Contents lists available at ScienceDirect

Future Generation Computer Systems

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

On arrival scheduling of real-time precedence constrained tasks on multi-processor systems using genetic algorithm

Pranab K. Muhuri [*], Amit Rauniyar, Rahul Nath

Department of Computer Science, South Asian University, New Delhi-10021, India

HIGHLIGHTS

- A novel formulation for the precedence constrained real-time scheduling problem.
- We model it as a dynamic constraint problem considering task arrival times.
- Propose a dynamic genetic algorithm for real-time systems to solve the problem.
- It finds feasible task schedule with minimised makespan ensuring deadline compliance.
- It uses the idea of variable length chromosome for varying number of task arrivals.
- Thorough comparative study with PSO, ABC and SA considering benchmark task sets.
- Provides an implementation method to tackle synchronization and missing deadlines.

ARTICLE INFO

Article history:
Received 25 May 2018
Received in revised form 5 September 2018
Accepted 7 October 2018
Available online 29 October 2018

Keywords:
Precedence constraints
Arrival time
Task allocation
Dynamic task scheduling
Real-time multiprocessor systems
Genetic algorithms

ABSTRACT

This paper models the problem of precedence based real-time task scheduling as a dynamic constraint problem. It presents a new scheduling approach, termed as 'Dynamic Genetic Algorithm for Real-Time Scheduling' (^{d}GA-RTS). The significant feature of ^{d}GA-RTS is that it can handle dynamic as well as static scheduling of inter-dependent tasks for real-time systems. In parallel or distributed systems, the main aim of dynamic task scheduling is to allocate processors to a given set of tasks to execute them within optimized completion times without violating the task dependencies, if any. The ^{d}GA-RTS schedules the tasks in the waiting list when any new task arrives in the scheduler and minimizes the overall schedule length of the task set ensuring deadline compliance. We also illustrate an implementation technique to deal with synchronization problems in multi-processor systems. In order to exhibit the applicability of our approach, we perform experiments with suitable benchmark as well as synthetic test cases. Further, we conduct extensive simulations and compare the results with different performance metrics. The comparative study of the results with existing approaches indicates that our proposed approach is more efficient in generating feasible solutions.

1. Introduction

Every task in a real-time system (RTS) needs to be executed within a specific time frame, called 'deadline'. Violation of a deadline may cause severe damage to the system, and create catastrophe to the surroundings, where such a system is employed. RTSs are mainly of two types: hard RTS and soft RTS. Hard RTSs require strict compliance of the task deadlines. That is, in a hard RTS, every task must be processed within the deadline; else, there may be system failure causing significant losses. In soft RTSs, the computational outputs may remain useful even if some deadlines are violated up to certain extents [1]. Hence, scheduling of tasks is an important module of the RTS design which ensures that the system operation is safe and purposeful.

Real-time tasks may have different characteristics, on the basis of which they are termed 'pre-emptive' or 'non-pre-emptive', 'periodic' or 'aperiodic', 'interdependent (precedence based)' or 'independent', etc. Task scheduling in RTS is either static or dynamic. Static task scheduling is usually feasible only for periodic tasks. Here, all required task characteristics viz. data dependencies, communication cost, processing time, and synchronization requirements, etc. are known prior to scheduling [2]. However, aperiodic tasks are highly event-generated and their characteristics are not known a priori. Thus, static scheduling approaches are not that much useful for aperiodic tasks. Therefore, execution of such tasks on parallel processors needs a proper mechanism that can dynamically allocate processors and schedule the tasks. When new task(s) arrives, such a scheduler should be able to dynamically generate a new feasible schedule.

Real-time tasks are often assumed as independent, for simplicity by the researchers. However, in practical domains, tasks

[*] Corresponding author.
E-mail addresses: pranabmuhuri@cs.sau.ac.in (P.K. Muhuri), amitrauniyar90@gmail.com (A. Rauniyar), rahul.nath@outlook.com (R. Nath).

https://doi.org/10.1016/j.future.2018.10.013
0167-739X/© 2018 Elsevier B.V. All rights reserved.

are very much inter-dependent. To depict the inter-dependencies among tasks topologically, DAG (directed acyclic graph) is a very useful tool [3]. Each task is a distinct unit of job which may take dependencies from/to one or more tasks. The preceding task is called parent and the task which depends on it, is called the child task. Also, a real-time task needs to be conditioned for execution as soon as it arrives in the system. It is necessary to trigger the scheduler on arrival of a task so that it can be scheduled with minimum delay to ensure deadline compliances and fulfil the input requirements of waiting child task. Hence, scheduling of DAG in real-time heterogeneous systems on the basis of arrival times makes the environment dynamic and more complex. Despite several studies on task scheduling problems, GAs have not yet been explored much for dynamic task scheduling in RTS for precedence-constrained tasks represented by DAGs. The goal of dynamic task scheduling on real-time parallel processors is to allocate processor to a given task considering the task precedences and the deadlines to execute with minimum schedule length.

In this paper, a new scheduling approach, which we termed as 'Dynamic Genetic Algorithm for Real-Time Scheduling' (^{d}GA-RTS), is presented. ^{d}GA-RTS can suitably handle DAGs in RTS with task deadlines for generating dynamic schedule giving minimum possible makespan and ensures deadline compliances. It revises the schedule when any new task arrives at the scheduler. The designing of ^{d}GA-RTS is based on the motivation obtained from the concept of immigration scheme considered in [4]. In this new algorithm, we have introduced a novel idea of variable chromosome length [5] for variations in the task numbers when new task(s) arrive. The main feature of ^{d}GA-RTS is that it can handle dynamic as well as static scheduling of inter-dependent tasks for RTS. Also, the algorithm can be applied to single as well as multi-processor systems. Multi-processor computing systems can be both homogeneous and heterogeneous. Therefore, we have considered both scenarios for experiments in our work. We have performed experiments using suitable task sets and demonstrated the applicability of the proposed method for dynamic and static scheduling considering different performance metrics. Later on, we present an implementation technique which is capable of managing the synchronization problem that may arise because of the arrival of some new task(s) before the termination of the scheduling algorithm. Also, we discuss about few traditional techniques which can handle unfeasible solutions and deadline violations in terms of exception handler for real-time systems.

The major contributions of this paper are highlighted below:

- Modelled the problem of precedence based real-time tasks scheduling as a dynamic constraint problem on the basis of their arrival times.
- The model considered the arrival time to find the schedule and calculated lateness of each task to examine the feasibility of the solutions obtained.
- Proposed a scheduler model for handling dynamic behaviour of the problem.
- Developed a decimal-encoded GA, which we have named as "Dynamic Genetic Algorithms for Real-time Systems (^{d}GA-RTS)", to find solutions for the problem.
- We have also solved the problem by implementing few commonly used evolutionary algorithms viz., Simulated Annealing (SA), Particle Swarm Optimization (PSO) and Artificial Bee Colony (ABC).
- We have compared results obtained from PSO, ABC and SA with our ^{d}GA-RTS technique using four different performance metrics on different benchmark task sets.
- Included an implementation method that can tackle synchronization problem and deadline violations as an exception handling.

Rest of the paper is organized as follows. In Section 2, a brief review of the existing literature is given. Section 3 discusses the formulation of the problem and presents the details of the proposed GA based solution scheme for dynamic task scheduling in RTS. It also demonstrates the working of the proposed approach using a simple example. Section 4 presents a comparative study of the results with a discussion on several performance measures along with simulation results on larger task sets for both static and dynamic cases. A way to deal with the synchronization problems and exception handling during implementation is described in Section 5. Section 6 concludes the paper with discussions on the results. It also highlights few future research directions in the context of the addressed problem.

2. Literature review

Genetic algorithms (GA) are widely used for optimization and search problems including task scheduling problems [6–8]. A number of research works were reported for static task scheduling in RTSs using GAs. However, there were not many reported study on the use of GAs for dynamic task allocation and scheduling in RTSs. Some well-known approaches that used GAs for task scheduling problems are now discussed briefly.

In one of its very early application to the scheduling problem, Monnier et al. [9] implemented a GA to solve a complex real-time periodic tasks scheduling problem in order to get the optimal schedule in a distributed environment. In [10], Choe et al. presented a GA based approach for iterative rescheduling to boost the performance under real-time constraints, which reportedly outperformed the simulated annealing based approach. In [11], Oh et al. proposed a GA based solution technique to find optimal schedules for task executions on multi-processor systems with an aim to minimize the required number of processors and the tardiness of tasks. Several hybridized forms of GA were also developed to gain improvement in forming schedules for multiprocessor task scheduling. In [12], Omara et al. developed two hybrid GAs viz. CPGA (critical path genetic algorithm) and TDGA (task duplication genetic algorithm) for the task scheduling problem in general computing systems. CPGA could minimize the execution length and balance the loads, whereas TDGA used the idea of task duplication to optimize the communication overhead between the tasks. Recently, Zhang et al. [13] designed a GA based solution technique for load-aware resource allocation and DAG task scheduling (LA-RATS) strategy for cloudlet systems. The results of their experiments showed that the approach performed well in minimizing the monetary cost, total turnaround time and enhanced deadline satisfaction for delay tolerant and sensitive applications. A novel hybridized GA for fault-tolerant scheduling to guarantee the compliance of real-time task deadlines was developed by Samal et al. [14]. They designed a model of scheduler and the primary backup fault-tolerant method considering real-time properties of task to tackle the faults. The solutions achieved after the experiment on several task sets demonstrated that their technique was capable of producing improved solutions. Akbari et al. [15] proposed a GA based technique with new functions of genetic operators and tuning of initial population to optimize the schedule length with lower repetitions of solutions. They performed simulations on random graphs in heterogeneous system and measured several performance metrics. The comparative studies of their results indicated significant enhancement with respect to other scheduling algorithms.

One of the very early and significant considerations of GA for dynamic load balancing and scheduling was by Zomya et al. [16], which projected better result than first fit heuristic approach. In [17], Cheng et al. presented a GA based approach for dynamic real-time multiprocessor scheduling problem considering the resource and time constraints with a feasible energy function. Ma et al. [18] implemented an improved dynamic version of GA to address the dynamic task scheduling problem in cloud computing to enhance its performance. Their model for the cloud environment could produce an effective throughput with reduction in the execution times. A GA based dynamic scheduling scheme was proposed by Page et al. [19] that illustrated the minimization of execution times on distributed heterogeneous processors considering system resources at the arrival of tasks. Similarly, Page et al. [20] proposed a dynamic scheduling approach that combines GA with eight common heuristics on heterogeneous multiprocessor systems to minimize the execution times. Their algorithm was designed for scheduling batches of tasks which could be remapped pre-emptively to the processors. Nayak et al. [21] proposed a hybrid GA with BFO for multi-processor task scheduling and reported significant improvements over other approaches. Nath et al. proposed GA based approaches for energy efficient scheduling under uncertain situation and static scenarios in [22–24]. A scheduling scheme for non-identical processors was proposed by Wen et al. [25] that considered the hybridization of GA with neighbourhood search technique. In [4], Cheng et al. solved a dynamic load balancing problem, where a number of GAs were executed in series. Zheng et al. [26] formulated a problem of scheduling DAG workflows to process big data on cloud with an aim to minimize the monetary cost and satisfy the deadline constraints. Through extensive simulations, authors showed that their proposed cost-efficient scheduling algorithm was superior to other existing solutions. An improved version of GA termed as Multi-Stage Composite GA was proposed by Li et al. [27] to solve random assignment problem for resource allocation by tackling the slow and premature convergence of considered evolutionary technique. They even found that the approach could be suitable for scheduling in manufacturing process, data mining and other complex numerical optimization [28] systems. Li et al. [29], studied one of the classical optimization problems, random assignment problem for resource allocation in manufacturing and management process. They developed a GA based solution approach considering the compound effect of mean and variance to make a decision. The results of their experiments show the improvement in efficiency and convergence. A DAG based static scheduling algorithm for distributed system using a hybrid GA was proposed in [30]. For task scheduling and partitioning of data in a heterogeneous environment, Suresh et al. proposed a real-coded hybrid GA in [31]. Similarly, Biswas et al. [32] reported a DAG based real-time task scheduling approach using Bayesian optimization algorithm that minimizes total makespan and tardiness of a schedule. Chaudhary et al. [33] studied the context to map the theory, concepts and principles of evolutionary strategies in system science, integrating GA as an artificial intelligent tool. Through, their study they also presented a scope of GA for a wide group of business applications and commercial software. GA, being one of the popular and efficient evolutionary algorithms (EA), was applied to many other significant works [28,34–38] for the study of task assignment and resource allocation problems in recent times.

3. Problem formulation and working of the proposed genetic algorithmic solution approach

In this section, we first describe the model of the directed acyclic graphs, which is used in our problem formulation for representing the task characteristics. Then, we describe the procedure followed for the generation of the task deadlines. Also, we briefly explain the steps of a traditional GA. The elements of the GA used for the directed acyclic graphs are explained next. Finally, we have explained the working of the proposed algorithm.

The notations and the abbreviations used for the explanation and the mathematical formulation of the problem are mentioned in Table 1.

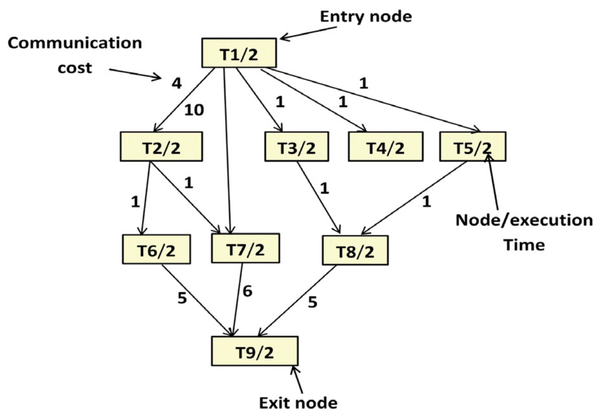


Fig. 1. A typical DAG based task graph.

3.1. The model of directed acyclic graph

Kwok et al. [3] presented a model of DAG and performed comparisons to benchmark several graphs for task scheduling algorithms. The newly arrived task may be dependent or independent of the tasks already scheduled. If the newly arrived tasks are dependent, then it forms a task graph $G = (V, E)$, which can be denoted by a weighted DAG with V as the set of nodes and E as the set of directed edges. A node is a set of instructions termed as task. An edge starts from a parent node and sinks at a child node. Nodes without parents are known as entry nodes, whereas nodes without any child are called exit nodes. Fig. 1 shows a typical DAG.

A child node may have one or more parent nodes as it forms a child–parent relationship between the nodes. The child does not execute until it is all parents complete their executions, as the processed data of parents may be input for the child node. The computation cost of a node n_i is represented by (n_i) time. The precedence from parent to child can be represented as $n_{i\rightarrow}n_j$, where n_i is a parent of n_j child. The weight on an edge is known as the communication cost, which is denoted by $C(n_{i,}n_j)$ [3,30]. This communication cost is accounted when n_i and n_j are scheduled on different processors; otherwise, communication cost is considered as zero. If the node n_i is scheduled on processor P_j, then the start time and finish time of node n_i are represented by $ST(n_{i,}P_j)$ and $FT(n_{i,}P_j)$, respectively. After all the nodes are scheduled, the schedule length defined as $max_i\{FT(n_i, P_j)\}$, across all processors is calculated. The objective of DAG scheduling is to find an allocation of the tasks to different processors and their start times so that the total execution length is minimized considering that the precedence constraints are preserved and deadlines are complied.

3.2. Precedence based task scheduling: Dynamic constraint problem

The scheduling of precedence based task graph as a dynamic constraint problem is more practical and realistic. Here, the task characteristics such as arrival time, starting time, execution time and deadlines are not known in advance. We define the problem as a series of instances adapting the similar strategy described in [39] for a dynamic constraint problem [40,41].

A centralized framework is designed, where the central scheduler has knowledge about all processors and tasks to be allocated. The central scheduler evaluates the schedule for all the available tasks at a particular instance. The problem (at that instance) is to schedule n precedence based tasks $\{T_1, T_2, \ldots, T_n\}$ on m number of distributed processors $\{P_1, P_2, \ldots, P_m\}$. Each task T_i has an arrival time $AT\ (T_i) \geq 0$, starting time $ST(T_i, P_j)$ on processor P_j, execution time $ET(T_i) > 0$ and a deadline, $deadline\ (T_i) \geq AT\ (T_i) + ET(T_i)$.

Table 1
Notations, abbreviations and definitions.

Notations	Definitions
T_i	ith task
P_j	jth processor
m	Number of processors
n	Number of tasks
CC	Communication cost
*	Time at which ^{d}GA-RTS is invoked
$C\ (T_i, T_k)$	Communication cost between a child T_i and the parent T_k
LT	List of tasks according to topological order
$AT\ (T_i)$	Arrival time of task T_i
$ET(T_i)$	Execution time of task T_i
$ST(T_i, P_j)$	Start time of task T_i on processor P_j
$FT\left(T_i, P_j\right)$	Finish time of task T_i on processor P_j
$RT\left(P_j\right)$	Ready time of processor P_j
DAG	Directed Acyclic Graph
$DAT\ (T_i)$	Data Arrival Time of the task T_i
ITQ	Initial Task Queue
FTQ	Finished Task Queue
PSW	Processor Status Window
GA	Genetic Algorithm
PSO	Particle Swarm Optimization
SA	Simulated Annealing
ABC	Artificial Bee Colony Algorithm
avg_exe	Average amount of time needed to execute asset of tasks
level/depth	Depth of the tasks in the task graph
deadline (T_i)	Time within which processing of a task T_i must be completed
Completion Time	Finishing time of a task
Lateness	Difference between the completion times and the deadlines. The lateness of task T_i is given by, $lateness\ (T_i) = FT\ (T_i\ , P_j) - deadline\ (T_i)$
Total lateness	$\sum_{i=1}^{n} lateness(T_i)$, i.e. the sum of the lateness of all the tasks in a schedule
Average total lateness	Total lateness averaged over the number of populations
Laxity	$Laxity = deadline\ (T_i) - \{ET\ (T_i) - t\}$, where t is the time for which task T_i has already executed
dlaxity	Array of size of number of tasks in LT consisting of arbitrary scalar values of tasks according to their level
$ABC_A^{m_1,m_2}$	Area between the curves of algorithm m_1 and m_2

Let P_k be the processor on which the k^{th} parent task T_k of task T_i (child) is scheduled. So, child cannot execute until its parent has finished. Data Arrival Time (DAT) of task T_i at processor P_l is given by:

$$DAT(T_i) = \max\{FT\ (T_k, P_k) + C\ (T_i, T_k)\}\ ; k = 1, \ldots, N \quad (1)$$

$$\text{If } l = k,\ \text{then } C\ (T_i, T_k) = 0 \quad (2)$$

where l and k are the processor indices on which child and parent tasks are scheduled respectively and N is the number of parents of task T_i. Also, the communication cost is not taken into account if the parent and child tasks are scheduled on the same processor.

The main objective is to obtain a feasible schedule with minimum possible makespan, i.e. to assign starting times $ST(T_i, P_j)$ to all available tasks and calculate $FT(T_i, P_j)$ for each T_i;$1 \le j \le m$, $1 \le i \le n$, so that the following constraints are satisfied:

i. $\forall T_i, ST(T_i, P_j) \ge AT\ (T_i)$
ii. $ST(T_i, P_j) \ge AT\ (T_i) + DAT(T_i)$
iii. No preemption is allowed *i.e.*, $FT\left(T_i, P_j\right) = ST\left(T_i, P_j\right) + ET(T_i)$
iv. Each processor can only process one task at a time. Therefore, the number of tasks being processed at a time in the system does not exceed m, i.e.

$$\sum_{i=1,2,\ldots,n;j=1,2,\ldots,m} X_{ij}\ (CLK) \le m,\ \text{where}$$

$$X_{ij}\ (CLK) = \begin{cases} 1, & if\ T_i\ is\ running\ on\ P_j \\ 0, & otherwise \end{cases} \quad (3)$$

Here, CLK is the clock time. The objective function is the makespan of the scheduled tasks, which is defined to evaluate the quality of the schedule. It is defined as

$$S\_length = \max\left(FT\left(T_i, P_j\right)\right); i = 1, \ldots, n\ and\ j = 1, \ldots, m \quad (4)$$

where, *S_length* is a variable used to denote the schedule length or makespan of the given tasks.

