J. Parallel Distrib. Comput. 70 (2010) 13–22

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Genetic algorithms for task scheduling problem

Fatma A. Omara [a], Mona M. Arafa [b,*]

[a] Computer Science Department, Faculty of Computers & Information, Cairo University, Egypt

[b] Mathematics Department, Faculty of Science, Banha University, Egypt

ARTICLE INFO

Article history:
Received 21 January 2007
Received in revised form
21 September 2009
Accepted 22 September 2009
Available online 6 October 2009

Keywords:
Evolutionary computing
Genetic algorithms
Scheduling
Task partitioning
Graph algorithms
Parallel processing

ABSTRACT

The scheduling and mapping of the precedence-constrained task graph to processors is considered to be the most crucial NP-complete problem in parallel and distributed computing systems. Several genetic algorithms have been developed to solve this problem. A common feature in most of them has been the use of chromosomal representation for a schedule. However, these algorithms are monolithic, as they attempt to scan the entire solution space without considering how to reduce the complexity of the optimization process. In this paper, two genetic algorithms have been developed and implemented. Our developed algorithms are genetic algorithms with some heuristic principles that have been added to improve the performance. According to the first developed genetic algorithm, two fitness functions have been applied one after the other. The first fitness function is concerned with minimizing the total execution time (schedule length), and the second one is concerned with the load balance satisfaction. The second developed genetic algorithm is based on a task duplication technique to overcome the communication overhead. Our proposed algorithms have been implemented and evaluated using benchmarks. According to the evolved results, it has been found that our algorithms always outperform the traditional algorithms.

1. Introduction

The problem of scheduling a task graph of a parallel program onto a parallel and distributed computing system is a well-defined NP-complete problem that has received a large amount of attention, and it is considered one of the most challenging problems in parallel computing [11]. This problem involves mapping a Directed Acyclic Graph (DAG) for a collection of computational tasks and their data precedence onto a parallel processing system. The goal of a task scheduler is to assign tasks to available processors such that precedence requirements for these tasks are satisfied and, at the same time, the overall execution length (i.e., make span) is minimized [29]. Generally, the scheduling problem could be of the following two types: static and dynamic.

In the case static scheduling, the characteristics of a parallel program such as task processing periods, communication, data dependencies, and synchronization requirement are known before execution [18]. According to the dynamic scheduling, a few assumptions about the parallel program should be made before execution, then scheduling decisions have to be taken on-the-fly [21]. The work in this paper is solely concerned with the static scheduling problem. A general taxonomy for static scheduling algorithms has been reviewed and discussed by Kwok and Ahmad [18]. Many task scheduling algorithms have been developed with moderate complexity as a constraint, which is a reasonable assumption for general purpose development platforms [23,17,20,22]. Generally, the task scheduling algorithms may be divided in two main classes; greedy and non-greedy (iterative) algorithms [9]. The greedy algorithms only attempt to minimize the start time of the tasks of a parallel program. This is done by allocating the tasks into the available processors without back tracking. On the other hand, the main principle of the iterative algorithms is that they start from an initial solution and try to improve it. The greedy task scheduling algorithms may be classified into two categories; algorithms with duplication, and algorithms without duplication. One of the common algorithms in the first category is the Duplication Scheduling Heuristic (DSH) algorithm [11]. The DSH algorithm works by first arranging nodes in a descending order according to their static *b*-level, then determining the start-time of the node on the processor without duplication of any ancestor. After that, DSH attempts to duplicate ancestors of the node during the duplication time slot until the slot is used up or the start-time of the node does not improve. However, one of the best algorithms in the second category is the Modified Critical Path (MCP) algorithm [28]. The MCP algorithm computes at first the ALAPs of all the nodes, then creates a ready list containing ALAP times of the nodes in an ascending order. The ALAP of a node is computed by computing the length of the Critical Path (CP) and then subtracting the *b*-level of a node from it. Ties are broken by considering the minimum ALAP time of the

* Corresponding author.
E-mail addresses: f.omara@fci-cu.edu.eg (F.A. Omara), m_h_banha@yahoo.com (M.M. Arafa).

0743-7315/$ – see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2009.09.009

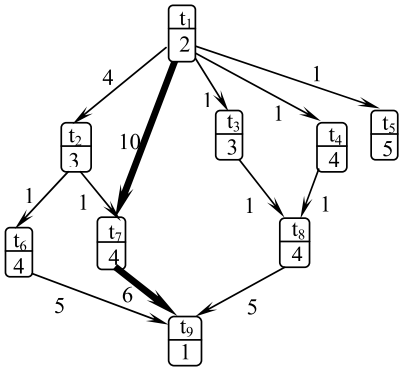

Fig. 1. Example of DAG.

children of a node. If the minimum ALAP time of the children is equal, ties are broken randomly. According to the MCP algorithm, the highest priority node in the list is picked up and assigned to a processor that allows the earliest start time using an insertion approach. Recently, Genetic Algorithms (GAs) have been widely reckoned as useful meta-heuristics for obtaining high quality solutions for a broad range of combinatorial optimization problems including the task scheduling problem [29,18]. Another merit of genetic search is that its inherent parallelism can be exploited to further reduce its running time [5]. The basic principles of GAs were first laid down by Holland [10], and after that they have been well described in many texts. The GA operates on a population of solutions rather than a single solution. The genetic search begins by initializing a population of individuals. Individual solutions are selected from the population, then are mated to form new solutions. The mating process is implemented by combining or crossing over genetic material from two parents to form the genetic material for one or two new solutions; this transfers the data from one generation of solutions to the next. Random mutation is applied periodically to promote diversity. The individuals in the population are replaced by the new generation. A fitness function, which measures the quality of each candidate solution according to the given optimization objective, is used to help determining which individuals are retained in the population as successive generations evolve [13]. There are two important but competing themes that exist in a GA search; the need for selective pressure so that the GA is able to focus the search on promising areas of the search space, and the need for population diversity so that important information (particular bit values) is not lost [19,7].

Recently, several GAs have been developed for solving the task scheduling problem, the main distinction between them has been the chromosomal representation of a schedule [25,6,14,3,16, 26,29]. In this paper, we propose two hybrid genetic algorithms which we designate as the Critical Path Genetic Algorithm (CPGA) and the Task Duplication Genetic Algorithm (TDGA). Our developed algorithms show the effect of the amalgamation of greedy algorithms with the genetic algorithm meta-heuristic. The first algorithm, CPGA, is based on how to use the idle time of the processors efficiently, and reschedule the critical path nodes to reduce their start time. Then, two fitness functions have been applied, one after the other. The first fitness function is concerned with minimizing the total execution time (schedule length), and the second one is concerned with satisfying the load balance. The second algorithm, TDGA, is based on task duplication principle to minimize the communication overheads.

