Int J Parallel Prog (2012) 40:225–257
DOI 10.1007/s10766-011-0179-0

An Immune-based Genetic Algorithm with Reduced Search Space Coding for Multiprocessor Task Scheduling Problem

Mohsen Ebrahimi Moghaddam · Mohammad Reza Bonyadi

Received: 16 November 2010 / Accepted: 1 July 2011 / Published online: 30 July 2011

Abstract Multiprocessor task scheduling is an important problem in parallel applications and distributed systems. In this way, solving the multiprocessor task scheduling problem (*MTSP*) by heuristic, meta-heuristic, and hybrid algorithms have been proposed in literature. Although the problem has been addressed by many researchers, challenges to improve the convergence speed and the reliability of methods for solving the problem are still continued especially in the case that the communication cost is added to the problem frame work. In this paper, an Immune-based Genetic algorithm (*IGA*), a meta-heuristic approach, with a new coding scheme is proposed to solve *MTSP*. It is shown that the proposed coding reduces the search space of MTSP in many practical problems, which effectively influences the convergence speed of the optimization process. In addition to the reduced search space offered by the proposed coding that eventuate in exploring better solutions at a shorter time frame, it guarantees the validity of solutions by using any crossover and mutation operators. Furthermore, to overcome the regeneration phenomena in the proposed GA (generating similar chromosomes) which leads to premature convergence, an affinity based approach inspired from Artificial Immune system is employed which results in better exploration in the searching process. Experimental results showed that the proposed *IGA* surpasses related works in terms of found makespan (20% improvement in average) while it needs less iterations to find the solutions (90% improvement in average) when it is applied to standard test benches.

M. Ebrahimi Moghaddam (✉) · M. R. Bonyadi
Computer Engineering Department, Electrical and Computer Engineering Faculty,
Shahid Beheshti University, G.C. Velenjak, 1983963113 Tehran, Iran
e-mail: m_moghadam@sbu.ac.ir

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

Keywords Multiprocessor task scheduling problem (MTSP) · Makespan · Genetic algorithm (GA) · Meta-heuristic

Abbreviations

AT (Affinity threshold)	When the SR for two chromosomes is smaller than this threshold, the worst one is re-initialized
CAPET (Coding based on addressing Potentially executable tasks)	The proposed coding
Clr (Closeness ratio)	It shows that the fitness for chromosomes is becoming close to each other
DAG (Directed acyclic graph)	is a directed graph to present a MTSP
GA (Genetic algorithm)	A population-base optimization method
IGA (Immune-based genetic algorithm)	is the proposed GA enhanced by an affinity mechanism
MTSP (Multiprocessor task scheduling problem)	A famous and classical scheduling problem
nt	Number of tasks
np	Number of processors
PET (Potentially executable Tasks)	A queue that contains all tasks which can be executed now that are the tasks which all of their predecessors have been executed
Petc	The length of PET
PSc (Processors sequence)	is an array of processors which its elements are prepared by decoding the genes in a chromosome genes in a chromosome
SR (Similarity ratio)	Shows the similarity for two chromosomes
SSR (Search space ratio)	is a ratio which compares the search space between two coding
TS (Task sequence)	is an array of tasks which its elements are prepared by decoding the genes in a chromosome

1 Introduction

Scheduling problems in multiprocessor, parallel, and distributed systems are classified in the NP-hard class of problems [1]. These problems are employed in different important applications such as information processing, weather forecasting, image processing, database systems, process control, economy, and operation research. In these problems, there are several tasks that should be assigned to different processors in a way that (i) the semantics are preserved, and (ii) the total run-time of all processes is minimized by considering the task dependencies and communication costs [1–5]. The input of these problems is usually considered as a directed acyclic graph (*DAG*)

which provides precedence, dependency, priority among tasks, and cost for communication between two relative tasks. With regards to these aspects, all available multiprocessor task scheduling methods try to allocate limited or undetermined number of processors to a set of tasks such that the minimum execution time for all tasks is attained. The total run-time of all processes is called makespan in literature.

To solve the Multiprocessor Task Scheduling Problem (*MTSP*), many heuristic and meta-heuristic methods have been proposed so far, each of which contain their merits and drawbacks. The basic idea of heuristic methods is to determine an optimal order of tasks regarding to their execution priority. Afterwards, the tasks are executed on appropriate processors in the determined order. Therefore, it is seen that each heuristic approach is composed of two main parts: finding an order of the tasks and allocating a processor to each of them. Also, meta-heuristics such as Genetic Algorithms (GAs), Ant Colony Optimizations (ACO), Particle Swarm Optimization (PSO) are good candidates to be applied to *MTSP* [6–8]. Although there is immense number of methods to confront the *MTSP*, solving the problem more faster and reliable has remained as an open research field which encourages the researchers for further investigations.

Here, we propose a new and efficient coding scheme and genetic method to solve the problem when communication costs exist. The proposed coding technique is called *CAPET* (Coding based on Addressing the Potentially Executable Tasks) and works based on addressing a queue of potentially executable tasks. It is shown that *CAPET* reduces the search space in many practical cases in comparison to other existing coding. In addition, the simplicity of the *CAPET* enabled us to use simple GA operators which influence the execution time of the algorithm. Moreover, to overcome the regeneration phenomena that is usually occurred in GA, an artificial immune system operator is employed to help exploring better results from the search space [9,10]. The proposed method was applied on several benchmarks and results were compared to several state-of-the-art algorithms. Experimental results showed that the proposed method improved the results of related works in terms of makespan and number of needed generations.

The rest of the paper has been organized as follows. Some backgrounds including the multiprocessor task scheduling problem and genetic algorithm are introduced in Sect. 2. Section 3 surveys several previous methods for the *MTSP*. The proposed genetic approach is explained in Sect. 4. Section 5 is dedicated to the experimental results and comparison with other methods, finally we conclude the paper in Sect. 6.

2 Background

2.1 Multiprocessor Task Scheduling Problem

As mentioned earlier, the multiprocessor task scheduling problem (*MTSP*) focuses on achieving an order of tasks so that the makespan is minimized. Difficulty of this problem depends on various parameters such as topology of the task graph for the problem at hand, dependencies conditions among tasks, topology of the multiprocessor system, tasks execution time, communication cost among processors, and etc. [11,12]. Generally, *MTSP* is defined as follows: there are several processors (*np*) and some

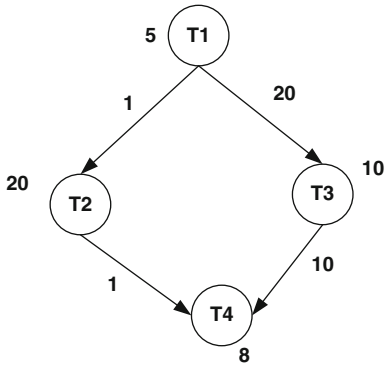

Fig. 1 A typical Multiprocessor task scheduling problem of four tasks presented by a DAG, each task has an execution time and a communication cost in the relation with others

related tasks (*nt*), the tasks can be executed if and only if all of their predecessors have been executed so far. Also, each task needs a certain amount of time for execution and if a task and its successors executed on two different processors, the successors should be executed after a transition time called communication cost. It is worth mentioning that the communication cost is considered if and only if the task and its last predecessors are executed on different processors, otherwise it is considered as zero. In fact, the communication cost is the cost of switching between processors. The aim of this problem is to find an order of tasks and best processor allocation for each task in a way that executing the tasks in the found order on the offered processors leads to minimum makespan.

Usually, the input problem is described by a Directed Acyclic Graph (*DAG*) that represents dependencies and communication cost among available tasks, as well as execution time of each one. An example of such *DAG* has been shown in Fig. 1. In this figure four tasks are available ($nt = 4$), e.g. task T1 is the predecessor of T2 and T3 and the communication cost between them is 1 and 20 respectively while the execution time of task T1 is 5. Task T4 has two predecessors (T2 and T3) and it can be executed after they executed both.

In the MTSP, the number of processors is *np*) should be specified. Therefore, the goal of solving problem is to find an order of tasks and the best processor to execute each task, without violating the precedence constraints.

Because all combination of processors can be assigned to each sequence of tasks, the problem space of the MTSP grows exponentially with regards to *np*. Also, without considering the precedence constraint of tasks, the problem space is grown with a factorial coefficient of *nt*. All in all, we can say that $nt!{*}np^{nt}$ is an upper bound for the space of the MTSP that is happened when all sequences of tasks are valid. This occurs when no precedence constraint exists. Note that the actual problem space for a specific MTSP depends on the precedence constraints of the tasks in its *DAG* but it is smaller than the mentioned upper bound. The lower bound for the problem space of MTSP happens when all tasks are related to each other in a chain which has been shown in Fig. 2.

In this case, there is just one valid sequence of tasks that reduces the problem space to np^{nt}.

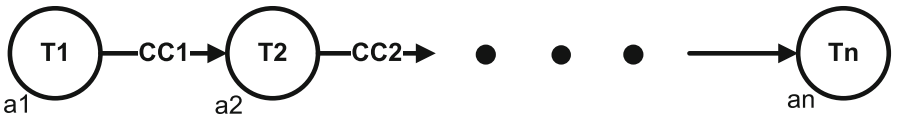

Fig. 2 When tasks are linked together in a chain, there is just one valid sequence of tasks which reduces the problem space

2.2 Genetic Algorithm

Genetic Algorithm is an evolutionary computational method proposed by Rechenberg and Holland [13]. This method imitates the process of biological evolution in nature, and classified as random search techniques. The Genetic based methods are appropriate for exploring large search spaces, therefore, they can effectively used when an NP-Hard or NP-Complete problem is in hand. In such methods, various candidate solutions are chased during the search procedure in the system, and the population evolves until a candidate solution satisfies a predefined criteria. Each solution (individual) is represented as a sequence (chromosome) of elements (genes) and is assigned a fitness value based on the value given by an evaluation function. The way that the problem is mapped into a sequence of numbers is called "coding" procedure. The fitness value measures how close the individual is to the optimum solution. To calculate the fitness value, each chromosome is "decoded" and evaluated. A set of individuals constitutes a population that evolves from one generation to the next through the creation of new individuals and deletion of some old ones. The process starts with an initial population created in some way, e.g. through a random process. Evolution can take two forms:

I. Crossover

Two selected chromosomes can be combined by a crossover operator; the result is replaced with the lowest fitness chromosome in the population. Selection of each chromosome is performed by an algorithm that ensures the selection probability is proportional to the fitness of chromosome. New chromosome has the chance to be better than the replaced one. The process is oriented towards the sub-regions of the search space, where optimal solution is supposed to exist [13].

II. Mutation

In mutation process, a gene from a selected chromosome is randomly changed. This provides additional chances of entering unexplored sub-regions. Finally, the evolution is stopped when either the goal is reached or a maximum generation has been spent.