At a particular CLK, if schedule for n tasks is to be found on m processors, then the instance of scheduling problem will be a static constraint problem (SCP), as everything about a task is known at its arrival. However, the arrival of tasks is not known in advance, so at every CLK, system checks for the arrival of tasks. If the tasks have arrived at CLK, the central scheduler is triggered to find a feasible schedule. Therefore, the problem becomes series of instances of scheduling problem with the progress of time. This particular scenario makes the problem dynamic in nature as sequence of such instances (SCP) occurs over time. Thus, we can design the complete task graph scheduling problem as a dynamic constraint problem.

At CLK, each task T_i from the set of tasks $\{T_1, \ldots, T_n\}$ has an execution time $ET(T_i)$ and *deadline* (T_i). In order to generate a feasible schedule with minimum possible makespan, we assign a starting time and allocate processors to all tasks which arrive in the system following the topological order. If any new task arrives at CLK, then a new solution is generated combining the newly arrived tasks with some of the already scheduled tasks for which $ST(T_i, P_j) > CLK$ and whose executions have not yet been started on the allocated processors. This leads to the formation of a newer schedule with updated processor allocations. On the other hand, the tasks $\{T_1, \ldots, T_m\}$, which are already being processed, i.e. whose executions started at times $ST(T_i, P_j) < CLK, j = 1, \ldots, m$ and but not finished their execution yet, i.e., $FT(T_i, P_j) = ST(T_i, P_j) + ET\ (T_i) > CLK$, can no longer be rescheduled. Thus, the goal becomes, finding an optimal schedule such that all tasks can be scheduled and executed by assigning starting time to all the unscheduled tasks at CLK so that all the constraints are satisfied. A given task T_i, which has not yet started execution, may get different values for the $ST(T_i, P_j)$ in subsequent CLK if any new task satisfying the constraints is available.

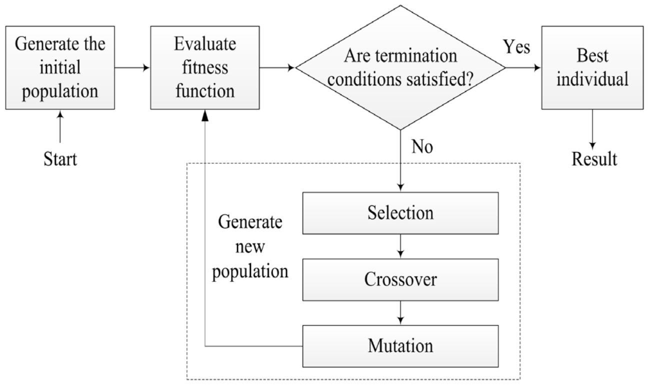

Generate the initial population
Start
Evaluate fitness function
Are termination conditions satisfied?
Yes
Best individual
Result
No
Generate new population
Selection
Crossover
Mutation

Fig. 2. Flowchart of a traditional genetic algorithm.

3.3. Deadline generation

The task deadlines for all task graphs are generated using the method proposed in [42]. The pseudo code of the procedure used for this purpose is as follows:

```
For i = 1 to n
    Take a task $T_i$ from LT
    deadline $(T_i) = depth(T_i) * avg\_exe * dlaxity(T_i)$
End for
```

Here, n is the number of tasks available in the array LT (list of tasks), $depth(T_i)$ denotes the level of the task T_i in the given DAG and $dlaxity\ (T_i)$ is a scalar value according to the level of task T_i. Since, large values of $dlaxity(T_i)$ may result in increased laxity in the system, we have generated $dlaxity(T_i)$ value randomly between [1, $n/max(level)$]. Thus, each level may have its own $dlaxity(T_i)$ value.

3.4. Genetic algorithm

GA is an efficient metaheuristic search and optimization tool based on the principle of natural evolution [6]. It operates on a pool of solutions termed as population and scans the entire search space through a guided random search technique. This leads to attain reasonable global minima or maxima within a bounded search space [8]. It is applied to numerous real-world problems. Fig. 2 shows various steps of a traditional GA in the form of a flowchart. In GAs, parameters or genes are coded using binary or real numbers. Then a problem specific fitness function (also called, objective function) is designed, which is later used to evaluate the fitness of the individuals. Then, selection technique is applied to select individuals with best fitness values. The selected individuals then go through genetic operations of crossover and mutation to generate offspring possibly with better fitness.

3.5. Genetic algorithm for dynamic task allocation and scheduling in RTS

Here, we are going to introduce a new GA based approach for dynamic task scheduling in RTS termed as 'Dynamic Genetic Algorithm for Real-Time Scheduling' (^{d}GA-RTS). We have used the basic principles of the GA proposed in [12]. For dependent tasks, DAT (Data Arrival Time) is calculated to ensure the complete execution of parents before child nodes get the processors. The population is initialized by assigning tasks to random processors and scheduling them. Then, fitness value is calculated to select the best chromosome for the formation of offsprings through crossover and mutation.

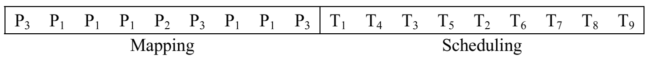


Fig. 3(a). A chromosome model.

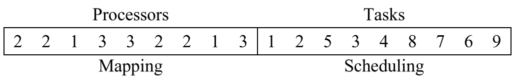


Fig. 3(b). A typical chromosome for the task graph shown in Fig. 3(b) (generated by our code).

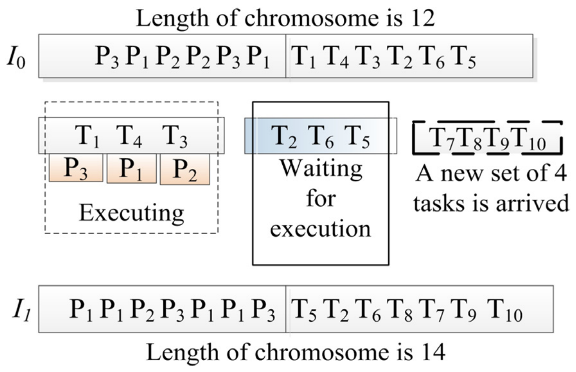

Length of chromosome is 12
I_0
P3 P1 P2 P2 P3 P1 | T1 T4 T3 T2 T6 T5
T1 T4 T3
P3 P1 P2
Executing
T2 T6 T5
Waiting for execution
T7 T8 T9 T10
A new set of 4 tasks is arrived
I_1
P1 P1 P2 P3 P1 P1 P3 | T5 T2 T6 T8 T7 T9 T10
Length of chromosome is 14

Fig. 4. Chromosome length at different point of time.

3.5.1. Elements of GA for task allocation and scheduling

The details of our GA encoding for DAG is illustrated below, which also gives the details of the fitness function and the procedure for its evaluation.

(i) *Representation of Genes:* We used decimal number to represent genes of chromosome. A chromosome is generated randomly consisting of two parts: Mapping and Scheduling. The mapping part contains processor indices where the tasks are to be run, whereas the scheduling part determines the sequence for processing the tasks. Let, n be the number of tasks available. Thus, the length of the chromosome becomes twice the number of the tasks available, i.e. $2n$. We have used a variable length chromosome [5] to encode the variations in the number of tasks due to the arrival of new tasks.

The elements of the mapping part are generated randomly from $\{1, 2, \ldots, m\}$, where m is the total number of processors. Here, we consider a 3-processor RTS, so $m = 3$. Similarly, the scheduling part is chosen randomly to maintain the precedence order of the tasks (available) in the graph. Then, the mapping part and the scheduling part are clubbed together to represent a complete chromosome as shown in Fig. 3(a). It demonstrates that task T_1 is scheduled on processor P_3, T_2 on P_2, T_3 on P_1 and so on. Fig. 3(b) represents a typical chromosome generated by our code for the task graph shown in Fig. 1.

At a particular instant, the number of arrived tasks along with tasks which are waiting may be different from the earlier instance. Therefore, the chromosome needs to be encoded with different variables for the new set of tasks if change has occurred. Thus, the size of the chromosome along with the values of genes changes over the time.

Fig. 4 illustrates an example for chromosome encoding at two different instances I_0 and I_1. Suppose at I_0 six different tasks, $\{T_1, T_2, T_3, T_4, T_{5,} T_6\}$ arrive into the system; so, a chromosome is encoded which is of length 12. Let us assume, a distributed system with three processors $\{P_1, P_2, P_3\}$, all with their ready time zero (i.e. all three processors are free). Also, say out of six tasks, T_1, T_4 and T_3, are assigned to processor P_3, P_1, and P_2 respectively, for execution. So, the remaining three tasks T_2, T_6 and T_5 stay in waiting state. They will be executed only when all required inputs and the assigned processor is free. Meanwhile, $\{T_7, T_8, T_9, T_{10}\}$ arrives in the system; so, a change is sensed and algorithm starts from the scratch, beginning with the encoding of chromosome. At I_1, a chromosome is encoded with new decision variables considering the arrived tasks (T_7, T_8, T_9, T_{10}) and the tasks waiting for execution (T_2, T_6, T_5). Thus, the length of the chromosome at I_1 becomes 14 for a total of 7 different tasks.

(ii) *Population Initialization:* Each chromosome in the population is considered as a candidate solution. Therefore, a certain number of chromosomes are generated randomly to make the initial population. The set of chromosomes represent diverse solutions that help to scan the search space.

(iii) *Fitness function and evaluation:* Here, the main aim is to minimize the schedule length/makespan such that eventually the lateness of all the tasks is also optimized. That is, the goal is to find a schedule of tasks which minimizes the total completion time of n tasks on m number of processors. Thus, smaller the schedule length, better is the solution. Therefore, here we consider the following function of the schedule length (S_length) as the fitness function:

$$Fn = (a/S\_length) \tag{5}$$

Here, a is a constant and S_length is determined by the Eq. (4). The fitness value (Fn) of each chromosome is attached to it. From Eq. (5) we can observe that, lower the S_length value, higher will be the fitness of a solution.

The pseudo code of the task scheduler for determining the starting time of the tasks and the schedule lengths using ^{d}GA-RTS is as follows [12]:

```
1. RT $(P_j)$ // Ready time of processors
2. Let LT be a list of tasks according to the topological order of DAG.
3. For i = 1 to n
      a. Remove the first task $T_i$ form the list LT.
      b. For j = 1 to m
            Processors
              If $T_i$ is scheduled to processor $P_j$
                 $ST(T_i, P_j) = max\{RT(P_j), DAT(T_i)\}$
                 $FT(T_i, P_j) = ST(T_i, P_j) + ET(T_i)$
                 $RT(P_j) = FT(T_i, P_j)$
              End If
            End For
   End For
      c. S_Length = max(FT)
```

Here, $RT(P_j)$ represents the ready time of the processor P_j, $ST(T_i, P_j)$ is the start time for the execution of task T_i on processor P_j, $DAT(T_i)$ is the data arrival time of task T_i, $FT(T_i, P_j)$ is the completion time of task T_i on the processor P_j, $ET(T_i)$ is the time required to execute task T_i, LT is the list of tasks from scheduling part of a chromosome in topological order and S_Length is the time at which all the tasks in LT end up their execution i.e. the maximum finishing time.

(iv) *Selection scheme*: After evaluating the fitness, the chromosomes with better fitness score are selected. From the various available approaches, binary tournament selection [43] is used for the selection. Here, two chromosomes are selected randomly for further operations. This selection process is repeated for all the individuals in the population.

(v) *Genetic operators*: There are two important genetic operators: crossover and mutation. We now explain each of them below:

a. Crossover: As we have considered an environment where tasks are dependent, it forms a task graph topology with different levels. As discussed earlier, chromosomes are divided into two parts, mapping and scheduling. Every chromosome is likely to crossover with a probability pc. The probability of crossover is taken as $pc = 0.8$. Two chromosomes from the population are selected using tournament selection and a number r is generated randomly between [0, 1]. If $r \leq 0.5$, then crossover is done on the mapping part of the chromosome, otherwise it is performed on the scheduling part.

The mapping part goes under the single point crossover. An integer is generated randomly between 1 to n to select a crossover point. The parts of chromosomes on the right of crossover point are then exchanged to produce two new offsprings. Fig. 5 shows an example of one point crossover on the mapping part, where the scheduling part is also attached with the chromosome to show the changes in the processor allocations before and after the crossover operation.

If $r > 0.5$, then ordered crossover [12,44] operator is applied to the scheduling part of the chromosome and a random point p is chosen such that $n < p < 2n$. First, pass the left segment from the Parent1 to the offspring, and then construct the right segment of the offspring according to the order of the right segment of Parent2 as shown in Fig. 6.

b. Mutation: A single chromosome is selected randomly for mutation at a time. Each position in the mapping part is likely for mutation operation with a probability $pm = 0.02$. Fig. 7 shows a random mutation operation performed on a chromosome. It changes the allocation of a task from one processor to another. After the mutation operator, the assignment of the task T_5 is changed from processor P_1 to processor P_3.

After genetic operations, fitness values of the generated offspring(s) are evaluated and the worst individuals in the population are replaced with offsprings having better fitness values.

(vi) *Terminating Criteria:* Evolution of population will stop once the terminating criteria are met. The scheduler should produce schedules within reasonable period. Two conditions for terminating ^{d}GA-RTS are applicable viz., (i) fixed number of generations, and (ii) when the makespan value is unchanged even after a large number of generations. Our study is focused only on the first criteria.

3.5.2. The scheduler model

We now explain the scheduler model, which is an improved version of the one described in [45]. The schematic diagram of the scheduler for a distributed environment is shown in Fig. 8. Assume that all the tasks arrive at a central processor where a scheduling algorithm called central scheduler is encapsulated to find a schedule.

Arrived tasks first enter into a queue called Initial Task Queue (ITQ) and the central scheduler is triggered simultaneously. In this model, only the execution times and the deadlines of the task along with their precedence and communication costs are known when they have entered the ITQ. It is the central schedule which is responsible for mapping the real-time tasks to different processors and their scheduling on individual processors for execution in such a way that the total schedule length is minimized. The generated schedule is also maintained at the ITQ in scheduled order. Thus, the utility of the ITQ is to hold the tasks in scheduled order until they are dispatched for processing on allocated processors. Also, a list is maintained consisting the details of the processors, called Processor Status Window (PSW). It maintains a list of running and idle processors with the details of tasks running or completed their execution on different processors. A task is dispatched for processing once the allocated processor is free and all of its required inputs are available. In addition to the ITQ and the PSW, the scheduler also manages a Finished Task Queue (FTQ) to maintain the details of the

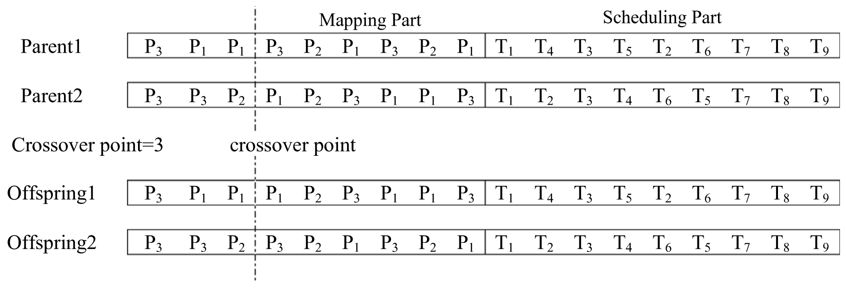


Fig. 5. Single point crossover.

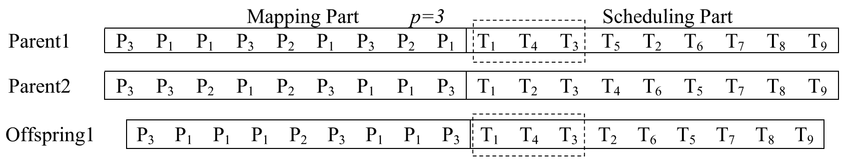


Fig. 6. Ordered crossover.

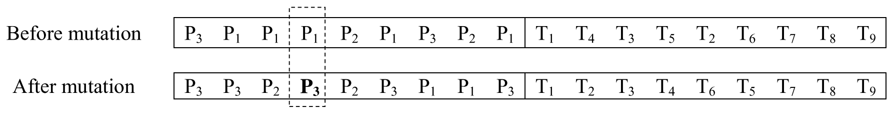


Fig. 7. Mutation.

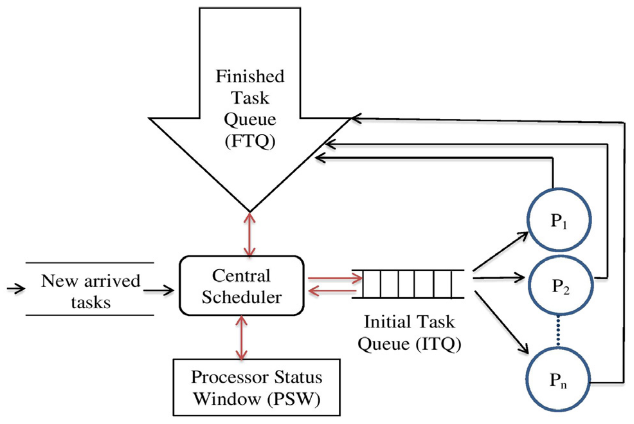


Fig. 8. Task scheduler for dynamic task scheduling in RTS.

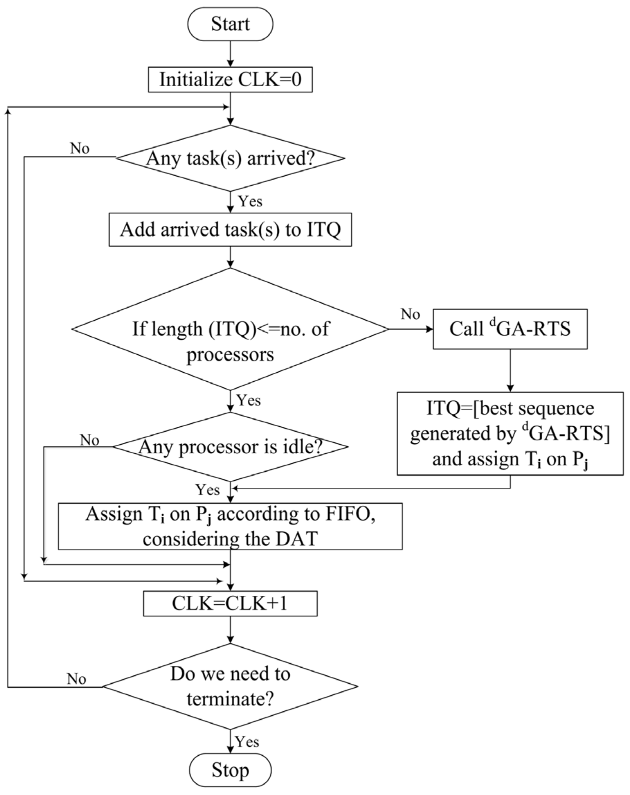

Start
Initialize CLK=0
Any task(s) arrived?
No
Yes
Add arrived task(s) to ITQ
If length (ITQ)<=no. of processors
No
Call dGA-RTS
ITQ=[best sequence generated by dGA-RTS] and assign Ti on Pj
Yes
Any processor is idle?
No
Yes
Assign Ti on Pj according to FIFO, considering the DAT
CLK=CLK+1
Do we need to terminate?
No
Yes
Stop

Fig. 9. Flowchart of the proposed algorithm.

tasks which have completed their execution. It gets updated with all the information such as processor number, completion time and parents of a task once it completes execution on a particular processor. These data at the FTQ will help to decide the data arrival time, starting time and sequence of processing of task waiting at the ITQ.

