Int J Parallel Prog (2009) 37:462–487
DOI 10.1007/s10766-009-0107-8

A Bipartite Genetic Algorithm for Multi-processor Task Scheduling

Mohammad Reza Bonyadi · Mohsen Ebrahimi Moghaddam

Received: 31 August 2008 / Accepted: 15 May 2009 / Published online: 3 June 2009

Abstract Until now, several methods have been presented to optimally solve the multiprocessor task scheduling problem that is an NP-hard one. In this paper, a genetic-based algorithm has been presented to solve this problem with better results in comparison with related methods. The proposed method is a bipartite algorithm in a way that each part is based on different genetic schemes, such as genome presentation and genetic operators. In the first part, it uses a genetic method to find an adequate sequence of tasks and in the second one, it finds the best match processors. To evaluate the proposed method, we applied it on several benchmarks and the results were compared with well known algorithms. The experimental results were satisfactory and in most cases the presented method had a better makespan with at least 10% less iterations compared to related works.

Keywords Genetic algorithm · Multiprocessor · Task scheduling · Makespan

1 Introduction

Scheduling problem in multiprocessor, parallel, and distributed systems is an NP-hard problem. The scheduling problem is employed to solve problems, such as information processing, whether forecasting, image processing, database systems, process control, economics, and operation research. The efficient communication and well-organized assignments of jobs to processors are important concerns in solving multiprocessor

M. R. Bonyadi · M. Ebrahimi Moghaddam (✉)
Electrical and Computer Engineering Department, Shahid Beheshti University G.C.,
Velanjak Ave, Tehran, Iran
e-mail: m_moghadam@sbu.ac.ir

M. R. Bonyadi
e-mail: m_bonyadi@std.sbu.ac.ir

task scheduling problems (MTSPs) [46]. Also, in this problem, the limited or undetermined processors should be assigned to some tasks such that minimum total execution time of all tasks is achieved.

The multiprocessor task scheduling problem is defined as follows:

There are *Nt* tasks and *Np* processors. The tasks are related to each other such that some tasks cannot start to process while its precedence is not accomplished. The process is preemptive (the process is not stopped while the task is processing). After a task is processed, its successor task may be processed but only after a predefined time (called communication cost). It is worth noting that each processor can process just one task simultaneously. The input of such a problem is usually considered as a directed acyclic graph (DAG) which provides precedence, dependency, and priority tasks together with communication cost among tasks and their precedence.

To solve this problem and achieve the minimum execution time, many heuristic and Metaheuristic methods have been presented. In all of these methods, the basic idea is to determine an order for tasks based on their execution priority. The approach of each method is different with regards to various aspects of input DAG [6,17,27,41]. After finding an optimal order, first task of the queue process is selected and an appropriate processor is allocated to it. Therefore, each heuristic approach is comprises two parts: firstly, finding an optimal order of tasks and secondly, allocating an appropriate processor to tasks.

As already mentioned, multiprocessor scheduling is an NP-hard problem [46]. Genetic algorithm (GA) has been investigated as a method to find an optimal solution for the NP-hard problems [11,32]. Thus, several methods have presented to solve this problem based on GAs [12,15,21,25,43].

In this paper, we present a novel and efficient genetic-based method to improve the results of related works in terms of makespan and number of convergence iterations that is called bipartite genetic algorithm (BGA). In BGA, the problem is split into two parts: the first part contains a population of processor numbers while the second one includes a population of task sequences. Each part is altered using a GA according to the other part. In fact, these parts cooperate with each other to solve the problem such that in each iteration, the best match chromosome for all elements in the first population is sought in the second population and vice versa. The proposed method was applied on several test benches and the results were satisfactory and it outperformed several related works.

The rest of paper is organized as follows: in Sect. 2, some of the previous studies on the multiprocessor task scheduling are presented. In Sect. 3, the proposed method is introduced and its various parts are defined. Section 4 consists of parameter setting process and the experimental results. Finally, we conclude this study in Sect. 5.

2 Related Works

There are several approaches to solve multiprocessor task scheduling, such as heuristic approaches [13,18,47], evolutionary or population-based approaches [12,14,15,21,34,42,43] and hybrid methods [3]. In this section, several presented methods in

the field of multiprocessor task scheduling from 1995 are reviewed. Each method is introduced and its preferences and shortcomings are debated briefly.

In [2], critical path fast duplication (CPFD) algorithm as a heuristic one has been presented. This algorithm is based on a classical technique in finding critical paths in DAG and it gives high priority to tasks which are placed in DAG critical path (critical path in DAG is the longest one between the first and one of the last nodes in terms of execution time and communication cost).

In [19], another method has been proposed which schedules the tasks on several processors, based on a greedy heuristic approach. The method consists of three phases: assignment, unrolling, and scheduling. The authors showed that their algorithm has better performance in comparison with an optimal branch and bound method.

The genetic-based methods have attracted a lot of researcher attention in solving the MTSP [12,14,15,21,34,42,43]. One of the main differences in these genetic approaches is in their chromosome (solution) representations; different representations of genomes and chromosomes have been discussed and studied with their prominences and shortcomings. Another important difference among genetic methods is in their genetic operators, such as crossover and mutation. Using different crossover and mutation methods for reproducing the offspring is strongly dependent upon the chromosome representation which may lead to the production of legal or illegal solutions. Another important point in designing a GA is simplicity of algorithm and complexity of evolutionary optimization process.

The first and most important work which has used GA for multiprocessor tasks scheduling is a classic work called Hou, Ansari and Ren (HAR) algorithm [12]. The height of tasks in input DAG is considered as the main feature in the proposed work. Each chromosome is composed of several strings. The number of strings corresponds to the number of processors and each string shows a schedule of some tasks based on the height of that task in DAG. The height concept is used to restrain the possible violation of precedence condition, i.e., the tasks are ordered according to their heights in each string. Based on this definition, the precedence condition is always satisfied and a simple crossover structure can be applied on the problem solutions (precedence-aware task schedules) to produce legal schedules. While this algorithm is very simple in terms of computational complexity, but it cannot ensure that the search space is global so that some feasible schedules are not reachable at all [49]. After HAR, in 1995 another GA-based method has been published [45]. The main difference between this study and the above-mentioned work is in their chromosome representation. The authors have been presented a GA using a matrix genome encoding to schedule distributed tasks. This algorithm provided better scheduling results than its previous works.

Another work that presents a new technique based on problem-space genetic algorithms (PSGAs) has been introduced in [7]. It utilizes the search power of GAs in combination with list scheduling heuristics, such as ISH [20] and DSH [20]. The results showed that the combinatorial could work efficiently and scheduled the tasks on several processors as well as the other methods. The methods presented in [1,3], also, tried to combine the heuristic and genetic approaches to solve this problem.

After the success of PSGA in 1996, in the same year, another genetic-based method which exploited some previous distributed techniques to develop a distributed GA was introduced. The main aim of this study has been to develop a distributed genetic

algorithm system (DGAS) that evaluates the performance of real time trial scheduling algorithms on a targeted system instead of simply simulating the performance [48].

In 1998, two other genetic-based methods have been carried out. In the first work [33], a comparison of genotype representations was prepared and genotype representation was divided to two different models and these models were compared in terms of respective quality of results and time of convergence. The proposed models were direct and indirect. In the direct representation, the schedule is represented and manipulated directly by the genetic operators. In this representation, the genotype is just like the phenotype. However, in indirect genotype representation, only the decision on how to build the schedule is encoded in the chromosome. In the second work, an evolutionary approach for multiprocessor scheduling of dependent tasks was stated [28]. The authors presented an approach for pre-runtime scheduling of periodic tasks on multiple processors in a hard real-time system.

Another genetic-based multiprocessor scheduling method in 1998 has been presented in [42]. The authors of this paper claimed that the task duplication is a useful technique for shortening the length of schedules. In addition, they added new genetic operators to the GA to control the degree of replication of tasks.

In [34], Corrga et al. tried to overcome the three drawbacks of the HAR which were presented in [12]. These drawbacks were: (1) the searches in the global solution space were weak, (2) load balancing between processors has not been observed in the case of initializing population, and (3) there was no knowledge about the quality of individuals, because in creating individuals only the validity of individuals had been observed and the quality of them had not been monitored. These authors have presented combined genetic list (CGL) algorithm that was a combined approach. In this method, some knowledge about the scheduling problem was introduced. This knowledge was represented by the use of list heuristic and improved the GA in the genetic operators. These improvements removed the mentioned drawbacks of the HAR algorithm to a large degree. The chromosome structure in this method was just like the structure in HAR. Nevertheless, CGL removed the HAR problems and it was a suitable algorithm in terms of solutions quality in spite of causing a heavy computational load in crossover and mutation operators. The prepared results demonstrated that the CGL method was more useful in generating better solutions in comparison with pure genetic or list heuristic approaches.