3 Related Works

Since now, different approaches have been employed to solve *MTSP* such as heuristic algorithms [14–16], evolutionary approaches [7,8,11,17–20], and hybrid methods [21–27]. In this section, a brief survey study on some solutions of *MTSP* is represented, in which it is tried to investigate the best ones in terms of their results. This section is divided into two sub-sections. In the first sub-section, heuristic methods are surveyed while second subsection contains a review of meta-heuristic methods.

3.1 Heuristic Methods for Solving the Multiprocessor Task Scheduling Problem

There are various heuristic algorithms for *MTSP* [12,28–31]. The best heuristic approaches are based on task list technique [28–35]. In this technique, a list of tasks in descending priority order is made. Then, a task is removed from the head of sorted list and is assigned to a processor. These methods are classified as: static and dynamic.

In the static approaches, a static order is assigned to each problem and tasks are removed from the head of priority list till the last node comes out [35–38]. In the dynamic approaches, the order of the list is not constant and it might be changed dynamically after executing each task. Dynamic approaches such as [1,16,39] can reach better makespan in compare to static ones while they have higher time complexity due to updating the priorities after assigning processor to the tasks... In the following, some of them are overviewed.

In [40], a scheduling method has been proposed which uses list scheduling heuristic called ISH (Insertion Scheduling) followed by DSH (Duplication scheduling) that is a task duplication method [40]. Another method which uses task duplication technique is CPFD (Critical Path Fast Duplication) [41]. This work is based on a classical technique in finding the critical path in the *DAG* of input problem. CPFD assigns higher priority to the tasks which are placed in the critical path. Kasahara and Narita proposed another algorithm based on the critical path and most immediate successor in the input *DAG* [15]. Also, a method based on a dynamic critical path was proposed in [16]. In this work, at first the critical path of the *DAG* is determined and then the schedule on each processor is rearranged dynamically based on this path such that a suitable processor for a task in the critical path is assigned by looking ahead the potential start times of the remaining tasks on that processor. MCP (Modified Critical Path) which has been presented by Wu and Gajski [42] is another work in the class of heuristic algorithms for *MTSP* that was based on critical path.

Apart from the task duplication and critical path approaches, there are two other attributes, called *t-level* (top level) and *b-level* (bottom level), for assigning priority to the processors [29,31,34]. These attributes are elicited from the input problem *DAG* and frequently used in related researches. The *t-level* of node n_i in the input *DAG* is the length of a longest path from an entry node to n_i (excluding n_i). The *b-level* of node n_i is the length of a longest path from n_i to an exit node and hence the *b-level* of a node is bounded from above by the length of *DAG* critical path [32]. Besides the mentioned approaches, there is another frequently-used parameter in the *MTSP* that is called ALAP (As Late As Possible) start time [16,42], which defines the longest possible execution time that a task can be postponed.

There are several other features and definitions of *DAG*s [39] which are used in heuristic methods such as HLFET (Highest Level First with Estimated Times), HLFNET (Highest Levels First with No Estimated Times), Random (the assigned tasks priority are random), SCFET (Smallest Co-levels First with Estimated Times) and SCFNET (Smallest Co-levels First with No Estimated Times) [34], CP/MISF (critical path/most immediate successors first) [15], HNF (Heavy Node First) and WL (Weighted Length) [28]. All of these attributes act based on level concept in the *DAG* and without consideration of communication cost. Moreover, DF/IHS [15], EZ (Edge-zeroing) algorithm [43], LC (Linear Clustering) algorithm [44], DSC (Dominant Sequence Clustering)

algorithm [1,45], MD (Mobility Directed) [42], DCP (Dynamic Critical Path) [16], ETF (Earliest Task First) [14] and greedy heuristics [46] are other heuristic methods.

Although many fast heuristic methods have been proposed in the past four decades, these methods are not considered as intense as before because they do not have good results in all cases. Therefore, nowadays applying combinatorial optimization algorithms such as GA, meta-heuristics, and hybrid methods for solving *MTSP* has attracted a lot of attentions.

3.2 Genetic Based Methods for Solving the Multiprocessor Task Scheduling Problem

Whereas genetic algorithm has tackled different engineering problems successfully [47], solving *MTSP* using this evolutionary computing approach has attracted many attentions and various related studies have been reported in the literature [7,8,11,17–20]. Three main differences can be considered among GA-based methods for solving *MTSP*: (1) chromosome representations (coding technique), (2) genetic operators such as crossover and mutation which are dependent to the chromosome representation and (3) the simplicity of the algorithm that affects the complexity of evolutionary optimization process. In the following paragraphs, some of these methods are surveyed.

One of the first methods which utilized GA to solve *MTSP* was presented by Hou, Ansari and Ren (HAR) [8]. In HAR algorithm, the task height in the input *DAG* is the main feature. The chromosomes in this method have a simple structure and each chromosome is composed of several strings. Each string shows a schedule of some tasks based on their heights; also, the number of strings is equal to the number of processors. Despite the simplicity and low computational complexity, the algorithm does not seek the problem space thoroughly and some feasible schedules are not reachable at all [33]. By re-designing the chromosome representation, another algorithm was proposed in [48] which improved the performance of HAR. Also, Correa et al. [19] proposed a method to overcome the drawbacks of HAR. They proposed CGL (Combined Genetic List) as a combinatorial approach which consists of improved GA with introduction of some knowledge about the scheduling problem by list heuristics in genetic operators. This work used a chromosome that its structure was similar to proposed chromosomes in HAR, but it used knowledge augmented genetic operators. Although the presented method overcame the problems of HAR and it was a suitable algorithm in terms of solutions quality, it caused a heavy computational load in crossover and mutation operators.

Combining GA and list scheduling led to a new algorithm called Problem-space GA (PSGA) [27]. This method is one of the pioneer approaches in hybrid GAs to solve *MTSP* which encouraged researchers to design other combinational algorithms. For example, in [20], a GA-based multiprocessor scheduling in combination with task duplication approach has been presented. The authors showed that task duplication is a useful technique for shortening the length of schedules. This study proposed some genetic operators to control the degree of tasks replication.

In 2000 and 2001, two genetic based methods were proposed that considered load balancing in parallel systems [49,50]. 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. In these methods, some important parameters in load balancing like memory locality, scheduling overhead, threshold policies, information exchange criteria, inter-processor communication and their effects on load balancing have been considered.

With advances in genetic methods for solving *MTSP*, some researchers tried to modify the conventional approaches of genetic algorithm and combine them with other problem solving techniques such as divide and conquer. In 2003, a genetic approach called PGA (Partitioned Genetic Algorithm) has been proposed [17], which divides the input *DAG* to some partial graphs using a b-level partitioning algorithm, and each of these separated parts is solved separately using GA. Afterwards subgroups are cascaded and formed the final solution through a conquer algorithm. As it was claimed by authors, a better scheduling time in comparison to a pure GA was reached by their method. Beside this work, two other articles were published in 2003 [33,51]. In [33], a genetic algorithm called TEOL (Task Execution Order List) was presented in which all feasible schedules were surely reachable with the same probability. The aim of this work was overcoming the drawbacks of two previous works ([8] and [19]). Its results showed its enhanced shortening execution time in comparison to [8], but it had similar execution time or in some cases worse results in compare to [19].

After the third year of new century, many attempts were made by researchers for solving the problem with better optimality [6,18,52–54]. These methods used new selection techniques [52], new chromosome representations [8,18], dynamic and parallel GAs [53,54], investigating the stagnant state of GA and facing shortcomings using memetic algorithm [6], variable length chromosomes for GA [22], Bipartite genetic algorithm(*BGA*) [1], and Priority-based Multi-Chromosome (*PMC*) [18]. To evaluate the results of proposed method in the present paper, we selected *BGA*, *PMC* and *Incremental GA* that are the most recent and have better results in compare with others; therefore, in next paragraph these methods are surveyed in more details.

Wu et al. [8] proposed an incremental genetic technique which reached valid solutions gradually. Main contribution of their paper was presenting a novel, flexible and adaptive chromosome representation. In this method, each solution or individual was shown by a set of cells, each of which consisted of a pair (t, p) that t and p correspond with processor and task numbers respectively, i.e., each task should be executed on its couple processor. Flexibility of each chromosome is due to this fact that each individual may have a different length in comparison to others'. This feature formed the incremental procedure of this algorithm in constructing valid solutions. Another important issue about this method is that duplicating tasks in an individual is acceptable.

One of the recent genetic based methods that considered communication costs has been presented in [18]. The authors of this paper proposed the extension of the priority-based coding method as priority-based multi-chromosome (*PMC*). Moreover, a new crossover method which was compatible with this new encoding was proposed, 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 [33]. As far as we studied, *PMC* method is one of the best works that has been presented so far. It has a simple chromosome structure that embeds all required information for scheduling in only one dimension. The suitable and intelligible design of GA operators resulted in valid solution production and also algorithm time reduction. In [9], we defined a new and fast heuristic for the problem, which was used to provide the initial population in initialization phase. The used coding scheme and GA operators in this paper were the same as those in PMC. In this work, an Immune –based operator were adjoined to the GA to facilitate the convergence process. This operator worked based on the Euclidean distance between chromosomes and decision about omitting a chromosome was made based on a threshold.

Also, recently we have published a new GA based method called Bipartite Genetic Algorithm (*BGA*) which efficiently deals with *MTSP* [1]. The method contains two parts; finding an optimized sequence of tasks and assigning processors. In fact, there are two populations, one for tasks sequences and the other for processors, which co-operate with each other to solve the problem. First, the population of tasks sequences is evolved in a certain number of generations. This process is evaluated according to the best fit processor array in the population of processors. Then, the population of processors is evolved by another genetic algorithm which tries to find best processor array for the sequences in the tasks population. After a certain number of generations, both parts are redone until the optimum sequence of tasks and the best fit processor array obtained [1].

4 Proposed Genetic Method

The proposed genetic method is based on a new and efficient chromosome structure. Because of the simplicity in chromosome representation, the method consumes low execution time and converges faster in comparison to related approaches (See Sect. 5). In this section, at first we describe the proposed coding and discuss its usefulness. After that, other parts of the proposed genetic algorithm such as its reproduction operators and selection algorithm are thoroughly explained. In addition, we pose an immune mechanism that is combined with other population-based methods to overcome the regeneration phenomena which is described in Sect. 4.3.

4.1 Coding/Decoding of Solutions (Chromosome Representation)

There are two aspects in *MTSP* that should be addressed by a coding solution: a sequence of tasks which does not violate the precedence constraints in the input problem and a corresponding processor assignment to this sequence which minimizes the makespan of the presented schedule. It has been shown that coding can affect the difficulty of genetic operators as well as the algorithm potential for convergence to an optimal schedule [51].