3.5.2.1 Working of the proposed scheduler model

The working of the proposed scheduler is illustrated in the form of a flowchart in Fig. 9. The scheduler starts from the clock 0. At clock $(CLK) = 0$, if any task arrives, the central scheduler adds the newly arrived task to ITQ and checks for the idle processor. The task arrived may be independent or dependent on some other task(s). Here two scenarios may arise.

First, if the number of task in the ITQ is less than or equal to the number of the free processors, the tasks are allocated to these processors according to FIFO (first in first out) considering the precedence of the tasks. Secondly, if the number of tasks arrived is greater than the number of processors, then the ^{d}GA-RTS, encapsulated in the central scheduler, will be invoked to find the schedule with the minimum schedule length that ensures deadline compliance. The parameters passed to ^{d}GA-RTS are the number of tasks, precedence information, ready time of each processor and *DAT* for each task. The schedule obtained will be maintained

Clock	*0**	*1*	*2**	*3*	*4*	*5*
Processor1	$T_1(2)$	$T_1(1)$	$T_4(4)$	$T_4(3)$	$T_4(2)$	$T_4(1)$

Fig. 10. Typical clock timings.

in the ITQ. The first task in the queue will be assigned a specific processor only if the respective processor is free. The PSW will be updated with this information for further execution. Generally, the *DAT* for independent task is 0. So, it will be assigned directly to a free processor. But for a dependent task, if the required input is available, then it will be assigned to a processor. Otherwise, it will have to wait in the ITQ till the required data is available. That means, the task at a particular clock in the ITQ will be scheduled to the processor considering the *DAT* and the ready time of the individual processors. As shown in Fig. 10, after increment of every clock unit, the remaining execution time of the tasks will be decreased by a unit. The remaining time for completing task T_i at time t is denoted as $T_i(k)$. The bold line on the boundary of column shows the finishing length of the task. For example, $T_1(2)$ starts at clock 0 and finishes at the end of clock 1 with total execution time of 2 units. It also shows that the processor is free and new task is dispatched from ITQ for execution. Also, at every clock increment, the scheduler will check for newly arrived tasks and whether any processor is free. If any new task is arrived, then the scheduler will make a revisit to the schedule. Otherwise, the task already dispatched on the processors will continue execution. The ITQ is updated only on the arrival of a new task. ^{d}GA-RTS will be called only if number of tasks in the ITQ is larger than the number of processors. The ^{d}GA-RTS terminates at a finite number of iterations.

3.5.2.2 Example to illustrate the working of the proposed technique using a synthetic task sets of 15 tasks

We now consider a set of 15 tasks (STG2) with randomly generated synthetic task parameters. The characteristics for each task are mentioned in Table 2. The first column represents the task number, second column gives the execution time of each task and the communication cost of the tasks is given in the third column. The fourth column represents the parent of task T_i and the fifth column shows the clock at which the task T_i will arrive at the scheduler. Columns 6, 7 and 8 give the deadline, dlaxity and level of a particular task. Tasks T_1 and T_2 are independent tasks. We consider a multiprocessor RTS with three processors for executing the task set with population size 10. Fig. 11 illustrates the run time anomaly of parallelized tasks. Fig. 12 shows the scheduled tasks in the ITQ at different clocks.

At the beginning of clock 0, first seven tasks T_1 to T_7 arrive at the scheduler. At this time, all the three processors are free i.e. ready time of each processor is 0. Since the number of tasks arrived at this clock is greater than the number of processors available, ^{d}GA-RTS will be invoked. The generated schedule is (P_1 P_2 P_3 P_2 P_2 P_3 P_1 | T_1 T_2 T_5 T_3 T_6 T_4 T_7), which gives a makespan of 11 units. The tasks $T_1(2)$ and $T_2(3)$ are dispatched on P_1 and P_2 for execution. The PSW is updated. P_3 remains free as task T_3 and T_4 are children of T_1. So, they cannot run until T_1 is completed. Hence, the precedence constraints of the tasks T_3 and T_4 are maintained, i.e., required input data for their execution will only be available after parent T_1 finishes its execution. So, the ordering of the remaining tasks in ITQ will be (P_3 P_2 P_2 P_3 P_1 | T_5 T_3 T_6 T_4 T_7). At CLK 1, no task is arrived and the DAT of T_3 and T_4 is not met since $T_1(1)$ has not finished yet. So, ITQ remains same. At the end of clock 1, T_1 is finished and P_1 is free. PSW is updated with status of P_1 as idle and FTQ is updated with the finishing time of T_1 and number of the processor on which it was executed.

Table 2
Characteristics of the set of 15-tasks.

Task	ET	CC	Parent	AT	Deadline	Dlaxity	Level (depth)
T_1	2	...	...	0	4	1	1
T_2	3	...	...	0	4	1	1
T_3	3	1	T_1	0	16	2	2
T_4	4	1	T_1	0	16	2	2
T_5	5	2	T_2	0	24	3	2
T_6	5	4	T_2	0	24	3	2
T_7	5	4	T_1	0	24	3	2
T_8	4	2	T_4	2	36	3	3
T_9	1	5	T_3	2	36	3	3
T_{10}	1	4	T_6	2	36	3	3
T_{11}	10	2	T_7	7	36	3	3
T_{12}	8	4	T_5	7	36	3	3
T_{13}	4	1	T_8	7	64	4	4
T_{14}	6	4	T_2	8	24	3	2
T_{15}	5	2	T_9	8	64	4	4

At the beginning of clock 2, T_8, T_9 and T_{10} arrive at ITQ. The length, i.e., number of tasks, in the ITQ is greater than the no. of processors. So, ^{d}GA-RTS is invoked by sending the tasks to the ITQ with all required inputs. The schedule obtained is (P_1 P_3 P_2 P_1 P_2 P_3 P_1 P_3 | T_4 T_3 T_7 T_6 T_5 T_9 T_8 T_{10}), where the schedule length is of 16 units. The ready time of P_1, P_2 and P_3 are 0, 1 and 0, respectively. So, the first task in the ITQ is dispatched to the respective idle processor; i.e., $T_4(4)$ is scheduled on P_1. The communication cost is taken as zero, i.e., not considered, as parent T_1 was also completed on the same processor. At the end of the execution of all the tasks, two separate DAGs are formed with roots at T_1 and T_2. The dependent (or child) tasks of these two DAGs will arrive at different clocks of time. Fig. 13 gives the generated DAGs. The execution length of the best schedule generated by ^{d}GA-RTS at different clocks when task arrives is shown in Fig. 14. However, the communication cost for $T_3(4)$, which is scheduled on P_3 is taken into account since its parents T_1 was completed on a different processor P_1. Hence, the total execution length of T_3 will be equal to execution time plus communication cost, i.e. $3 + 1 = 4$ at the current state. The ITQ will remain with (P_2 P_1 P_2 P_3 P_1 P_3 | T_7 T_6 T_5 T_9 T_8 T_{10}). The same procedure is followed for every clock. The total execution length of the task set is 34 units as visible from the task schedule shown in Fig. 11. Task is dispatched for running on scheduled processor only if the required data is available and the processor on which it is scheduled is free. As shown in Fig. 15, the lateness of all the tasks are negative. That is, whole task set will finish execution without any deadline miss. We see that the proposed approach has capably solved the dynamic task allocation problem for the given task set and produces a near optimal schedule which gives us best possible makespan value ensuring complete deadline compliance.

4. Simulations and results discussion

A number of simulation experiments are conducted to investigate the functioning of the proposed solution technique. Our experiments have been based on benchmark task sets [46]. We have presented experimental results for two different cases: static and dynamic, with suitable task sets. For static case, all tasks are assumed to arrive at a particular instance of time, whereas in the dynamic case we consider arbitrary arrival time for each task. Numerical and graphical results are discussed in detail and significant findings are highlighted.

4.1. Complexity analysis

In order to evaluate the efficiency of the proposed method, we compare it with three different state-of-the-art evolutionary

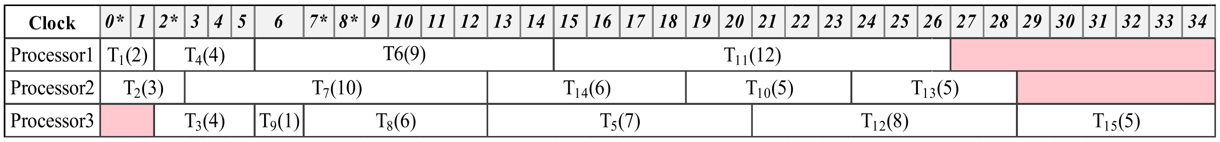
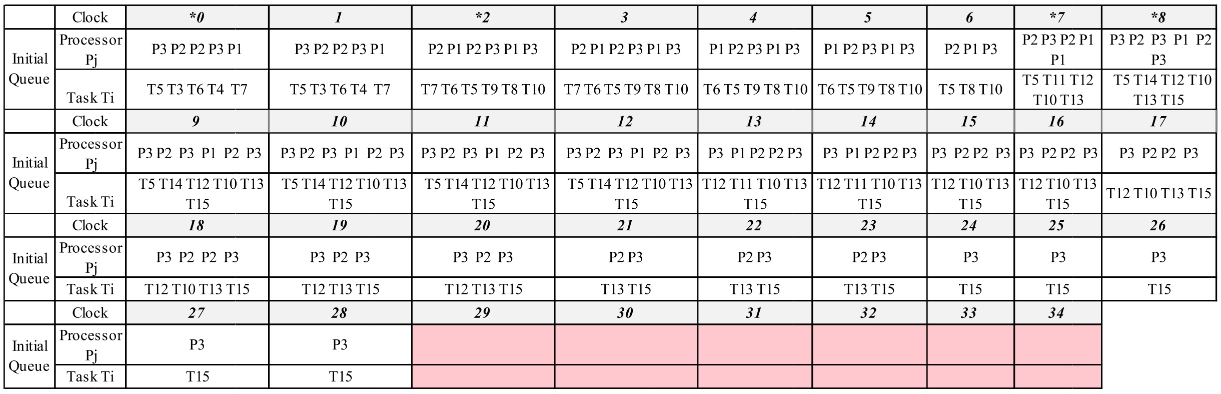


Fig. 11. Runtime anomaly of parallelized tasks.



	Clock	*0	1	*2	3	4	5	6	*7	*8
Initial Queue	Processor Pj	P3 P2 P2 P3 P1	P3 P2 P2 P3 P1	P2 P1 P2 P3 P1 P3	P2 P1 P2 P3 P1 P3	P1 P2 P3 P1 P3	P1 P2 P3 P1 P3	P2 P1 P3	P2 P3 P2 P1 P1	P3 P2 P3 P1 P2 P3
	Task Ti	T5 T3 T6 T4 T7	T5 T3 T6 T4 T7	T7 T6 T5 T9 T8 T10	T7 T6 T5 T9 T8 T10	T6 T5 T9 T8 T10	T6 T5 T9 T8 T10	T5 T8 T10	T5 T11 T12 T10 T13	T5 T14 T12 T10 T13 T15
	Clock	9	10	11	12	13	14	15	16	17
Initial Queue	Processor Pj	P3 P2 P3 P1 P2 P3	P3 P2 P3 P1 P2 P3	P3 P2 P3 P1 P2 P3	P3 P2 P3 P1 P2 P3	P3 P1 P2 P2 P3	P3 P1 P2 P2 P3	P3 P2 P2 P3	P3 P2 P2 P3	P3 P2 P2 P3
	Task Ti	T5 T14 T12 T10 T13 T15	T5 T14 T12 T10 T13 T15	T5 T14 T12 T10 T13 T15	T5 T14 T12 T10 T13 T15	T12 T11 T10 T13 T15	T12 T11 T10 T13 T15	T12 T10 T13 T15	T12 T10 T13 T15	T12 T10 T13 T15
	Clock	18	19	20	21	22	23	24	25	26
Initial Queue	Processor Pj	P3 P2 P2 P3	P3 P2 P3	P3 P2 P3	P2 P3	P2 P3	P2 P3	P3	P3	P3
	Task Ti	T12 T10 T13 T15	T12 T13 T15	T12 T13 T15	T13 T15	T13 T15	T13 T15	T15	T15	T15
	Clock	27	28	29	30	31	32	33	34	
Initial Queue	Processor Pj	P3	P3							
	Task Ti	T15	T15							

Fig. 12. Initial task queue at an instance during runtime.

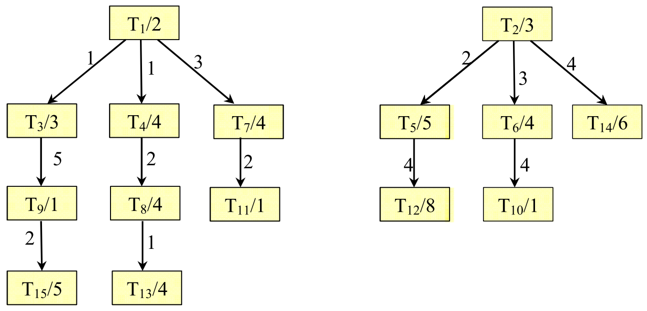


Fig. 13. Two DAGs formed at the end of simulation of all 15 tasks.

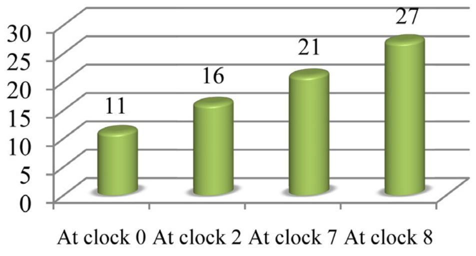


Fig. 14. Sequence execution length at different clock time by ^{d}GA-RTS.

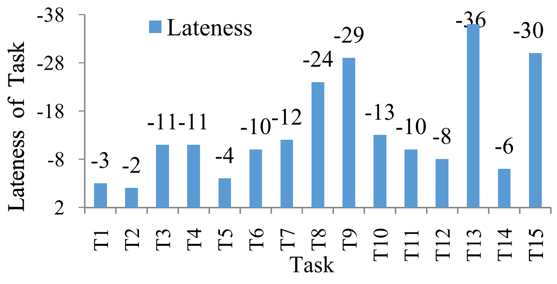


Fig. 15. Lateness of each task as per best schedule found.

methods: ABC (Artificial Bee Colony) Algorithm [47–49], PSO (Particle Swarm Optimization) algorithm [50] and SA (Simulated Annealing) algorithm [51]. The standard framework design of the algorithms is used for our experiments. Few significant elements like random generation of decision variables for encoding of chromosomes, fitness function evaluation and sorting of population are designed as per the addressed problem's requirement. We have implemented the purposed approach as well as other three approaches (ABC, PSO, and SA), considered for comparison, on MATLAB 2013a. Each of the algorithms starts with initialization of population (or colony or, swarm) of fixed size, say P. A common technique is followed to randomly generate the variables of chromosome for all algorithms. So, the time complexity of population initialization is same for every algorithm. Also, the sorting procedure used for each algorithm is of same complexity. Hence, we do not show the complexity of population initialization and sorting, as they remain a common factor in every case.

For, a system with n tasks and m processors, the complexity of fitness evaluation function (see pseudo code in 3.5.1 (iii)) for an individual will be $O(nm)$. The lateness calculation of all n tasks of an individual takes $O(n)$ complexity. Therefore, the time complexity of fitness calculation along with lateness of task of an individual becomes of $O(nm) + O(n)$, i.e. $O(nm)$.

The GA operations include tournament selection which has $O(1)$ time complexity for selecting two parents, one-point crossover and one point mutation operation of $O(1)$ and $O(1)$ complexity respectively. Fitness calculation within GA along with lateness has $O(nm)$ time complexity. Therefore, while running GA, an operation with time complexity $T = O(nm)$ for each iteration is needed.

The main operations involved in an iteration of PSO after particles initialization phase are fitness evaluation of $O(nm)$, the velocity and position updating of a particle is of $O(1)$. Therefore, the time complexity for updating the velocity and the position of P particles will be $O(P)$. Hence, the time complexity for each iteration of the PSO algorithm will be $O(Pnm)$.

After initialization of colony of bees, P recruited bees goes through $O(Pnm)$ time complexity to find neighbour food source. The new position of all P bees is updated with $O(P)$ and fitness is evaluated with $O(nm)$ along with the lateness calculation. The onlooker bees selects a food source depending on the probability obtained from fitness value provided by employed bees which is of $O(1) \times O(nm)$, i.e. the time complexity becomes $O(Lnm)$ for L number of Onlooker bees. The scout bees are identified using abandon criteria out of all P bees and the solution failing to the limit criterion goes for new search. This involves cost of $O(P)$. Also, the best solution updating is of cost $O(P)$ after comparing each solution. Therefore, the time complexity of ABC will be $O(nm\,(P+L))$for each iteration.

After initialization and evaluation, SA operations include perturbation to neighbour with $O(P)$ time complexity for P individuals on each temperature to accept the changes, the fitness evaluation with lateness is $O(nm)$, annealing to reach equilibrium state is $O(PlogP)$. Thus, for an iteration, the computational complexity of SA will be $O(P\log Pnm)$.

We conclude that the time complexity for I iterations of GA, PSO, ABC and SA will be $O(Inm)$, $O(IPnm)$, $O(Inm(P+L))$ and $O(IP\log Pnm)$ respectively. We have performed empirical study on different sets of tasks and compared the computational time taken by each algorithm.

4.2. Parameter settings of algorithms

The parameter settings for the implementation of the algorithms for experiments are mentioned in a tabular form in Table 3. The simulations were conducted on a desktop PC with 4 GB RAM and Intel(R) Core(TM) i5-2400 CPU.

4.3. Task sets

In order to check the feasibility of our proposed approach on larger task sets, experiments with standard task graphs [46] are conducted. For static case, we consider a Robot Control Program (STG1) with 90 tasks. For the dynamic case, we consider a synthetic task set (STG2), Sparse matrix solver (STG3) and SPEC Fpppp (STG4) with 15, 98 and 336 tasks respectively. The individuals in the population pool determine the diversity and landscape of the search space, which evolves to find better solution over the generations. Thus, we experiment with different combinations of population size and number of generations to analyse the behaviour of each algorithm. Table 4 presents the details about the task sets used for the experiments. For STG1, STG3 and STG4, we have fixed the population size at 10,100 and 200. For STG2, we set lesser value of population sizes, which are 5, 10 and 15 to accommodate diversified combination of individuals as it consists of only 15 tasks. Each set of population for every task set is evolved over 100, 500 and 1000 number of fixed iterations. Eventually, we come across nine different instances of each task set for the experiments.