In 2001, the load balancing problem which has high importance in parallel and concurrent systems was thoroughly investigated in [50]. The load balancing includes partitioning a program into smaller tasks that can be executed concurrently and mapping each of these tasks to a computational resource, such as a processor. The authors used a GA to solve the dynamic load balancing problem. In this method, some parameters, such as threshold policies, information exchange criteria, and inter-processor communication and their effects on load balancing were considered. Another work that considers load balancing, memory locality, and scheduling overhead issues was published in 2000 [10].

With advances in genetic methods for scheduling the tasks on several processors, some works tried to change the conventional approach of GA. They combined other problem solving techniques, such as divide and conquer mechanism with GA. In 2003, a modified genetic approach called partitioned genetic algorithm (PGA) was proposed

[21]. In PGA: at first, the input DAG is divided into partial graphs using a b-level partitioning algorithm and each of these separate parts is solved individually using GA. After that, a conquer algorithm cascades the subgroups and forms the final solution. The authors claimed that the PGA leads to a better scheduling time in comparison with a chaste GA and a similar performance with common GA. Along the same lines, some other genetic-based works which were published in 2003 are [49] and [35]. In [49], a new GA called task execution order list (TEOL) was presented to solve the scheduling problem in parallel multiprocessor systems. The TEOL guarantees that all feasible search space is reachable with the same probability. Other heuristics can be combined with TEOL to improve its performance because TEOL is based on the restrain of the predecessor relationship of input DAG. This work tried to overcome the problems of two prior works that presented in [43] and [34]. Its prepared results showed that it could gain some improvements in shortening execution time toward [43], but it had similar execution times or in some cases weak results in comparison with [34]. Besides, it is worth mentioning that this work reached a noticeable reduction in time of solving scheduling problem in comparison with [34]. In the same manner, some other works tried to schedule partially ordered parts of DAG. Making and forming these parts is very intricate and some works, such as [5,39,43], and [4] have struggled to employ this technique in different ways. For example, an incremental approach was used in [43]. In this method, each solution or individual is shown using a set of cells and each cell consists of a pair (t, p) that p is corresponded with a processor and t is related to a task. Main contribution of this paper is presenting a novel, flexible and adaptive chromosome representation. Another important issue about this method is permission of duplicating tasks in an individual. Non-string representation of the solutions for scheduling problems together with genetic operators that were selected via a hybrid mechanism have been used in [5]. This partitioning technique can be very useful although it makes the scheduling problem more complicated.

In the fourth year of new millennium, some other new methods which used different ways for scheduling multiprocessor tasks using GA were proposed [24,26,29]. As a sample, the method presented in [29], sets up a GA approach for scheduling problem in a multilayer multiprocessor environment. In multilayer multiprocessor task scheduling, it is required that tasks or jobs go through more than one stage where each stage has several parallel processors. This kind of multiprocessor task scheduling is widely used specially in industrial and computing applications. Also, in this research, a new crossover operator has been introduced.

In 2005, a new dynamic task scheduling using GAs for heterogeneous distributed computing was proposed [30]. This dynamic scheduler used GA for finding a minimum execution time. The term "dynamic" is used since the algorithm can work in an environment with dynamically changing resources and adapts to variable system resources.

In [16], a novel representation called fixed task, alternative processors (FTAP) was proposed to schedule the tasks. In this work, the length of chromosomes is intelligently adapted according to the given problem. The main focus of this paper is on the effect of solution representation on solving problems and its speed and accuracy and universality through the problem space.

One of the recent works that was performed in 2007 on multiprocessor task scheduling using GA is a study that employs MTSP in dynamically reconfigurable hardware [31]. In this work, a pair of two dimensional strings is used in representation of chromosome. In this study, it is shown that the GA is more efficient than list scheduling algorithm for scheduling tasks onto a dynamically reconfigurable hardware.

Many GAs have been applied for scheduling multiprocessor tasks but just a few of them considered the communication cost so far [14]. One of the most recent and important works on the multiprocessor task scheduling using GA was presented in [14] wherein the authors proposed the extension of the priority-based coding method as the priority-based multi-chromosome (PMC). In this method, each gene in each chromosome was a number in the interval $[1, p \times t]$ where p was the number of processors and t was the number of tasks and there was no duplication in each chromosome. In this presentation, each gene index represented the task number. Hence, a new crossover method compatible with this new encoding was proposed which was a permutation based crossover and was called weight mapping crossover (WMX). The priority-based encoding is the knowledge of how to handle the problem of producing encoding that can treat the precedence constraints efficiently [49]. As far as it has been studied and based on a vast investigation on GA methods for multiprocessor scheduling, the PMC method is one of the best works that has ever been performed in terms of its simple chromosome structure and the suitable and intelligible design of GA operators. Their designed chromosome structure could simply convey all of the required information for scheduling in only one dimension and just using an array structure. The specific proposed chromosome structure and crossover has resulted in valid solution production and also a reduced algorithm time. In this respect, we have used PMC method for comparing our approach results and validating our proposed method.

Also, with increasing interests for solving the multiprocessor task scheduling, the other search methods and techniques have been applied to solve multiprocessor scheduling problem. Some of these approaches use a combinatorial algorithms and some of them try to exert new optimization algorithms, such as ACO, PSO, AIS, mean field annealing and memetic algorithm [5,9,22,36–39].

To improve the results of related work in case of makespan and number of convergence iterations; in this paper, we proposed a genetic based method to solve multi processor task scheduling that is called BGA. The proposed method consists of two parts; each part is based on independent genetic schemes. The experimental results showed that BGA improved the makespan of several standard benchmarks with regard to related works. In the next section, BGA is introduced and its parameters are defined.

3 Proposed Method

In this section, the proposed method for multiprocessor task scheduling is presented. The proposed method consists of two parts and is called BGA. The first part finds the best valid sequence of tasks independently of processors and is called GAS (GA for task sequences). The second part finds the corresponding processors by improving the best fitness value and is called *GAP* (GA for processors). In each generation of the

BGA, the *GAS* and *GAP* are run for predefined number of generations. The *GAS* and *GAP* are described in the following sections.

Since the proposed method consists of several parameters, all parameter acronyms are defined again in "Appendix".

3.1 The GAS: A GA to Find the Best Match Task Sequences

First, we introduce *PopS* and *PopP* that are two important parameters in *GAS*. *PopS* is a population which contains predefined number of tasks sequences while *PopP* is a population which contains predefined number of processor arrays. The length of each array in *PopP* is equal to the length of each task sequence in *PopS*. *GAS* finds a set (population) of task sequences which has the maximum compatibility with the available processor arrays in the *PopP*. In each generation of *GAS*, the GA operators are applied on *PopS*. Each task sequence in *PopS* is combined with all processor arrays in current *PopP* and produces several pairs. Then, for each pair, the makespan is computed and the processor array that minimizes the makespan of the pair is used as the best match for that sequence. After finding the best match for all sequences, the GA operators (crossover and mutation) are applied on *PopS* to improve the makespan of population. In other words, in each generation of *GAS*, the attempt is on evolving *PopS* in a way that the makespan of the best match pairs is improved. Whenever *GAS* is running, *PopP* is not changed and the reproduced sequences are matched with the current *PopP*.

In *GAS*, the representation of task sequences in *PopS* and the fitness value for each sequence are very consequential. In addition, adequate reproduction operators are necessary to attain the best performance and find better solutions in less iteration. The proposed coding scheme, fitness function and reproduction operators for *GAS* are presented as follows.

3.1.1 Coding Scheme for GAS

The proposed coding scheme (representation of chromosomes) for each task sequence is a permutation of task numbers. As an instance, Fig. 1b shows a corresponding chromosome for a nine tasks problem in Fig. 1a. The chromosome in Fig. 1b shows an order for executing the tasks independent from the processors. As it was mentioned, each chromosome in *PopS* contains a permutation of the tasks; each of them is processed (to schedule on processors) according to their appearance. Therefore, each task in the chromosome should appear before all of its children and after all of its parents. Thus, some permutations of the tasks may not be admissible and the validation state of the chromosomes should be attended. Hence, a validation phase after producing the chromosomes is considered. The validation procedure changes the order of the tasks in a chromosome to make it valid. It is clear that if a chromosome is valid by itself, the validation procedure does not change it. The validation procedure is introduced in Algorithm 3.