The remainder of this paper is organized as follows: Section 2 gives a description for the model of task scheduling problem. An implementation of the standard GA is presented in Section 3. Our developed CPGA is introduced in Section 4. Section 5 presents the details of our TDGA algorithm. Performance Evaluation of our developed algorithms with respect to MCP algorithm, and DSH algorithm is presented in Section 6. Conclusions are given in Section 7.

Table 1
Selected benchmark programs.

Benchmarks programs	No_tasks	Source	Note
Pg_1	100	[30]	Random graphs
Pg_2	90	[30]	Robot control program
Pg_3	98	[30]	Sparse matrix solver

2. The model for task scheduling problem

The model of the parallel system to be considered in this work can be described as follows [18]: The system consists of a limited number of fully connected homogeneous processors. Let a task graph G be a Directed Acyclic Graph (DAG) composed of N nodes $n_1, n_2, n_3, \ldots, n_N$. Each node is termed a task of the graph which in turn is a set of instructions that must be executed sequentially without preemption in the same processor. A node has one or more inputs. When all inputs are available, the node is triggered to execute. A node with no parent is called an entry node and a node with no child is called an exit node. The computation cost of a node n_i is denoted by (n_i) weight. The graph also has E directed edges representing a partial order among the tasks. The partial order introduces a precedence-constrained DAG and implies that if $n_i \rightarrow n_j$, then n_j is a child, which cannot start until its parent n_i finishes. The weight on an edge is called the communication cost of the edge and is denoted by $c(n_i, n_j)$. This cost is incurred if n_i and n_j are scheduled on different processors and is considered to be zero if n_i and n_j are scheduled on the same processor. If a node n_i is scheduled to processor P, the start time and finish time of the node are denoted by $\mathrm{ST}(n_i, P)$ and $\mathrm{FT}(n_i, P)$ respectively. After all nodes have been scheduled, the schedule length is defined as $\max\{\mathrm{FT}(n_i, P)\}$ across all processors. The objective of the task scheduling problem is to find an assignment and the start times of the tasks to processors such that the schedule length is minimized and, in the same time, the precedence constrains are preserved. A Critical Path (CP) of a task graph is defined as the path with the maximum sum of node and edge weights from an entry node to an exit node. A node in CP is denoted by CP Nodes (CPNs). An example of a DAG is represented in Fig. 1, where CP is drawn in bold.

3. The *S*tandard *G*enetic *A*lgorithm (SGA)

Before presenting the details of our developed algorithms, some principles which are used in the design are discussed.

Definition. Any task cannot start until all parents have finished. Let P_j be the processor on which the kth parent task t_k of task t_i is scheduled. The Data Arrival Time (DAT) of t_i at processor P_i is defined as:

$$\mathrm{DAT} = \max\{\mathrm{FT}(t_k, P_j) + c(t_i, t_k)\}; \quad k = 1, \ldots, \mathit{No\_parent} \tag{1}$$

where, *No_parent* is the number of t_i's parents.

$$\text{If } i = j \text{ then } c(t_i, t_k) \text{ equals zero.} \tag{2}$$

The parent task that maximizes the above expression is called the *favorite predecessors* of t_i and it is denoted by $\mathit{favpred}(t_i, P_j)$.

The benchmark programs which have been used to evaluate our algorithms are listed in Table 1.

3.1. The SGA implementation

The SGA has been implemented first. This algorithm is started with an initial population of feasible solutions. Then, by applying some operators, the best solution can be found after some generations. The selection of the best solution is determined according to the value of the fitness function. According to this SGA, the

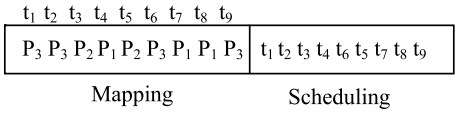

Fig. 2. Representation of a chromosome.

Table 2
A comparison between roulette wheel and tournament selection.

Benchmark programs	Roulette wheel selection	Tournament selection
Pg_1	301.6	283.7
Pg_2	1331.6	969
Pg_3	585.8	521.8

chromosome is divided into two sections; mapping and scheduling sections. The mapping section contains the processors indices where tasks are to be run on it. The schedule section determines the sequence for the processing of the tasks. Fig. 2 shows an example of such a representation of the chromosome. Where, tasks t_4, t_7, t_8 will be scheduled on processor P_1, tasks t_3, t_5 will be scheduled on processor P_2, and tasks t_1, t_2, t_6 and t_9 will be scheduled on processor P_3. The length of the chromosome is linearly proportional to the number of tasks.

3.1.1. Genetic formulation of SGA

Initial population

The initial population is constructed randomly. The first part of the chromosome (i.e. mapping) is chosen randomly from 1 to *No_Processors* where the *No_Processors* is the number of the processors in the system. The second part (i.e. the schedule) is generated randomly such that the topological order of the graph is preserved.

Fitness function

The main objective of the scheduling problem is to minimize the schedule length of a schedule.

$$\text{Fitness function} = (a/S\_Length) \tag{3}$$

where a is a constant and *S_Length* is the schedule length which is determined by the following equation:

$$S\_Length = \max(\text{FT}[t_i]) : \quad i = 1, \ldots, K_{\text{No\_tasks}}. \tag{4}$$

The pseudo code of The Task Schedule using SGA is as follows:

Function schedule_length

1. $\forall \text{RT}[P_j] = 0$ // RT is the ready time of the processors.
2. Let LT be a list of tasks according to the topological order of DAG.
3. For $i = 1$ to No_Tasks // No_Tasks is number of tasks in DAG
   a. Remove the first task t_i form list LT.
   b. For $j = 1$ to No_Processors // No_Processors is number of Processors
      If t_i is scheduled to processor P_j
      $\text{ST}[t_i] = \max\{\text{RT}[P_j], \text{DAT}(t_i, P_j)\}$
      $\text{FT}[t_i] = \text{ST}[t_i] + \text{weight}[t_i]$
      $\text{RT}[P_j] = \text{FT}[t_i]$
      End If
   End For
   End For
   c. $S\_Length = \max(\text{FT})$

Example. By considering the chromosome represented in Fig. 2 as a solution of a DAG that is represented in Fig. 1, the *Fitness* function that is defined by Eq. (3) has been used to calculate the schedule length (see Fig. 3).

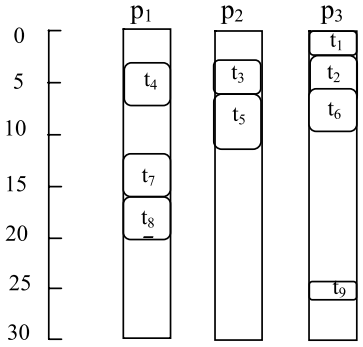

Fig. 3. The schedule length.

