Available online at www.sciencedirect.com

ScienceDirect

Computers & Operations Research 35 (2008) 976–993

computers & operations research

www.elsevier.com/locate/cor

A comparison of multiprocessor task scheduling algorithms with communication costs

Reakook Hwang[a], Mitsuo Gen[b,*], Hiroshi Katayama[a]

[a]*Department of Industrial and Management Systems Engineering, Graduate School of Science and Engineering, Waseda University, Tokyo 169-8555, Japan*

[b]*Department of Information Architecture, Graduate School of Information, Production and System, Waseda University, Kitakyushu 808-0135, Japan*

Available online 11 July 2006

Abstract

Both parallel and distributed network environment systems play a vital role in the improvement of high performance computing. Of primary concern when analyzing these systems is multiprocessor task scheduling. Therefore, this paper addresses the challenge of multiprocessor task scheduling parallel programs, represented as directed acyclic task graph (DAG), for execution on multiprocessors with communication costs. Moreover, we investigate an alternative paradigm, where genetic algorithms (GAs) have recently received much attention, which is a class of robust stochastic search algorithms for various combinatorial optimization problems. We design the new encoding mechanism with a multi-functional chromosome that uses the priority representation—the so-called priority-based multi-chromosome (PMC). PMC can efficiently represent a task schedule and assign tasks to processors. The proposed priority-based GA has show effective performance in various parallel environments for scheduling methods.

© 2006 Elsevier Ltd. All rights reserved.

Keywords: Multiprocessor task scheduling; Genetic algorithm; Priority-based multi-chromosome (PMC)

1. Introduction

The utilization of parallel processing systems these days, in a vast variety of applications, is the result of numerous breakthroughs over the last two decades. The development of parallel and distributed systems has lead to there use in several applications including information processing, fluid flow, weather modeling, database systems, real-time high-speed simulation of dynamical systems, and image processing. The data for these applications can be distributed evenly on the processors of parallel and distributed systems, and thus maximum benefits from these systems can be obtained by employing efficient task partitioning and scheduling strategies.

The multiprocessor task scheduling problem considered in this paper is based on the deterministic model, which is the execution time of tasks and the data communication time between tasks that are assigned; and, the directed acyclic task graph (DAG) that represents the precedence relations of the tasks of a parallel processing system well known as the NP-complete problem. Many heuristic based methods and approaches to the task scheduling dilemma

* Corresponding author. Tel.: +81 93 692 5273; fax: +81 93 692 5021.
E-mail address: gen@waseda.jp (M. Gen).

0305-0548/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2006.05.013

have been proposed [1–7]. The reason for such proposals is because the precedence constraints between tasks can be non-uniform therefore rendering the need for a uniformity solution. We assume that the parallel processor system is uniform and non-preemptive; that is, task scheduling and allocation onto homogeneous multiprocessor systems where as each processor completes the current task before the new one starts its execution.

Recently, evolutionary approaches have been developed to solve this problem. For example, a genetic algorithms (GA) based approach can better locate a near optimal solution than a list schedule in most cases [8–10]. The proposed GA point of concern is a search algorithm based on the principles of evolution and natural genetics. In this case, the GA combines the exploitation of the past results with the new areas of space search exploration. By using survival of the fittest techniques, combined with a structured yet randomized information exchange, a GA can mimic some of the innovative attributes of a human search. Even if applied by several GAs for the multiprocessor scheduling problem, few resembling this problem have ever been published to be solved using the task graph with the communication cost.

However, for a more realistic problem, we may assume that communication delays between processors are possible. When two communicating tasks are mapped to the same processor the communication delay becomes zero because the data transfer is effective; but, when mapped to different processors the communication delay is represented. For this problem, we proposed the extension of the priority-based coding method as the priority-based multi-chromosome (PMC), which has been noted thus far as PMC—how to encode a problem formula into a chromosome which conditions all subsequent steps in genetic algorithms—a key issue at stake in this paper. The priority-based encoding [11] is the knowledge of how to handle the problem of producing encoding that can treat the precedence constraints efficiently. For the PMC to work adequately we design a new crossover method where weight mapping crossover (WMX) determines mapping relations by using their opposite string parts.

This paper is organized, as follows: In Section 2, *multiprocessor task scheduling problem* the general models of a DAG and its formulas are discussed. Section 3 presents, *related works* as traditional list scheduling algorithms. In Section 4, *the proposed genetic algorithm* we design a genetic algorithm for the multiprocessor scheduling problem in coding methods. In Section 5, *genetic operators* genetic operators are proposed. Section 6, *experimental results* shows the experimental results in comparison to other scheduling methods. We conclude this work in Section 7, *conclusions*.

2. Multiprocessor task scheduling problem

In the multiprocessor task scheduling problem, assigning priority to tasks is very important for both heuristic algorithms and search algorithms, has great influence over the scheduling result and the real parallel processing time. In most past methods, assigning priority takes account of only the processing time of each task. This is because including the communication time between tasks substantially increases the number of combinations to be searched as candidates for solutions. However, in practical cases of parallel executions the communication overhead should be considered if two tasks, that have a precedence relation, are mapped onto different processors. Therefore, it is expected during scheduling time to find a much better solution in the early stage of a search. This is done by assigning priority to each task that takes account of communication overhead and by allocating tasks on to available processors using the priorities. The problem with optimal task scheduling of a DAG, and with a parallel processing system using m processors, is the assignment of the computation tasks to processors in such a way that precedence relations are maintained; also, that all tasks are completed in the shortest possible time as presented in the following mathematical formulation (Fig. 1):

$$\begin{aligned} \min \quad & f = \max_j \{t_j\} \\ \text{s.t.} \quad & t_k - p_k - d_{jk} \geqslant t_j, \quad T_j \succ T_k \ \ \forall j,k, \\ & t_j \geqslant 0 \quad \forall j. \end{aligned}$$

