dx.doi.org/10.1109/71.207593

Wolfram, S., 1983. Cellular automata. *Los Alamos Sci.*, **9**:2-27.

Wu, M.Y., Gajski, D.D., 1990. Hypertool: a programming aid for message-passing systems. *IEEE Trans. Parall. Distr. Syst.*, **1**(3):330-343. http://dx.doi.org/10.1109/71.80160