3.1.2. Genetic operators

In order to apply crossover and mutation operators, the selection phase should be applied first. This selection phase is used to allocate reproductive trials to chromosomes according to their fitness. There are different approaches that can be applied in the selection phase. According to the work in this paper, fitness-proportional roulette wheel selection [12] and tournament selection [8] are compared such that the better method is used (i.e., produces the shortest schedule length). In the roulette wheel selection, the probability of selection is proportional to the chromosome's fitness. The analogy with a roulette wheel arises because one can imagine the whole population forming a roulette wheel with the size of any chromosome's slot is proportional to its fitness. The wheel is then spun and the figurative "ball" thrown in. The probability of the ball coming to rest in any particular slot is proportional to the arc of the slot and thus to the fitness of the corresponding chromosome. In binary tournament selection, two chromosomes are picked at random from the population. Whichever has the higher fitness is chosen. This process is repeated for all the chromosomes in the population.

Table 2 contains the results for a comparison between these two selection methods using 4 processors for each benchmark program listed in Table 1. According to the results listed in Table 2, the tournament selection method produces schedule length smaller than that produced by the roulette wheel selection. Therefore, the tournament selection method is used in the work of this paper.

Crossover operator

Each chromosome in the population is subjected to crossover with probability μ_c. Two chromosomes are selected from the population, and a random number RN $\in [0, 1]$ is generated for each chromosome. If RN $< \mu_c$, these chromosomes are subjected to the crossover operation using one of two kinds of the crossover operators; that is single point crossover and order crossover operators. Otherwise, these chromosomes are not changed. The pseudo code of the crossover function is as follows.

Function crossover

1. Select two chromosomes chrom1 and chrom2
2. Let RN a random real number between 0 and 1
3. If RN < 0.5 /* operators probability
   Crossover_Map (chrom1, chrom2)
   Else
   Crossover_Order (chrom1, chrom2)

According to the crossover function, one of the crossover operators is used.

Crossover map

When the single crossover is selected, it is applied to the first part of the chromosome. By having two chromosomes, a random integer number called the crossover point is generated from 1 to *No_Tasks*. The portions of the chromosomes lying to the right of

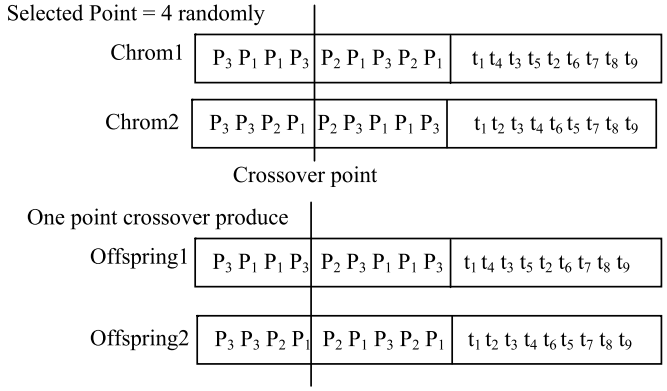

Fig. 4. One point crossover operator.

the crossover point are exchanged to produce two offspring (see Fig. 4).

Order crossover

When the order crossover operator is applied to the second part of the chromosome, a random point is chosen. First, pass the left segment from the chrom1 to the offspring, then construct the right fragment of the offspring according to the order of the right segment of chrom2 (see Fig. 5 as an example).

Mutation operator

Each position in the first part of the chromosome is subjected to mutation with a probability μ_m. Mutation involves changing the assignment of a task from one processor to another. Fig. 6 illustrates the mutation operation on chrom1. After the mutation operator is applied, the assignment of task t_4 is changed from processor P_3 to processor P_1.

4. The Critical Path Genetic Algorithm (CPGA)

Our developed CPGA algorithm is considered a hybrid of GA principles and heuristic principles (e.g., given priority of the nodes according to ALAP level). On the other hand, the same principles and operators which are used in the SGA algorithm have been used in the CPGA algorithm. The encoding of the chromosome is the same as in SGA, but in the initial population the second part (schedule) of the chromosome can be constructed using one of the following ways:

1. The schedule part is constructed randomly as in SGA.
2. The schedule part is constructed using ALAP.

These two ways have been applied using benchmark programs listed in Table 1 with four processors. According to the comparative results listed in Table 3, it is seen that the priority of the nodes by ALAP method outperforms the random one in the most cases.

By using ALAP, the second parts of the chromosomes become static along the population. So, the crossover operator is restricted to the one point crossover operator.

Three modifications have been applied in the SGA to improve the scheduling performance. These modifications are:

1. Reuse idle time,

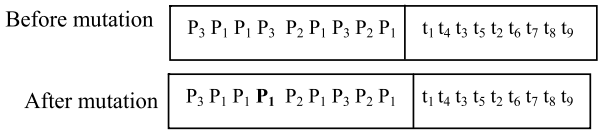

Fig. 6. Mutation operator.

Table 3
A comparison between random and order ALAP order methods.

Benchmark programs	Random order	ALAP order
Pg_1	183.4	152.3
Pg_2	848.5	826.4
Pg_3	301.8	293.8

2. Priority of the CPNs, and
3. Load balance.

Reuse idle time modification:

This modification is based on the insertion approach [11] where the idle time of the processors is used by assigning some tasks to idle time slots. This modification is implemented in our algorithm using *Test_Slots* function. The pseudo code of this function is as follows:

Function Test_Slots

1. Let LT be a list of ready tasks
2. Initially the idle_time list is empty and S_idle_time = 0, E_idle_time = 0
3. While the list LT is not empty, get a task t_i from the head of the list
   a. Min_ST = ∞
   b. For each processor P_j
      If t_i is scheduled to P_j
         Let this_ST be equal to the start time of t_i on P_j
         If this_ST < Min_ST Then Min_ST = this_ST
      If the idle_time list of P_j is not empty
         For each time_slot of the idle_time list
            If (E_idle_time-S_idle_time) >= weight [t_i]
            && DAT(t_i, P_j) < S_idle_time
               Then schedule t_i in the idle_time and
               update the S_idle_time and E_idle_time
               Let st_time be the start time of the task *ti*
               equal to S_ideal_time
            End If
   c. If st_time < Min_ST Then Min_ST = st_time

Example. Consider the schedule represented in Fig. 3. The processor P_1 has an idle time slot; the start of this idle time (S_idle_time) is equal to 7 while its end time (E_idle_slot) is equal to 12. On the other hand, the weight (t_8) = 4 and DAT (t_8, P_1)= S_idle_slot = 7. By applying the modification, t_8 can be rescheduled to start at time 7. The final schedule length according to this modification becomes 23 instead of 26 (see Fig. 7).