4.4. Performance measures

Measuring the performance of an algorithm properly, is important in evolutionary optimization to evaluate how well the algorithm meets the intended goals. It is also very important to compare the algorithms to quantify the difference of performance between them. It helps to infer new information and learn about characteristics of the algorithms for the considered problem. Here, we will analyse the results obtained using few common performance measure which are discussed briefly below.

i. *Best-of-generation*: Best-of-generation (BOG) is an optimality based performance measure to evaluate the ability of the algorithms in finding the solutions with the best objective values. This measure provides quantitative comparison of algorithms' performance [52,53]. For each algorithm, at every generation, the best schedule is selected and the best makespan value of the current generation is recorded. The complete performance of EA based on optimality for R runs is averaged for measuring the online performance of the algorithm and can be calculated by using Eq. (6):

$$\overline{F}_{BOG}=\frac{1}{G}\times\sum_{i=1}^{G}\left(\frac{1}{R}\times\sum_{j=1}^{j=R}F_{BOG_{ij}}\right) \tag{6}$$

where, $\overline{F}_{BOG}$ is the online performance of the algorithm for the addressed problem, i.e., the average of best-of-generation fitness over the runs. G is the number of generations, R is the number of runs of algorithm and $F_{BOG_{ij}}$ is the best-of-generation fitness value of generation i of run j. Thus, it would be also reasonable to compare EA's according to the average makespan, since in dynamic environments the diversity of population fluctuates with the evolution. The average fitness is the mean of the fitness values of the entire population. The arrival of offspring (possibly better) into the current pool of individuals changes the population and we get new average population fitness at every generation. It is useful in depicting the convergence of individuals near to the best fitness value. We have also observed the total lateness and its average over the number of populations for each generation to ensure the feasibility of the algorithm for real-time systems.

ii. *Offline Performance:* The offline performance is used to determine the quality of solution produced between periods. In dynamic environment, it is necessary to observe the change in the optimum to encapsulate the variability in the results and compare EAs on the basis of the best solutions they can produce during each period. Mathematically, it can be defined as the average of the best solutions found at the period k over a given number periods P. If $f(Best_k)$ denotes the makespan of the best solution found in the current period k and P is the number of period, then the offline performance off can be calculated as follows:

$$off\;=\frac{1}{P}\times\sum_{k=1}^{P}f\,(Best_k) \tag{7}$$

iii. *Accuracy:* Accuracy or relative error measures the location of the best solution within an interval of upper bound (worst known solution) and lower bound (best known solution) in the search space. The main advantage of this measure is that it uses the min–max normalization to rescale the solution fitness from its initial values $[Max_t, Min_t]$ to [0, 1]. It helps to reduce the possible biases caused due to difference of fitness at different generations. Thus, higher the accuracy value, the better is the EA's performance. The accuracy of an algorithm a_t at generation t is calculated by the following formula:

$$a_t=\frac{f\,(BOG_t)-Min_t}{Max_t-Min_t} \tag{8}$$

where, $f\,(BOG_t)$ is the best objective value at tth generation. In our case, we consider makespan value as input for calculating accuracy of the algorithm.

iv. *Area Between Curves:* The population in a dynamic environment may alter due to arrival of immigrants but still the properties like best fitness or $\overline{F}_{BOG}$ may depend on the current solutions, previous population or successive one. $\overline{F}_{BOG}$ provides the result which encapsulates the behaviour of the algorithm. In searching problem, it is possible for two or more algorithms to provide same $\overline{F}_{BOG}$ with different search behaviours. Therefore, Enrique et al. [54] proposed a new measure, Area Between Curves ($ABC_A^{m_1,m_2}$) to compare the performance of a pair of algorithms in dynamic environment by quantifying the distance between the performance curves. $ABC_A^{m_1,m_2}$ between two algorithms m_1 and m_2 is the integral of the difference between two functions $Am_1\,(x)$ and $Am_2\,(x)$ over entire generations.

$$ABC_A^{m_1,m_2}=\frac{1}{G}\cdot\int_1^G Am_1\,(x)-Am_2\,(x)\,dx \tag{9}$$

Here, $Am_i\,(x)$ where $i\in\{1,2\}$, can be replaced by any measure. In our case, we use $\overline{F}_{BOG}$ to define the area as it will explain the difference between the algorithms having closer $\overline{F}_{BOG}$ values. A negative $ABC_A^{m_1,m_2}$ value infers that m_1 curve is lower than m_2, otherwise it is higher.

Table 3
Parameter settings of the algorithms for simulation.

Parameter settings			
^{d}GA-RTS	PSO	SA	ABC
probability of crossover $(pc) = 0.8$	inertia coefficient $(w) = 1$	maximum number of sub-iterations $= 5$	acceleration coefficient $(a) = 1$
probability of mutation $(pm) = 0.1$	damping ratio of w (wdamp) $= 0.99$	initial temperature $(t0) = 0.1$	abandonment limit parameter $(l) = 0.5 \times$ colony size[a]
tournament size $= 2$	personal acceleration coefficient $(c1) = 2$	temperature reduction rate $(tr) = 0.99$	number of onlooker bees $= 0.5 \times$ colony size
number of offsprings $(nc) = 2 \times$ round($pc \times$ population size/2)	social acceleration coefficient $(c2) = 2$	number of neighbours per individual $= 5$	number of employed bees $= 0.5 \times$ colony size
number of offsprings $(nm) = 2 \times$ round($pm \times$ population size)	max. velocity $(maxV)$ = randomly chosen between 1 and 10 Min. Velocity $(minV) = -maxV$	mutation rate $= 0.1$	number of scout bee $= 0.5 \times$ colony size

[a]For STG1, STG3 and STG4: simulation was conducted with population size/swarm size/colony size of 10, 100 and 200 for 100, 500 and 1000 generations. For STG2: simulation was conducted with population size/swarm size/colony size of 5, 10 and 15 for 100, 500 and 1000 generations.

Table 4
Details about task sets used for experiment.

Task Set	No. of tasks	Represented as	Case	Population size for experiment
Robot control program	90	STG1	Static	10, 100 & 200
Synthetic task	15	STG2	Dynamic	5, 10 & 15
Sparse matrix solver	98	STG3	Dynamic	10, 100 & 200
SPEC Fpppp	336	STG4	Dynamic	10, 100 & 200

4.5. *Static case: Scheduling of precedence based tasks without arrival time*

A benchmark Robot Control Program [46] is taken for the simulation. The problem consists of a task set with 90 tasks which are represented by the standard task graph shown in Fig. 16. The arrival time for all 90 tasks in the set is assumed to be zero i.e., all tasks arrive at the same time in the system. This gives the scenario of static task allocation and scheduling. Table 5 consists of detailed characteristics of the tasks. We present the experimental results for single run of ^{d}GA-RTS using STG1. A fixed set of communication costs are considered for the weight of edges between the dependent tasks.

4.5.1. *Experiment with a fixed population size over different number of generations for single run of ^{d}GA-RTS*

Here, we have considered a three processor RTS where the ready times of all the three processors are initially taken as zero. We set initial populations size to 200 and number of generations as 100, 500 and 1000. The crossover probability of (pc) and the mutation probability (pm) were set at 0.8 and 0.02, respectively. Considering the optimality of the solution produced, we have observed that, with the increase in the number of generations, significant improvements in the results are found.

The experiment for the 100 generations gives a reduction in the makespan value from 853 to 830. Analysing the result after 1000 generations, we see the convergence with the makespan value of 829 units. Fig. 17 illustrates the best schedule found after the 1000th generations. Fig. 18(a) gives the makespan versus number of generations plot, whereas Fig. 18(b) gives the plot of average makespan versus the number of generations.

In both the cases, a significant reduction in the makespan value is visible. An improvement is achieved in the average makespan for increasing generations. However, reduction is slow as number of generations increases. We have examined the feasibility of the generated schedule with the best makespan value obtained after 1000 generations. It is found that all the tasks complete execution well before the deadlines. Thus, we can conclude that the generated schedule is best with minimum makespan value of 829 and average lateness of −539.778 per task. Fig. 18(c) shows the plot of the total lateness averaged over the number populations for 100, 500 and 1000 generations. On the other hand, Fig. 18(d) shows the average lateness per task in the best schedule after 100, 500 and 1000 generations. The lateness of each task of the best schedule found after 100, 500 and 1000 generations are shown in the form of histogram in Fig. 19. The comparative results for 100, 500 and 1000 generations are summarized in Table 6. As shown in bold in the third row of Table 6, we could find the best value of the makespan, total lateness and the respective average values after 1000 generations.

4.5.2. *Experiment with a fixed number of generation using different population sizes for single run of ^{d}GA-RTS*

Furthermore, we have analysed the results for different population size of 10, 100 and 200 keeping the number of generations fixed at 1000. Table 7 gives the comparative results for different population sizes for 1000 generations.

Fig. 20(a)–(d) show the comparative graphs for best makespan, average makespan, average total lateness and average lateness, respectively. We can observe from these figures that, with the increase in the population size convergence becomes faster. Thus, we see that our proposed approach is able to solve the benchmark robot control program and produce the efficient schedule with minimum possible scheduling results.

4.5.3. *Results for 30 runs and comparative study*

We have implemented ABC, PSO and SA for the addressed problem to compare the efficiency of proposed technique. It is very difficult to analyse EAs theoretically as they are heuristics and often unpredictable for similar parameters at different instances. So, we perform 30 independent runs of each algorithm for every instance of all the task sets.

We recorded the BOG, i.e., best schedule's makespan and total lateness value of each generation and calculated the $\overline{F}_{BOG}$ for each experiment. In order to observe the convergence of evolution, we have also noted average makespan, average total lateness, average lateness and total computation time for performance analysis. Table 8 represents the average values of different measurements of each algorithm for comparative study. The result of simulation shows that with increase in the generations, there are improvements in the scheduling results for all the cases. Also, it can be observed that with the increase in the population, the rate of

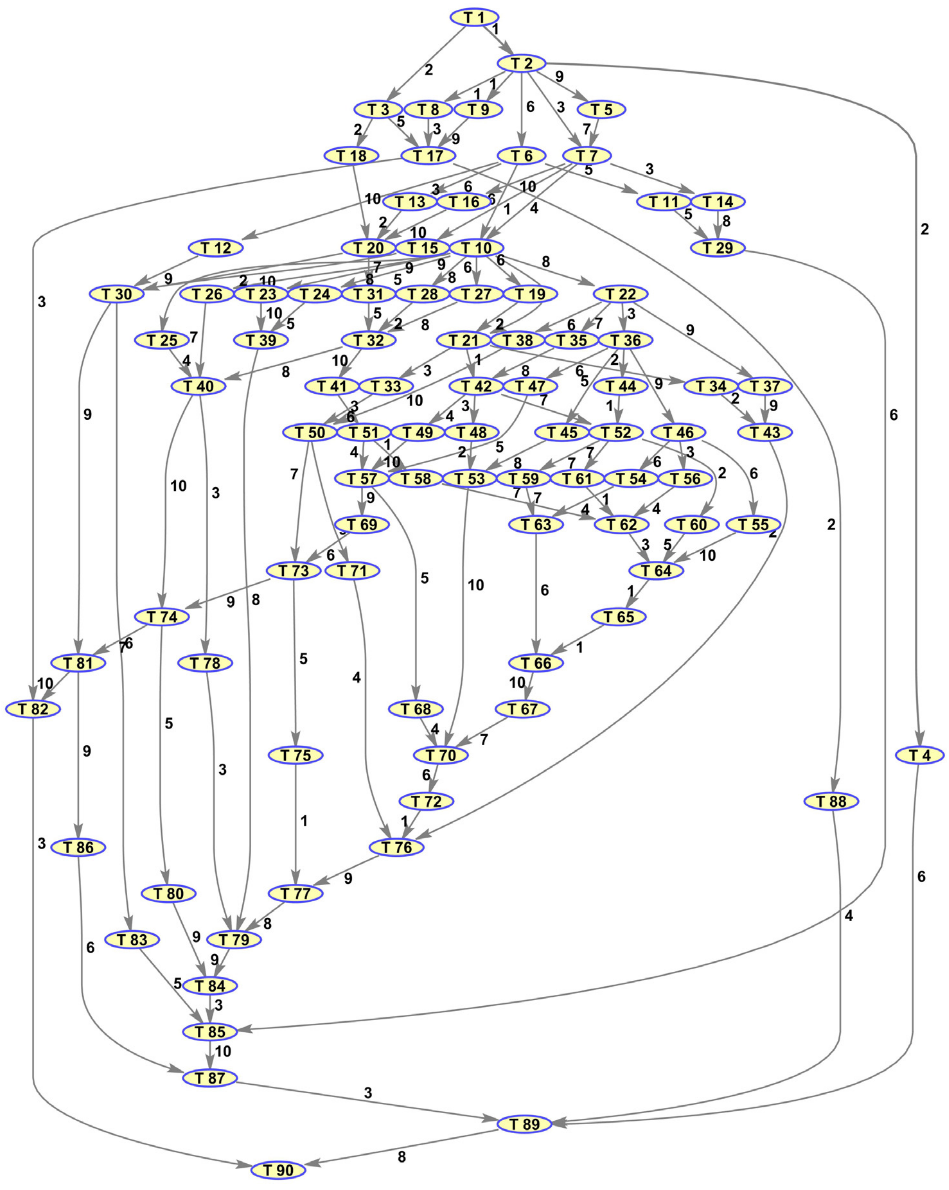


Fig. 16. Task graph of the benchmark robot control program with 90 Tasks.

P1	T18	T6	T11	T16	T25	T19	T29	T28	T27	T30	T36	T21	T83	T47	T34	T32	T52	T48	T56	T40	T71	T57	T68	T64	T75	T81	T66	T86	T77	T89	T90
P2	T5	T4	T17	T12	T14	T88	T15	T10	T22	T38	T37	T31	T46	T45	T49	T41	T78	T60	T59	T69	T74	T65	T80	T82	T70	T76	T79	T84			
P3	T1	T3	T2	T8	T9	T13	T7	T23	T24	T26	T20	T39	T35	T42	T33	T44	T50	T55	T54	T43	T53	T61	T51	T58	T63	T62	T73	T67	T72	T85	T87

Fig. 17. Best task assignment and schedule obtained after 1000 generations.

convergence is faster as the best makespan value and average makespan value is lesser for large population size compared to small population size. The experiment shows significant improvement at the end of evolution for all the techniques. Boldface value indicates the best value of the respective algorithm.

From Table 8, we can observe that for population size 10 and generation 100, ^{d}GA-RTS has achieved better results compared to SA, ABC and PSO. On the other hand, with increase in the number of generations, SA has been able to achieve solutions with minimum makespan but not much varied from ^{d}GA-RTS. However, PSO has completed its execution quicker than others but has not been able to achieve better measurement values. Also, the completion time for SA is quite larger than ^{d}GA-RTS for every case. Thus, we can conclude that our approach has worked well for the static case. A comparative plots for running time of all four algorithms for different instances of STG1 has been illustrated in Fig. 21. Fig. 22 graphically illustrates the comparison of the algorithms over best makespan, average makespan, total lateness of best schedule and average total lateness for 200 populations and 1000 generations for STG1.

4.5.4. Experiments on larger task sets for static case

To demonstrate the scalability of our proposed technique, we have performed simulation using ten randomly selected comparatively larger task sets with different number of nodes and levels from [46]. The schedules for these benchmark task sets have been generated without considering any communication cost. Therefore, we generate a set of communication costs for each task set

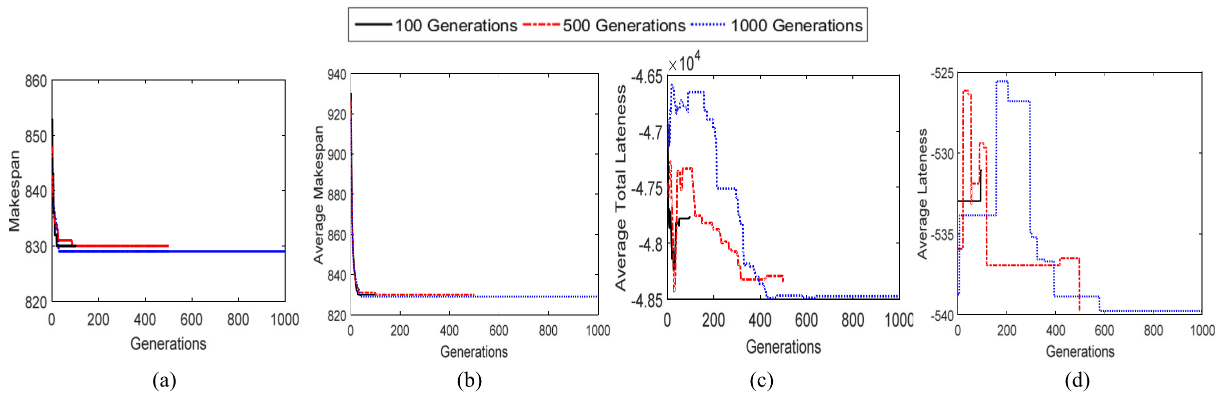


Fig. 18. (a) Best makespan, (b) Average makespan of population, (c) Average total lateness of population and (d) Average lateness in the best schedule with varying generations.

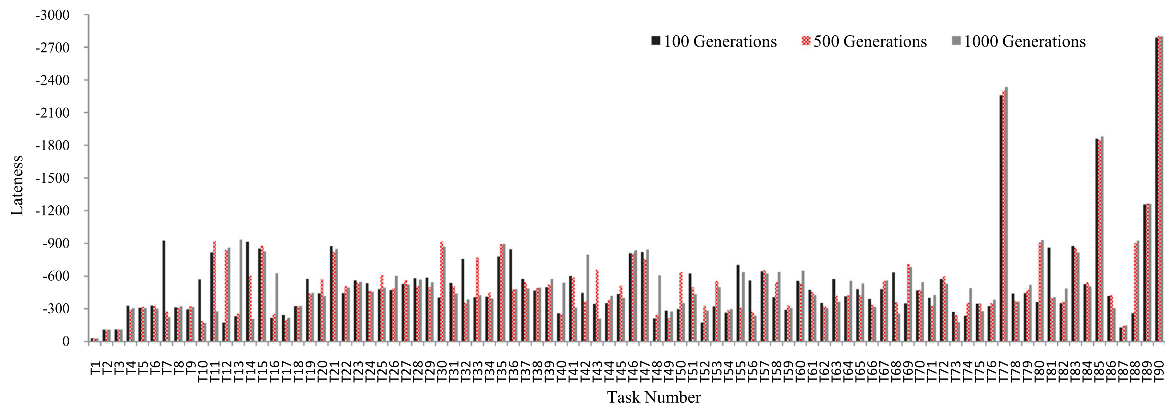


Fig. 19. Lateness of each task in the best schedule obtained after 100, 500 and 1000 generations.

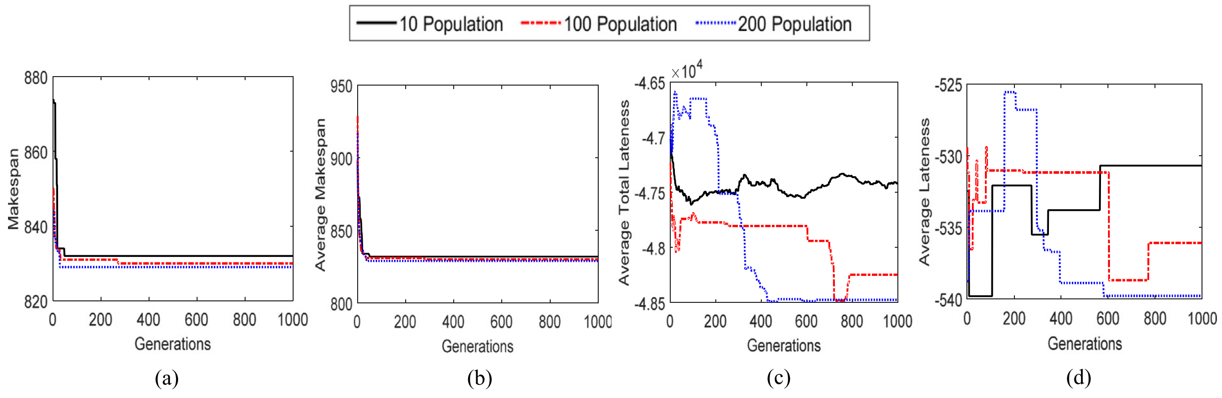


Fig. 20. (a) Best makespan, (b) Average makespan, (c) Average total lateness and (d) Average lateness in the best schedule for population size 10, 100 and 20.