In this paper, a coding called *CAPET* (Coding based on Addressing the Potentially Executable Tasks) is proposed. *CAPET* uses a one dimensional array of *nt* (number of tasks) integer numbers in the interval [1, np^*m] (m is defined in next paragraphs)

Fig. 3 An Example of the proposed chromosome structure, each number is generated randomly based on the mentioned notes. Using these random numbers two facts may be discovered, first a task for executing (Eq. 1), and second a processor for executing this task (Eq. 2)

for each chromosome. These numbers are interpreted in decoding phase to extract a sequence of tasks and their corresponding processors.

To decode a chromosome a queue is used which contains all potentially executable tasks. This queue is called *PET* and consists of tasks that all of their predecessors have been executed and they are ready to be executed now. *PET* initially is filled by the tasks at the first level of input *DAG* (these tasks have no predecessor). Also, as it was mentioned, each gene of a chromosome contains an integer in the interval $[1, np^*m]$ where m is the maximum length of the *PET* and np is the number of processors. Each gene is interpreted to address a task for execution according to Eq. (1):

$$TS_i = PET[(C_i \bmod petc) + 1] \tag{1}$$

In this equation, $petc$ is the size of *PET*, C_i is ith integer in the chromosome (ith gene), *PET* is the defined queue, and TS_i is the task that should be executed now. The symbol PET[.] refers to the indexes of the array PET. Note that *PET* is indexed from 1 to nt but the result of *mod* operator is ranged from 0 to $nt - 1$, hence, the array index has been added by one. To assign an appropriate processor to task TS_i, this gene (C_i) is interpreted by the following equation:

$$PSc_i = \left(\left\lfloor \frac{C_i}{petc} \right\rfloor \bmod np \right) + 1 \tag{2}$$

In this equation, np is the number of processors and PSc_i is the processor that should be assigned to the x_i. Indeed, by applying Eqs. 1 and 2 on each gene C_i, the pair (TS_i, PSc_i) is extracted. Because the value of C_i is in the interval $[1, np^*m]$, the outcome of the term $\left\lfloor \frac{C_i}{petc} \right\rfloor$ is in the interval $[0, np + e]$ or equal to zero; therefore, in Eq. (2) *mod* operator has been used to find the corresponding processors. After applying Eqs. (1) and (2) on C_i and executing the TS_i on processor PSc_i, the queue (PET) is updated and this process continues until all genes in the chromosome are analyzed.

We will show that by using this coding/decoding procedure, the chromosomes are always valid and no repairing procedure is needed. Also, *CAPET* does not need to be permutation of numbers (repetition of the genes values is allowed). Hence, a simple version of crossover and mutation can be used in reproduction phase. Figure 3 shows a sample chromosome which presents a solution candidate of a nine task problem shown in Fig. 4. *Algorithm 1* shows the step by step procedure for decoding phase of a chromosome.

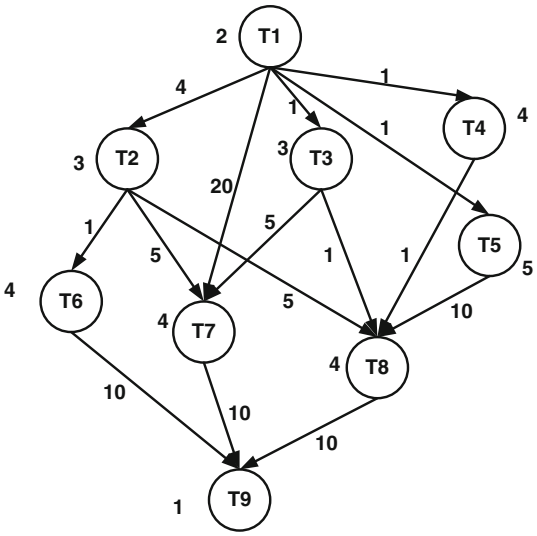

Fig. 4 An example problem with nine tasks and communication costs that is shown by a DAG

Algorithm 1- Chromosome Decoding:

Input: the chromosome C, DAG of input problem, m (maximum length for the PET), np (number of processors)

Output: a valid task sequence (TS), *a processor schedule (PSc) corresponded to TS*

Begin

1) Find all nodes in the DAG which can be scheduled now, they are the tasks in the first height of the DAG for the initial stage. Add these nodes to PET (Potential Executable Task Set);

2) For i=1 to chromosome length (nt)

2.1) TS (i) =PET [(C_i mod petc)+1];

2.2) PSc (i) = [(C_i div petc) mod np] + 1;

2.3) Discard TS(i) from PET;

2.4) Adjoin all potential executable tasks (the tasks that can be scheduled now) to PET;

3) End for

4) Output: TS (a valid task sequence), PSc (Processor Schedule corresponded to TS)

End of algorithm 1

Table 1 shows the step by step decoding procedure (*Algorithm 1*) of the chromosome in Fig. 3 when the problem in Fig. 4 is considered as input problem. In addition, the Gantt chart of decoded solution is illustrated in Fig. 5. It is worth mentioning that in this example the number of processors was considered as 2 ($np = 2$) and the value of m is 4 (the procedure of computing the value of m is presented in the next paragraphs).

From the table, it is seen that in the first step the PET contains all tasks which have no precedence (in this example, task 1). Then, the first gene is interpreted by Eq. 1 and the first task is selected for execution. As it is seen, the only task in the PET is selected for execution. Afterwards, by using Eq. 2, a processor is assigned to this task. As it has been shown in the table, the first processor has been assigned to the first task. Then, the PET is updated (column 2 of the table) which shows the executable tasks after executing the first task. According to the second gene, Eqs. 1, 2, PET [1] is the selected task for execution and it should be executed on processor 1. This procedure continues until all genes are processed. In each step, a task from PET is selected and a processor is assigned to this task.

Table 1 The step by step proposed decoding procedure for the chromosome in Fig. 3

Seq.	PET	Length of *PET*	Chromosome [Seq] (underprocessing gene)	To be schedule task	Processor	Task schedule
1	{1}	1	2	PET((2%1)+1)= PET(1)=1	((2/1)%2)+1=1	{1}
2	{2,3,4,5}	4	8	PET((8%4)+1) = PET(1)=2	((8/4)%2)+1=1	{1,2}
3	{3,4,5,6}	4	4	PET((4%4)+1) = PET(1)=3	((4/4)%2)+1=2	{1,2,3}
4	{4,5,6,7}	4	6	PET((6%4)+1) = PET(3)=6	((6/3)%2)+1=2	{1,2,3,6}
5	{4,5,7}	3	7	PET((7%3)+1) = PET(2)=5	((7/3)%2)+1=1	{1,2,3,6,5}
6	{4,7}	2	5	PET((5%2)+1) = PET(2)=7	((5/2)%2)+1=1	{1,2,3,6,5,7}
7	{4}	1	7	PET((7%1)+1) = PET(1)=4	((7/1)%2)+1=2	{1,2,3,6,5,7,4}
8	{8}	1	8	PET((8%1)+1) = PET(1)=8	((8/1)%2)+1=1	{1,2,3,6,5,7,4,8}
9	{9}	1	6	PET((6%1)+1) = PET(1)=9	((6/1)%2)+1=1	{1,2,3,6,5,7,4,8,9}

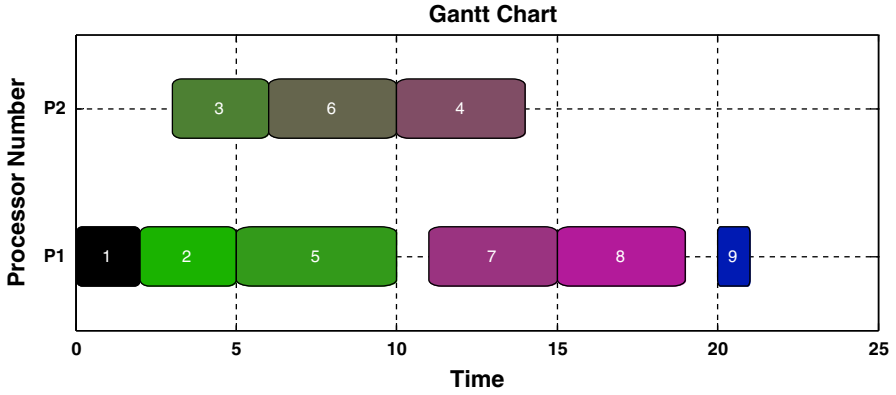

Fig. 5 The Schedule Generated by the proposed algorithm for the problem in Fig. 4 by coding in Fig. 3

It is worth mentioning that *CAPET* covers the problem space thoroughly if the value of m is calculated correctly. To show the capability of the *CAPET* in covering all the problem space we have to show that the chromosomes, which have been coded by this coding scheme, can be decoded in a way that

(1) They can address all feasible sequences of tasks
(2) They can assign all combination of processors to these tasks.

First of all, we have to show that all feasible sequences of tasks are reachable by decoding phase. In other words, if we show that each task can appear in all indexes of the decoded sequence (TS in Algorithm 1) we can say that all sequences of the tasks can be achieved. To show this, note that all tasks in *TS* are the tasks which their predecessors have been executed. In fact, *TS(i)* contains a task which all of its predecessors have been executed. This is because of this fact that the task in *TS(i)* has been selected according to Eq. 1 which selects the tasks from PET that contains the executable tasks, the tasks which their predecessors have been executed. Hence, *TS(i)* will be a task which is executable. Now, we have to show that all feasible sequences of the tasks are reachable. Consider we are interpreting ith gene in the input chromosome (C_i). As it was mentioned, in each step of the decoding algorithm the PET just contains executable tasks. Also, C_i carries an integer in the interval $[1 \ldots np^*m]$ which is used to extract the tasks from PET. By considering this fact that m is bigger than or equal to the number of tasks in PET (Maximum length of PET), np^*m is always bigger than or equal to the length of the PET. Thus, for various values of C_i, the term $[(Ci \ \ mod \ \ petc) + 1]$ can generate all numbers between 1 and the current length of PET. Consequently, we can say that all indexes in PET have this chance to be addressed by C_i. So, all feasible order of tasks can be generated by this decoding procedure.

At this time, we are going to show that the interpretation of each gene for addressing the processors (Eq. 2) is in a way that a task in kth index of the queue has this chance to be scheduled on any processor. In fact, by using this equation, all processors have this chance to be assigned to all tasks. To prove the latter claim, consider that task i

is in the kth index of the queue. We are trying to show that ith gene can address this task and assign all processors to this task. In the worst case, *petc* (the current size of PET) is equal to m which leads to address this task (task i that is kth index of PET) by exactly np integers in the interval $[1, np^*m]$ that are k, $k+petc$, …, $k+(np-1)*petc$. In fact, by considering each of these values in C_i, the kth task in PET is addressed when Eq. 1 is used. The quotient of these integers is an integer in the interval $[0, np-1]$ when they are divided by *petc* (Eq. 2 uses this quotient). Hence, Eq. 2 can address all processors for each desired task in worst case. Note that, in all *DAG*s, the tasks are labeled from left to right according to their heights.