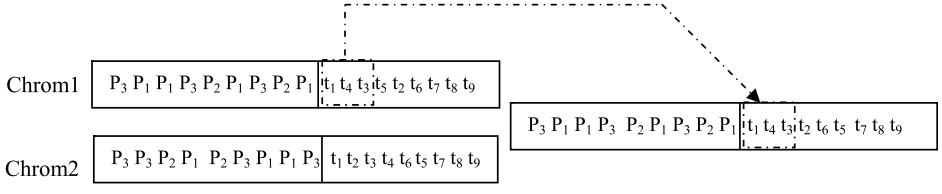

Fig. 5. Order crossover operator.

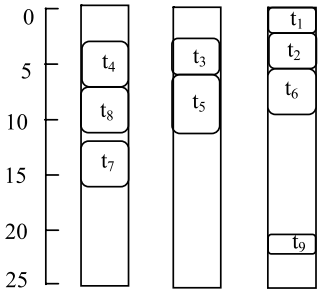

Fig. 7. The schedule length after applying the test_slots function is reduced from 26 to 23.

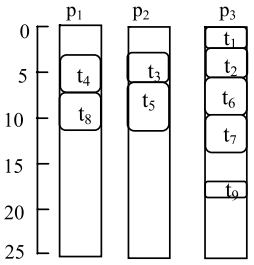

Fig. 8. The schedule length after applying the reschedule of the CPNs function is reduced from 23 to 17.

Priority of CPNs modification:

According to the second modification, another optimization factor is applied to recalculate the schedule length after giving high priorities for the (CPNs) such that they can start as early as possible. This modification is implemented using a function called *Reschedule_CPNs* function. The pseudo code of this function is as follows:

Function Reschedule_CPNs

1. Determine the CP and make a list of CPNs
2. While the list of CPNs is not empty DO
   Remove the task t_i from the list
   Let VIP = $favpred(t_i, P_j)$
   If VIP is assigned to processor P_j
   Then The task t_i is assigned to processor P_j
   End If

Example. We apply the *Reschedule_CPNs* function on the scheduling presented in Fig. 7. According to the DAG presented in Fig. 1, it is found that the CPNs are t_1, t_7, and t_9. Task t_1 is the entry node and it has no predecessor and the *favpred* of t_7 is task t_1. Task t_7 is scheduled to processor P_1. Also the *favpred* of t_9 is t_8, but at the same time it starts early on the processor P_3, so t_9 has not moved. The final schedule length is reduced to 17 instead of 23 (see Fig. 8).

Load balance modification:

Because the main objective of the task scheduling is to minimize the schedule length, it is found that several solutions can produce the same schedule length, but the load balance between processors might not be satisfied in some of them. The aim of load balance modification is that to obtain the minimum schedule length and, in the same time, the load balance is satisfied. This has been satisfied by using two fitness functions one after the other instead of one fitness function. The first fitness function deals with minimizing the total execution time, and the second fitness function is used to satisfy load balance between processors. This function is proposed in [15] and it is calculated by the ratio of the maximum execution time (i.e. schedule length) to the average execution time over all processors.

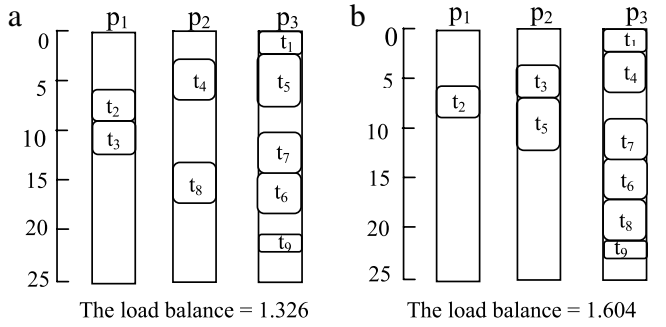

Fig. 9. According to balance fitness function solution (a) is better than solution (b).

If the execution time of processor P_j is denoted by $E\_time[P_j]$, then the average execution time over all processors is:

$$\text{avg} = \sum_{j=1}^{No\_processors} E\_time[P_j]/No\_processors. \tag{5}$$

So the load balance is calculated as

$$load\_balance = S\_Length/\text{Avg}. \tag{6}$$

Suppose that two task scheduling solutions are given in Fig. 9. The schedule length of both solutions is equal to 23.

Solution a : $\text{Avg} = (12 + 17 + 23)/3 \approx 17.33$
$Load\_balance = 23/17.33 \approx 1.326$
Solution b : $\text{Avg} = (9 + 11 + 23)/3 \approx 14.33$
$Load\_balance = (23/14.33) \approx 1.604$.

According to the balance fitness function, solution (a) is better than solution (b).

Adaptive μ_c and μ_m parameters

Srinivas and Patnaik [24] have proposed an adaptive method to tune crossover rate μ_c and mutation rate μ_m, on the fly, based on the idea of sustaining in diversity in a population without affecting its convergence properties. Therefore; the rate μ_c is defined as:

$$\mu_c = \frac{k_c(f_{\max} - f_c)}{(f_{\max} - f_{\text{avg}})} \tag{7}$$

and the rate μ_m is defined as:

$$\mu_m = \frac{k_m(f_{\max} - f_m)}{(f_{\max} - f_{\text{avg}})} \tag{8}$$

where,

$f_{\max}$ is the maximum fitness value, f_{avg} is the average fitness value
f_c is the fitness value of the best chromosome for the crossover
f_m is the fitness value of the chromosome to be mutated and, k_c and k_m are positive real constants less than 1.

The CPGA algorithm has been implemented into two versions: the first version has been done using static parameters ($\mu_c = 0.8$ and $\mu_m = 0.02$), and the second version has been done using adaptive parameters. Table 4 represents the comparison results between these two versions. According to the results, it is seen that by using adaptive parameters (μ_c and μ_m), one can help preventing a GA from getting stuck at local minima. So, using the adaptive method is batter than using static values of μ_c and μ_m.

Table 4
A comparison between static and dynamic μ_c, μ_m parameters.

Benchmark programs	Dynamic parameters	Static parameters
Pg_1	148	152.3
Pg_2	785.6	826.4
Pg_3	288.2	293.8

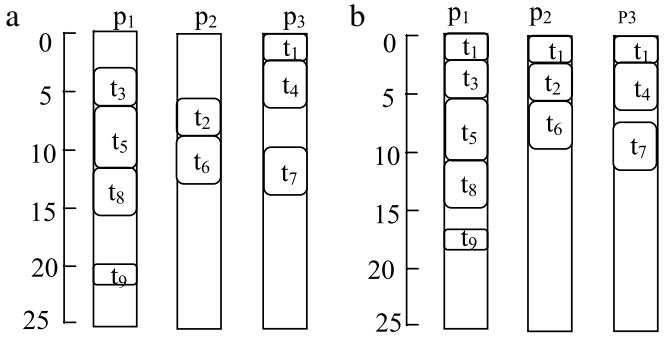

Fig. 10. (a) Before duplication (schedule length = 21) (b) After duplication (schedule length = 18).

$$(t_2, P_1)(t_3, P_2)(t_4, P_1)(t_2, P_2)(t_4, P_2)$$

Fig. 11. An example of the chromosome.