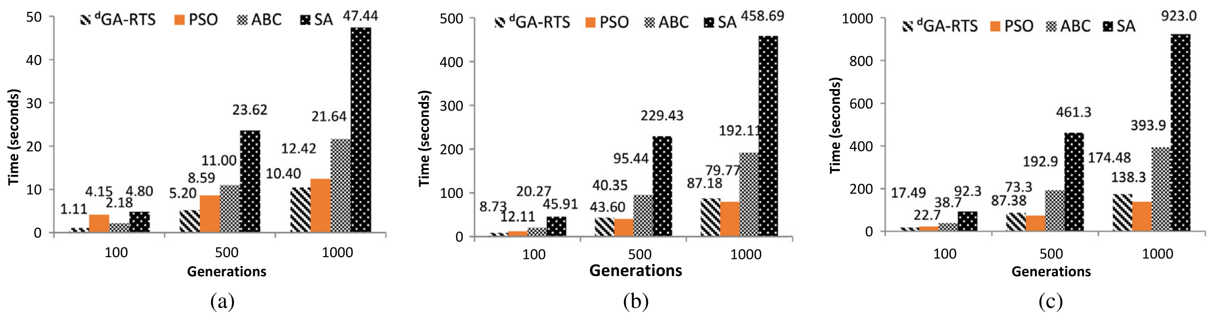


Fig. 21. Running time of the algorithms for ^{d}GA-RTS, PSO, ABC and SA for different population size : (a) 10, (b) 100 and (c) 200 over 100, 500 and 1000 generations for the STG1 task set.

Table 5
Task characteristics for the benchmark robot control program.

Task	ET	Parent	CC	Deadline	Dlaxity	Level	Task	ET	Parent	CC	Deadline	Dlaxity	Level
T1	0	. . .	. . .	28	1	1	T46	32	T36	9	672	3	8
T2	1	T1	1	56	1	2	T47	66	T36	6	672	3	8
T3	1	T1	1	112	2	2	T48	15	T42	3	756	3	9
T4	5	T2	2	336	4	3	T49	24	T42	4	756	3	9
T5	5	T2	9	336	4	3	T50	39	T33,T38	6,10	756	3	9
T6	10	T2	6	336	4	3	T51	28	T41	3	1120	4	10
T7	28	T2 , T5	3,7	336	3	4	T52	40	T42,T44	7,1	756	3	9
T8	5	T2	1	336	4	3	T53	12	T45,T48	8,2	1120	4	10
T9	10	T2	1	336	4	3	T54	111	T46	6	756	3	9
T10	1	T6 ,T7	6,4	560	4	5	T55	84	T46	6	756	3	9
T11	57	T6	5	336	3	4	T56	38	T46	3	756	3	9
T12	66	T6	3	336	3	4	T57	39	T47,T49,T51	5,10,4	924	3	11
T13	38	T6	6	336	3	4	T58	28	T51	1	924	3	11
T14	15	T7	3	980	7	5	T59	69	T52	7	1120	4	10
T15	24	T7	1	980	7	5	T60	42	T52	2	1120	4	10
T16	10	T7	10	980	7	5	T61	10	T52	7	840	3	10
T17	15	T3,T8,T9	5,3,9	336	3	4	T62	24	T56,T58,T61	4,7,1	1008	3	12
T18	10	T3	2	336	4	3	T63	12	T54,T59	4,7	924	3	11
T19	10	T10	6	672	4	6	T64	39	T55,T60,T62	10,5,3	1456	4	13
T20	24	T13,T16,T18	2,10,10	672	4	6	T65	42	T64	1	1176	3	14
T21	40	T10,T19	5,2	784	4	7	T66	12	T63,T65	6,1	1260	3	15
T22	32	T10	8	672	4	6	T67	28	T66	10	1344	3	16
T23	57	T10	8	672	4	6	T68	24	T57	5	1008	3	12
T24	15	T10	5	672	4	6	T69	40	T57	9	1008	3	12
T25	10	T10	9	672	4	6	T70	24	T53,T67,T68	10,7,4	952	2	17
T26	38	T10	9	672	4	6	T71	24	T50	9	1120	4	10
T27	53	T10	6	672	4	6	T72	28	T70	6	1008	2	18
T28	10	T10	8	672	4	6	T73	40	T50,T69	7,6	1092	3	13
T29	12	T11,T14	5,8	672	4	6	T74	40	T40,T73	10,9	1176	3	14
T30	39	T12,T15,T20	9,10,2	784	4	7	T75	38	T73	5	1176	3	14
T31	4	T20	7	784	4	7	T76	24	T43,T71,T72	2,4,1	1596	3	19
T32	24	T27,T28,T31	8,2,5	1120	5	8	T77	36	T75,T76	1,9	1120	2	20
T33	24	T21	3	1120	5	8	T78	10	T40	3	1120	4	10
T34	15	T21	5	1120	5	8	T79	24	T39,T77,T78	8,8,3	1176	2	21
T35	10	T22	7	784	4	7	T80	10	T74	5	1260	3	15
T36	32	T22	3	784	4	7	T81	12	T30,T74	9,7	1260	3	15
T37	57	T22	9	784	4	7	T82	40	T17,T81	3,10	1344	3	16
T38	66	T22	6	784	4	7	T83	24	T30	6	1120	5	8
T39	12	T23,T24	10,5	784	4	7	T84	8	T79,T80	9,9	3080	5	22
T40	39	T25,T26,T32	4,7,8	756	3	9	T85	24	T29,T83,T84	6,5,3	1288	2	23
T41	28	T32	10	756	3	9	T86	24	T81	9	1344	3	16
T42	40	T21,T35	1,8	672	3	8	T87	36	T85,T86	10,6	2688	4	24
T43	12	T34,T37	2,9	756	3	9	T88	24	T17	2	980	7	5
T44	10	T36	2	224	1	8	T89	24	T4,T87,T88	6,3,4	2100	3	25
T45	57	T36	5	672	3	8	T90	0	T82,T90	3,8	3640	5	26

Table 6
Comparative results for 100, 500 and 1000 generations with population size 200.

Generations	Best makespan		Best total lateness		Average makespan		Average total lateness		Average lateness		Time
	From	To	From	To	From	To	From	To	From	To	
100	853	830	−47 968	−47 796	930.390	830	−46 934.900	−47 766.400	−532.978	−531.067	17.472
500	848	830	−48 234	−48 576	926.545	830	−47 460.700	−48 344.000	−535.933	−539.733	87.454
1000	844	**829**	−48 488	**−48 580**	917.115	**829**	−46 975.800	**−48 474.100**	−538.756	**−539.778**	174.300

Table 7
Comparative results for 10, 100 and 200 population size over 1000 generations.

Population size	Best makespan		Best total lateness		Average makespan		Average total lateness		Average lateness		Time
	From	To	From	To	From	To	From	To	From	To	
10	874	832	−47 922	−47 762	898.400	832	−47 016.47	−47 420.49	−532.467	−530.689	10.343
100	850	830	−44 392	−44 472	928.820	830	−44 344.71	−44 472.32	−532.467	−530.689	87.095
200	844	**829**	−48 488	**−48 580**	917.115	**829**	−46 975.80	**−48 474.10**	−538.756	**−539.778**	174.300

using uniform distribution for dependent tasks to find solutions by the proposed approach. The arrival time of all the tasks are considered to be zero to make the scenario as static i.e. all tasks arrive at the same instance of time. We perform experiment substituting ^{d}GA-RTS, PSO, ABC and SA at the central scheduler. The population is set to 100 for 1000 generations for every task set and the specific number of processors as mentioned in [46] is used for respective task sets.

Table 9 comparatively provides the best makespan, best total lateness and total execution time averaged over 30 independent runs of each algorithm for every task set. It can be observed from Table 9 that SA produces better solutions among all four algorithms but takes the largest execution time. However, ^{d}GA-RTS finishes its execution within much lesser time and produces solutions with best makespan values very nearer to SA. PSO and ABC are unable to generate solutions with better quality in reasonable

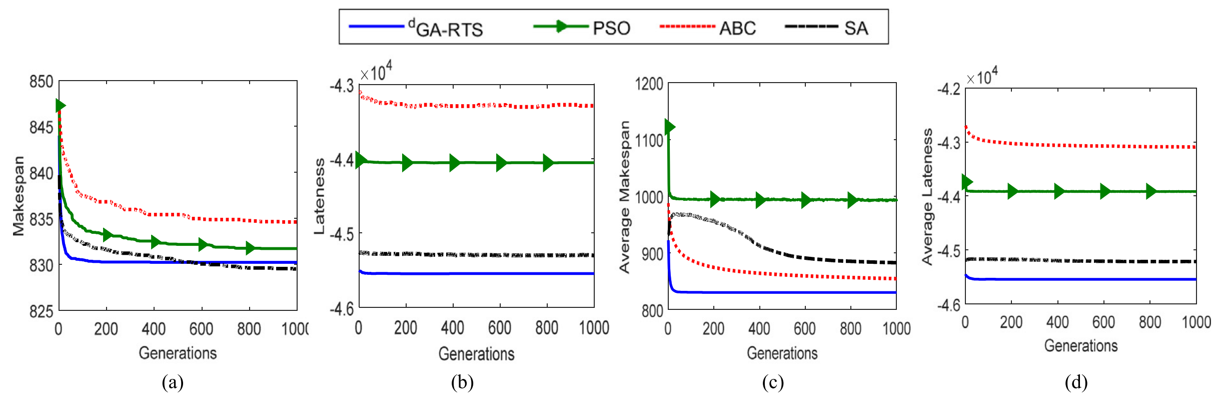


Fig. 22. Comparative results of the performance with ^{d}GA-RTS, PSO, ABC algorithm and SA: (a) Best Makespan, (b) Best Total Lateness, (c) Average Makespan and (d) Average Total Lateness for population size of 200 for 1000 Generations for STG1.

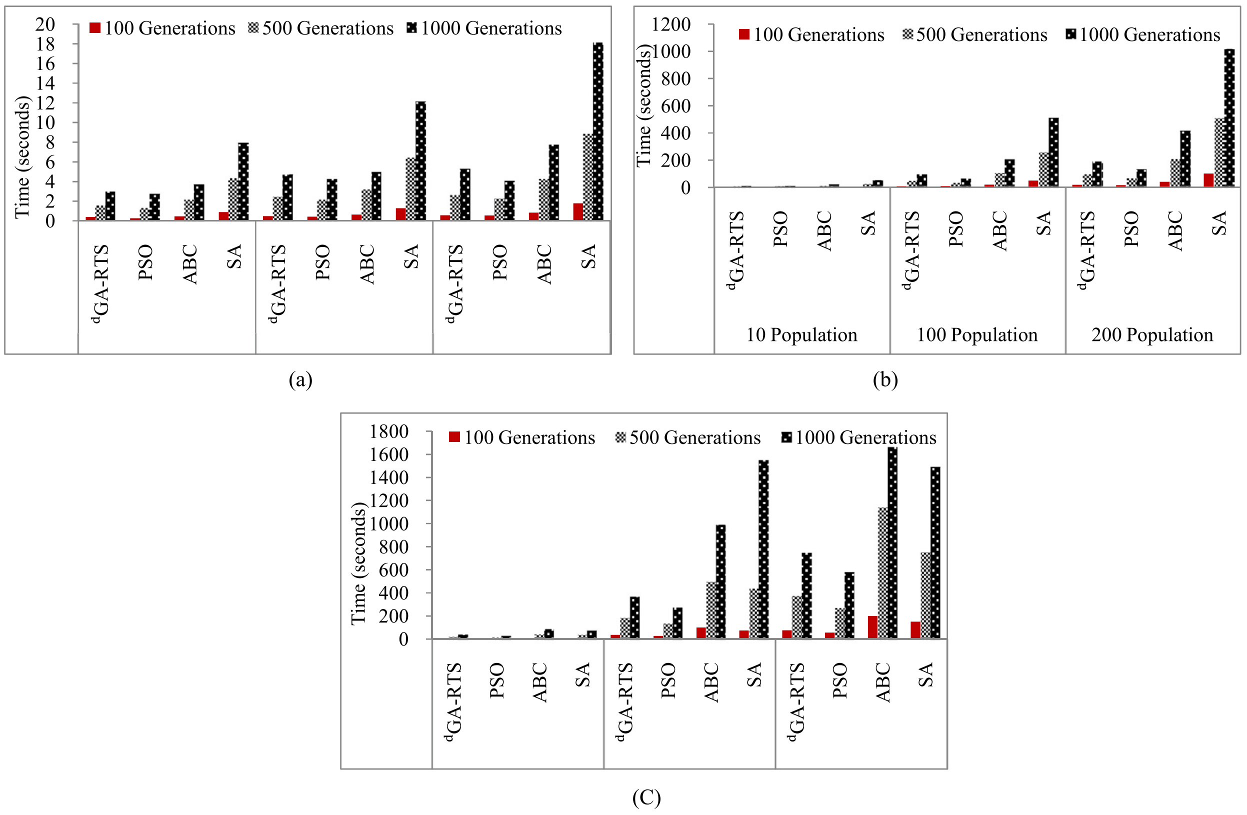


Fig. 23. Running time of ^{d}GA-RTS, PSO, ABC and SA for different population size with varying generations for (a) STG2, (b) STG3 and (c) STG4 task sets.

time in comparison to ^{d}GA-RTS. The results obtained using PSO is better than that of ABC and also, the execution time of ABC is longer than PSO. Hence, we can conclude that ^{d}GA-RTS has been able to generate solutions of better quality among all four algorithms with context to total execution time and best makespan values. Further, we also noted the total lateness value of best schedule and found that every algorithm is able to produce a feasible solution with negative lateness value.

4.6. Dynamic case: Scheduling of precedence based tasks with arrival times

We have performed experiments using STG2, STG3 and STG4 for dynamic case simulations. The communication costs and the arrival times are generated randomly using uniform distribution to fulfil the requirements of input parameters for the solving techniques. Tables 10–12 shows the comparative results for STG2, STG3 and STG4.

These tables consist of combined results for different instances of population versus number of generations for each of the four algorithms. We start analysing the results from Table 10 for the synthetic task set STG2. Here, SA has achieved better $\overline{F}_{BOG}$, but again, taking larger completion time than the other three techniques. The values obtained using ^{d}GA-RTS are very much nearer to that achieved using SA, where ^{d}GA-RTS takes quite lesser completion time in comparison to SA, ABC and PSO. Fig. 23(a)–(c) give the comparative plots for running time of all the four algorithms for different instances of STG2, STG3 and STG4, respectively. Similarly, ^{d}GA-RTS has produced schedules with minimum $\overline{F}_{BOG}$ for large task sets of STG3 and STG4 for 100 generations with different population sizes as mentioned in Tables 11 and 12, respectively.

Table 8
Comparative results of ^{d}GA-RTS, PSO, ABC algorithm and SA for STG1 with 90 tasks.

Population size	Generations	100						500						1000					
	Algo	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time
10	^{d}GA-RTS	**835.700**	−43 178.000	**841.558**	−43 177.980	−479.755	**1.113**	833.800	−**45 395.350**	**836.305**	−**45 395.610**	−**504.390**	**5.201**	832.633	−41 705.350	**836.028**	−41 705.170	−463.390	**10.404**
	PSO	839.100	−**45 464.010**	997.750	−**45 361.320**	−**504.015**	4.151	835.700	−42 423.420	998.189	−42 286.670	−471.370	8.589	834.667	−**44 126.970**	997.919	−**43 983.340**	−**490.300**	12.419
	ABC	855.133	−43 060.880	908.778	−42 971.440	−477.460	2.177	842.700	−43 162.560	910.056	−43 092.560	−479.600	10.999	841.233	−43 135.120	909.484	−43 107.400	−479.280	21.643
	SA	835.733	−43 762.160	938.334	−43 696.300	−485.514	4.801	**831.033**	−45 016.330	958.536	−44 994.520	−500.180	23.619	**830.100**	−42 610.860	962.377	−42 593.230	−473.450	47.440
100	^{d}GA-RTS	**831.033**	−44 436.607	**835.954**	−44 437.614	−493.751	**8.733**	**830.267**	−**44 570.303**	**831.483**	−**44 570.049**	−**495.226**	43.597	830.700	−43 424.367	**831.453**	−43 424.173	−482.493	87.181
	PSO	834.800	−**45 504.567**	993.518	−**45 366.778**	−**504.075**	12.109	833.033	−42 696.577	995.148	−42 554.197	−474.406	**40.352**	832.133	−**46 204.417**	994.264	−**46 066.403**	−**513.382**	**79.769**
	ABC	841.633	−43 144.652	876.354	−42 979.212	−477.547	20.269	836.300	−43 249.693	877.881	−43 051.013	−480.552	95.438	835.267	−43 287.413	877.896	−43 078.111	−480.971	192.113
	SA	833.567	−45 036.223	882.734	−44 951.473	−499.461	45.911	830.600	−43 462.823	923.761	−43 401.301	−482.920	229.432	**829.667**	−44 748.450	938.654	−44 687.446	−497.205	458.687
200	^{d}GA-RTS	**830.400**	−**46 180.930**	**834.161**	−**46 183.190**	−**513.147**	**17.488**	**830.167**	−**45 201.013**	**831.382**	−**45 201.427**	−**502.233**	87.379	830.200	−**45 545.417**	**830.794**	−**45 543.875**	−**506.060**	174.483
	PSO	834.300	−44 336.687	993.956	−44 208.714	−491.208	22.678	832.500	−44 783.467	994.209	−44 636.524	−497.594	**73.251**	831.700	−44 052.760	994.204	−43 916.752	−489.475	**138.282**
	ABC	839.200	−43 215.868	866.862	−42 992.984	−477.700	38.659	835.800	−43 244.133	867.763	−43 070.163	−480.490	192.913	834.600	−43 286.933	867.615	−43 093.975	−480.966	393.916
	SA	832.367	−44 253.230	859.174	−44 139.347	−490.437	92.264	830.633	−43 345.027	893.958	−43 259.518	−481.611	461.345	**829.500**	−45 301.217	914.038	−45 214.956	−503.347	923.041

Table 9
Comparative results for 30 runs of ^{d}GA-RTS, PSO, ABC and SA for different task sets.