It is noticeable that the chromosomes in the initialization phase are generated by a process which always produces a legal chromosome. In other words, the *PopS* is initialized by a random process but in such a way just to produce valid sequences.

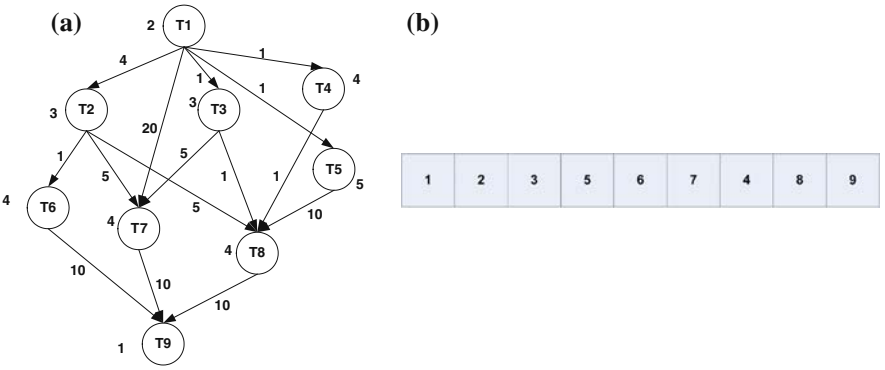

Fig. 1 **a** A sample problem with 9 tasks that is shown by DAG. **b** Corresponding chromosome with (**a**) in *PopS*; each task in the chromosome should be appeared before all of its children and after all of its parents

To attain this objective i.e., randomly producing valid chromosomes, the following algorithm is proposed.

Algorithm 1 (Initialize *PopS*)

Input: PsS (Population Size), DAG (Problem Task relations), output: PopS
*1) Make PopS as an empty matrix with the size PsS*Nt*
2) Make Temp as an empty, temporary array that its size is equal to the number of tasks in DAG
3) Temp ← all tasks in DAG which can be scheduled now (here, just the root tasks)
4) For i = 1 to PsS
a. For j = 1 to Nt
i. R ← a random number between 1 and the current length of Temp
ii. PopS (i, j) ← the R^{th} index of Temp
iii. Discard the R^{th} index of Temp
iv. Temp ← Temp ∪ all tasks in DAG which can be scheduled now
b. End for
5) End for
6) Return PopS

In this algorithm, *PsS* shows the expected population size, Nt stands for number of tasks, and DAG shows the problem tasks relations that are considered as algorithm inputs. Using this algorithm, the *PopS* is initialized randomly and all chromosomes are valid. Hence, no validation process is needed after initializing the *PopS*.

3.1.2 Fitness Values for GAS

The fitness values for the sequences in *PopS* are computed according to the processor arrays in *PopP*. First, a processor array in current *PopP* is found for each sequence in *PopS* (a pair of sequence and its corresponding processors) which minimizes the makespan. Then, this makespan is defined as the fitness value for the sequence. The following pseudo code realizes this procedure:

Algorithm 2 (*GAS* Fitness)

Input: PopS, Pop; Output: Fitness values for PopS
1) For each sequence S in PopS
a. For each processor array P in PopP
i. Compute the makespan of the pair (S, P) (called $M_{S,P}$)
ii. If the $M_{S,P}$ is the best found makespan for S so far, update the fitness of this sequence (S) as the $M_{S,P}$
b. End for
2) End for
3) Return all found fitness values

3.1.3 Reproduction Operators for GAS

The reproduction operators consist of crossover and mutation which are proposed to use in *GAS* for producing new offspring. As it was pointed out earlier, the *GAS* employs a permutation-based chromosome model. This induces us to use the operators which are compatible with this chromosome structure. In this paper, two crossover operators are used: weighted mapped crossover (WMX) [14] and "Order base" [8]. Also, three mutation models are used (i.e. swap, scramble, and inversion). The results show that the "Order based" crossover in combination with "Swap" mutation has the best performance in comparison with other combinations. The used crossover and mutation operators are presented as follows:

3.1.3.1 Crossover Operators

1. WMX: in this crossover model, first, two cutting points are randomly selected as a substring and then the values of these substrings are checked for contents and order. The order of the substring that is opted from the first parent is used to reorder the corresponding substring in second parent and vice versa [14].
2. Order-based crossover: in this crossover model, two cutting points are selected in first parent and the substring between these points is copied in the first offspring. Then, starting from the second crossover point in the second parent, the remaining unused numbers in the first offspring are copied according to the order that appears in the second parent, wrapping around at the end of the list. The creating process for the second child is in an analogous manner with the parents' role being reversed [8].

3.1.3.2 Mutation Operators

1. Swap: in this mutation, two genes are selected and their contents are replaced [8].
2. Scramble: this mutation model uses two randomly selected points in each chromosome and the content of this substring is randomly arranged and replaced by the old substring [8].

3. Inversion: in this mutation model, two points are selected in each individual and the substring between these two points are reversed and replaced by the old substring [8].

The mutation and crossover operators in *GAS* are applied with the probability *PmS* and *PcS*, respectively. Because of the precedence constraints in multiprocessor task scheduling and the permutation-based coding scheme for tasks, some of the offspring which are produced via the mentioned crossovers and mutations may be invalid. Therefore, a validation phase is needed to construct valid sequences according to the produced chromosomes. In fact, in each iteration of the *GAS*, each chromosome is validated via a validation procedure. In the validation phase, first, each chromosome in *PopS* is processed to find its best matched array processor in *PopP* and to produce an array of pairs. Then, if an element of this array cannot be scheduled in its place (the parent of the task is not scheduled yet), it is placed in a queue. In each iteration of the validation phase, this queue is processed to find the pairs which can be scheduled. The remaining pairs in the queue are scheduled in their appearance order in the queue after scheduling all the elements of the array. The following algorithm shows the validation procedure.

Algorithm 3 (Validation)

Input: an array of pairs P (a sequence of tasks and its best matched array processor); Output: validated form of P (called RP)

1) Set the Q as an empty queue and RP as an empty array
2) For all elements P_i in P
  a. If P_i can be scheduled (all parents of the i^{th} task in P are scheduled) then
    i. $RP \leftarrow RP \cup P_i$
  b. Else
    i. Place P_i in Q
  c. End if
  d. For all elements Q_i in Q
    i. If Q_i can be scheduled (all parents of the i^{th} task in Q are scheduled) then
      1. $RP \leftarrow RP \cup Q_i$
      2. Remove Q_i from Q
    ii. End if
  e. End For
3) End For
4) While Q is not empty
  a. For all elements Q_i in Q
    i. If Q_i can be scheduled (all parents of the i^{th} task in Q are scheduled) then
      1. $RP \leftarrow RP \cup Q_i$
      2. Remove Q_i form Q
    ii. End if
  b. End For

5) End while
6) Return the tasks in RP

The "while" loop in line 4 of the algorithm is necessary because after the "for" loop, it is probable that some tasks remain in Q and have not been scheduled yet. In this procedure, RP contains the validate chromosomes and Q is a temporary queue that is used in procedure.

3.1.4 The Step by Step Procedure of GAS

Underneath, *GAS* is presented step by step:

Algorithm 4 (GAS)

Inputs: PopS, PopP, GNS (maximum number of generations) and SGNS (number of successive iteration which algorithm is not improved); Output: PopS, NSS (number of iteration which GAS is run)
1) Initialize the parameters PmS, PcS
2) NSS=0
3) While stopping criteria are not satisfied
a. Calculate the fitness values for PopS (Algorithm 2)
b. Perform crossover with the probability PcS according to Sect. 3.1.3 on PopS
c. Perform mutation with the probability PmS according to Sect. 3.1.3 on PopS
d. Validate the offspring (Algorithm 3)
e. NSS = NSS + 1
4) End for
5) Return the PopS and NSS

The stopping criteria for *GAS* are the maximum number of generations (*GNS*) and number of successive iteration that the algorithm not make any improvement thereof (S*GNS*). In this algorithm, the *NSS* is the number of iterations which *GAS* is run. Other parameters have been defined in last section (also "Appendix").