5. The Task Duplication Genetic Algorithm (TDGA)

Even with an efficient scheduling algorithm, some processors might be idle during the execution of the program because the tasks assigned to them might be waiting to receive some data from the tasks assigned to other processors. If the idle time slots of a waiting processor could be effectively used by identifying some critical tasks and redundantly allocating them in these slots, the execution time of the parallel program could be further reduced [1].

According to our proposed algorithm, a good schedule based on task duplication has been proposed. This proposed algorithm called the Task Duplication Genetic Algorithm (TDGA) employs a genetic algorithm for solving the scheduling problem.

Definition. At a particular scheduling step; for any task t_i on a processor P_j

If EST($favpred(t_i, P_j)$) + weight($favpred(t_i, P_j)$) < EST(t_i, P_j)

Then EST(t_i, P_j) can be reduced by scheduling $favpred(t_i, P_j)$ to P_j.

This definition could be applied recursively upward the DAG to reduce the schedule length.

Example. To clarify the effect of the task duplication technique, consider the schedule presented in Fig. 10(a) for the DAG in Fig. 1, the schedule length is equal to 21. If t_1 is duplicated to processor P_1 and P_2 the schedule length is reduced to 18 (see Fig. 10(b)).

5.1. Genetic formulation of the TDGA

According to our TDGA algorithm, each chromosome in the population consists of a vector of order pairs (t, p) which indicates that task t is assigned to processor p. The number of order pairs in a chromosome may vary in length. An example of a chromosome is shown in Fig. 11. The first order pair shows that task t_2 is assigned to processor P_1, and the second one indicates that task t_3 is assigned to processor P_2, etc ....

According to the duplication principles, the same task may be assigned more than once to different processors without duplicating it in the same processor. If a task processor pair appears more than once on the chromosome, only one of the pairs is considered. According to Fig. 11, the task t_2 is assigned to processor P_1 and P_2.

Table 5
A comparison between the methods (HD and RD).

Benchmark programs	HD	RD
Pg_1	493.9	494.1
Pg_2	1221	1269.5
Pg_3	641.2	616.2

Definition. Invalid chromosomes are the chromosomes that do not contain all the DAG tasks. These invalid chromosomes might be generated.

Initial population

According to our TDGA algorithm, two methods to generate the initial population are applied. The first one is called Random Duplication (RD) and the second one is called Heuristic Duplication (HD). According to RD, the initial population is generated randomly such that each task can be assigned to more than one processor.

In the HD, the initial population is initialized with randomly generated chromosomes, while each chromosome consists of exactly one copy of each task (i.e. no task duplication). Then, each task is randomly assigned to a processor. After that, a duplication technique is applied by a function called the Duplication_Process. The pseudo code of the Duplication_Process function is as follows:

Function Duplicatin_Process

1. Compute SL for each task in the DAG
2. Make a list S_list of the tasks according to SL in descending order
3. Take the task t_i from S_list
4. While S_list is not empty
   a. If t_i is assigned to processor P_j
      IF $favpred(t_i, P_j)$ is not assigned to P_j
      IF (time_slot(t_i, P_j) $\geq$ weight($favpred(t_i s, P_j)$)
      assign $favpred(t_i, P_j)$ to P_j

According to the implementation results using RD and HD methods, it is found that two methods produce nearly the same results. Therefore, the HD method has been considered in our TDGA algorithm (see Table 5).

Fitness function

Our fitness function is defined as 1/*S-length*, where *S-length* is defined as the maximum finishing time of all tasks of the DAG. The proposed GA assigns zero to an invalid chromosome as its fitness value.

Genetic operators

Crossover operator

Two point crossover operator is used. According to the two point crossover, two points are assigned to bind the crossover region. Since each chromosome consists of a vector of task processor pair, the crossover exchanges substrings of pairs between two chromosomes. Two points are randomly chosen and the partitions between the points are exchanged between two chromosomes to form two offspring. The crossover probability μ_c gives the probability that a pair of chromosome will undergo a crossover. An example of a two point crossover is shown in Fig. 12.

Mutation operator

The mutation probability μ_m indicates the probability that an order pair will be changed. If a pair is selected to be mutated, the processor number of that pair will be randomly changed. An example of mutation operator is shown in Fig. 13.

Chrom1 (t_3, P_2) (t_1, P_2) | (t_4, P_1) (t_1, P_1) | (t_2, P_2)
Chrom 2 (t_1, P_1) (t_3, P_1) | (t_4, P_2) (t_4, t_2) | (t_2, P_1) (t_1, P_2)

Crossover point1 Crossover point2

Two points 2 and 4 are generated randomly, two point crossover operator produce

Offspring1 (t_3, P_2) (t_1, P_2) | (t_4, P_2) (t_4, t_2) | (t_2, P_2)
Opsspring2 (t_1, P_1) (t_3, P_1) | (t_4, P_1) (t_1, P_1) | (t_2, P_1) (t_1, P_2)

Crossover point1 Crossover point2

Fig. 12. Example of two point crossover operator.

Chrom (t_3, P_2) $(t_1, \mathbf{P_2})$ (t_4, P_1) (t_1, P_1) (t_2, P_2)

Offspring (t_3, P_2) $(t_1, \mathbf{P_1})$ (t_4, P_1) (t_1, P_1) (t_2, P_2)

Fig. 13. Example of mutation operator.

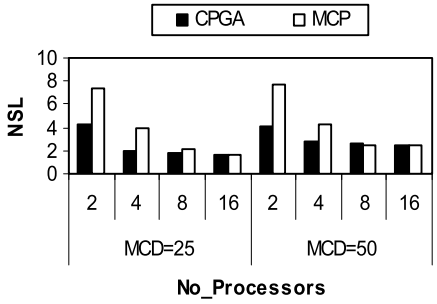

Fig. 14. NSL for Pg_1 and MCD 25 and 50.

6. Performance evaluation

6.1. The problem environment

To evaluate our proposed algorithms, we have implemented them using an Intel processor (2.6 GH) using C++ language. The algorithms are applied using task graphs of specific benchmark application programs which are taken from a Standard Task Graph (STG) archive [30] (see Table 1). The first two programs of this STG set consists of task graphs generated randomly Pg_1, the second program is the robot control (Pg_2) as an actual application program and the last program is the sparse matrix solver (Pg_3). Also, we considered the task graphs with random communication costs. These communication costs are distributed uniformly between 1 and a specified maximum communication delay (MCD). The population size is considered to be 200, and the number of generations is considered to be 500 generations.