To sum up, we can say that each gene in the proposed chromosome structure have this chance to address all tasks with assigning all combination of processors.

Let us back to finding the value of maximum length of *PET* (m). Because designing an algorithm for finding an upper bound for *petc* in an input *DAG* is not easy in general, to calculate the value of m we used a preprocessing procedure in combination with an adaptive approach. Indeed, before start of optimization algorithm, the value of m is set as nt which is its maximum possible value (the length of *PET* is always smaller than nt). Then, several chromosomes are generated (in the simulations we generated 1,000 chromosomes) randomly and all of them are decoded. In this way, we calculate the maximum length of the *PET* for these chromosomes and use this value plus one as the value of m. Afterwards, the initial population is generated using this value of m and the optimization algorithm (*IGA*) is started. *Algorithm 2* shows the preprocessing procedure.

Algorithm 2- Initial value of m:

```
Input: Population p, DAG of the problem
Output: initial value of m
Begin
    m=nt
    generate 1000 chromosomes randomly
    decode the chromosomes
    m= the maximum value for the petc
    m=m+1
End of algorithm 2
```

Because this algorithm does not guarantee to test all combinations of integers in chromosomes, the calculated value of m may be smaller than the maximum possible length of PET. Hence, it has been added by 1 as a confidence bound. Because this confidence bound may not be enough as well, the value of m is updated in the evolution procedure adaptively by following criteria: if there is a sequence of tasks and processors which needs a queue with size m, the value of m is added by one and the new value of m is used thereafter. Note that if the initial value of m be smaller than the maximum possible length of PET, the mutation operator gives us this chance to get missed genes values in next generations. As an example, the value of m for the input *DAG* in Fig. 4 is 4. In addition, our simulations showed that in more than 98% of tests, the initial value of m was the correct one and is not updated in the evolution procedure. Moreover, the exact value of m can be easily calculated for many practical problems (see Sect. 4.3).

4.2 Proposed Method Overview and Its Operators

The proposed algorithm pursues the conventional GA mechanism that is shown in *Algorithm 3*.

Algorithm 3-Conventional genetic algorithm pseudo code:

```
Calculate the initial value of m using Algorithm 2
Initialize Population;
While terminating condition is not emerged
Calculate objective value (fitness) for all of the chromosomes;
Perform Selection;
Perform cross over;
Perform mutation;
Update m:
If there is any chromosome which needs a PET for decoding that its length is equal to m then m=m+1
End of algorithm 3
```

Each step of the algorithm is described in next sub-sections. Also, the fitness values (Eq. 3) for the chromosomes is calculated according to their makespan, therefore, the smaller value of the fitness is desired.

$$Fitness(C) = makespan(Decoding(C)) \tag{3}$$

The decoding procedure has been described in *Algorithm 1*. In this procedure, the maximum needed length of PET for all chromosomes is stored and is used to update the value of m.

4.2.1 Selection Algorithm

In this study, we used a famous selection method called roulette wheel [55]. In roulette wheel, the chromosomes are selected according to their fitness values in a way that if the fitness value for chromosome a is better than the fitness value for chromosome b, the chromosome a participates in further evaluations with a greater probability. This selection method allows *GA* to give more chance to good individuals with better fitness values to be survived while the individuals which their fitness values are unfavorable have small chance to participate in reproduction, but they are not removed permanently. In other words, poor chromosomes have a chance to participate in reproduction phase as well as privileged chromosomes have, but with a lower probability. In consequent, we can say that through the roulette wheel procedure, some of the good individuals are reproduced, whereas some of the bad ones are eliminated and hence, the population is likely to be dominated by high-quality individuals.

4.2.2 Reproduction Operators

Because of the simple structure of chromosomes that is non-permutation, using complex crossover operators is not needed. In the proposed method, we used simple crossover operators such as “two points” and “one point”. In n-point crossover, two chromosomes are selected and exchange their subsequences with each other. This process is performed with probability P_c.

Also, we used two well-known mutation methods in this paper, "Creep" and "Random Resetting". In the creep mutation, a gene in a chromosome is increased by a random value while in Random Resetting the value of a gene in a chromosome is randomly selected from the problem space. The mutation operator is applied to each gene of each chromosome with probability p_m. Moreover, in this study, the condition of attending a determined number of generations is applied as terminating criterion. We have to note that the population is initialized via a random process using the initial value of m.

After applying the mutation and crossover, some new chromosomes are prepared. To produce the new population, best chromosomes among ones in the current population are selected together with all new generated ones (generated by the mutation and crossover operators).

4.3 Proposed Immune Genetic Algorithm (*IGA*)

In genetic algorithm, when a population is formed, some similar solutions may be created which cause the search process traps in a local optimum. Also, it is possible that the chromosomes become similar [10,56] while the algorithm is in progress. This phenomenon is called "regeneration", in this case, the search process stocks in a local optimum. Therefore, evolution process improves slowly or even become stagnant. To overcome the regeneration problem, we can check the similarity of solutions in the evolution procedure and eliminate solutions which are similar to the previous ones and substitute new solutions instead. Indeed, when the similarity of two chromosomes is more than a predefined threshold, one of these chromosomes are replaced by a new regenerated one. This method is nominated as an affinity mechanism. Simulations results showed that the proposed affinity mechanism together with presented GA structure caused a significant improvement in compare to the pure GA. There are two important points that should be considered for this mechanism. The first point is "What is the similarity?" and the other is "The value of threshold". Here, the number of tasks that are in the same place and should be executed on the same processor is defined as the similarity between two chromosomes. The following algorithm is introduced which returns the similarity ratio.

Algorithm 4: Similarity Ratio

Input: Chromosomes C_1 and C_2
Output: Similarity Ratio (SR)
[$TS_1 PSs_1$]=Decoding (C_1)
[$TS_2 PSs_2$]=Decoding (C_2)
Counter=0
For i=1 to length of C_1
    If $TS_{1,i}$ is $TS_{2,i}$ and $PSs_{1,i}$ is $PSs_{2,i}$
        Counter=Counter+1
    End if
End for
$SR = \frac{Counter}{Length(C_1)}$
End of algorithm 4

If *Similarity Ratio (SR)* is more than *Affinity Threshold (AT)* then the worse chromosome between C_1 and C_2 is reinitialized. It is worth mentioning that *SR* is calculated

for all chromosomes in the current population. *Algorithm 5* shows how the procedure of affinity mechanism works.

Algorithm 5: Affinity Mechanism

```
Input: Population Pop
Output: Reformed Pop
For i=1 to number of chromosomes
    For j=1 to number of chromosomes
        If (Similarity Ratio (Pop_i, Pop_j) > AT)
            If fitness (Pop_i) < fitness (Pop_j)
                Reinitialize (Pop_j)
            Else
                Reinitialize (Pop_i)
            End if
        End if
    End for
End for
End of algorithm 5
```

The affinity mechanism is applied to the population when one of the following conditions occurs:

- The Closeness Ratio (*Clr*) is less than a specified threshold.
- The best chromosome has not been improved for last t iterations.

The Closeness Ratio (*Clr*) is determined as closeness of chromosomes by Eq. 4.

$$Clr = \frac{\text{Mean of chromosomes}'\text{fitnesses - The fitness of best chromosome}}{\text{Mean of chromosomes}'\text{fitnesses}} \tag{4}$$

In the proposed method, we supposed that when *Clr* is less than 0.01 or the best chromosome has not changed in last 100 (t) iterations, *Algorithm 5* is executed. The specified values have been identified by experiments. The process of finding similarity (SR) and reinitializing the chromosomes (Algorithm 5) is added up to the defined genetic method to strengthen the exploration performance of the evolution process.

Table 2 compares the coding structure and operators of the proposed *IGA* with *BGA*, *Incremental GA* and *PMC* methods. This table compares the search space, feasibility of chromosomes, and operators used in the mentioned algorithms.

As it has been shown in Table 2, both *BGA* [1] and *Incremental GA* [8] utilizes two-part coding (*Incremental GA* utilizes a pair for each gene) to represent the solutions. In contrast, the coding in *PMC* and *CAPET* need a one-dimensional chromosome. Also, *PMC* exerts a specific crossover that sustains the feasibility of chromosome structures but *CAPET* does not need such operator. The search space that is sought by *Incremental GA* and *BGA* is very close to the actual problem space while the *PMC* seeks a rather bigger space. In the *CAPET*, the space that is sought depends on the value of m. Our experiments showed that the value of m is a value that leads to a highly smaller space in many practical problems (See the end of this section and "Appendix"). However, in the *Incremental GA* method, some chromosomes may contain an incompatible length and they cannot be decoded. Moreover, 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, while in the proposed IGA which utilizes the

Table 2 Comparison of IGA, PMC, Incremental GA and BGA with several important aspects

	IGA (*CAPET*)	*BGA* [1]	*Incremental GA* [8]	*PMC*[18]
Needed parts for chromosomes	1	2	2	1
All problem space is searched?	Yes	Yes	Yes	Yes
Search space Size	$(m*np)^{nt}$	$nt!*np^{nt}$	$nt!*np^{nt}$	$P(nt*np, nt)$ *where P is Permutation*
Any specific type of GA operators needed	No	Yes	No	Yes
Feasibility after GA operators	Always feasible	Needs validation	Sometimes Not feasible	Always feasible
How treat invalid chromosomes	Always valid	Validate them	Discard them	Always valid

CAPET in its coding, all chromosomes are always valid and all constraints are satisfied automatically. In addition, the *BGA* needs validation phase while *IGA* does not.