Algo			^{d}GA-RTS			PSO			ABC			SA		
Task set	Nodes	Processor	Best makespan	Best total lateness	Execution time	Best makespan	Best total lateness	Execution time	Best makespan	Best total lateness	Execution time	Best makespan	Best total lateness	Execution time
proto062.stg	591	6	974.367	−11 304.465	410.400	1013.033	−11 112.027	683.860	1012.567	−10 283.708	1269.411	973.000	−11 360.182	3215.186
proto066.stg	771	6	1371.867	−127 338.126	873.373	1420.167	−126 311.673	1342.718	1421.000	−125 926.018	2571.795	1369.667	−129 471.412	6243.099
proto164.stg	844	7	1172.800	−12 678.464	929.493	1217.000	−13 154.077	1402.964	1217.633	−10 826.994	2699.750	1170.933	−13 331.001	6586.633
proto126.stg	1143	4	3033.567	−191 096.977	1890.501	3081.300	−190 124.872	2762.614	3083.300	−189 050.690	5372.268	3032.567	−191 756.774	13 193.285
proto128.stg	1362	5	2750.567	−8083.081	1859.289	2812.800	−7947.614	2737.050	2814.267	−12 351.989	5311.042	2750.000	−8287.551	12 999.307
proto189.stg	1540	4	3644.667	−102 288.660	3389.650	3680.967	−106 812.650	4736.232	3682.333	−100 942.946	9233.727	3643.400	−103 067.158	22 673.515
proto188.stg	1781	3	6009.233	−294 139.163	4783.864	6009.833	−291 832.872	6489.208	6033.433	−295 249.971	12 658.574	6008.733	−299 889.688	30 893.202
proto039.stg	1992	6	3161.267	−70 300.918	5828.627	3216.600	−70 030.378	7887.251	3107.567	−59 248.586	16 179.088	3161.800	−68 984.501	38 814.391
proto148.stg	2122	7	3195.467	−473 227.580	6956.829	3298.133	−468 300.820	9448.219	3297.933	−458 368.935	20 811.083	3191.750	−479 381.525	44 880.498
proto144.stg	2418	6	4306.767	−439 837.823	9385.026	4419.833	−446 867.998	12 369.549	4110.567	−425 575.154	23 934.613	4304.800	−441 431.371	24 215.477

Table 10
Comparative results of ^{d}GA-RTS, PSO, ABC algorithm and SA for STG2 with 15 tasks.

Population size	Generations	100						500						1000					
	Algo	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time
5	^{d}GA-RTS	34.533	−170.433	**35.749**	−170.280	−11.362	0.381	33.800	−179.100	**34.156**	−179.147	−11.940	1.545	34.000	−176.433	**34.118**	−176.513	−11.762	2.967
	PSO	34.600	−175.767	45.866	−123.820	−11.718	**0.264**	33.233	−181.233	45.826	−121.547	−12.082	**1.324**	33.133	−182.667	45.755	−116.293	−12.178	**2.711**
	ABC	34.433	−176.300	36.635	−**172.367**	−11.753	0.456	33.367	−182.500	36.566	−178.960	−12.167	2.151	33.267	−182.167	36.492	−177.853	−12.144	3.694
	SA	**32.200**	−**187.933**	39.392	−159.067	−**12.529**	0.891	**31.833**	−**191.133**	40.979	−**186.760**	−**12.742**	4.345	**31.700**	−**189.167**	41.449	−**184.473**	−**12.611**	7.900
10	^{d}GA-RTS	33.467	−185.300	**34.319**	−**185.127**	−12.353	0.484	33.600	−180.967	**33.824**	−180.960	−12.064	2.462	33.667	−179.933	**33.467**	−180.933	−11.996	4.671
	PSO	33.967	−178.467	45.560	−122.490	−11.898	**0.418**	33.000	−188.300	45.797	−121.797	−12.553	**2.153**	32.867	−193.867	45.761	−122.430	−12.924	**4.234**
	ABC	33.767	−183.500	35.019	−176.430	−12.233	0.629	33.033	−189.400	35.099	−180.517	−12.627	3.188	32.933	−189.233	34.853	−182.640	−12.616	5.007
	SA	**32.167**	−**193.333**	35.055	−151.823	−**12.889**	1.295	**31.733**	−**191.233**	36.323	−**182.937**	−**12.749**	6.417	**31.700**	−**194.967**	37.507	−**186.103**	−**12.998**	12.121
15	^{d}GA-RTS	33.500	−187.333	34.253	−**187.518**	−12.489	0.570	33.233	−189.067	**33.758**	−**189.398**	−12.604	2.626	33.200	−188.767	**33.298**	−188.796	−12.584	5.277
	PSO	33.400	−187.467	45.561	−124.516	−12.498	**0.547**	32.733	−**196.467**	45.590	−121.604	−**13.098**	**2.229**	32.533	−**199.667**	45.632	−121.727	−**13.311**	**4.069**
	ABC	33.433	−188.233	34.624	−178.564	−12.549	0.836	32.633	−194.567	34.597	−186.989	−12.971	4.221	32.500	−198.733	34.243	−**189.440**	−13.249	7.775
	SA	**32.000**	−**194.900**	**33.930**	−150.658	−**12.993**	1.787	**31.767**	−193.167	35.009	−176.658	−12.878	8.826	**31.633**	−194.600	35.923	−182.544	−12.973	18.099

Table 11
Comparative results of ^{d}GA-RTS, PSO, ABC algorithm and SA for STG3 with 98 tasks.

Population size	Generations	100						500						1000					
	Algo	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time
10	^{d}GA-RTS	649.833	−14 456.130	**652.826**	−14 456.143	−147.512	**1.345**	647.633	−14 523.107	**649.336**	−14 523.130	−148.195	**6.394**	647.867	−**17 487.347**	**649.329**	−**17 487.289**	−**178.442**	12.477
	PSO	650.900	−**15 634.567**	751.583	−**15 569.950**	−**159.536**	2.862	649.033	−**15 985.880**	751.228	−**15 919.851**	−**163.121**	7.610	647.800	−15 426.573	751.014	−15 352.577	−157.414	**11.540**
	ABC	667.333	−13 670.133	715.682	−13 453.603	−139.491	2.391	658.967	−13 729.767	714.838	−13 620.233	−140.100	11.313	655.000	−13 643.167	715.258	−13 637.923	−139.216	21.557
	SA	**648.700**	−13 990.297	710.160	−13 955.852	−142.758	5.573	**646.167**	−14 827.497	717.102	−14 813.284	−151.301	27.081	**646.867**	−16 407.657	718.147	−16 395.718	−167.425	53.207
100	^{d}GA-RTS	**646.400**	−14 394.777	**648.485**	−14 394.719	−146.885	**9.678**	646.133	−14 025.643	**646.827**	−14 027.003	−143.119	47.713	646.200	−13 801.523	**646.655**	−13 801.917	−140.832	95.358
	PSO	647.833	−13 829.297	747.177	−13 749.943	−141.115	10.621	646.733	−**15 904.203**	747.926	−**15 827.479**	−**162.288**	**33.411**	646.533	−12 947.367	747.373	−12 868.859	−132.116	**61.710**
	ABC	653.533	−13 790.233	692.693	−13 455.322	−140.717	20.715	650.833	−13 824.633	692.679	−13 614.521	−141.068	104.628	650.067	−13 803.000	692.544	−13 660.761	−140.847	206.375
	SA	647.000	−**15 134.153**	680.542	−**15 088.206**	−**154.430**	51.545	**646.000**	−14 265.923	702.237	−14 230.327	−145.571	257.379	**646.000**	−**14 719.393**	715.882	−**14 690.263**	−**150.198**	508.860
200	^{d}GA-RTS	**646.033**	−**16 205.937**	**648.379**	−**16 206.935**	−**165.367**	19.073	**646.000**	−**15 196.587**	**646.564**	−**15 194.651**	−**155.067**	95.205	**646.000**	−14 044.840	**646.323**	−14 043.986	−143.315	189.942
	PSO	647.333	−15 079.163	746.679	−15 003.187	−153.869	**16.905**	646.300	−13 640.653	746.488	−13 564.067	−139.190	**66.071**	646.100	−13 538.940	746.164	−13 461.887	−138.152	**132.824**
	ABC	651.100	−13 829.467	683.093	−13 445.672	−141.117	41.587	649.767	−13 778.800	683.518	−13 603.664	−140.600	208.032	649.033	−13 873.967	683.679	−13 655.112	−141.571	414.669
	SA	646.333	−13 382.133	666.654	−13 330.291	−136.552	102.823	**646.000**	−14 103.023	684.811	−14 046.118	−143.908	507.919	**646.000**	−**16 064.447**	699.507	−**16 023.559**	−**163.923**	1016.997

Table 12
Comparative results of ^{d}GA-RTS, PSO, ABC algorithm and SA for STG4 with 336 tasks.

Population size	Generations	100						500						1000					
	Algo	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time	Makespan	Total lateness	Average makespan	Average total lateness	Average lateness	Time
10	^{d}GA-RTS	**2381.033**	−**118 148.080**	**2394.089**	−**118 147.865**	−**351.631**	4.424	2379.433	−**117 195.403**	**2385.384**	−**117 195.361**	−**348.796**	21.199	2379.467	−115 962.250	**2385.094**	−115 962.411	−345.126	42.149
	PSO	2388.633	−115 934.757	2755.757	−115 878.797	−345.044	**4.260**	2382.533	−110 691.093	2760.499	−110 531.875	−329.438	**15.524**	2381.233	−**118 245.613**	2760.876	−**118 137.147**	−**351.921**	**29.079**
	ABC	2465.000	−105 567.150	2646.536	−97 364.615	−314.188	8.388	2433.300	−102 010.050	2649.968	−95 185.675	−303.601	41.880	2432.667	−97 135.650	2651.496	−95 130.495	−289.094	90.051
	SA	2383.800	−114 962.287	2654.184	−114 919.417	−342.150	7.760	**2377.500**	−116 055.430	2695.385	−116 016.513	−345.403	37.698	**2377.067**	−114 007.307	2700.239	−113 986.526	−339.307	74.798
100	^{d}GA-RTS	**2377.467**	−110 654.690	**2382.629**	−110 654.275	−329.329	37.162	2377.333	−106 423.210	**2378.976**	−106 419.865	−316.736	185.527	2377.400	−109 623.927	**2378.926**	−109 627.884	−326.262	370.756
	PSO	2382.600	−**117 670.190**	2746.535	−**117 560.131**	−**350.209**	**27.987**	2379.600	−**119 938.173**	2750.309	−**119 774.549**	−**356.959**	**135.670**	2378.933	−107 703.257	2750.321	−107 548.558	−320.545	**276.829**
	ABC	2410.167	−104 645.000	2579.027	−95 712.138	−311.443	100.387	2400.100	−99 677.100	2579.805	−94 617.589	−296.658	501.199	2392.900	−100 775.550	2579.719	−94 579.494	−299.927	991.672
	SA	2379.867	−111 131.310	2510.799	−111 011.035	−330.748	73.528	**2377.167**	−113 459.757	2611.536	−113 407.446	−337.678	437.781	**2377.000**	−**117 120.227**	2647.726	−**117 078.245**	−**348.572**	1553.924
200	^{d}GA-RTS	**2377.233**	−112 709.193	**2381.130**	−112 716.426	−335.444	74.911	**2377.267**	−111 375.483	**2378.211**	−111 373.617	−331.475	374.234	**2377.000**	−**119 044.317**	**2377.869**	−**119 013.842**	−**354.299**	748.542
	PSO	2380.533	−**119 524.773**	2746.869	−**119 453.405**	−**355.728**	**55.122**	2378.800	−105 540.070	2745.956	−105 383.414	−314.107	**270.598**	2378.433	−106 634.967	2747.965	−106 535.484	−317.366	**578.714**
	ABC	2399.300	−106 155.500	2548.605	−95 811.783	−315.939	199.392	2393.400	−101 426.950	2549.353	−94 347.622	−301.866	1142.470	2393.333	−98 413.550	2549.248	−94 436.342	−292.897	1665.921
	SA	2379.467	−118 686.003	2444.280	−118 635.071	−353.232	151.245	2377.300	−**113 590.913**	2512.896	−**113 456.444**	−**338.068**	754.255	**2377.000**	−117 112.513	2549.546	−117 066.056	−348.549	1493.157

Table 13
Comparative results of 30 runs of ^{d}GA-RTS, PSO, ABC and SA for different task sets.

Algo			^{d}GA-RTS			PSO			ABC			SA		
Task set	Nodes	Processor	Best makespan	Best total lateness	Execution time	Best makespan	Best total lateness	Execution time	Best makespan	Best total lateness	Execution time	Best makespan	Best total lateness	Execution time
proto001.stg	473	4	1179.867	−9519.591	545.323	1206.300	−10 085.010	731.454	1201.700	−9454.340	953.565	1179.300	−9712.825	2015.006
proto065.stg	681	6	1211.933	−156 104.165	969.738	1258.933	−15 5124.878	1585.057	1258.567	−153 631.411	2386.517	1208.133	−153 879.050	5157.598
proto114.stg	897	8	1164.000	−14 057.713	1292.093	1231.400	−16 673.880	1758.286	1234.300	−1281.695	2703.653	1155.833	−11 425.603	5844.684
proto070.stg	1152	6	1972.033	−55 663.384	2569.506	2027.200	−58 266.300	3615.510	2023.800	−53 220.926	5836.396	1970.733	−57 095.675	13 115.558
proto076.stg	1385	8	1830.000	−281 115.456	3840.181	1906.933	−283 255.140	5336.658	1903.467	−263 462.326	8651.110	1579.333	−247 258.038	17 043.682
proto035.stg	1545	4	3842.267	−124 061.361	4595.294	3881.667	−126 457.197	6354.780	3879.500	−122 546.195	10 485.916	3841.500	−127 058.944	23 639.212
proto037.stg	1719	4	4086.067	−174 895.768	5685.038	4119.800	−170 559.945	7868.886	4124.367	−165 940.735	12 037.903	3949.800	−164 101.097	36 570.709
proto091.stg	2059	8	2759.600	−23 541.312	7571.059	2855.133	−20 643.975	10 265.929	2845.967	−23 351.417	17 182.728	2202.267	−19 498.958	33 466.661
proto146.stg	2235	9	2611.633	−186 873.871	5521.363	2718.967	−18 055.562	7368.087	2728.300	−184 539.688	10 966.857	2600.033	−182 688.557	27 521.312
proto143.stg	2490	7	3734.367	−476 728.661	11 965.111	3847.200	−475 013.392	16 274.064	3846.800	−437 356.746	35 254.162	3480.433	−423 196.453	62 821.403

The evolution of ABC had been much slower taking longer computation time and low quality schedules in comparison to ^{d}GA-RTS and PSO. On the other hand, PSO has completed its execution within lesser computation time almost for every instance but with very low improvements in $\overline{F}_{BOG}$ compared to SA and the proposed ^{d}GA-RTS based approach. Again, it can be observed that for both task sets, the schedule lengths obtained from ^{d}GA-RTS are very close to those of SA.

We have also observed the average makespan for each generation for 30 independent runs of each algorithm. Though, there is considerable improvement in the makespan values, the overall population evolution is slow as number of generations increases. As discussed earlier, the optimum value changes periodically with a change in the environment. After the arrival of a new task, the possible schedule length may change, which affects the convergence rate. This also changes the overall diversity and average fitness at every generation. Thus, we record the average of makespan and average total lateness of population at every generation for analysis. The results obtained show that our proposed approach has produced population pools with individuals nearer to the best possible solution in the search space.

It can be witnessed from Tables 8 and 10–12 that the average makespan for all the task sets is better using ^{d}GA-RTS compared to SA, PSO and ABC. In order to check the applicability of our approach to RTS, we have computed the total lateness of best schedules and the average total lateness for every generation. It can be seen that all four algorithms have met the deadlines with lateness being negative for all the instances. Also, we have checked the average lateness of the best schedule to ensure that all tasks have completed their executions within viable amount of time before their respective deadlines. Therefore, we can conclude that ^{d}GA-RTS based approach has been able to produce feasible schedules in most of the cases within reasonable time for given instances and fulfils the real-time constraints.

4.6.1. *Extensive experiments for dynamic case*

Similar to the static case, we select ten task sets randomly (different from the static case) from [46] to demonstrate the feasibility of the proposed approach for larger task sets with dynamic arrival times. Therefore, a fixed set of communication costs for edge weight between dependent tasks and arrival time for each task is generated using uniform distribution. The simulation is done using population size of 100 for 1000 generations. For every task set, 30 independent runs of each algorithm is performed. The average measurement values obtained after 30 runs of each algorithm are presented in Table 13 for comparative analysis.

Here also we can observe that ^{d}GA-RTS and SA are able to produce better solutions when compared to PSO and ABC for each set of tasks. The results obtained using ^{d}GA-RTS and SA are almost similar with very minor difference. On the other hand, ^{d}GA-RTS takes very less time compared to SA to complete its execution for every task set. In the dynamic case also, PSO and ABC have not been able to generate better solutions compared to ^{d}GA-RTS. However, results achieved using PSO and ABC are much similar for every instance but ABC take much longer time compared to PSO. The solutions generated after simulations demonstrate the feasibility of our approach as lateness value in all cases is negative for every task. This proves our approach to be a suitable technique to solve the addressed problem as it produces better solutions within lesser time.

4.7. *Comparison based on performance measures*

We have compared the results using four different performance measures and conducted a statistical analysis. The improvement

Table 14
Performance measure values for STG2.

Measures	^{d}GA-RTS	PSO	ABC	SA
Offline	**34.1852**	45.7765	34.6573	37.0357
Accuracy	0.98636	0.98603	**0.98641**	0.98498
S.D.	**0.25434**	0.46368	0.43679	0.28084

Table 15
Performance measure values for STG3.

Measures	^{d}GA-RTS	PSO	ABC	SA
Offline	**646.318**	746.315	682.302	707.602
Accuracy	0.91333	0.91141	**0.93915**	0.91754
S.D.	**0.36872**	0.70268	1.28928	0.38779

Table 16
Performance measure values for STG4.

Measures	^{d}GA-RTS	PSO	ABC	SA
Offline	**2377.84**	2751.03	2549.28	2563.49
Accuracy	0.98936	0.98563	**0.99253**	0.99132
S.D.	**0.9353**	1.84267	3.08691	1.15424

in the result incurs with the varying generations. We have also observed that better results are found for large population size and large number of generations. Since offline and accuracy are used especially for measuring EA's performance in dynamic environment [52,55], we have conducted experiments on results obtained with STG2, STG3 and STG4. We have determined offline, accuracy and standard deviation values (to measure variability in results) of average best makespan for 30 runs for each generation. We have measured the values over the instances: 200 population size and 1000 generations for STG3 and STG4, and 15 population size and 1000 generations for STG2.

Tables 14–16 give the different measurement values for STG2, STG3 and STG4 task sets as mentioned above. The offline value of ^{d}GA-RTS is minimum for all the cases, i.e. our approach has generated better schedules compared to other three techniques. Similarly, the accuracy value for all the studied algorithms achieved good performance with more than 90% of accuracy in all the cases. It can be observed that ABC has obtained better accuracy values in all cases but not with much difference from the accuracy of other three algorithms. For STG3 and STG4, instances ^{d}GA-RTS and SA have performed better than PSO with accuracy values of SA being higher but closer to that of ^{d}GA-RTS. From all the tables, we can see that again, ^{d}GA-RTS and SA have been able to generate schedules near to the best solutions with lesser variability in the results than PSO and ABC.

For calculating $ABC_{A}^{m_1,m_2}$ measure, we consider results obtained from the instance with population size of 200 and generations 1000 for STG1, STG3 and STG4. For STG2, we have studied the results obtained using population size of 15 for 1000 generations. $ABC_{A}^{m_1,m_2}$ is applied to every pair of algorithms rendering to $\overline{F}_{BOG}$ (best makespan of each generation averaged over 30 runs) with property $m = \overline{F}_{BOG}$ using Eq. (9). Table 17 presents the results obtained for each pair of algorithms from which we can observe the following:

- Beside $ABC_{A}^{m_1,m_2}$ value for STG2 being positive, ^{d}GA-RTS obtains negative values for SGT1, SGT3 and SGT4 with all other algorithms, i.e. the curve of ^{d}GA-RTS lies under the curves of all other algorithms. $ABC_{A}^{m_1,m_2}$ values attained (using SGT1, SGT3 and SGT4) by ^{d}GA-RTS paired to ABC is least compared to the negative values obtained pairing with PSO and SA. That means the curve of ABC lies above to the curves of SA, while PSO and ^{d}GA-RTS are distant.

Table 17
Comparison of $ABC_A^{m_1,m_2}$ values achieved by each pair of algorithms.