6.2. The developed CPGA evaluation

Our algorithm CPGA, and one of the best greedy algorithms, called the MCP algorithm have been implemented and compared. Firstly, a comparison between the CPGA and MCP algorithms with respect to the Normalized Schedule Length (NSL) with different number of processors has been carried out. The NSL is defined as [2]:

$$\mathrm{NSL} = \frac{S\_Length}{\sum_{n_i \in CP} \mathrm{weight}(n_i)} \tag{9}$$

where *S_Length* is the schedule length and weight (n_i) is the weight of the node n_i. The sum of computation costs on the CP represents a lower bound on the schedule length. Such a lower bound may not always be possible to achieve, and the optimal schedule length may be larger than this bound.

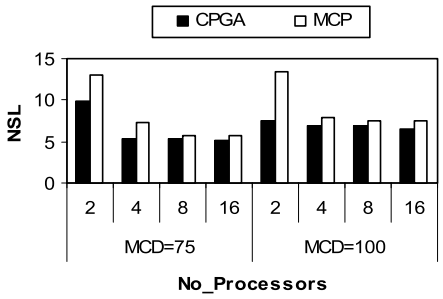

Fig. 15. NSL for Pg_1 and MCD 75 and 100.

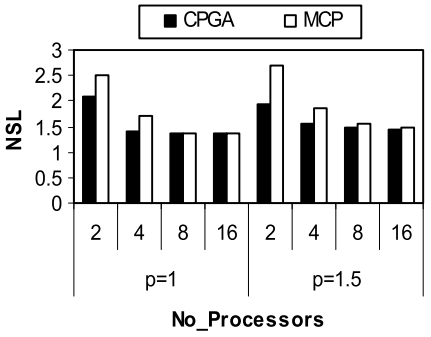

Fig. 16. NSL for Pg_2 and two values of ρ.

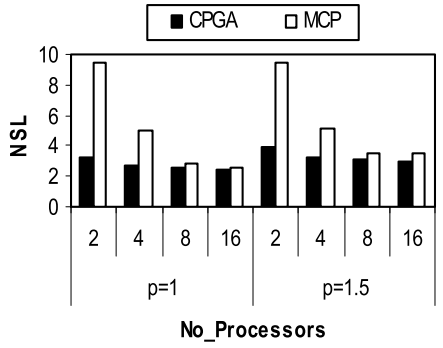

Fig. 17. NSL for Pg_3 and two values of ρ.

Secondly, the performance of the CPGA and MCP are measured with respect to speedup [4]. The speedup can be estimated as:

$$S(p) = \frac{T(1)}{T(\mathrm{P})} \tag{10}$$

where, $T(1)$ is the time required for executing a program on a uniprocessor computer and $T(P)$ is the time required for executing the same program on a parallel computer with P processors.

The NSL for CPGA and MCP algorithms using 2, 4, 8, and 16 processors for Pg_1 and different MCD (25, 50, 75, and 100) are given in Figs. 14 and 15. Also the NSL for Pg_2 and Pg_3 graphs with two different numbers of ρ are given in Figs. 16 and 17 respectively.

Figs. 14–17 show that the performance of our proposed CPGA algorithm is always outperformed MCP algorithm. According to the obtained results, it is found that the NSL of all algorithms is increased when the number of processors is increased. Although, our CPGA is always the best, and it achieves a lower bound when the communication delay is small.

Figs. 18–21 present the speedup of CPGA, and MCP algorithms using Pg_1, Pg_2 and Pg_3 respectively.

According to Figs. 18–21, our proposed CPGA algorithm provides better speedup than that for the MCP algorithm in most cases.

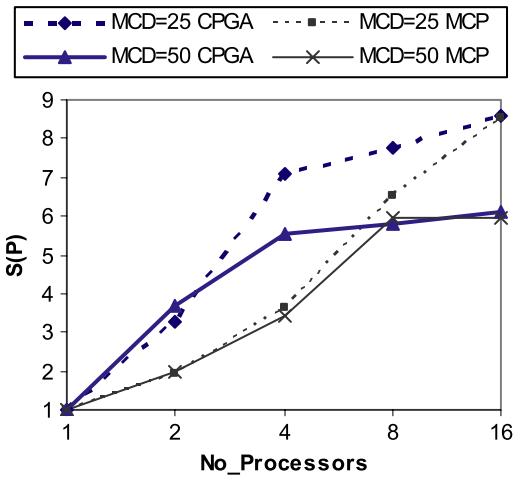

Fig. 18. Speedup for Pg_1 and MCD 25 and 50.

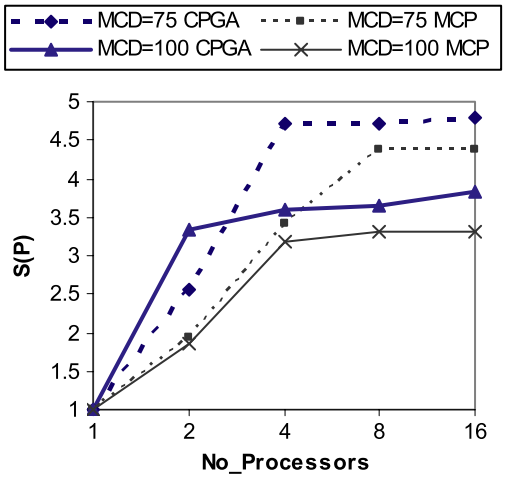

Fig. 19. Speedup for Pg_1 and MCD 75 and 100.

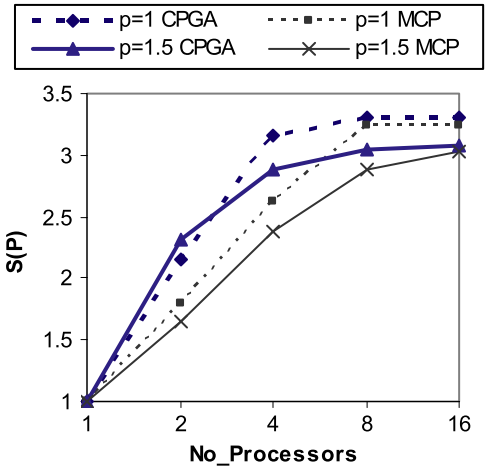

Fig. 20. Speedup for Pg_2 and two values of ρ.

Generally, the speedup increases when the number of processors increases. In some cases the speedup is greater than the number of processors (i.e. super speedup) [27]. Finally, because of the communication overhead, the increasing speedup is not generally linear.

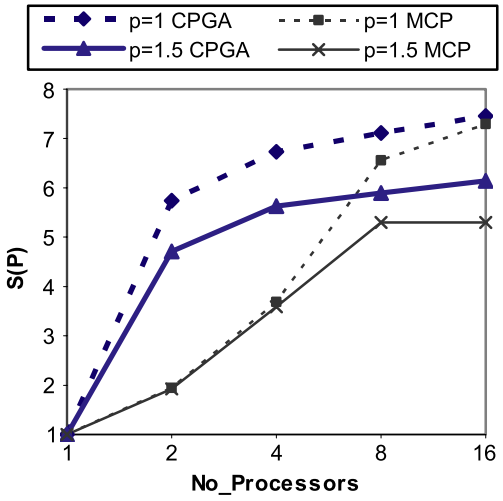

Fig. 21. Speedup for Pg_3 and two values of ρ.