3.2 The *GAP*: A GA for Finding the Best Match Processor Arrays

The *GAP* is employed to evolve the *PopP* to generate the processor arrays which are more compatible with the sequences in *PopS*. In fact, in each generation of *GAP*, the new offspring are produced via the GA operators. The *PopP* is combined with *PopS* in each generation and best pair matches are found. Then the fitness values for processor arrays in *PopP* are computed using the makespan for the best match pairs. The coding scheme, fitness function, and reproduction operators are introduced as follows.

3.2.1 Coding Scheme for GAP

Having in mind that each element in *PopS* contains *Nt* (*Nt* is number of tasks) task, the coding scheme for the chromosomes *GAP* is an array which contains *Nt* cells and the value of each cell is between one and *Np* (*Np* is the number of processors). Each cell value shows the processor number for the corresponding task to run on. Figure 2 shows an instance for this coding. This chromosome is the best match for the chromosome in Fig. 1b. The pair array of this array processor and the chromosome in Fig. 1b is presented in Fig. 3.

As an example, pair (3, 1) in Fig. 3 indicates that the task 3 has to be executed in processor 1 after executing the first task in processor 2 (with regard to Fig. 1a). The communication cost should be considered to calculate the makespan. Figure 4 shows the Gantt chart for this pair array where its makespan is equal to 21.

3.2.2 Fitness Value for GAP

The fitness values for the sequences in *PopP* are computed according to the current sequences in *PopS*. First, we try to find a sequence in current *PopS* for each chromosome in *PopP* (a pair of sequence and its corresponding processors) which minimizes the makespan. Then we define this makespan as the fitness value for the processor array. The following pseudo code shows this procedure:

2	2	1	2	1	2	1	2	2

Fig. 2 An instance of chromosomes in *PopP*. In this example, it is considered that there are two processors to process the tasks of Fig. 1a. This figure shows the chromosome that is matched with chromosome in Fig. 1b

(1, 2)	(2, 2)	(3, 1)	(5, 2)	(6, 1)	(7, 2)	(4, 1)	(8, 2)	(9, 2)

Fig. 3 The pair array of the chromosomes in Figs. 1b and 2. This pair is used to find the makespan and fitness for these chromosomes

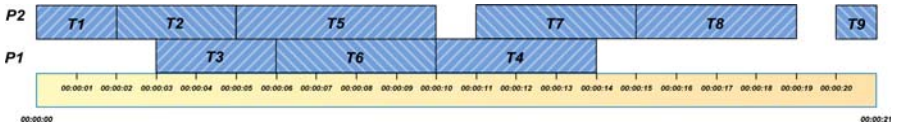

Fig. 4 The scheduling results for the problem in Fig. 1a. The results have been obtained by using BGA in 100 iterations when there were two processors. This figure corresponds with Fig. 3

Algorithm 5 (*GAP* Fitness)

Input: PopS, PopP; Output: Fitness values for PopP
1) For each sequence P in PopP
a. For each processor array S in PopS
i. Compute the makespan of the pair (S, P) (called $M_{S,P}$)
ii. If $M_{S,P}$ is the best found makespan for S so far, update the fitness of this processor array (P) as the $M_{S,P}$
b. End for
2) End for
3) Return all found fitness values

In this procedure, $M_{S,P}$ is the best found makespan of each pair (S, P) where S is a chromosome in *PopS* and P is a chromosome in *PopP*. The definitions of other parameters are same as last sections (see "Appendix").

3.2.3 Reproduction Operators for GAP

According to the simple coding scheme for *GAP*, a simple crossover and mutation can efficiently work to improve the *PopP*. Here, the "One Point" crossover and the "Uniform" mutation have been used. In one-point crossover, two parents are selected via a selection procedure and one cutting point is chosen via a random process. Then the second part of the parents is replaced and two offspring are produced. The crossover operator is employed with the probability *PcP*. In the uniform mutation, each chromosome is mutated with the probability Pm (here *PmP*). In this process, one gene of the selected chromosome changes to a value that is selected randomly from the problem space.

3.2.4 The Step by Step Procedure of GAP

The following algorithm presents the *GAP*

Algorithm 6 (*GAP*)

Input: PopS, PopP, GNP (max number of generations) and SGNP (number of successive generations that algorithm is not improved), output: PopP, NSP (Number of iterations that GAS is run)
1) Initialize the parameters PmP, PcP.
2) NSP = 0;
3) While stopping criteria are not satisfied
a. Calculate the fitness values for PopP(Algorithm 5)
b. Perform crossover with the probability PcP according to Sect. 3.2.3 on PopP
c. Perform mutation with the probability PmP according to Sect. 3.2.3 on PopP
d. NSP = NSP + 1
4) End for
5) Return the PopP and NSP

The stopping criteria for *GAP* are the maximum number of generations (*GNP*) and number of successive iteration that the algorithm does not make any improvement thereof (*SGNP*). *PcP* and *PmP* show the probability of crossover and mutation, respectively. In this algorithm, the *NSP* is the number of iterations which *GAS* is run. It is worthwhile to note that the population of processors (*PopP*: population of processors) is initialized via a random process. Here, no validation process is needed. The definitions of other parameters are same as last sections (see "Appendix" also).

3.3 The Proposed BGA

The following pseudo code shows the proposed BGA.

Algorithm 7 (BGA)

Input: DAG, output: best order of tasks and their corresponding processors with minimum makespan

1) Initialize the parameters PmP, PmS, PcP, PcS, PsP, PsS, GNP, GNS, SGNS, SGNP, IT, SIT
2) Initialize PopS population by Algorithm 1.
3) Initialize PopP population randomly.
4) NS = 0
5) While stopping criteria are not emerged (NS < IT and the algorithm is improved for SIT last iterations)
  a. [PopS NSS] = GAS(PopS, PopP, GNS, SGNS)
  b. NS = NS + NSS
  c. [PopP NSP] = GAS(PopS, PopP, GNP, SGNP)
  d. NS = NS + NSP
6) End while
7) Return the best task sequence in PopS with its corresponding processor array in PopP as the best solution

In this algorithm, *Pm* is the mutation rate, *Pc* the crossover *Ps* the population size, *GN* the maximum number of generations and *SGN* the maximum number of successive generations that the algorithm (*GAS* or *GAP*) is not improved in. In all of these parameters, the postfix term P indicates that the parameter is relevant to the *GAP* and the postfix term *S* shows that the parameter is relevant to *GAS*. As an example, *PmP* means and *PmS* the same rate in *GAS*. *NS* is a counter that shows number of iterations in algorithm. The definitions of other parameters are same as last section and they are introduced in "Appendix". The algorithm goes on while predefined criteria do not emerge. In the implementation, we use two criteria to stop the algorithm:

1. The maximum number of iterations (*IT*)
2. Number of successive iterations which the fitness value is not improved in (*SIT*)

The first condition will emerge when *NS* is bigger than or equal to *IT*. The second condition emerges when the algorithm could not find better solutions in the last predefined iterations (*SIT*). In fact, if *NS* is bigger than *IT* or the algorithm is not improved for the last *SIT* iterations, the algorithm stops.

4 Experiments and Results

4.1 Experiment Environment

In this part, the results of experiments are presented. The proposed algorithm has been implemented in Matlab environment and the simulations were performed on a personal computer with 512 MB of RAM and 1.8 GHz AMD CPU. First, a sample graph was used to set the parameters of the proposed BGA. The parameters *PmS*, *PcS*, crossover and mutation models were adjusted via some experiments. To evaluate BGA and compare its results with related works, standard task graph (STG) test bench [40] was used. The communication costs are not included in STG so they were produced via a random process. The resulted communication costs are available online in [51]. In addition, task scheduling problems that are shown in Table 1 were used in evaluation and comparison. In Table 1, each problem has been marked as *P#*. Also, the number of tasks (no. of tasks) for each one, and the communication costs have been given. The communication costs for P5, P6 and P7 have been declared in [20,23] and [44], respectively, as well. Except for P (5–7), each problem corresponds to a well known mathematical problem described in the last column of the Table 1. The problems listed in Table 1 usually were used in the literature to evaluate multiprocessor task scheduling methods [43]. The problems presented in Figs. 1a and 7 were other problems that have been used to evaluate the methods. Each method was run 10 times and min, max and average of makespan together with number of generations were compared.