As we just mentioned, the search space that is addressed by *CAPET* is smaller than that of the *BGA* and *Incremental GA* in many cases. Also, by considering the points in Sect. 2.1, the problem space for MTSP depends on the structure of the *DAG* of the problem, which is bounded between $nt!*np^{nt}$ and np^{nt}. Nevertheless, the search space that is addressed by Incremental GA and BGA is equal to the upper bound of the problem space. On the other hand, the space that is sought by CAPET is $m^{nt^*}np^{nt}$. To compare the addressed spaces by *CAPET* and *BGA* and *Incremental GA* methods, the following equation is defined:

$$SSR = \frac{m^{nt}np^{nt}}{nt!np^{nt}} = \frac{m^{nt}}{nt!} \xrightarrow{\log_{10}} \log_{10}^{SSR} = nt\log_{10}^{m} - \underbrace{\log_{10}^{(nt!)}}_{\sum_{i=1}^{nt}\log_{10}^{i}} \longrightarrow SSR$$

$$= 10^{nt\log_{10}^{m} - \sum_{i=1}^{nt}\log_{10}^{i}} \tag{5}$$

In this equation, *SSR* is the Search Space Ratio, *nt* is the number of tasks, *np* is the number of processors and *m* is the maximum length of *PET*. According to this equation, if $SSR < 1$, it means that the *CAPET* addresses a smaller space and other methods seek a bigger space that is not necessary. Our experiments showed that in many standard benchmarks the value of *SSR* is smaller than 1 (see "Appendix"). As an example, the effect of the *CAPET* on the search space of distributing the tasks of Gauss-Jordan method on multi-processors is demonstrated here. Figure 6 shows the Gauss-Jordan *DAG* in a general view:

It is obvious that in this problem the maximum value for *petc* is *s* (that is number of levels in input *DAG*) because the maximum number of independent nodes that can

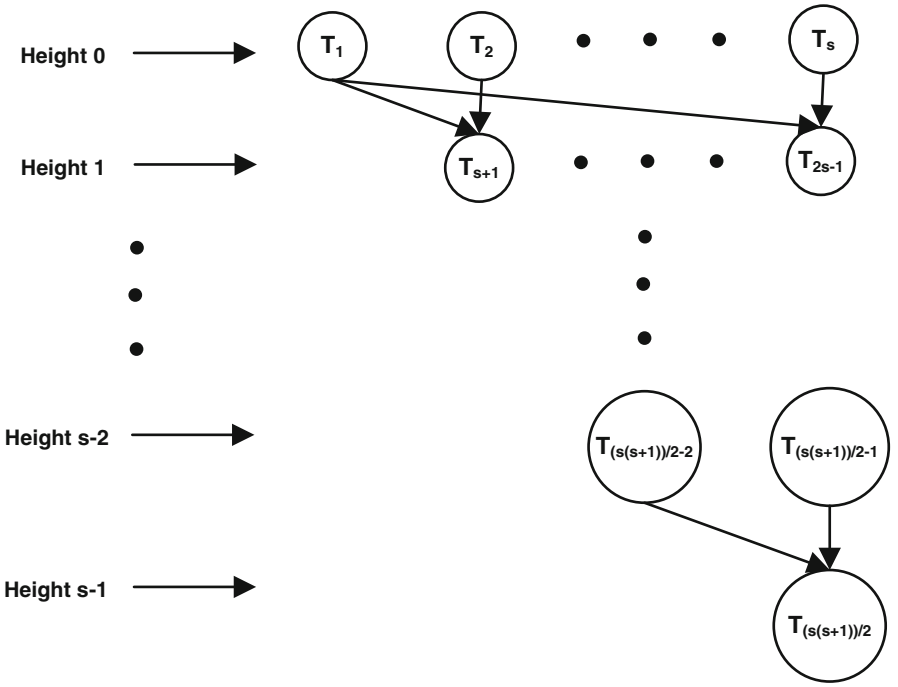

Fig. 6 The general form of the *DAG* for Gauss-Jordan problem

be executed together are in the first height of the *DAG*. Also, using *Algorithm 2*, the value of m for these kinds of *DAGs* which contain s levels (heights) is equal to the number of nodes in the first height which is s. Hence, the space that is addressed by the *CAPET* is $s^{nt}{}^{*}np^{nt}$ while this space for two other methods is $nt!{}^{*}np^{nt}$ in which nt is $\frac{s(s+1)}{2}$. Thus, by substituting in Eq. 5 we have:

$$SSR = 10^{nt\log_{10}^{m} - \sum_{i=1}^{nt}\log_{10}^{i}} \overset{m=s,nt=\frac{s(s+1)}{2}}{\longrightarrow} SSR = 10^{\frac{s(s+1)}{2}\log_{10}^{s} - \sum_{i=1}^{\frac{s(s+1)}{2}}\log_{10}^{i}} \tag{6}$$

Figure 7 shows the curve of *SSR* as a function of $s$for this equation.

It is seen that the *CAPET* addresses a space that is highly smaller than that of the other methods. For instance, in a 465 tasks Gauss-Jordan problem (30 levels) an algorithm which utilizes the *CAPET* for its coding seeks 10^{350} times smaller space in comparison to the other coding. This fact helps the optimization method to converge much faster to better solutions. For more information about the problem space that is sought by the *CAPET* in other *DAG*s see "Appendix". Note that, although the CAPET seeks a considerably smaller space, it addresses all feasible sequences and thoroughly covers the problem space.

The following algorithm shows the procedure of the proposed IGA using CAPET.

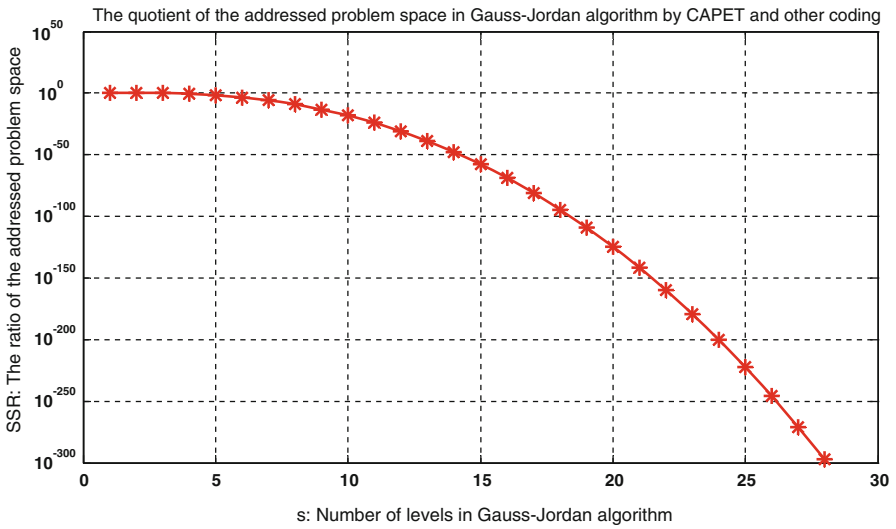

Fig. 7 The ratio of the addressed space by the CAPET to other methods

Algorithm 6: Solve MTSP by IGA

Input: DAG of the Problem, Number of processors (np), Number of tasks (nt)
Output: Gantt chart for executing the tasks
Use Algorithm 2 to find an appropriate value for m
Initialize the population
For i=1 to max_gen
Do selection as described in Section 4.2.1
Do crossover and mutation as explained in Section 4.2.2 and produce new chromosomes
Decode the chromosomes according to Algorithm 1 and calculate fitness (Section 4.2)
Replace the population by the best chromosomes among the previous chromosomes and newly generated ones
Apply Algorithm 4 and 5 to the current population if the related conditions met (Section 4.3)
Update the value of m (maximum PET length plus one)
End for
Decode the best chromosome and draw the Gantt chart
End of Algorithm 6

5 Simulation and Experimental Results

To evaluate the proposed method, we implemented it in *MATLB 7.6* environment and the results were compared with related works. We used a PC with 1.66 Ghz of CPU ad 2 GB of RAM in all tests.

Parameters The used parameters in simulations are as follows:

- *Closeness Ratio* (*Clr*)*: 0.01*
- *Number of iterations without improvement to apply Affinity Mechanism: 100*
- *Pc: 0.9*
- *Pm: 0.01*
- *Population Size: min*(*np*nt, 400*)
- *Mutation/Crossover methods: Random Reset/One point*

- *Affinity Threshold* (AT): $\min\left(0.95, \frac{nt-1}{nt}\right)$

The values of these parameters have been adjusted by trial to get the best results. Also, the value of number of iterations is different for each test and has not been mentioned here. About AT, we have to note that if the value of nt be a small number (say 10 for example), the value of SR is always smaller than 0.95 and hence the re-initialization procedure is just considered for the chromosomes that are exactly the same as each other (SR=1). Hence, the value $\min\left(0.95, \frac{nt-1}{nt}\right)$ has been considered for AT (when $nt = 10$ then $AT = 0.9$ and the chromosomes which their similarity ratio for them is more than 0.9 are reinitialized). The population is initialized randomly in all runs.

Related Works for Comparison: To show the effectiveness of the proposed method, the methods in [1,8,18,20,42,57,58] were selected for comparison purposes because to the best of our knowledge these methods have gained valuable results for the problem, they consider communication cost in their original form, and they are highly cited in the literature.

In all tests, we applied the proposed method 10 times on the problems and the results were reported.

The comparison part is composed of three main phases. At first, the proposed IGA is compared with pure GA (without affinity mechanism) to show the effects of affinity mechanism. Afterwards, results of comparison with some traditional heuristic list scheduling algorithms are reported. Finally, the proposed method is compared with other state-of-the-art GA-based methods.

Test Benches: To compare the presented method with other related works, we applied it on various set of test benches that were often used in literature. These test problems were stemmed from [1,18,20,40,42,57,59]. Some selected problems for comparison purposes have been listed in Table 3. Also, Fig. 8 shows an 18 tasks problem which was stated in [18] and used in evaluation of proposed method. In addition, for further comparisons, we used some problems from [42,59]. These problems belong to the set of problems called STG which does not contain communication cost in their original form. The communication costs have been considered in the same way that was reported in [1].

5.1 Effects of Affinity Mechanism

As it was mentioned, at first we compare the results of the IGA and pure GA when both of them use CAPET for their chromosome coding. The results have been reported in Table 4. The used data base stemmed from STG [59].

The averages have been calculated over 10 runs. From the table, it is clear that the *IGA* found the known optimum solutions in all selected cases at least for 4 runs in all 10 runs. Also, the *IGA* met the optimum solutions more than 7.3 times over 10 runs in average in all cases while this average is 4.4 for pure GA (40% improvement). This shows that the immune mechanism has a significant impact on the performance of the algorithm. Figure 9 shows the Gantt chart (output of our program) for the problem "Rand0002" when 4 processors were considered.