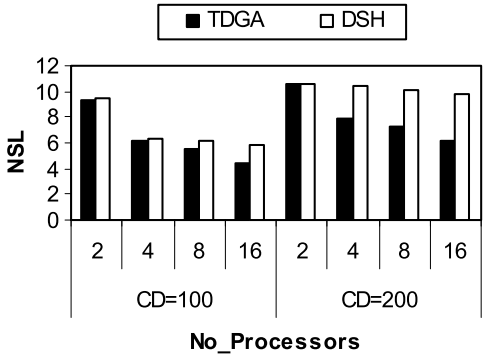

Fig. 22. NSL for Pg_1 and CD = 100 and 200.

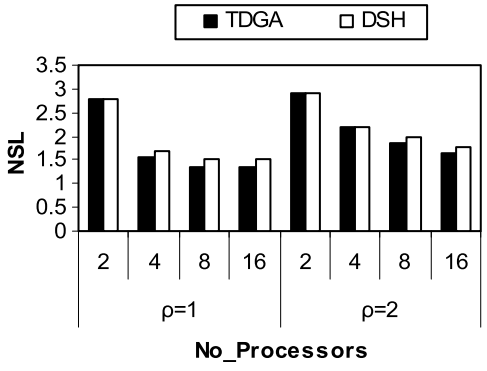

Fig. 23. NSL for Pg_2 and $\rho = 1$ and 2.

6.3. The developed TDGA evaluation

To measure the performance of the TDGA, a comparison between our TDGA algorithm, and one of the well known heuristic algorithms based on task duplication called DSH algorithm has been done with respect to NSL and speedup.

To clarify the effect of task duplication in our TDGA algorithm, the same benchmark application programs Pg_1, Pg_2, and Pg_3 listed in Table 1 have been used with high communication delay.

The NSL for TDGA, and DSH algorithms using 2, 4, 8, and 16 processors for Pg_1 with two values of Communication Delay (CD) (CD = 100 and 200) is given in Fig. 22. Also the NSL for Pg_2 and Pg_3 are given in Figs. 23 and 24.

According to the results in Figs. 22–24, it is found that our TDGA algorithm outperforms the DSH algorithm, especially when the

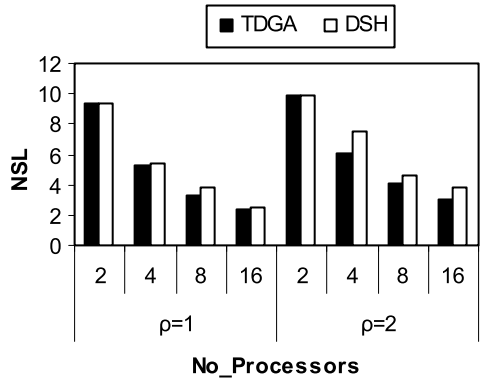

Fig. 24. NSL for Pg_3 and $\rho = 1$ and 2.

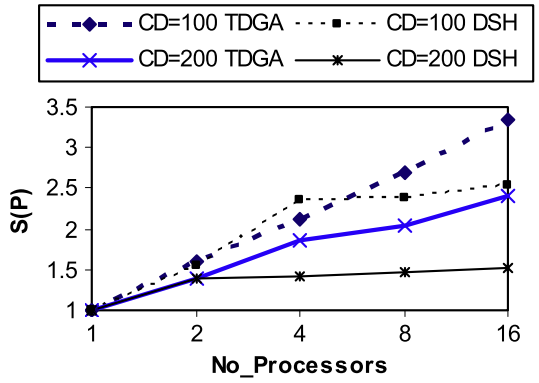

Fig. 25. Speedup for Pg_1 and $\rho = 1$ and 2.

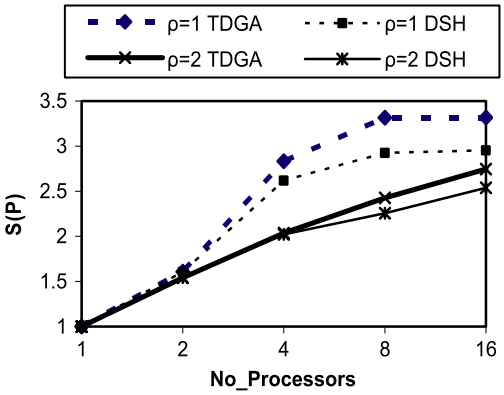

Fig. 26. Speedup for Pg_2 and $\rho = 1$ and 2.

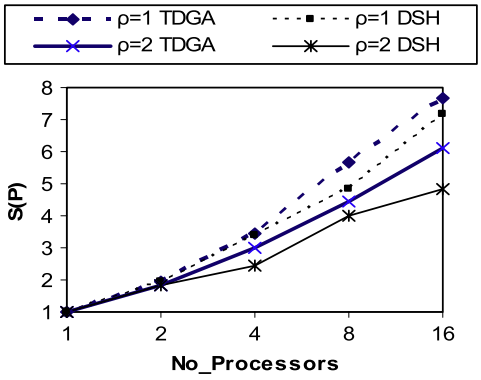

Fig. 27. Speedup for Pg_3 and $\rho = 1$ and 2.

number of communications as well as the number of processors increases.

The speedup of our TDGA algorithm and DSH algorithm is given in Figs. 25–27 for Pg_1, Pg_2, and Pg_3 respectively.

The results reveal that the performance of our TDGA algorithm has always outperformed the DSH algorithm. Also, the TDGA speedup is nearly linear, especially for random graphs.

7. Conclusions

In this paper, an implementation of a standard GA (SGA) to solve the task scheduling problem has been presented. Some modifications have been added to this SGA to improve the scheduling performance. These modifications are based on amalgamating heuristic principles with the GA principles. The first developed algorithm which has been called the Critical Path Genetic Algorithm (CPGA) is based on rescheduling the critical path nodes (CPNs) in the chromosome through different generations. Also, two modifications have been added. The first one is concerned with how to use the idle time of the processors efficiently, and the second one is concerned with satisfying the load balance among processors. The last modification is applied only when there are two or more scheduling solutions with the same schedule length are produced.

A comparative study between our CPGA, and one of the standard heuristic algorithms called the MCP algorithm has been presented using standard task graphs and considering random communication costs. The experimental studies show that the CPGA always outperforms the MCP algorithm in most cases. Generally, the performance of our CPGA algorithm is better than the MCP algorithm.