4.2 Parameter Setting

The problem P7 in Table 1 is used to find the best parameters values of the BGA. Table 2 illustrates the results of running BGA to solve problem P7. The crossover and mutation model for *GAS* are mentioned in Table 2. The parameters were initialized by the following values:

Table 1 Some task scheduling problems that were used to evaluate the proposed method

Problem	No. of tasks	Communication costs	Source	Description
P1	15	25 (fixed)	Tsuchiya et al. [42]	Gauss-Jordan algorithm
P2	15	100 (fixed)	Tsuchiya et al. [42]	Gauss-Jordan algorithm
P3	14	20 (fixed)	Tsuchiya et al. [42]	LU decomposition
P4	14	80 (fixed)	Tsuchiya et al. [42]	LU decomposition
P5	15	Variable for each graph edge	Kruatrachue and Lewis [20]	–
P6	17	Variable for each graph edge	Mouhamed [23]	–
P7	18	Variable for each graph edge	Wu and Gajski [44]	–
P8	16	40 (fixed)	Wu and Gajski [44]	Laplace
P9	16	160 (fixed)	Wu and Gajski [44]	Laplace

The communication cost of each problem has been presented except for P5–P7. The communication costs are fixed in some ones and variable for others. The source column shows the reference of the problem and the description column shows the corresponding well known problem

- *GNS* and *GNP*: 20
- *SGNS* and *SGNP*: 10
- *PsS* and *PsP*: 20
- *PmS* and *PmP*: 0.2
- *PcS* and *PcP*: 0.8
- Maximum iteration for BGA (*IT*): 1000
- Number of successive iterations without improvement to stop the algorithm (*SIT*): 100

The used crossover for *GAP* was "one point" and the used mutation was "uniform". Table 2 indicates that the algorithm finds the best solutions when the crossover was order-based and the mutation was swap. Hence, in all implementations, we use these operators for *GAS*. In addition, to find the best values for crossover and mutation rates, the *PcS* and *PmS* have been iteratively. The value of *PcS* was changed in the interval [0.5, 0.9] with the step size 0.1. Figure 5 shows the results of variations versus makespan average that is average of 10 runs of the BGA to solve P7. The value of other parameters was as follows:

- *GNS* and *GNP*: 20
- *SGNS* and *SGNP*: 10
- *PsS* and *PsP*: 20
- *PmS* and *PmP*: 0.2

Table 2 The results of applying BGA on P7 using several models of mutation and crossovers in *GAS* part

Cross over	Mutation	Average number of iterations	Average of makespan
Order based	Swap	212	402
	Inversion	239	408
	Scramble	228	408
WMX	Swap	299	412
	Inversion	225	404
	Scramble	218	406

The results are average of running BGA 10 times. The makespan values are reported in time unit (e.g. second)

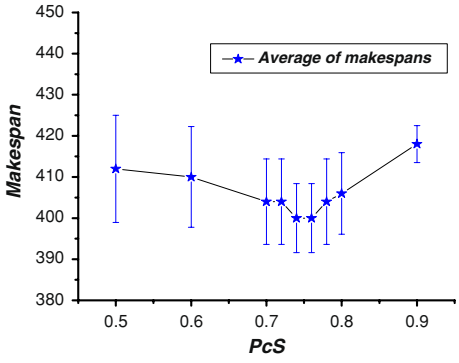

Fig. 5 The average makespan values that has been obtained by 10 times applying BGA on P7 versus variation of *PcS* in interval [0.5, 0.9]. The best result obtained when *PcS* was equal to 0.74

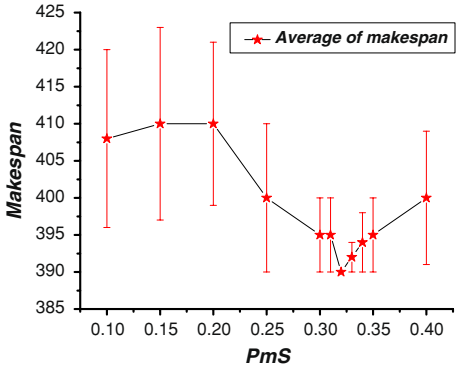

Fig. 6 The average makespan values that has been obtained by 10 times applying BGA on P7 versus variation of *PmS* in the interval [0.1, 0.4]. The best result obtained when *PmS* was equal to 0.32

- *PcP*: 0.8
- Maximum iteration for BGA (*IT*): 1000
- Number of successive iterations without improvement to stop the algorithm (*SIT*): 100

The used crossover and mutation for *GAS* was "order based" and "swap", respectively, while the "one point" and "uniform" was used for *GAP*. Figure 5 shows that the best performance of the BGA emerges when the value of *PcS* is 0.74. Since the interval [0.7, 0.8] was a critical range (the makespan of *PcS* 0.7 and the makespan of *PcS* 0.8 are close to each other), the *PcS* in this range has been changed with the step size 0.02. The best performance was achieved when the *PcS* was set to 0.74. Figure 5 shows the STD of average makespan also. Moreover, to determine the best value for *PmS*, a similar performed. In this experiment, the value of *PmS* was changed in the interval [0.1, 0.4]. Figure 6 shows the makespan average in 10 times running of BGA to solve P7 problem vs. *PmS* in the interval [0.1, 0.4] with the step size 0.05. Again, because the interval [0.3, 0.35] is a critical interval (the average for *PmS* = 0.3 and *PmS* = 0.35 are close), the step size in this interval was investigated in more details (0.01). Figure 6 also shows the STD (standard deviation) of makespan.

It is obvious in Fig. 6 that the best performance of the algorithm emerges when *PmS* is 0.32. A similar parameter setting is performed for *GAP* parameters (*PmP* and *PcP*). The experiments show that the best outcome is obtained when *PcP* is chosen as 0.8 and *PmP* was set as 0.2. Hence, in all experiments, the following parameters were used in BGA implementation:

- *PsS* and *PsP*: 20
- *GNS* and *GNP*: $2 \times PsS$ and $2 \times PsP$
- *SGNS* and *SGNP*: *GNS*/2 and *GNP*/2
- *PmS*: 0.32, *PmP*: 0.2
- *PcS*: 0.74, *PcP*: 0.8
- Maximum iteration for BGA (*IT*): $100 \times \max(PsS, PsP)$
- Number of successive iterations without improvement to stop the algorithm (*SIT*): $10 \times \max(PsS, PsP)$

The literature with which the BGA was compared, utilized 400 chromosomes in their populations. Hence, the *PsS* and *PsP* are chosen as 20 because in this case, we

can say that we have 400 chromosomes. About the values of *GNS* and *GNP*, we can say that, if the values for *GNS* and *GNP* are chosen as big numbers *IT*) and the value of *IT* are considered as a constant value, the improvement will slow down because the evolution process (*GAS* and *GAP*) is performed in a big number of generations on just one population (*PopS* or *PopP*). Thus, that population will be greatly compatible with the current other (*PopP* or *PopS*). We call this phenomenon as "over compatibility" which is something like "overtraining" phenomenon occurring in neural networks. Furthermore, if these values (*GNS* and *GNP*) are chosen as small values (according to *PsS* and *PsP*), the *GAS* and *GAP* will not have enough time to exploit the it can be. Hence, we use factor 2 to ensure that the *GNS* and *GNP* are small enough according to *IT* (about 1/50) and big enough according to *Ps* (twice), at the same time. A similar problem has been considered for the values of *SGNS* and *SGNP*.

4.3 Comparisons with Well Known Heuristics

In this section, the BGA is compared with some well known heuristics, such as modified critical path (MCP) [44], dominant sequence clustering (DSC) [46], mobility directed (MD) [44], dynamic critical path (DCP) [47], insertion scheduling heuristic (ISH) [20], duplication scheduling heuristic (DSH) [20], and CPFD [2]. Table 3 demonstrates the makespan of problem presented in Fig. 1a when it is solved by BGA, MCP, DSC, and MD. The results of BGA in all cases are better than the results achieved by mentioned heuristics. The BGA was run 10 times in each case (2, 3 and 4 processors) and the best makespan of each case has been reported in Table 3. *PsP* and *PsS* in this test are selected 10. The best results in Table 3 are shown by font. As it is shown in Table 3, the results of BGA is outperformed other methods.

Table 4 shows the results of the comparisons between the BGA and the ISH, DSH and CPFD. The problems of Table 1 were used for this experiment. In this case, the best makespan in 10 runs of the algorithm has been reported. As it is shown in Table 4, the BGA is always better than ISH algorithm, but its results are often worse than CPFD and DSH algorithms. The best result in each row is shown by Bold-Italic-Underline font.