Table 3 Nine selected problems which have been used in different scheduling methods, all of them are prepared in [8], the source column shows the original source of problem and description column shows the usage of problems

Problem	No. tasks	Communication costs	Source	Description
P1	15	25	Tsuchiya et al. [20]	Gauss-Jordan algorithm
P2	15	100	Tsuchiya et al. [20]	Gauss-Jordan algorithm
P3	14	20	Tsuchiya et al. [20]	LU decomposition
P4	14	80	Tsuchiya et al. [20]	LU decomposition
P5	15	Declared in graph	Kruatrachue and Lewis [40]	Not identified
P6	17	Declared in graph	Mouhamed [57]	Not identified
P7	18	Declared in graph	Wu and Gajski [42]	Not identified
P8	16	40	Wu and Gajski [42]	Laplace
P9	16	160	Wu and Gajski [42]	Laplace

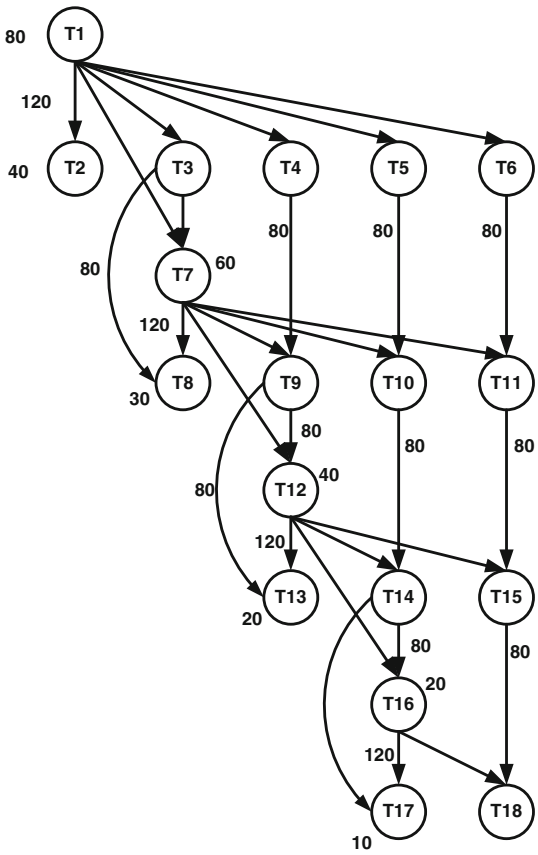

Fig. 8 A sample problem with 18 tasks [18]

Table 4 The result of applying proposed method on five selected problems from the STG dataset (50 tasks problems) in terms of makespan

Problem name	No. of processors	Known optimum	*IGA*			Proposed method without affinity mechanism		
			Best	Average	Number of runs that *IGA* found optimum	Best	Average	Number of runs that *GA* found optimum
Rand0002	4	71	***71***	***71.3***	***7***	72	74.3	0
	16	71	***71***	***71***	***10***	71	71.1	9
Rand0016	4	139	139	***141.2***	***5***	139	144.7	2
	16	139	139	***140.2***	***6***	139	141	4
Rand0019	4	139	***139***	***140.5***	***6***	140	143.5	0
	16	139	***139***	***139***	***10***	139	139	10
Rand0028	4	162	***162***	***163***	***5***	165	169	0
	16	160	***160***	***166***	***4***	160	166	4
Rand0043	4	61	61	***61***	***10***	61	62	5
	16	45	45	***45***	***10***	45	45	10

The number of processors was 4–16 and the IGA was run 10 times for each problem. The number of generations was 200 for these tests. The communication costs were 0 for all problems

Bold italicized numbers are the best values reported

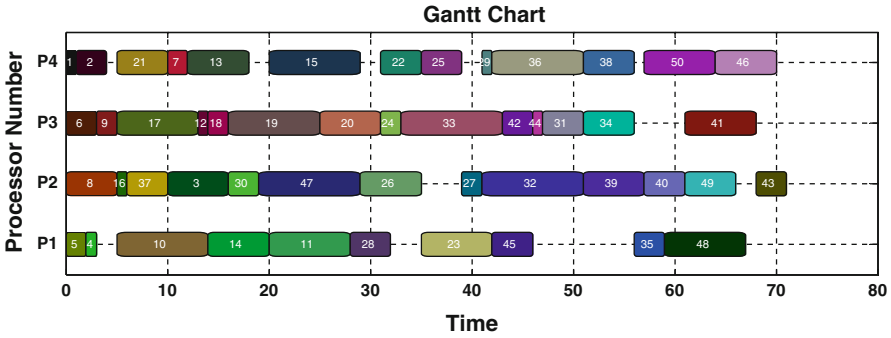

Fig. 9 The Gantt chart for a 50 tasks problem (Rand0002). The makespan on 4 processors was 71 (that is the optimum value for this problem)

Table 5 Comparative results among some previous heuristics and the proposed methods for problem in Figs. 4 and 8

Problem in:	Algorithms	MCP	DSC	MD	DCP	Proposed IGA		
Fig. 4	No. processors	3	4	2	2	2	3	4
	Best solution	29	27	32	32	**21**	**21**	**21**
Fig. 8	No. processors	4	6	3	3	3	4	6
	Best solution	520	460	460	**440**	**440**	**440**	**440**

The number of generations was 200. Best found solutions have been reported. The values are makespan of problems in time unit
Bold numbers are the best values reported

5.2 The Proposed IGA Versus Other Heuristic Methods

In this part, first, the proposed *IGA* is applied to two problems, have been shown in Figs. 4 and 8, and the results are compared with previous methods such as DSC [1], DCP [16], MCP, and MD [42]. Simulation results for different number of processors have been reported in Table 5. As it is clear in this table, the proposed *IGA* have a significant superiority in comparison to these heuristics. In each row of this table, the best achieved results have been bolded.

Also, the performance of the *IGA* was compared to some other previous heuristics such as Insertion Scheduling Heuristic (ISH) [40], Duplication Scheduling Heuristic (DSH) [40], and Critical Path Fast Duplication (CPFD) [41] in Table 6. The best value of each row has been bolded.

As Table 5 shows, IGA outperforms ISH, the only heuristic in the table without task duplication, in 6 cases over 9 and works same as this method in 3 remaining cases. But, in compare to DSH and CPFD, IGA works same in 4 cases and better than them in 1 case. This is because IGA does not use any task duplication but CPFD and DSH use this technique which strongly improves the performance of these methods. The listed methods in the table need a fraction of second (0.08 s for ISH, 0.1 s for CPFD and DSH in average) to solve the problem at hand while IGA needs a considerable amount of time (1.2 s in average) to converge in compare to them. It is worthwhile to mention that

Table 6 Comparative results among some previous heuristics and the proposed methods for nine different problems which are addressed in Table 3 when Number of processors = 4, and Number of generations = 500

Problem	Algorithm	ISH	DSH	CPFD	Proposed *IGA*	Cpu time for *IGA* (s)
P1	Best Solution	**300**	**300**	**300**	**300**	0.22
P2	Best Solution	500	**400**	**400**	420	0.73
P3	Best Solution	**260**	**260**	**260**	**260**	0.9
P4	Best Solution	400	**330**	**330**	360	1.53
P5	Best Solution	650	539	446	**438**	0.88
P6	Best Solution	41	**37**	**37**	**37**	1.47
P7	Best Solution	450	370	**330**	390	1.4
P8	Best Solution	**760**	**760**	**760**	**760**	1.67
P9	Best Solution	1220	**1030**	1040	1070	1.98

The values are makespan of problems in time unit
Bold numbers are the best values reported

heuristic methods create solutions by considering some predefined criteria and do not search the problem space which result in faster execution. However, iterative search algorithms (such as GA) consider a larger number of solutions for the problem at hand which is a time consuming procedure. In addition, about task duplication we have to say that these methods offer better makespan in scheduling tasks that is obtained at a cost of higher utilization of computational resources which might be available or not. Indeed, depending on the access type to the computation capacity, task duplication might be justifiable or not. For more information see [60]. Therefore, although DSH and CPFD outperform IGA in some cases, because of using task duplication, they are not applicable in all cases and this is a big constraint of these methods.

Figure 10a compares ISH, DSH, HLFET and IGA when all of them were applied to 15, 21, 28 and 36 tasks Gauss-Jordan elimination problem. Each task took 40 time units to be processed and all communication costs were 100 time units. The figure shows that IGA works better than the other methods especially when the number of nodes increases.

From the figure, it is seen that the results of IGA are much better than that of the ISH and HLFET in all cases. The results of IGA are the same as DSH in small problems while its results become better when the problem is big. In this case, the cpu time for the IGA is bigger than the cpu time for ISH, HLFET, and DSH. But, as it was mentioned, the results of IGA are better than these methods. Note that DSH uses task duplication while ISH, HLFET, and IGA do not use this technique. Figure10b shows the found schedules for Gauss-Jordan problem with 36 tasks by IGA.

5.3 The Proposed IGA Versus Different GA-based Methods

To compare *IGA* with other GA-based methods, three methods were selected: *PMC* [18], *BGA* [1], and *Incremental GA* [8]. These methods were selected because they have been presented recently and have better results among others. At first, the proposed *IGA* has been compared with the *PMC* [18] and *BGA* [1] when they were applied

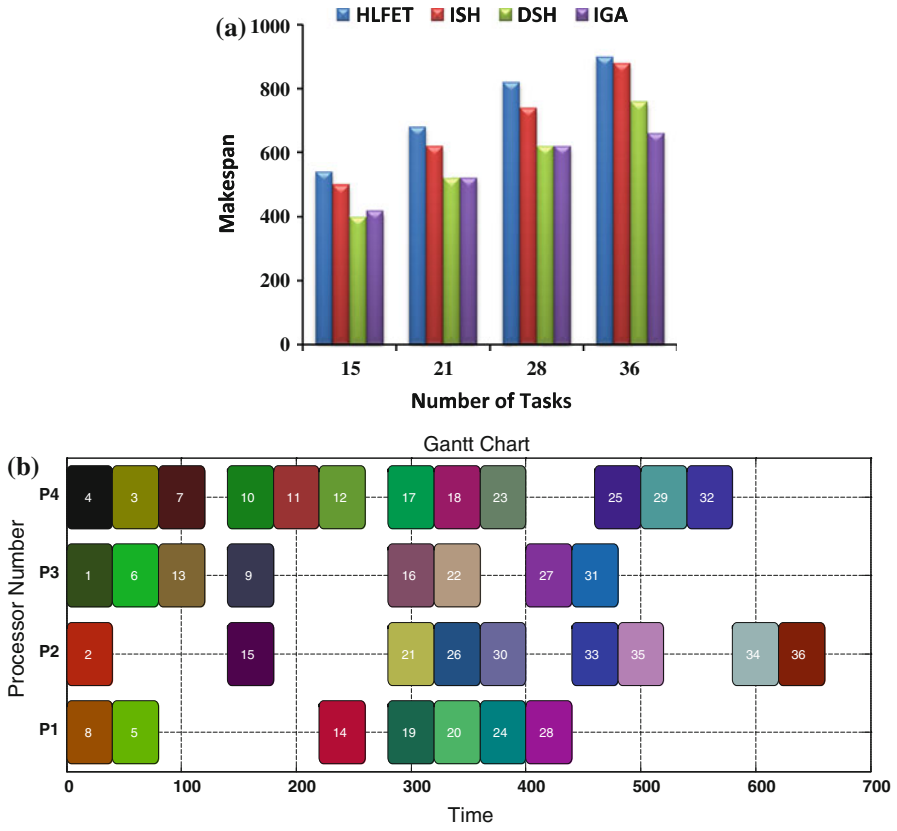

Fig. 10 Comparison results among ISH, DSH and IGA when they are applied to Gauss-Jordan problem. **a** various number of tasks, **b** result of scheduling for Gauss-Jordan 36 by IGA

to the problems in Figs. 4 and 8. The comparative results are presented in Table 7. In this table, some useful parameters such as best, worst, mean, and the standard deviation of solutions are reported and best value in each row has been bolded.