The second developed algorithm which is called the Task Duplication Genetic Algorithm (TDGA), is based on task duplication techniques to overcome the communication overhead. According to task duplication techniques, the communication delays are reduced and then the overall execution time is minimized, in the same time, the performance of the genetic algorithm is improved. The performance of the TDGA is compared with a traditional heuristic scheduling technique, DSH. The experimental studies show that the TDGA algorithm outperforms the DSH algorithm in most cases.

References

[1] I. Ahmad, Y. Kwok, A new approach to scheduling parallel programs using task duplication, in: Proceeding of the 23rd International Conf. on Parallel Processing, August 1994, North Carolina State University, NC, USA, 1994.
[2] I. Ahmad, Y. Kwok, Benchmarking and comparison of the task graph scheduling algorithms, J. Parallel Distrib. Comput. 95 (1999) 381–422.
[3] I. Ahmed, M.K. Dhodhi, Task assignment using a problem-space genetic algorithm, Concurrency, Pract. Exp. 7 (1995) 411–428.
[4] S.G. Akl, Parallel Computation: Models and Methods, Prentice-Hall, Inc., 1997.
[5] S.M. Alaoui, O. Frieder, T.A. EL-Ghazawi, Parallel genetic algorithm for task mapping on parallel machine, in: Proc. of the 13th International Parallel Processing Symposium & 10th Symp. Parallel and Distributed Processing IPPS/SPDP Workshops, April 1999, San Juan, Puerto Rico, 1999.
[6] S. Ali, S.M. Sait, M.S.T. Benten, GSA: Scheduling and allocation using genetic algorithm, in: Proceedings of the Conference on EURO-DAC with EURO WDHL 1994, Grenoble, 1994, pp. 84–89.
[7] T. Back, U. Hammel, H.-P. Schwefel, Evolutionary computation: Comments on the history and current state, IEEE Trans. Evol. Comput. 1 (1997) 3–17.
[8] T. Blickle, L. Thiele, A mathematical analysis of tournament selection, in: Proceeding of the 6th International Conf. on Genetic Algorithms ICGA95, Morgan Kaufmann, San Francisco, CA, 1995.
[9] P. Bouvry, J. Chassin, D. Trystram, Efficient Solutions for Mapping Parallel Programs, CWI-Center for Mathematics and computer science, Amsterdam, The Netherlands, 1995, published in Euro-Par.
[10] T.H. Corman, C.E. Leiserson, R.L. Rivests, Introduction to Algorithms, MIT Press, 1990.
[11] H. El-Rewini, T.G. Lewis, H.H. Ali, Task Scheduling in Parallel and Distributed Systems, Prentice-Hall International Editions, 1994.
[12] A.T. Haghighat, M. Nikravan, A hybrid genetic algorithm for process scheduling in distributed operating systems considering load balancing, The IASTED Conference on Parallel and Distributed Computing and Networks PDCN, Innsbruck, Austria, 2005.

[13] J.H. Holland, Adaptation in Natural and Artificial Systems, Ann Arbor. Univ. of Michigan Press, 1975.
[14] E.H. Hou, N. Ansari, H. Ren, A genetic algorithm for multiprocessor scheduling, IEEE Trans. Parallel Distrib. Syst. 5 (1994) 113–120.
[15] S. Kumar, U. Maulik, S. Bandyopadhyay, S.K. Das, Efficient task mapping on distributed heterogeneous systems for mesh applications, in: Proceedings of the International Workshop on Distributed Computing, Kolkata, India, 2001.
[16] Yu. Kwok, High performance algorithms for compile-time scheduling of parallel processors, Ph.D. Thesis, Hong Kong University, 1997.
[17] Y. Kwok, I. Ahmad, Dynamic critical path scheduling: An effective technique for allocating task graphs to multi-processors, IEEE Trans. Parallel Distrib. Syst. 7 (1996) 506–521.
[18] Y. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. 31 (1999) 406–471.
[19] D. Levine, A parallel genetic algorithm for the set partitioning problem, Ph.D. Thesis in Computer Science, Department of Mathematics and Computer science, Illinois Institute of Technology, Chicago, USA, 1994.
[20] F.A. Omara, A. Allam, An efficient tasks scheduling algorithm for distributed memory machines with communication delays, J. Inf. Technol. 4 (2005) 326–334.
[21] M.A. Palis, J.C. Liou, S. Rajasekaran, S. Shende, S.S.L Wei, Online scheduling of dynamic trees, Parallel Process. Lett. 5 (1995) 635–646.
[22] A. Radulescu, A.J.C van Gemund, Low cost task scheduling for distributed memory machines, IEEE Trans. Parallel Distrib. Syst. 13 (2002) 648–658.
[23] G.C. Sih, E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Trans. Parallel Distrib. Syst. 4 (1993) 75–87.
[24] M. Srinivas, L.M. Patnaik, Adaptive probabilities of crossover and mutation in genetic algorithm, IEEE Trans. Syst. Man Cybern. 24 (4) (1994) 656–667.
[25] E.G. Talbi, T. Muntean, A new approach for the mapping problem: A parallel genetic algorithm, www.citessr.ist.psu.edu/, 1993.
[26] T. Tsuchiya, T. Osada, T. Kikuno, Genetic-based multiprocessor scheduling using task duplication, Microprocess. Microsyst. 22 (1998) 197–207.
[27] B. Wilkinson, M. Allen, Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers, Pearson Prentice Hall, 2005.
[28] M. Wu, D.D. Gajski, Hypertool: A programming aid for message-passing systems, IEEE Trans. Parallel Distrib. Syst. 1 (1990) 381–422.
[29] A.S. Wu, H. Yu, S. Jin, K.-C. Lin, G. Schiavone, An incremental genetic algorithm approach to multiprocessor scheduling, IEEE Trans. Parallel Distrib. Syst. 15 (2004) 824–834.
[30] http://www.Kasahara.Elec.Waseda.ac.jp/schedule/.

Prof. Fatma A. Omara is a Professor of the Computer Science Department and the Vice Dean for Community Affairs and Development at the Faculty of Computers and Information, Cairo University. She has published over 35 articles in prestigious international journals and conference proceedings. She has served as a Chairman and a member of the Steering Committees and Program Committees of several national Conferences. Prof. Omara is a member of the IEEE and the IEEE Computer Society. She has supervised over 30 Ph.D. and M.Sc. theses. Prof. Omara's interests are Parallel and Distributing Computing, Parallel Processing, Distributed Operating Systems, Task scheduling, High Performance Computing, and Cluster and Grid Computing.

Mona M. Arafa is an Associate Lecturer in the Mathematics Department, at the Faculty of Science at Benha University in Egypt. Ms. Arafa's interests include parallel and distributing computing, parallel processing, distributed operating systems, high performance computing, databases, and data mining.