4.4 Comparisons with GA Based Algorithms

In order to compare BGA with other GA-based methods, two such methods have been considered that their results were better than any other ones (i.e., incremental GA [43]

Table 3 Results of applying BGA and some well known heuristics on the problem in Fig. 1a

Algorithms	MCP	DSC	MD	DCP	BGA		
No. of processors	3	4	2	2	2	3	4
Best solution	29	27	32	32	***21***	***21***	***21***

The BGA could find less makespan. The values in row "Best solution" show the best makespan of problem achieved in 10 separate runs of algorithms. The makespan values are reported in time unit (e.g. second)

Table 4 The comparison results between the BGA and some heuristics in terms of the best gained makespan to solve the problems presented in Table 1

Problem	Algorithm	ISH	DSH	CPFD	BGA
P1	Best solution	300	300	300	300
P2	Best solution	500	***400***	***400***	420
P3	Best solution	***260***	***260***	***260***	270
P4	Best solution	400	**330**	**330**	360
P5	Best solution	650	***359***	446	440
P6	Best solution	41	***37***	***37***	***37***
P7	Best solution	450	370	***330***	390
P8	Best solution	760	***670***	760	790
P9	Best solution	1220	***1030***	1040	1088

Indeed, the BGA was run 10 times and the best result is mentioned. The makespan values are reported in time unit (e.g. second)

Table 5 Comparison of PMC, incremental GA, and BGA based on structural aspects

	BGA	Incremental GA [43]	PMC [14]
Needed parts for chromosomes	2	2	1
Is all problem space searched?	Yes	Yes	Yes
Search space size	$Nt! \times Np^{Nt}$	$Nt! \times Np^{Nt}$	P($Nt \times Np$, Nt) where P is permutation
Is any specific type of GA operators needed?	Yes	No	Yes
Feasibility after GA operators	Needs validation	Sometimes not feasible	Always feasible
How treat invalid chromosomes?	Validate them	Discard them	Always valid

and PMC [14]). Table 5 compares the structure of BGA with these methods. Also, Table 5 consists of problem space, feasibility of chromosomes, and the comparisons among operators.

As shown in Table 5, both BGA (the proposed method) and incremental GA [43] employ two-part chromosomes (incremental GA utilizes a pair for each gene) to present and code the problem. In contrast, the PMC utilizes a one-dimensional chromosome but its search space is larger.

The search space that is sought by incremental GA and BGA is very close to the actual problem space. Moreover, the BGA needs permutation-based GA operators in *GAS* part while *GAP* part of BGA exploits simple GA operators. Hence, a validation phase is needed in BGA to validate the task sequences where afterward all chromosomes become valid. On the other hand, the PMC exerts a specific crossover that sustains the feasibility of chromosome structures. However, in the incremental GA method, some chromosomes might have an incompatible length and they cannot be decoded. In this case, the crossover does not apply and the original chromosomes are used. Also, the precedence constraints are not considered in designing the operators in incremental GA and the method employs a penalty function in calculation of fitness values.

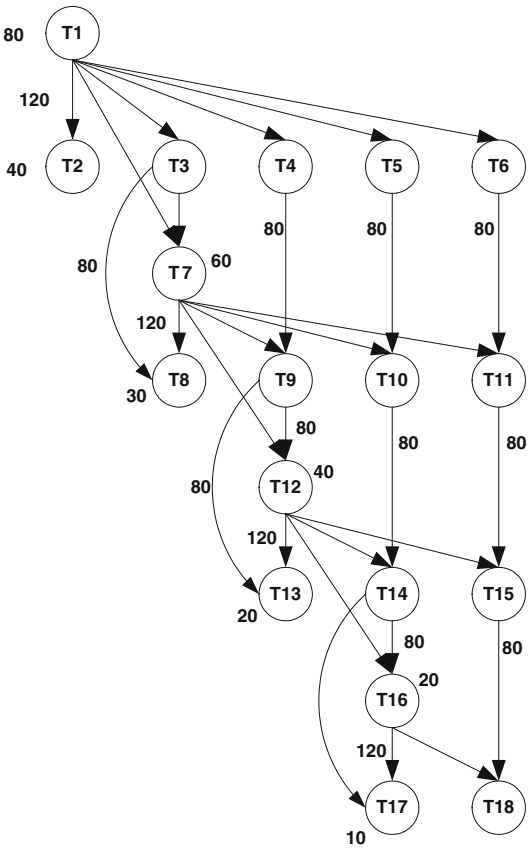

Fig. 7 A task scheduling problem with 18 tasks which was introduced in [14]

To compare the results of BGA with the mentioned related methods, three test benches were employed. At first, the BGA and PMC were applied on two problems shown in Figs. 1a and 7 and their results were compared in terms of the STD and the mean of obtained makespan. Then these methods were applied on some test cases of STG [40]. Next, these methods (BGA, incremental GA, and PMC) were applied on test benches which have been presented in Table 1 and the results were compared.

Table 6 shows the results of applying BGA and PMC on the problem in Fig. 1a. It is worth mentioning that the results are the outcomes of running the algorithms 10 times independently. As it is shown in Table 6, BGA has better results in all cases; for example, the achieved makespan with three processors in mean case is 21 by BGA compared to 22.4 by PMC which shows 6% improvement. Also, the zero value of STD shows that BGA has a robust behavior in solving this problem. The best results in each row are shown by Bold-Italic-Underline font.

In this experiment, the parameters of PMC were set as follow:

- Population size: 400
- *Pc*: 0.7
- *Pm*: 0.3

Table 6 The comparison results between the BGA and PMC for the problem in Fig. 1a

No. of processors	BGA				PMC [14]			
	Best	Worst	Mean	Std	Best	Worst	Mean	Std
2	***21***	***21***	***21***	***0***	21	23	21.9	0.56
3	***21***	***21***	***21***	***0***	21	23	22.4	0.69
4	***21***	***21***	***21***	***0***	21	23	22.3	0.82

In this case, *PsP* and *PsS* were equal to 10. The values show the best, worst and mean makespan in 10 times running algorithms to solve the problem. The Std column shows standard deviation of makespan results in different running of algorithms. The makespan values are reported in time unit (e.g. second)

Table 7 The makespans obtained by applying PMC and BGA on the problem in Fig. 7

No. of processors	BGA				PMC [14]			
	Best	Worst	Mean	Std	Best	Worst	Mean	Std
2	***460***	***470***	***463***	***4.83***	460	520	491	20.78
3	***440***	***490***	***461***	***12.86***	490	600	522	35.21
4	***440***	***470***	***461***	***8.75***	500	580	544	25.90
6	***460***	***490***	***471***	***11.00***	510	580	556	19.55

The Best, Worst, Mean, and Std (standard deviation) values have been calculated over 10 times running the algorithms. The problem solved with different number of processors. The makespan values are reported in time unit (e.g. second)

- Maximum iterations: 2,000
- Number of iterations without improvement: 200

As it is shown in Table 6, the BGA found the best known optimum solution in all runs and surpassed the PMC in terms of the obtained mean value for the makespan of the problem.

In addition, BGA was compared with PMC by applying both algorithms on problem that has been shown in Fig. 7. Table 7 shows the results of applying BGA and PMC on the problem presented in Fig. 7 using several numbers of processors. In this case, each algorithm was run 10 times and the results were reported. The BGA could find better makespan in comparison to PMC in terms of average, worst and best solutions in all cases, for example, the achieved makespan with four processors in mean case is 461 by BGA compared to 544 by PMC which shows 15% improvement. Also, the values of STDs for BGA are quite smaller than the STDs for PMC which reveals the stability of BGA.