Table 7 obviously shows that *IGA* excels *PMC* and *BGA* in all cases in terms of the best, worst, mean, and standard deviation for achieved solutions. Moreover, with regard to the smaller *std* (standard deviation) for the found solutions by the *IGA*, we can say that the method has the better stability in comparison to *PMC* and *BGA*. Also, *IGA* have about 13 and 1% improvement in average with regards to *PMC* and *BGA* respectively.

Apart from these tests, to have more robust comparison and evaluation, the simulations have been performed on *STG* dataset [59]which contains bigger problems. We applied *IGA* on several problems in *STG* and the results have presented here. In the first test, we applied IGA and BGA on some problems from STG when 4–100 processors were considered for scheduling and the results have been reported in Table 8.

Table 7 Comparative results of proposed method, BGA and PMC for solving problems in Fig. 3(Fig. 7) when number of generations was 200 (500)

Problem in	No. of Processors	Proposed (IGA)				PMC [18]				BGA [1]			
		Best	Worst	Mean	Std	Best	Worst	Mean	Std	Best	Worst	Mean	Std
Fig. 4	2	**21**	21	**21**	0	21	21	22	0.5	21	23	21.9	0.56
	3	**21**	21	**21**	0	21	21	21	0	21	23	22.4	0.69
	4	**21**	21	**21**	0	21	21	21	0	21	23	22.3	0.82
Fig. 8	2	**440**	470	**451**	6.33	460	470	463	4.83	460	520	491	20.78
	3	**440**	450	**446**	5.08	440	490	461	12.86	490	600	522	35.21
	4	**440**	470	**460**	8.2	440	470	461	8.75	500	580	544	25.90
	6	**460**	490	**470**	10.75	460	490	471	11.00	510	580	556	19.55

The values are makespan of problems in time unit
Bold numbers are the best values reported

Table 8 Comparison of BGA and the proposed IGA

Problem name	No. of processors	No. of tasks	CP length	*IGA*			*BGA*		
				Best	Average	CPU	Best	Average	CPU
Rand0002	4	100	124	**140**	***140***	429	143	145.6	777
	16			124	124	194	124	124	757
Rand0016	4		63	**148**	***149***	432	150	155.4	728
	16			**63**	***63.2***	418	69	70.2	756
Rand0019	4		137	**260**	***266***	631	273	279	1018
	16			**137**	***138.1***	663	140	141	1089
Rand0028	4		32	**137**	***137***	166	138	139.1	363
	16			**45**	***46.7***	173	47	48.6	369
Rand0043	4		56	**245**	***247.1***	170	247	247.3	360
	16			**83**	***83.2***	177	85	85.9	337
Rand0043	16	300	194	**256**	***260***	4602	261	264	7178
	100			**194**	***194***	4629	253	255.3	14423
Rand0016	16	500	1269	1269	1269	23712	1269	1269	29906
	100			1269	***1269***	22997	1269	1274	31087
Average			267.9	**312.1**	***313.3***	4242	319.1	321.3	6367

The number of iterations was 200 in this test and the population size was 400 for both algorithms
Bold and bold italicized numbers are the best values reported

In this table, the column CPU shows the average of CPU time over 10 runs for 200 iterations. The table shows that the IGA converges to better solutions while its runtime is considerably smaller than BGA (33% improvement). Also, the CP length shows the critical path length for the problems. It is important to note that even with infinite number of processors, the makespan cannot be lower than the CP length and hence it can be considered as the minimum makespan for a schedule. The table

Table 9 The results of applying IGA, BGA, and PMC on STG dataset with communication costs

Number of Tasks	Problem name	Number of processors	*IGA*		*PMC* [18]		*BGA* [1]	
			Mean	Std	Mean	Std	Mean	Std
50	Rand0002	4	***95***	1	128.6	5.02	105	6.1
		8	***95.7***	1.34	136.4	4.44	104.2	1.5
	Rand0016	4	172.2	3.03	210.8	12.07	***166.6***	9.2
		8	169.8	6.22	189.4	11.71	***161***	4.3
	Rand0019	4	172	4.24	221.2	12.1	***167.2***	8
		8	***158.1***	2.16	203.4	3.20	176.46	7.5
	Rand0028	4	208	1.22	256.4	6.30	***207.2***	7.3
		8	***205.4***	0.89	239.2	8.78	206.6	7.9
	Rand0043	4	***65***	0	94.6	7.95	77	2.1
		8	***60.3***	0.89	80.2	1.64	71.4	2.9
Improvement in percent					20%		**3%**	
100	Rand0002	4	***193.4***	0.89	286.6	14.55	223.4	12.1
		8	***169.4***	4.97	262.4	13.86	215.8	7.12
	Rand0016	4	***146.8***	0.44	236.8	11.60	184.2	7.4
		8	***118***	0	199.6	8.61	154.4	6.65
	Rand0019	4	***262.2***	1.30	369.4	19.44	296.2	3.34
		8	***180***	0	274.8	12.49	224.6	1.5
	Rand0028	4	***138***	0	165	10.66	140.6	1.51
		8	***71***	0	108	7	89	0
	Rand0043	4	***245.2***	1.78	271.2	12.39	247	2.73
		8	***126.7***	1.64	159.4	3.57	134.2	3.08
Improvement in percent					30%		**13.5%**	
300	Rand0002	4	***460***	0	955	22.2	760	4.2
	Rand0016	4	***828.8***	8.04	1496.2	76.55	1006.2	43.9
Improvement in percent			**–**		**47%**		**27%**	
500	Rand0002	4	***1077.5***	3.5	2181	11.23	1231	9.68
Improvement in percent					50.5%		**12.5%**	

The values are makespan of problems in time unit
Bold and bold italicized numbers are the best values reported

shows that IGA can find results near to CP length when number of processors is large enough.

In addition, we applied the *IGA*, *PMC,* and *BGA* on several test problems stemmed from *STG* and the results have been reported in Table 9 when communication cost exists. The STG library contains 180 test cases for 50 to 500 tasks problems but because of the lack of space we reported the results for some randomly selected problems from the library. We have to note that the standard version of the *STG* does not contain communication costs. Hence, we added communication costs to this dataset which can be found in [1,61].

Table 10 Simulation Results of applying IGA method on problems in Table 3 in compare with Incremental GA, PMC, and BGA methods when number of processors were 4

Test problem	*IGA*		*Incremental GA* [8]		*BGA* [1]		*PMC* [18]	
	Ave.	Num of Gen.	Ave.	Num of Gen .	Ave.	Num of Gen.	Ave.	Num of Gen.
P1	300	***2.2***	300	682	300	254	300	304
P2	***424***	***44.2***	430	1011	440	320	472	375
P3	268	***12.2***	***263***	934	270	311	290	340
P4	368	***30.4***	370	1333	***365***	361	418	372
P5	***438***	***26***	445.9	871	440	311	539	393
P6	***37.00***	***54.4***	37.78	1375.46	37.2	374	38.4	384
P7	390	***18***	***380***	1316.62	390	380	424	375
P8	***760***	***45.8***	782	1168	790	280	810	330
P9	***1082***	***183.2***	1101	1627	1088	314	1232	380

The values are makespan of problems in time unit
Bold italicized numbers are the best values reported

In this test, the maximum generation for IGA was 500 while this parameter was 1,000 for BGA and PMC. Also, the number of chromosomes for the IGA was as mentioned in the start part of Sect. 5 while this value was 400 for PMC and BGA. As Table 9 shows, *IGA* surpasses the *PMC* and *BGA* in most cases in terms of the average of makespan (improvement 37 and 14% in average).

To have more comparison and simulations, *IGA* was applied to the problems in Table 3 and its results were compared with *Incremental GA* [8], BGA, and PMC. Table 10 shows the results. The population size for the *Incremental GA* was 400 and the maximum generation was 3,000 while the population size for the *IGA* was as mentioned in the beginning of Sect. 5 and the maximum generation was 500. The population size for BGA and PMC was 400 and the maximum number of generations was 500 for these methods.

These results show that the *IGA* excels the *Incremental GA*, *BGA,* and *PMC* in terms of average of found solutions in 6, 6, and 8 cases over all 9 cases (around 1, 1.2 and 10% improvement in average) respectively. Also, the *IGA* converges much faster (as it was predicted in Sect. 4) than the *Incremental GA*, *BGA,* and *PMC* in all (96, 85 and 87% improvement) cases. The average of CPU time in one run for Incremental GA was 163.3 s while this average was 37.8 s for IGA, 45.3 s for PMC and 69.5 s for BGA. Furthermore, the average of CPU time for finding the best solution was 63s for Incremental GA while this average was 3.5 s for the proposed IGA, 32.5 s for PMC, and 44.8 s for BGA.

6 Conclusions

Each multiprocessor task scheduling problem can be modeled by a *DAG* which shows the relation among tasks and determines the communication costs between them. Since now, various mechanisms based on heuristic and meta-heuristic approaches have been

presented that all of them try to schedule tasks on multiprocessors in an efficient way. In this paper, a new genetic-based method which introduces a novel chromosome structure was presented. The proposed structure was based on a novel and simple chromosome structure that makes the scheduling problem very simple and fast in compare to other well-known counterparts and facilitates the usages of GA operators. Indeed, the proposed coding scheme (called *CAPET*) could reduce the search space and hence it makes the convergence to better solutions much faster. The proposed method strengthened by using an innovative immune inspired approach called Affinity mechanism (hence, we call this method as *IGA*). Experimental results showed remarkable improvements in decreasing the execution time of parallel tasks in a multiprocessor system. The proposed method improved the makespan of task scheduling problems (around 20% improvement in average) while it needed lower number of generations (around 90% improvement in average) and time. Besides, simulation results demonstrated that the *IGA* would be a well organized mechanism in real experimental task scheduling systems where a huge problem is confronted. Moreover, adjoining the task duplication approach to the proposed algorithm can improve the performance of the proposed method. Also, enhancing the method/structure to work in heterogeneous scheduling frame work can be a valuable field to work on.

Appendix

In this appendix, we will present the reduction space ratio (SSR) for all test cases in STG dataset. In fact, the Geometric mean of SSR over 100 first problems in 50, 100, 300, and 500 task problems in the STG test bench has been calculated and reported (in 100 tasks problems in STG, there is just 59 files of problems and our report is just for these 59 problems). In addition, the mean of initial and final value of m have been reported in a table. Because the value of SSR for some test cases was very small, we used geometric mean rather than arithmetic mean. The formula for the geometric mean is as follow:

$$GM = \sqrt[n]{\prod_{i=1}^{n} SSR_i}$$

In this report, n was 100 (100 test cases) and SSR_i is the SSR for ith test case except for 100 tasks problems which contains 59 problems in the dataset.