Area between curve ($ABC_A^{m_1,m_2}$)				
Algorithms	Task set (size)	PSO	ABC	SA
dGA-RTS	STG1 (90)	**−2.2970**	**−5.4996**	**−0.3539**
	STG2(15)	0.3713	0.4237	1.5099
	STG3(98)	**−0.5609**	**−3.9906**	**−0.1061**
	STG4(336)	**−2.2451**	**−20.3443**	**−0.6220**
SA	STG1(90)	**−1.9431**	**−5.1456**	
	STG2(15)	**−1.1386**	**−1.0861**	
	STG3(98)	**−0.4548**	**−3.8846**	
	STG4(336)	**−1.6231**	**−19.7222**	
ABC	STG1(90)	3.2025		
	STG2(15)	**−0.0524**		
	STG3(98)	3.4298		
	STG4(336)	18.0991		

- Analysing the rows of SA, we find that its curve lies significantly lower to the curves of PSO and ABC for all the task sets with curve of SA being much lower for STG4 paired with ABC.
- ABC achieves negative value only for STG2 when paired with PSO. It obtains positive values for other task sets which brings the conclusion that the curve of ABC lies above the curve of the PSO being considerably above from STG4.

Thus, we can conclude that though all algorithms have been able to generate solutions, dGA-RTS performed better for larger task sets whereas ABC has not been able to provide better makespan. The curve of ABC lies above all other algorithms and under the curve of PSO. Similarly, the curve of SA lies below the curve of PSO and above the dGA-RTS for most of the cases. It shows that our approach is suitable to generate solutions for the minimization problem addressed here.

4.8. Statistical significance test

In order to assess the significant difference in the performance of the algorithms, we have performed ANOVA test with a confidence level of 95% using the best makespan achieved at the last generation of each of 30 runs. Here also, we have considered the same set of instances and their results as taken for the $ABC_A^{m_1,m_2}$ metric. Fig. 24(a)–(d) presents the comparative ANOVA test results for STG1, STG2, STG3 and STG4 instances respectively, where dGA-RTS, PSO, ABC and SA is denoted by 1, 2, 3 and 4 respectively on Y-axis.

After the analytical study of the algorithms, we can notice from Fig. 24(a) that dGA-RTS has significantly better mean than PSO and ABC but statistically equivalent to SA for STG1. Here, ABC has significantly worst mean than other three algorithms. From Fig. 24(b), it can be observed that mean of PSO are significantly similar to ABC, but dGA-RTS and SA have significantly different mean for STG2. Similarly, for STG3, means of dGA-RTS, PSO and SA are statistically equivalent, as shown in Fig. 24(c). The mean of ABC is significantly different from all three other algorithms. The ANOVA test result depicted in Fig. 24(d) for STG4 instance infers that dGA-RTS, PSO and SA have means significantly better than ABC. dGA-RTS and SA present comparatively lower values but at par with PSO. Thus, we can conclude that results of SA and dGA-RTS are significantly similar with SA bearing lower values than others.

5. A practical implementation technique to deal with synchronization issue for the addressed problem and exception handling due to deadline violations

In this section, we first introduce an implementation technique to handle the synchronization problems and then provide a discussion on how to handle the exceptions that may occur due to deadline violations.

5.1. An implementation technique to deal with synchronization problem

There may arise situations when the underlying algorithm may not reach sufficient iterations to find optimal schedules before the arrival of new set of tasks in the system. Thus, finding the best (with minimum makespan) possible feasible solution for the addressed scheduling problem becomes difficult. However, practically even if, at an instance, we may not reach up to the optimal solution, we may still get some feasible solutions within very less iterations (before any environmental change occurs). So, if the arrival of tasks is frequent, the best way to tackle would be to track the feasible solutions at every iteration, and record if any such solution is available.

Let us assume a multi-core programmed [56–59] central scheduler (multi-core processor) which governs the process of finding optimized feasible schedule at one core and the implementation at other core simultaneously [60–62]. This makes the scheduler efficient to carry out all its functions parallelly [60,62]. Fig. 25 represents a method to implement and tackle the problem of synchronization with changes in the environment at different instance of time t. Let $T_{Ii} = \{T_1, T_2 \ldots, T_N\}$ be the set of tasks arrived at instance Ii. We discuss the detailed procedure and major functions to be carried by the central scheduler at different instances indexed from A to G as marked in Fig. 25.

A. Once the system gets ready for specified computing, let us suppose at CLK_0, a set of tasks T_{I1} arrives into the system. The central scheduler gets triggered and the incorporated scheduling algorithm starts its process of optimization at the specific core of the processor. Along with optimizing the schedule, the algorithm also works to track feasible solution(s) at every iteration, i.e. *lateness* $(T_j) \le 0$ (where, $j = 1, 2, \ldots,$ no. of tasks in T_{I1} set) for every task of T_{I1}.
B. After a time period t_1 at G_{i+k} (where $i \in \mathbf{Z}$, $i \ge 1$; $k \in \mathbf{N}$, $k = 0, 1, \ldots, n$ and $k = 0$ at this instant) iteration, a feasible solution(s) for T_{I1} is found. The feasible solution with best makespan found so far is recorded. The process of optimization continues and the feasible solution is updated at every iteration until any new set of tasks arrive or the termination criteria is satisfied. The black broken downward arrow represents the instance at which a feasible solution is detected.
C. After t_2 time period, at CLK_t the time instance of G_{i+n} (where, $i + n < 1000$) iteration, a new set of task T_{I2} is arrived. Here, we need to stop the ongoing optimization process and update the ITQ with the feasible schedule found at G_{i+n-1}.
D. Now, at the same instance of time, the dedicated core of the central scheduler will execute to dispatch the scheduled tasks of T_{I1} (whose all required inputs and resources are available) from ITQ to available processors. On the other hand, the remaining undispatched tasks of T_{I1} at ITQ along with newly arrived tasks i.e. $\overline{T_{I1}} + T_{I2}$ will be sent to the core at which scheduling algorithm will run to optimize and find the solution. The grey vertical block shows a progressive bar which represents the parallel execution of dispatch module and scheduling algorithm at two different dedicated cores respectively.
E. The grey downward broken arrow illustrates the instance at which feasible solution(s) for $\overline{T_{I1}} + T_{I2}$ set is found and updated consecutively in the next generation.
F. If suppose, the action C does not happen i.e. no new set of tasks are arrived, the optimization process for T_{I1} would not be interrupted. So, the broken horizontal double headed arrow shows the execution of the scheduling algorithm till termination point i.e. $t_2 + t_3$ is the time taken for 1000 generations. The * represents the termination of algorithm

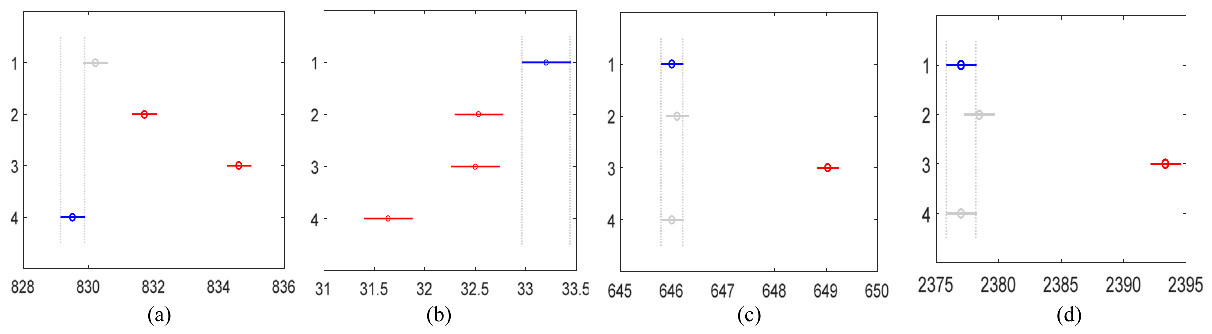


Fig. 24. Comparative results of ANOVA test for (a) STG1 with population size of 200, (b) STG2 with population size of 15, (c) STG3 with population size of 200 and (d) STG3 with population size of 200 over 1000 generations.

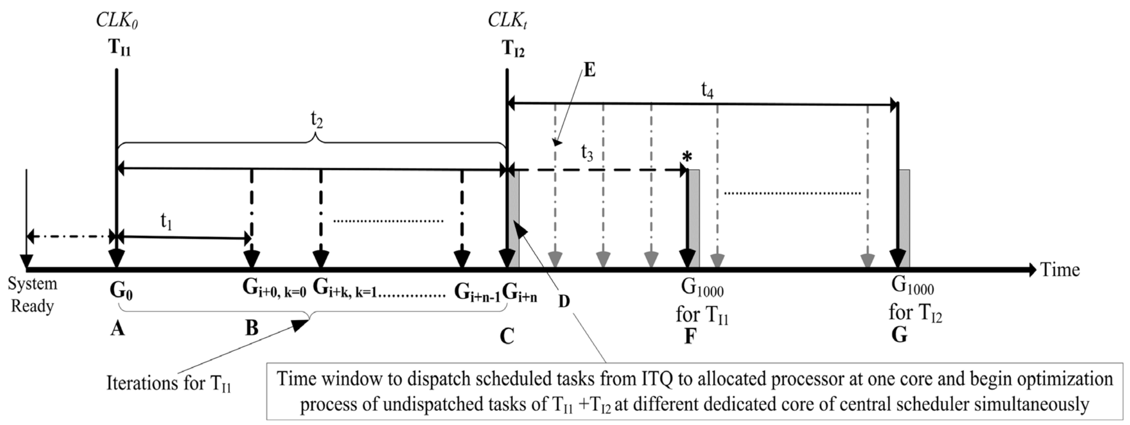

CLK0
T11
CLKt
T12
E
t4
t2
t3
*
t1
Time
System Ready
G0
A
Gi+0, k=0
Gi+k, k=1
B
Gi+n-1
Gi+n
C
D
G1000 for T11
F
G1000 for T12
G
Iterations for T11
Time window to dispatch scheduled tasks from ITQ to allocated processor at one core and begin optimization process of undispatched tasks of T11 +T12 at different dedicated core of central scheduler simultaneously

Fig. 25. A timeline depicting a dispatch rules to handle synchronization problem at different instance of time.

and finding best possible feasible solution. Also, the ITQ is updated. The grey progressive block shows dispatching of scheduled task on allocated available processors.

G. Similarly, after t_4 time period of optimization process for $\overline{T_{I1} + T_{I2}}$, the algorithm reaches to the termination point. Again, the ITQ is updated and tasks are allocated to the respective processors.

Therefore, to tackle the problem of synchronization for the addressed problem in real-time distributed multi-processor systems would be to keep track of feasible solutions and implement it as soon as a change is detected. This reduces the further delay and helps to meet real-time constraints. The advancement in parallel computing provides us much efficient resources to execute different modules simultaneously at different cores bearing shared memory. Further in Section 5.2 below, we have also conducted an experiment using STG1 (static case) for fixed running time of 30 s to observe how quickly the algorithms are able to generate a feasible solution.

5.2. An experiment to study efficiency of algorithms in terms of solution generation

In order to observe the behaviour of the algorithms in terms of results generated at different instances, we have simulated each algorithm for time duration of 30 s using STG1. We have also noted the results obtained at the end of 5, 10, 15, 20, 25 and 30 s during execution of each algorithm. Table 18 provides the experimental results of all the algorithms with values of best makespan, average makespan, best lateness and number of generations reached at the different instance of time.

We can observe from Table 18 that, with the increase in the generations, the results are optimized faster in ^{d}GA-RTS compared to other three algorithms. Proposed ^{d}GA-RTS completes 28, 56, 82, 110, 137 and 164 iterations at 5, 10, 15, 20, 25 and 30 s respectively. It is able to produce makespan value of 831 with lateness of tasks being negative at 5th second during execution. Similarly, it gets converged just before 56th generation with makespan values 829. Figs. 26–29 illustrates the best makespan, average makespan and best lateness values obtained with varying generations during 30 s execution of ^{d}GA-RTS, PSO, ABC and SA respectively.

In summary, we can say that in terms of running times, ^{d}GA-RTS, PSO, ABC and SA is the order in which these algorithms have performed. That is, ^{d}GA-RTS is the best performing algorithm among all the four. Similarly, in terms of best makespan generated within fixed time, performance-wise ordering of the algorithms is ^{d}GA-RTS, SA, PSO and ABC. Thus, we can conclude that the proposed approach performs better as it produces the feasible schedule faster than all others do.

5.3. Exception handling

Every task is a set of instructions which is coded to accomplish a specific goal. If a task is unprocessed within the given time window, it may lead to an unexpected error, also termed as exceptions. In real-time systems, deadline satisfaction is the main priority for the safety of the systems. It is unrealistic to say that a system design is completely invulnerable to such exceptions and unanticipated program conditions. Therefore, it is important to develop exception handlers which may help to detect, avoid and recover from any such unexpected conditions. The functionality of a task is known to the operating system of the domain, so most of the exceptions have default or programmed handling codes to recover from the error caused [14,63–66].

In dynamic scheduling problem, the deadline violation can be avoided for the task in a way that if a task's deadline is not guaranteed as per schedule, it will not be started. Hence, an exception

Table 18
Comparative results obtained at different instance of time using STG1.

Time at	5 seconds				10 seconds				15 seconds			
Algo	Makespan	Average makespan	Lateness	Iterations	Makespan	Average makespan	Lateness	Iterations	Makespan	Average makespan	Lateness	Iterations
^{d}GA-RTS	831	832.935	−52 866.6	28	829	829	−52 864.7	56	829	829	−52 864.7	82
PSO	839	1009.69	−41 739.4	25	834	995.075	−41 665.4	49	833	987.99	−41 764.4	74
ABC	845	938.685	−44 113.4	13	842	920.815	−44 079.8	25	842	909.31	−44 079.8	39
SA	836	963.355	−45 329	6	836	963.315	−45 329	11	836	956.44	−45 329	16
Time at	20 seconds				25 seconds				30 seconds			
Algo	Makespan	Average makespan	Lateness	Iterations	Makespan	Average makespan	Lateness	Iterations	Makespan	Average makespan	Lateness	Iterations
^{d}GA-RTS	829	829	−52 864.7	110	829	829	−52 864.7	137	829	829	−52 864.7	164
PSO	833	994.695	−41 764.4	98	833	993.53	−41 764.4	123	833	983.985	−41 764.4	148
ABC	842	902.995	−44 079.8	51	842	896.97	−44 079.8	63	842	892.535	−44 079.8	77
SA	836	957.4	−45 298.2	22	834	940.695	−45 366.2	27	834	926.71	−45 366.2	32

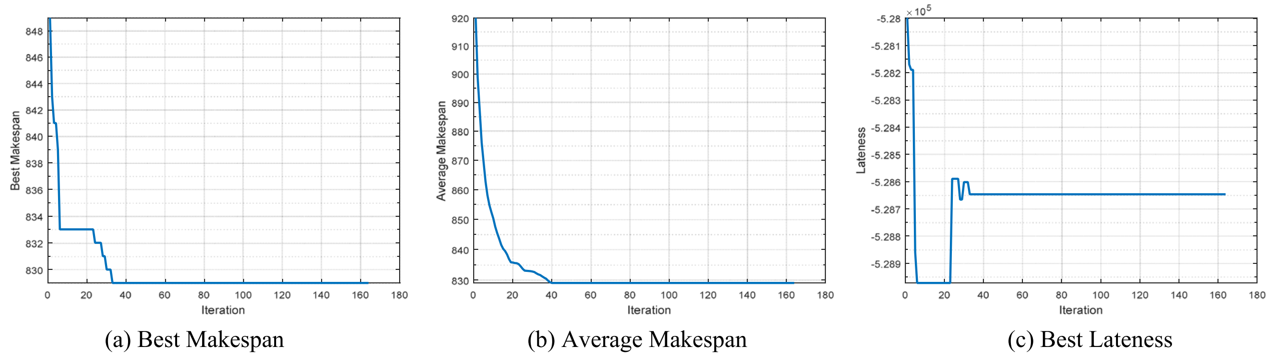


Fig. 26. Results obtained during 30 s run of ^{d}GA-RTS using STG1.

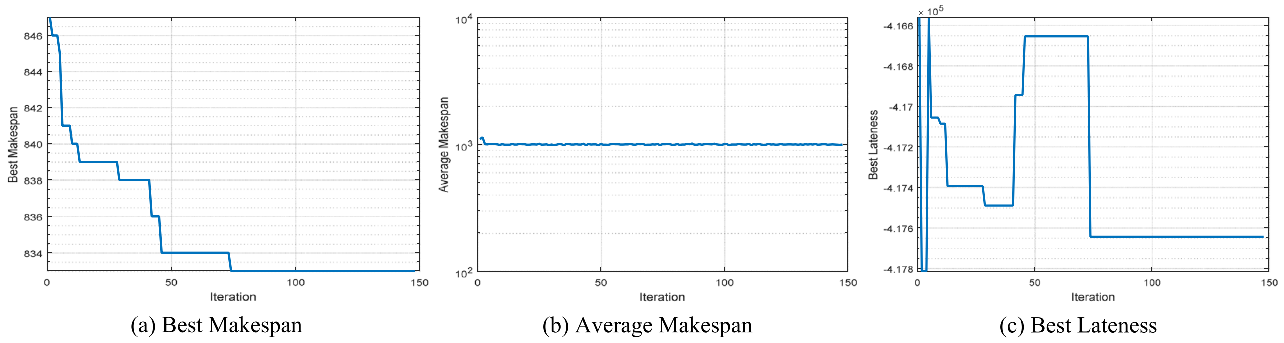


Fig. 27. Results obtained during 30 s run of PSO using STG1.