Table 8 shows the results of the BGA and PMC while both of them have been applied on some test cases of STG. Each algorithm was run 10 times and the average values of makespans have been reported. It shows that the BGA could find better makespan in comparison to PMC in all cases, for example, the average improvement to PMC when $Nt = 50$ is about 17.8%. Also, the STD of the the solutions for BGA is smaller than the PMC in most cases. This shows that the BGA is more stable in comparison to the PMC. The last row of Table 8 shows the results of applying the algorithms (BGA and PMC) on

Table 8 The results of applying BGA and PMC on some test cases of STG in 10 times running

Nt	Problem name	*Np*	BGA		PMC [14]		BGA improvement with regard to PMC (%)*
			Makespan average	STD	Makespan average	STD	
50	Rand0002	4	***105***	6.1	128.9	***5***	18.6
		8	***104.2***	***1.5***	136.4	4.44	23.6
	Rand0016	4	***166.6***	***9.2***	210.8	12	21.3
		8	***161***	***4.3***	189.4	11.71	14.99
	Rand0019	4	***167.2***	***8***	221.1	12	24.4
		8	***176.46***	7.5	203.4	***32***	13.24
	Rand0028	4	***207.2***	7.3	256.4	***6***	19.1
		8	***206.6***	***7.9***	239.2	8.78	13.6
	Rand0043	4	***77***	***2.1***	94.6	7	18.6
		8	***71.4***	2.9	80.1	***1.64***	10.8
Average of improvement in percent							17.8
100	Rand0002	4	***223.4***	***12.1***	286.6	14.55	22
		8	***215.8***	***7.12***	262.4	13.86	17.76
	Rand0016	4	***184.2***	***7.4***	236.8	11.6	22.3
		8	***154.4***	***6.65***	199.6	8.61	22.65
	Rand0019	4	***296.2***	***3.34***	369.4	19.44	19.8
		8	***224.6***	***1.5***	274.8	12.49	18.2
	Rand0028	4	***140.6***	***1.51***	165	10.66	15.2
		8	***89***	***0***	108	7	17.6
	Rand0043	4	***247***	***2.73***	271.2	12.39	9
		8	***134.2***	***3.08***	159.4	3.57	15.8
Average of improvement in percent							18.3
300	Rand0002	4	***760***	***4.2***	955	22.2	20.1

*The last column shows the improvement of BGA with regard to PMC. *Nt* shows number of tasks and *Np* shows number of processors. The makespan values are reported in time unit (e.g. second)

$\left(1 - \frac{\text{BGA makespan average}}{\text{PMC makespan average}}\right) \times 100$ in percentages

a 300-node graph. In this case, the STD value of algorithms is significant. As a matter of fact, it shows that the BGA in bigger problems has much better stability and reliability. The best results in each row of Table 8 are shown by Bold-Italic-Underline font.

To compare BGA with incremental GA, the test benches in Table 1 have been employed. Table 9 shows the results of applying BGA, PMC, and incremental GA on these test benches. In each case, the average results are the outcomes of 10 runs of these algorithms. According to Table 9, it is obvious that the BGA could find the acceptable solutions in much less iterations in comparison to incremental GA. In other words, incremental GA can find better makespan in some cases with much more iterations. Also, in some cases, the BGA could achieve the better average of makespan in comparison with this method. It is worth mentioning that the BGA is not improved in the last 200 iterations (second termination criterion) so the average iterations should be subtracted by 200. For instance, the BGA could find the makespan 300 for P1 in just 54 iterations in average while the incremental GA found this value in 682 iterations.

Table 9 The resultant makespans in solving problems in Table 1 by BGA, incremental GA, and PMC

Problem name	BGA		PMC [14]		Incremental GA [43]	
	Makespan average	Average of generations	Makespan average	Average of generations	Makespan average	Average of generations
P1	***300***	***254***	***300***	304	***300***	682
P2	440	***320***	472	375	***430***	1011
P3	270	***311***	290	340	***263***	934
P4	***365***	***361***	418	372	370	1333
P5	***440***	***311***	539	393	445.9	871
P6	***37.2***	***374***	38.4	384	37.78	1375
P7	390	***380***	424	375	***380***	1316
P8	790	***280***	810	330	***780***	1168
P9	***1088***	***314***	1232	380	1101	1627

Each algorithm was run 10 times and makespan average and average of generations are presented in corresponding column. The makespan values are reported in time unit (e.g. second)

Also, BGA found better makespan in compare to PMC in all problems. The best results in each row are shown by Bold-Italic-Underline font.

5 Conclusions and Future Works

In this paper, a new approach for multi-processor task scheduling has been presented. In this approach, a BGA was proposed which splits the problem into two sub problems: finding an adequate sequence of tasks and its best match processors to process the tasks. In fact, the problem space is split into two subspaces and each of them is solved and the results are combined to find the answer for the main problem. The algorithm has been implemented and its parameters were set by separate experiments to find the best performance. The results were compared with some recent GA-based and heuristic algorithms in terms of STD, average makespan, best obtained makespan and iterations. The experimental results showed that the BGA could find acceptable solutions for the problem in comparison with the recent algorithms and in some cases, the BGA works strongly better. The main virtue of BGA was its stability in big problems (Table 9). In addition, in some cases, the BGA could get better makespan for problems in much less iterations with the same population size in comparison with other related algorithms (Table 9). In future, we are going to develop a much stronger method which can solve all problems in their optimized values.

Appendix

Acronyms Used in the Paper

BAG Bipartite genetic algorithm (the name of the proposed method)
GAP Genetic algorithm for processors. This is a part of BGA
GAS Genetic algorithm for task sequences. This is a part of BGA

GNP Generation number for processors part (*GAP*). It shows the maximum number of generations that the *GAP* should be run

GNS Generation number for sequences (*GAS*). It shows the maximum number of generations that the *GAS* should be run

IT Maximum number of iterations the algorithm (BGA) can continue. It is compared with *NS* in each iteration

Np Number of processors

NS Number of iterations that *GAP* and *GAS* are run. It is equal to *NSS*+*NSP*

NSP Number of iterations that the *GAP* is run in the current call. After calling *GAP*, it should be ceased based on two stopping criteria. If it stops because it achieves *GNP* iterations, the *NSP* will be equal to *GNP* but if it terminates because it didnot improve for the last *SGNP* iterations, the *NSP* is not equal to *GNP*

NSS Number of iterations that the *GAS* is run in the current call. After calling *GAS*, it should be ceased after some iterations based on two stopping criteria. If it stops because it achieves *GNS* iterations, the *NSS* will be equal to *GNS* but if it terminates because it didnot improve for the last *SGNS* iterations, the *NSS* is not equal to *GNS*

Nt Number of tasks

PcP Crossover rate for *GAP*

PcS Crossover rate for *GAS*

PmP Mutation rate for *GAP*

PmS Mutation rate for *GAS*

PopP Population of processor arrays. Indeed, this is a pool of processor numbers. Each element in this population contains an array which its length is equal to the number of tasks

PopS Population of sequences. In fact, this is a pool of task sequences

PsP Population size for *GAP*. In fact, *PsP* is an integer that exhibits the number of chromosomes in *GAP*

PsS Population size for *GAS*. In fact, *PsS* is an integer that exhibits the number of chromosomes in *GAS*

SGNP Successive generation number for processor arrays (*GAP*). In fact it shows the maximum successive number of generations that the *GAP* can continue evolving without any improvement

SGNS Successive generation number for sequences (*GAS*). In fact it shows the maximum successive number of generations that the *GAS* can continue evolving without any improvement

SIT Number of successive iterations the algorithm (BGA) can continue without improvement

References

1. Ahmad, I., Dhodhi, M.K.: Multiprocessor scheduling in a genetic paradigm. Parallel Comput. **22**, 395–406 (1996). doi:10.1016/0167-8191(95)00068-2
2. Ahmad, I., Kwok, Y.: On exploiting task duplication in parallel program scheduling. IEEE Trans. Parallel Distrib. Syst. **9**(9), 872–892 (1998). doi:10.1109/71.722221