Because the result of $\prod_{i=1}^{100} SSR_i$ is very small and the calculation would result in zero (the floating point precision in computer might not be adequate), we considered this calculation by following simplifications:

$$GM = \sqrt[n]{10^{\log(\prod_{i=1}^{n} SSR_i)}} = \sqrt[n]{10^{\sum_{i=1}^{n} \log(SSR_i)}} = 10^{\frac{\sum_{i=1}^{n} \log(SSR_i)}{n}}$$

The base of log operator is 10. By substituting SSR_i from Eq. 5 we have:

$$GM = 10^{\frac{\sum_{i=1}^{n} \log(SSR_i)}{n}} \xrightarrow{Eq5} GM = 10^{\frac{nt}{n} \sum_{i=1}^{n} (\log(m_i)) - \sum_{j=1}^{nt} \log(j)}$$

Now, calculation of $\sum Log(SSR_i)$ will result in smaller error in our final calculation. Table Ap.1 shows the results over the mentioned problems.

To find the maximum value for m to guarantee SSR<1 we used the following formula which is directly derived from Eq. 5

$$SSR < 1 \rightarrow nt \log_{10}^{m} < \sum_{i=1}^{nt} \log_{10}^{i} \rightarrow m < 10^{\frac{\sum_{i=1}^{nt} \log_{10}^{i}}{nt}}$$

See "Appendix" Table Ap.1.

Table Ap.1 geometric mean for SSR over 359 problems in STG dataset

Test bench	Initial value of m	Final value of m	Maximum value of m to guarantee $SSR < 1$	Geometric mean of SSR	Number of problems in which $SSR > 1$
50 tasks problems	13.6	13.61	19.48325	$10^{-10.3561}$	13
100 tasks problems	24.1	24.22	37.9927	$10^{-25.18}$	12
300 tasks problems	24.3	24.36	111.7599	$10^{-221.1127}$	2
500 tasks problems	32.26	33.8	185.4269	$10^{-441.2946}$	3

References

1. Bonyadi, M.R., Moghaddam, M.E.: A bipartite genetic algorithm for multi-processor task scheduling, IJPP, Springer **37**(5), 462–487 (2009). doi:10.1007/s10766-009-0107-8
2. Kafil, M., Ahmad, I.: Optimal task assignment in heterogeneous distributed computing systems. IEEE Concurr. **6**, 42–51 (1998)
3. Thanalapati, T., Dandamudi, S.: An efficient adaptive scheduling scheme for distributed memory multicomputer. IEEE Trans. Parallel Distrib. Syst. **12**(7), 758–768 (2001)
4. Nissanke, N., Leulseged, A., Chillara, S.: Probabilistic performance analysis in multiprocessor scheduling. J. Comput. Control Eng. **13**(4), 171–179 (2002)
5. Corbalan, J., Martorell, X., Labarta, J.: Performance-driven processor allocation. IEEE Trans. Parallel Distrib. Syst. **16**(7), 599–611 (2005)
6. 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
7. 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
8. 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)
9. Azghadi, M.R., Bonyadi, M.R., Hashemi, S., Moghadam, M.E.: A hybrid multiprocessor task scheduling method based on immune genetic algorithm, ICST 5th, pp. 1–4. ACM, NY (2008). doi:10.4108/ICST.QSHINE2008.4263
10. Azimipour, M., Bonyadi, M.R., Eshghi, M.: Using Immune Genetic Algorithm in ATPG, AJBAS. ISSN: 1991-8178, pp. 920–928 (2008)
11. Hou, E.S.H., Ansari, N., Hong, R.: A genetic algorithm for multiprocessor scheduling. IEEE Trans. Parallel Distrib. Syst. **5**(2), 113–120 (1994)

12. Garey, M., Johanson, D.: Computers and Intractability: a Guide to the Theory of NP-Completeness. W. H. Freeman and Co., New York (1979)
13. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing, pp. 43–44. Springer (2003)
14. 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)
15. Kasahara, H., Narita, S.: Practical multiprocessing scheduling algorithms for efficient parallel processing. IEEE Trans. Comput. **33**, 1023–1029 (1984)
16. Kwok, Y.-K., Ahmad, I.: Dynamic critical path scheduling: an effective technique for allocating task graphs to multiprocessors. IEEE Trans. Parallel Distrib. Syst. **7**(5), 506–521 (1996)
17. Lee, Y.H., Chen, C.: A modified genetic algorithm for task scheduling in multi Processor systems. In: Proceedings of the Nineth Workshop on Compiler Techniques for High Performance Computing, 2003
18. Hwang, R., Gen, M., Katayama, H.: A comparison of multiprocessor task scheduling algorithms with communication costs. Comput. Oper. Res. **35**, 976–993 (2008)
19. Corrga, R.C., Ferreira, A., Rebreyend, P.: Scheduling multiprocessor tasks with genetic algorithm. IEEE Trans. Parallel Distrib. Syst. **10**(8), 825–837 (1999)
20. Tsuchiya, T., Osada, T., Kikuno, T.: Genetics-based multiprocessor scheduling using task duplication. J. Microprocess. Microsyst. **22**(3-4), 197–207 (1998)
21. 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)
22. 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 **2**(2), 11–14 (2005)
23. Ercan, M.F.: A hybrid particle swarm optimization approach for scheduling flow-shops with multiprocessor tasks. In: Proceedings of the International Conference on Information Science and Security, pp. 13–16 (2008)
24. Sutar, S., Sawant, J., Jadhav, J.: Task scheduling for multiprocessor systems using memetic algorithms. In: Proceedings of the Fourth International Working Conference Performance Modeling and Evaluation of Heterogeneous Networks (HET-NETs '06) (2006)
25. Salleh, S., Zomaya, A.Y.: Multiprocessor scheduling using mean-field annealing. Special Issue: Bio-Inspired Solutions to Parallel Processing Problems **14**(5-6), 393–408 (1998)
26. Man, L., Yang, L.T.: Hybrid genetic algorithms for scheduling partially ordered tasks ina multi-processor environment. In: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, (RTCSA '99), pp. 382–387 (1999)
27. Dhodhi, M.K., Ahmad, I.: A multiprocessor scheduling scheme using problem-space genetic algorithms. In: Proceedings of the IEEE International Conference on Evolutionary Computation, pp. 214–219 (1995)
28. Shirazi, B., Wang, M., Pathak, G.: Analysis and evaluation of heuristic methods for static task scheduling. J. Parallel Distrib. Comput. **10**(3), 222–232 (1990)
29. Gerasoulis, A., Yang, T.: A comparison of clustering heuristics for scheduling *DAG*'s on multiprocessors. J. Parallel Distrib. Comput. **16**(4), 276–291 (1992)
30. El-Rewini, H., Ali, H.H., Lewis, T.G.: Task scheduling in multiprocessor systems. IEEE Computer **28**(12), 27–37 (1995)
31. Ahmad, I., Kwok, Y.-K., Wu, M.-Y.: Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In: Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, pp. 207–213 (1996)
32. Kwok, Y., Ahmad, I.: Static Scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. **31**(4), 406–471 (1999)
33. Zhongiz, 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)
34. Adam, T.L., Chandy, K.M., Dickson, J.R.: A comparison of list scheduling for parallel processing systems. Commun. ACM **17**(12), 685–690 (1974)
35. Coffman, E.G.: Computer and Job-Shop Scheduling Theory. Wiley, NY (1976)
36. Gonzalez, M.J. Jr.: Deterministic processor scheduling. ACM Comput. Surv. **9**(3), 173–204 (1977)
37. Frinsen, D.K.: Tighter bounds for LPT scheduling on uniform processors. SIAM J. Comput. **16**(3), 554–560 (1987)
38. Graham, R.L., Lawler, E.L., Lenstra, J.K., kan, A.H.G.R.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Ann. Discret. Math. **5**, 287–326 (1979)

39. Sih, G.C., Lee, E.A.: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Trans. Parallel Distrib. Syst. **4**(2), 75–87 (1993)
40. Kruatrachue, B., Lewis, T.G.: Duplication Scheduling Heuristic, a New Precedence Task Scheduler for Parallel Systems. Technical Report, Oregon State University (1987)
41. Ahmad, I., Kwok, Y.: On exploiting task duplication in parallel program scheduling. IEEE Trans. Parallel Distrib. Syst. **9**(9), 872–892 (1998)
42. Wu, M.Y., Gajski, D.D.: Hypertool: a programming aid for message-passing systems. IEEE Trans. Parallel Distribut. Syst. **1**(3), 330–343 (1990)
43. Sarkar, V.: Partitioning and Scheduling Parallel Programs for Multiprocessors. MIT Press, Cambridge (1989)
44. Kim, S.J., Browne, J.C.: A general approach to mapping of parallel computation upon multiprocessor architectures. In: Proceedings of International Conference on Parallel Processing, pp. 1–8 (1988)
45. Yang, T., Gerasoulis, A.: List scheduling with and without communication delays. Parallel Comput. **19**(12), 1321–1344 (1993)
46. Kermia, O., Sorel, Y.: A Rapid Heuristic for Scheduling Non-Preemptive Dependent Periodic Tasks onto Multiprocessor. ISCA PDCS, pp.1–6 (2007)
47. Gen, M., Cheng, R.: Genetic Algorithm and Engineering Optimization. Wiley, New York (2000)
48. Wang, P.C., Korfhage, W.: Process scheduling using genetic algorithms. In: Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing, pp. 638–641. San Antonio, Texas, October 1995
49. 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)
50. Hamidzadeh, B., Kit, L.Y., Lilja, D.J.: Dynamic task scheduling using online optimization. IEEE Trans. Parallel Distrib. Syst. **11**(11), 1151–1162 (2000)
51. 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
52. Musnjak, M., Golub, M.: Using a set of elite individuals in a genetic algorithm. In: Proceedings of the 26th International Conference on Informafion Technology Interfaces, pp. 531–536 (2004)
53. Nedunchelian, R., Koushik, K., Meiyappan, N., Raghu, V.: Dynamic task scheduling using parallel genetic algorithms for heterogeneous distributed computing. In: The 2006 World Congress in Computer Science Computer Engineering, and Applied Computing (GCA'06) (2006)
54. 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'05) (2005)
55. Holland, J.H.: Adaption in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
56. Li, Y., Dai, Y., Mam X.: A heuristic immune-genetic algorithm for multimodal function optimization. In: Proceeding of CIMCA/IAWTIC Conference, pp. 36–40 (2005)
57. Al-Mouhamed, M.A.: 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)
58. Kruatrachue, B., Lewis, T.: Grain size determination for parallel processing. Software IEEE **5**(1), 23–32 (1988)
59. Standard task graph set is available online at: http://www.kasahara.elec.waseda.ac.jp/schedule
60. Jin, S., Schiavone, G., Turgut, D.: A performance study of multiprocessor task scheduling algorithms. J. Supercomput. **43**, 77–97 (2008). doi:10.1007/s11227-007-0139-z
61. http://faculties.sbu.ac.ir/~moghaddam/index.php/main/page/9#tsk_skd