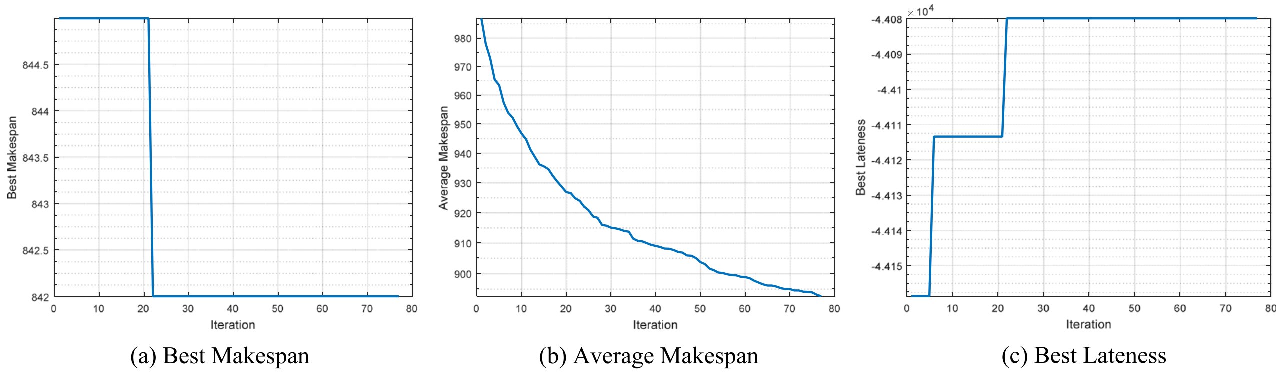


Fig. 28. Results obtained during 30 s run of ABC using STG1.

handler has to be executed although no time is left to prevent the damage caused by the error. One of the very early and most widely adapted schemes to handle such conditions is Task-Pair Scheduling model [65]. In this model, for every task, a pair of block similar to

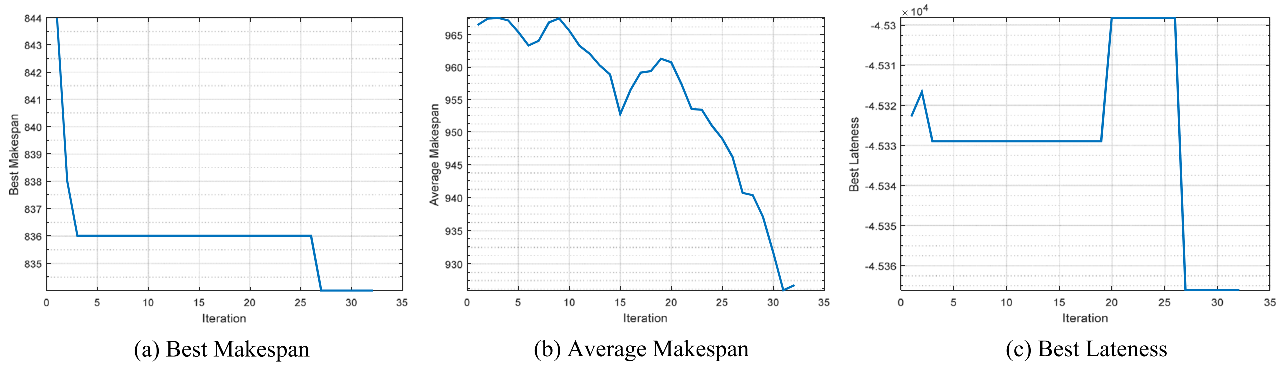


Fig. 29. Results obtained during 30 s run of SA using STG1.

try and catch is constructed viz., MainTask and ExceptTask. The MainTask consists of the body of actual task whereas ExceptTask bears the code for handling exceptions if deadline of a task is not guaranteed. The condition is to execute the MainTask only if the scheduler guaranteed the deadline; otherwise, ExceptTask will execute in maximum reasonable time before the given deadline of the task to handle the exceptions. The inconsistencies for execution of ExceptTask can be avoided by explicitly guaranteeing the timely execution of ExceptTask code as a separate task. Also, an object-oriented model is developed [66] to deal with real-time constraints and exception handling technique for easy and compatible design. There are similar designs adapted by different operating systems to handle exception handlings due to deadline violation [63]. These approaches can be easily deployed to manage unfeasible solutions for the smooth functioning of any system.

6. Conclusion and future research directions

This section summarizes the study done in this paper along with giving some insights to future research scopes in the domain of task scheduling.

6.1. Summary of the paper

This paper described precedence based tasks scheduling as a dynamic constraint problem and introduced a GA based dynamic task allocation and scheduling approach for real-time multiprocessor systems. Thus it has purposed a new scheduling scheme called 'Dynamic Genetic Algorithm for Real-Time Scheduling' (^{d}GA-RTS). The main feature of ^{d}GA-RTS is that it can handle dynamic as well as static scheduling of inter-dependent tasks for multi-processor RTS. The objective of dynamic task scheduling on parallel processors is to allocate a processor to a given task considering the task precedence and deadlines for execution within minimum possible period. Using benchmark as well as synthetic test cases we have demonstrated that the proposed ^{d}GA-RTS capably schedules the tasks in the waiting list when any new task is arrived in the scheduler. It minimizes the overall execution length of the task set ensuring overall deadline compliance. We have thoroughly compared the results obtained from the proposed ^{d}GA-RTS with three other popular evolutionary algorithms viz., simulated annealing, particle swarm optimization and artificial bee colony algorithm using four different performance measures. We presented the time complexity analysis of all these considered algorithms. Further, we also performed simulations using larger task sets to demonstrate the scalability of the proposed approach for static as well as dynamic case. Taking a composite look at the results, we can conclude that the proposed ^{d}GA-RTS produces better scheduling results than other three algorithms. We also presented an implementation approach to ensure deadline compliance along with a brief insight on how to handle the exception caused due to any possible deadline violations.

6.2. Future research directions

Task scheduling is a research area which is forever young because of ever expanding application domains of evolving systems. Everyday newer systems are developed requiring cutting edge planning and scheduling approaches for efficient and meaningful operations. Therefore, more and more research is needed on this important topic. In addition, the approach proposed in this paper is found to have not fast enough in generating the task schedules from the context of real-time systems. Thus, developing more efficient intelligent dispatching rules remains to be an interesting future research problem. Moreover, a practical implementation of central scheduler along with enhancement of its elements and overheads can be considered to make it more realistic. Further research may also focus on accommodating other features e.g. fault tolerance, power consumption, etc. in the form of additional objectives for better performance analysis of real-time systems or of more recent cyber–physical systems.

Acknowledgments

Authors gratefully acknowledge the valuable suggestions received from Prof. K Deb, Michigan State University, USA which has helped them in bringing significant novelty in the work specially the implementation scheme. The research is partially funded by the South Asian University, India in the form of start-up research fund. Second author is grateful to the DeITY, India for the financial support in the form of Visvesvaraya Ph.D. Fellowship. Third author is grateful to the CSIR, India for the financial support in the form of Senior Research Fellowship.

References

[1] J.W.S. Liu, Real-Time Systems, Pearson Education Asia, 2001, pp. 115–182.
[2] C.M. Krishna, Fault-tolerant scheduling in homogeneous REAL-TIME SYSTEMS, ACM Comput. Surv. 46 (4) (2014) 48.
[3] Kwok Yu-Kwong, Ishfaq Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, J. Parallel Dist. Comput. 59 (3) (1999) 381–422.
[4] Cheng Hui, Shengxiang Yang, Jiannong Cao, Dynamic genetic algorithms for the dynamic load balanced clustering problem in mobile ad hoc networks, Expert Syst. Appl. 40 (4) (2013) 1381–1392.
[5] Il Yong Kim, O.L. De Weck, Variable chromosome length genetic algorithm for progressive refinement in topology optimization, Struct. Multidiscip. Optim. 29 (6) (2005) 445–456.
[6] David E. Goldberg, Genetic Algorithms, Pearson Education India, 2006.
[7] Oduguwa Victor, Ashutosh Tiwari, Rajkumar Roy, Evolutionary computing in manufacturing industry: an overview of recent applications, Appl. Soft Comput. 5 (3) (2005) 281–299.
[8] Sastry Kumara, David E. Goldberg, Graham Kendall, Goldberg Graham Kendall Genetic algorithms. Search methodologies, Springer, Boston, MA, 2014, pp. 93–117.
[9] Yannick Monnier, J.P. Beauvais, A.M. Déplanche, A GA for scheduling tasks in a real-time distributed system, in: Euromicro Conf., 1998. Proc.s. 24th, Vol. 2, IEEE, 1998, pp. 708–714.

[10] Ri Choe, Hui Yuan, Youngjee Yang, Kwang Ryel Ryu, Real-time scheduling of twin stacking cranes in an automated container terminal using a genetic algorithm, in: Pro. of the 27th Annual ACM Symposium on Applied Computing, ACM, 2012, pp. 238–243.
[11] Oh Jaewon, Chisu Wu, GA-based real-time task scheduling with multiple goals, J. Tech. Phys. 71 (3) (2004).
[12] Fatma A. Omara, Mona M. Arafa, Genetic algorithms for task scheduling problem, J. Parallel Distrib. Comput. 70 (1) (2010) 13–22.
[13] Zhang Feifei, et al., A load-aware resource allocation and task scheduling for the emerging cloudlet system, Future Gener. Comput. Syst. (2018).
[14] Abhaya Kumar Samal, Rajib Mall, Chittaranjan Tripathy, Fault tolerant scheduling of hard real-time tasks on multiprocessor system using a hybrid genetic algorithm, Swarm Evol. Comput. 14 (2014) 92–105.
[15] Mehdi Akbari, Hassan Rashidi, Sasan H. Alizadeh, An enhanced genetic algorithm with new operators for task scheduling in heterogeneous computing systems, Eng. Appl. Artif. Intell. 61 (2017) 35–46.
[16] Albert Y. Zomaya, Yee-Hwei Teh, Observations on using GA for dynamic load-balancing, Parallel and Dist. Sys. IEEE Trans. on 12 (9) (2001) 899–911.
[17] Cheng Shu-Chen, Der-Fang Shiau, Yueh-Min Huang, Yen-Ting Lin, Dynamic hard-real-time scheduling using genetic algorithm for multiprocessor task with resource and timing constraints, Expert Syst. Appl. 36 (1) (2009) 852–860.
[18] Ma Juntao, et al., A novel dynamic task scheduling algorithm based on improved genetic algorithm in cloud computing. Wireless Communications, in: Networking and Applications, Springer, New Delhi, 2016, pp. 829–835.
[19] Andrew J. Page, Thomas J. Naughton, Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing, in: Parallel and Distributed Processing Symposium, 2005. Proc. 19th IEEE International, IEEE, 2005, 189a-189a.
[20] Andrew J. Page, Thomas M. Keane, Thomas J. Naughton, Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system, J. Parallel Distrib. Comput. 70 (7) (2010) 758–766.
[21] Nayak Sasmita Kumari, Padhy Sasmita Kumari, Siba Prasada Panigrahi, A novel algorithm for dynamic task scheduling, Future Gener. Comput. Syst. 28 (5) (2012) 709–717.
[22] Rahul Nath, Amit K. Shukla, Pranab K. Muhuri, Q. M. Danish Lohani, NSGA-II based energy efficient scheduling in real-time embedded systems for tasks with deadlines and execution times as type-2 fuzzy numbers, in: Fuzzy Systems (FUZZ), 2013 IEEE International Conference on, IEEE, 2013, pp. 1–8.
[23] Rahul Nath, Amit K. Shukla, Pranab K. Muhuri, Real-time power aware scheduling for tasks with type-2 fuzzy timing constraints, in: Fuzzy Systems (FUZZ-IEEE), 2014 IEEE International Conference on, IEEE, 2014, pp. 842–849.
[24] Amit K. Shukla, R. Nath, Pranab K. Muhuri, Muhuri Energy efficient task scheduling with Type-2 fuzzy uncertainty, in: Fuzzy Systems (FUZZ-IEEE), IEEE Int.Conf. on, IEEE, 2015.
[25] Wen Yun, Hua Xu, Jiadong Yang, A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system, Inform. Sci. 181 (3) (2011) 567–581.
[26] Zheng Wei, et al., Cost optimization for deadline-aware scheduling of big-data processing jobs on clouds, Future Gener. Comput. Syst. (2017).
[27] Li Fachao, Li Da Xu, Chenxia Jin, Hong Wang, Structure of multi-stage composite genetic algorithm (MSC-GA) and its performance, Expert Syst. Appl. 38 (7) (2011) 8929–8937.
[28] Li Fachao, et al., Intelligent bionic genetic algorithm (IB-GA) and its convergence, Expert Syst. Appl. 38 (7) (2011) 8804–8811.
[29] Li Fachao, et al., Random assignment method based on genetic algorithms and its application in resource allocation, Expert Syst. App. 39 (15) (2012) 12213–12219.
[30] Mohammad I. Daoud, Nawwaf Kharma, A hybrid heuristic–GA for task scheduling in heterogeneous processor networks, J. Parallel Distrib. Comput. 71 (11) (2011) 1518–1531.
[31] S. Suresh, Hao Huang, H.J. Kim, Hybrid real-coded genetic algorithm for data partitioning in multi-round load distribution and scheduling in heterogeneous systems, Appl. Soft Comput. 24 (2014) 500–510.
[32] Sajib K. Biswas, Amit Rauniyar, Pranab K. Muhuri, Multi-objective bayesian optimization algorithm for real-time task scheduling on heterogeneous multiprocessors, in: Evolutionary Computation (CEC), 2016 IEEE Congress on, IEEE, 2016, pp. 2844–2851.
[33] Sohail S. Chaudhry, Michael W. Varano, Lida Xu, Systems research, genetic algorithms and information systems. Systems Research and Behavioral Science: The Official Journal of the Int, Fed. Syst. Res. 17 (2) (2000) 149–162.
[34] Amit Rauniyar, Pranab K. Muhuri, Multi-robot coalition formation problem: Task allocation with adaptive immigrants based genetic algorithms, in: Systems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on, IEEE, 2016.
[35] Zang Wenke, et al., A cloud model based DNA genetic algorithm for numerical optimization problems, Future Gener. Comput. Syst. (2017).
[36] Mateja Ðumić, et al., Evolving priority rules for resource constrained project scheduling problem with genetic programming, Future Gener. Comput. Syst. (2018).
[37] Pranab K. Muhuri, Amit Rauniyar, Immigrants based adaptive genetic algorithms for task allocation in multi-robot systems, Int. J. Comput. Intell. Appl. 16 (04) (2017) 1750025.
[38] Jiang Yanxia, et al., Influencing factors for predicting financial performance based on genetic algorithms, Syst. Res. Behav. Sci 26 (6) (2009) 661–673.
[39] Rangsaritratsamee Ruedee, William G. FerrellJr, Mary Beth Kurz, Dynamic rescheduling that simultaneously considers efficiency and stability, Comput. Ind. Eng. 46 (1) (2004) 1–15.
[40] Dechter Rina, Avi Dechter, Belief maintenance in dynamic constraint networks, University of California. Computer Science Department, 1988.
[41] A. Hernández-Arauzo, J. Puente, R. Varela, J. Sedano, Electric vehicle charging under power and balance constraints as dynamic scheduling, Comput. Ind. Eng. 85 (2015) 306–315.
[42] Robert P. Dick, David L. Rhodes, Wayne Wolf, TGFF: task graphs for free, in: Proceedings of the 6th International Workshop on Hardware/software Codesign, 1998, pp. 97–101.
[43] David E. Goldberg, Kalyanmoy Deb, A comparative analysis of selection schemes used in genetic algorithms, Found. Genet Algorithms 1 (1991) 69–93.
[44] Pui Wah Poon, Jonathan Neil Carter, Genetic algorithm crossover operators for ordering applications, Comput. Oper. Res. 22 (1) (1995) 135–147.
[45] DI George Amalarethinam, G. . JJoyce Mary, A new DAG based dynamic task scheduling algorithm (DYTAS) for multiprocessor systems, Int. J. Comput. Appl. 19 (8) (2011) 24–28.
[46] http://www.kasahara.cs.waseda.ac.jp/schedule/.
[47] Dervis Karaboga, An idea based on honey bee swarm for numerical optimization. 200. Technical report-tr06, Erciyes university, computer engineering department, 2005.
[48] Karaboga Dervis, et al., A comprehensive survey: artificial bee colony (ABC) algorithm and applications, Artif. Intell. Rev. 42 (1) (2014) 21–57.
[49] Huo Jiuyuan, Liqun Liu, Yaonan Zhang, An improved multi-cores parallel artificial Bee colony optimization algorithm for parameters calibration of hydrological model, Future Gener. Comput. Syst. 81 (2018) 492–504.
[50] Kennedy James, Russell Eberhart, Particle swarm optimization, in: Neural Networks, 1995. Proceedings., IEEE International Conference on. Vol. 4, IEEE, 1995.
[51] Scott C. Kirkpatrick, Daniel Gelatt, Mario P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[52] NguyenTrung Thanh, Shengxiang Yang, Juergen Branke, Evolutionary dynamic optimization: A survey of the state of the art, Swarm Evol. Comp. 6 (2012) 1–24.
[53] Yu Xin, et al., Empirical analysis of evolutionary algorithms with immigrants schemes for dynamic optimization, Memetic Comput. 1 (1) (2009) 3–24.
[54] Alba Enrique, Sarasola Briseida, ABC, a new performance tool for algo. solving dynamic optimization problems, in: Evo. Comp. (CEC), 2010 IEEE Cong. on, IEEE, 2010.
[55] Ben-Romdhane Hajer, Enrique Alba, Saoussen Krichen, Best practices in measuring algorithm performance for dynamic optimization problems, Soft Comput. 17 (6) (2013) 1005–1017.
[56] Cucinotta Tommaso, Giuseppe Lipari, Lutz Schubert, Operating system and scheduling for future multicore and many-core platforms, Programming Multicore and Many-core Computing Systems 86 (2017).
[57] Lakshmanan Karthik, Dionisiode Niz, Ragunathan Rajkumar, Coordinated task scheduling, allocation and synchronization on multiprocessors, in: Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE, IEEE, 2009.
[58] Akhter Shameem, Jason Roberts, ason Roberts Multi-core programming, Vol. 33, Intel press, Hillsboro, 2006.
[59] Li Tong, et al., Efficient operating system scheduling for performance-asymmetric multi-core architectures, in: Supercomputing, 2007. SC'07. Proc. of the 2007 ACM/IEEE Conf. on, IEEE, 2007.
[60] Karthik S. Lakshmanan, Scheduling and Synchronization for Multi-core Real-time Systems, (Diss.), Carnegie Mellon University, 2011.
[61] Lakshmanan Karthik, Shinpei Kato, Ragunathan Rajkumar, Scheduling parallel real-time tasks on multi-core processors, in: Real-Time Systems Symposium (RTSS), 2010 IEEE 31st, IEEE, 2010.
[62] Song Insop, Sehjeong Kim, Fakhreddine Karray, A real-time scheduler design for a class of embedded systems, IEEE/ASME Trans. Mechatronics 13 (1) (2008) 36–45.
[63] Lang Jun, David B. Stewart, A study of the applicability of existing exception-handling techniques to component-based real-time software technology, ACM Trans. Programm. Lang. Syst. (TOPLAS) 20 (2) (1998) 274–301.
[64] https://users.ece.cmu.edu/~koopman/des_s99/exceptions/.
[65] H Streich, Taskpair-scheduling: An approach for dynamic real-time systems, in: Parallel and Distributed Real-Time Systems, 1994. Proc. of the Second Workshop on, IEEE, 1994.
[66] Romanovsky Alexander, Jie Xu, Brian Randell, Exception handling in object-oriented real-time distributed systems, in: Object-Oriented Real-Time Distributed Computing, 1998.(ISORC 98) Proceedings. 1998 First International Symposium on, IEEE, 1998.

Pranab K. Muhuri was born in Chittagong, Bangladesh. He received his Ph.D. degree in Computer Engineering in 2005 from IT-BHU [now Indian Institute of Technology, (BHU)], Varanasi, India. He is currently an Associate Professor at the Department of Computer Science, South Asian University, New Delhi, India, where he is leading the computational intelligence research group. His current research interests are mainly in real-time systems, computation intelligence, especially fuzzy systems, evolutionary algorithms, perceptual computing, and machine leaning. Pranab is an active member of the IEEE Computer Society, IEEE Computational Intelligence Society, and IEEE SMC Society. Currently, he is serving as a member of the Editorial Board of the Applied Soft Computing Journal.

Amit Rauniyar received his Bachelor degree in Computer Applications from HNB Garhwal University in 2012 and M.Sc. Computer Science in 2015 from South Asian University (SAU). Currently, he is a Visvesvaraya Reserach Fellow (funded by Ministry of Electronics & IT, Govt. of India) and pursuing Ph.D. degree in Computer Science from SAU. Amit is an active member of IEEE Computational Intelligence Society and IEEE Systems, Man and Cybernetics Society. His research interests include evolutionary algorithms, multi-factorial optimization, real-time and multi-robots systems.

Rahul Nath received his B.Sc.(Hons.) degree from Delhi University and M.Sc. degree in Computer Science from South Asian University (SAU) in 2011 and 2013, respectively. Currently, he is a Ph.D. scholar in the Dept. of Computer Science, SAU, New Delhi. Rahul is a CSIR Senior Research Fellow. He is an active member of IEEE Computational Intelligence Society and IEEE Systems, Man and Cybernetics Society. His research interests include evolutionary computation, multi-objective/many objective/bilevel optimization, real-time systems, and pollution-routing problem.