3. Bonyadi, M.R., Rahimi Azghadi, M., Hashemi, S., Ebrahimi Moghadam, M.: A hybrid multiprocessor task scheduling method based on immune genetic algorithm. In: Qshine 2008 Workshop on Artificial Intelligence in Grid Computing (2008). doi:10.4108/ICST.QSHINE2008.4263
4. Braun, T.D., Siegel, H.J., Beck, N.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J. Parallel Distrib. Comput. **61**, 810–837 (2001). doi:10.1006/jpdc.2000.1714
5. Chen, H.,Cheng, A.K.: Applying ant colony optimization to the partitioned scheduling problem for heterogeneous multiprocessors. Special Issue: IEEE RTAS 2005 Work-in-Progress, vol. 2, issue 2, pp. 11–14 (2005)
6. Corbalan, J., Martorell, X., Labarta, J.: Performance-driven processor allocation. IEEE Trans. Parallel Distrib. Syst. **16**(7), 599–611 (2005). doi:10.1109/TPDS.2005.85
7. Dhodhi, M.K., Ahmad, I.: A multiprocessor scheduling scheme using problem-space genetic algorithms. In: Proceedings of IEEE International Conference on Evolutionary Compututions, pp. 214–219 (1995)
8. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computation, 1st edn. Springer, Natural Computing Series (2003)
9. Ercan, M.F.: A hybrid particle swarm optimization approach for scheduling flow-shops with multiprocessor tasks. In: International Conference on Information Science and Security, pp. 13–16 (2008)
10. Hamidzadeh, B., Kit, L.Y., Lilja, D.J.: Dynamic task scheduling using online optimization. IEEE Trans. Parallel Distrib. Syst. **11**(11), 1151–1162 (2000). doi:10.1109/71.888636
11. Holland, J.H.: Adaption in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
12. Hou, E.S.H., Ansari, N., Hong, R.: A genetic algorithm for multiprocessor scheduling. IEEE Trans. Parallel Distrib. Syst. **5**(2), 113–120 (1994). doi:10.1109/71.265940
13. Hwang, J., Chow, Y., Anger, A., Lee, C.: Scheduling precedence graphs in systems with inter-processor communication times. SIAM J. Comput. **8**(2), 244–257 (1989). doi:10.1137/0218016
14. Hwang, R., Gen, M., Katayama, H.: A comparison of multiprocessor task scheduling algorithms with communication costs. Comput. Oper. Res. **35**, 976–993 (2008). doi:10.1016/j.cor.2006.05.013
15. Hwang, R.K., Gen, M.: Multiprocessor scheduling using genetic algorithm with priority-based coding. In: Proceedings of IEEJ Conference on Electronics, Information and Systems (2004)
16. Jelodar, M.S., Fakhraie, S.N., Montazeri, F., Fakhraie, S.M., Ahmadabadi, M.N.: A representation for genetic-algorithm-based multiprocessor task scheduling. In: IEEE Congress on Evolutionary Computation, pp. 16–21 (2006)
17. Kafil, M., Ahmad, I.: Optimal task assignment in heterogeneous distributed computing systems. IEEE Concurr. **6**, 42–51 (1998). doi:10.1109/4434.708255
18. Kasahara, H., Narita, S.: Practical multiprocessing scheduling algorithms for efficient parallel processing. IEEE Trans. Comput. **33**, 1023–1029 (1984). doi:10.1109/TC.1984.1676376
19. Kermia, O., Sorel, Y.: A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor. ISCA PDCS, pp. 1–6 (2007)
20. Kruatrachue, B., Lewis, T.G.: Duplication scheduling heuristic, a new precedence task scheduler for parallel systems. Technical Report, Oregon State University (1987)
21. Lee, Y.H., Chen, C.: A Modified genetic algorithm for task scheduling in multi processor systems. In: The Ninth Workshop on Compiler Techniques for High Performance Computing (2003)
22. Man, L., Yang, L.T.: Hybrid genetic algorithms for scheduling partially ordered tasks in a multi-processor environment. In: 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99), pp. 382–387 (1999)
23. Mayez, A.: Al-Mouhamed, lower bound on the number of processors and time for scheduling precedence graphs with communication costs. IEEE Trans. Softw. Eng. **16**(12), 1390–1401 (1990). doi:10.1109/32.62447
24. Meijer, M.: Scheduling parallel processes using genetic algorithms. Master thesis in the field of artificial intelligence, University of Amsterdam, February 2004
25. Montazeri, F., Salmani-Jelodar, M., Fakhraie, S.N., Fakhraie, S.M.: Evolutionary multiprocessor task scheduling. In: Proceedings of the International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06) (2006)
26. Musnjak, M., Golub, M.: Using a set of elite individuals in a genetic algorithm. In: 26th International Conference on Information Technology Interfaces, pp. 531–536 (2004)

27. Nissanke, N., Leulseged, A., Chillara, S.: Probabilistic performance analysis in multiprocessor scheduling. J. Comput. Contr. Eng. **13**(4), 171–179 (2002). doi:10.1049/cce:20020403
28. Nossal, R.: An evolutionary approach to multiprocessor scheduling of dependent tasks. Special Issue: Bio-inspired Solutions to Parallel Processing Problems, pp. 383–392 (1998)
29. Oguz, C., Ercan, M.F.: A genetic algorithm for multi-layer multiprocessor task scheduling. In: TENCON 2004, IEEE Region 10 Conference, vol. 2, pp. 168–170 (2004)
30. Page, A.J., Naughton, T.J.: Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2005)
31. Qu, Y., Soininen, J.P., Nurmi, J.: A genetic algorithm for scheduling tasks onto dynamically reconfigurable hardware. In: IEEE International Symposium on Circuits and Systems (ISCAS 2007), pp. 161–164 (2007)
32. Rechenberg, I.: Cybernetic solution path of an experimental problem. Royal Aircraft Establishment, Library Translation No. 1122, August 1965
33. Rebreyend, P., Sandnes, F.E., Megson, M.: Static Multiprocessor Task Graph Scheduling in the Genetic Paradigm: A Comparison of Genotype Representations, Parallel Emergent and Distributed Architecture Laboratory (PEDAL). The University of Reading, UK (1998)
34. Ricardo, C.: Corrga, Afonso Ferreira and Pascal Rebreyend, scheduling multiprocessor tasks with genetic algorithm. IEEE Trans. Parallel Distrib. Syst. **10**(8), 825–837 (1999). doi:10.1109/71.790600
35. Rinehart, M., Kianzad, V., Bhattacharyya, S.S.: A modular genetic algorithm for scheduling task graphs. Technical Report UMIACS-TR-2003-66. Institute for Advanced Computer Studies, University of Maryland at College Park (June) (2003)
36. Ritchie, G.: Static multi-processor scheduling with ant colony optimization & local search. Master of science thesis, artificial intelligence, University of Edinburgh (2003)
37. Salleh, S.,Zomaya, A.Y.: Multiprocessor scheduling using mean-field annealing. Special Issue: Bio-inspired Solutions to Parallel Processing Problems, vol. 14, issue 5–6, pp. 393–408
38. Sivanandam, S.N., Visalakshi, P., Bhuvaneswari, A.: Multiprocessor scheduling using hybrid particle swarm optimization with dynamically varying inertia. Int. J. Comput. Sci. Appl. **4**(3), 95–106 (2007)
39. Sutar, S., Sawant, J., Jadhav, J.: Task scheduling for multiprocessor systems using memetic algorithms. In: 4th International Working Conference Performance Modeling and Evaluation of Heterogeneous Networks (HET-NETs '06) (2006)
40. Standard Task Graph Set is available online at: http://www.kasahara.elec.waseda.ac.jp/schedule
41. Thanalapati, T., Dandamudi, S.: An efficient adaptive scheduling scheme for distributed memory multicomputer. IEEE Trans. Parallel Distrib. Syst. **12**(7), 758–768 (2001). doi:10.1109/71.940749
42. Tsuchiya, T., Osada, T., Kikuno, T.: Genetics-based multiprocessor scheduling using task duplication. J. Microprocess. Microsyst. **22**(3–4), 197–207 (1998)
43. Wu, A.S., Yu, H., Jin, S., Lin, K.-C., Schiavone, G.: An incremental genetic algorithm approach to multiprocessor scheduling. IEEE Trans. Parallel Distrib. Syst. **15**(9), 824–834 (2004). doi:10.1109/TPDS.2004.38
44. Wu, M.Y., Gajski, D.D.: Hypertool A programming aid for message-passing systems. IEEE Trans. Parallel Distrib. Syst. 1(3), 330–343 (1990). doi:10.1109/71.80160
45. Wang, P.C., Korfhage, W.: Process scheduling using genetic algorithms. In: 7th IEEE Symposium Parallel and Distributed Processing, Texas, San Antonio, pp. 638–641, October 1995
46. Yang, T., Gerasoulis, A.: DSC: scheduling parallel tasks on an unbounded number of processors. IEEE Trans. Parallel Distrib. Syst. **5**(9), 951–967 (1994)
47. Yo-Kwong, K.: Ishfaq Ahmad, dynamic critical path scheduling: an effective technique for allocating task graphs to multiprocessors. IEEE Trans. Parallel Distrib. Syst. **7**(5), 506–521 (1996). doi:10.1109/71.503776
48. Yue, K., Lilja, D.J.: Designing multiprocessor scheduling algorithms using a distributed genetic algorithm system. Technical Report No. HPPC-96-03, University of Minnesota, High performance Parallel Computing Research Group, May 1996
49. Zhong, Y.W., Yang, J.G.: A genetic algorithm for tasks scheduling in parallel multiprocessor systems. In: Proceedings of the Second International Conference on Machine Learning and Cybernetics, pp. 1785–1790 (2003)
50. Zomaya, A.Y., Teh, Y.H.: Observations on using genetic algorithms for dynamic load-balancing. IEEE Trans. Parallel Distrib. Syst. **12**(9), 899–911 (2001). doi:10.1109/71.954620
51. http://faculties.sbu.ac.ir/~moghadam/STG