In order to formulate an integrated model, the following indices and parameters are introduced:

Indices:

i: processor index, $i = 1,2,\ldots,m$
j,k: task index, $j,k = 1,2,\ldots, n$

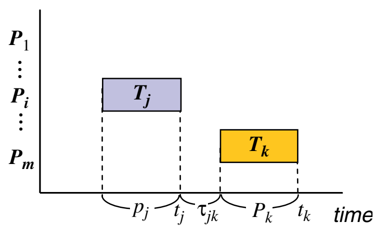


Fig. 1. Time chart of DAG.

Parameters:

f: makespan
n: number of tasks
m: number of processors
p_j: the processing time of task T_j
τ_{jk} : the communication time between T_j and T_k
$\succ$: represents a precedence relation between tasks; that is, $T_j \succ T_k$ means that T_j precedes T_k
pre(j): the set of predecessors of task T_j

$$d_{jk} = \begin{cases} \tau_{jk} & \text{if task } T_j \text{ and } T_k \text{ are assigned to different processor} \\ 0 & \text{otherwise} \end{cases}$$

Decision variables:

t_j: the completion time of task T_j

3. Related work

Here, three of the more traditional list scheduling heuristics that considered communication costs are introduced. There are many approaches that can be employed in static scheduling [12–14]. The basic idea is to make an ordered list of nodes by assigning them priorities, and then to repeatedly execute the following two steps until a valid schedule is obtained:

(1) Select from the list the node with the highest priority for scheduling.
(2) Select a processor to accommodate this node.

In more realistic cases, a scheduling algorithm needs to address a number of issues. It should exploit the parallelism by identifying the task graph structure and take into consideration task granularity, arbitrary computation, and communication costs.

The priorities are determined statically before the scheduling process begins. In the scheduling process, the node with the highest priority is chosen for scheduling. In the second step, the best possible processor, that is, the one which allows the earliest start time is selected to accommodate this one. The main problem with list scheduling an algorithm is that static priority assignment may not always order the nodes for scheduling according to their relative importance. Therefore, timely scheduling of the nodes can lead to a better schedule eventually. On the other hand, the drawback of a static approach is that an inefficient schedule may be generated if a relatively less important node is chosen for scheduling before the more important ones—static priority assignment may not capture the variation of the relative importance of nodes during the scheduling process. For example, consider the task graph shown in Fig. 2. Here, a schedule is produced using the highest levels first with estimated times (HLFET) algorithm [1], which determines the priority of a node by computing its level. The level of a node is the largest sum of computation costs along a path from the node to an exit node. The node with a higher level gets a higher priority. The HLFET algorithm schedules nodes in the order of: T_1, T_2, T_3, T_4. The schedule is shown in Fig. 3 in which all the nodes are scheduled to one processor;

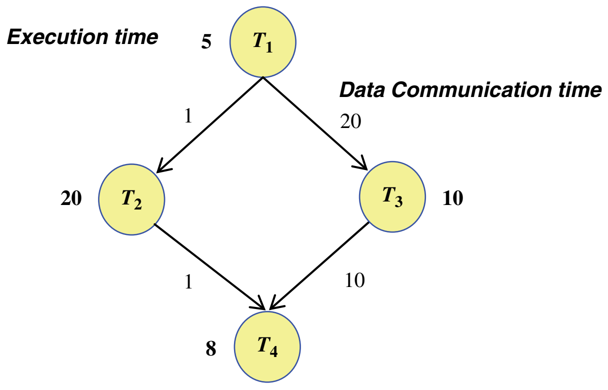


Fig. 2. DAG with communication costs.

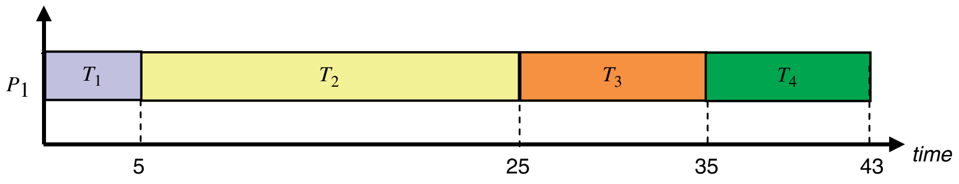


Fig. 3. Gantt Chart: the schedule generated by HLFET and MCP algorithm.

Table 1
Level value of task nodes

T_j	p_j	$ASAP_j$	$ALAP_j$	SL_j
1	5	0	0	53
2	20	6	24	29
3	10	25	25	28
4	8	45	45	8

SL_j: the static level of task T_j; CP: the critical path of task graph; $ASAP_j$: the as-soon-as-possible start time T_j; $ALAP_j$: the as-late-as-possible start time $T_j = CP - SL_j$.

the schedule length is 43 time units. However, the schedule length can be reduced, as shown in Fig. 5, if we schedule the nodes in the order of: T_1, T_3, T_2, T_4. In the second scheduling step, T_3 is a relatively more important node than T_2 because if it is not scheduled to the start time of T_4 it will be delayed due to the large communication costs along the path T_1, T_3, T_4. Thus, the HLFET algorithm does not precisely identify the most important node at each scheduling step because it orders nodes while assigning each of them a static attribute that does not depend on the communication among nodes.

Furthermore, this section illustrates three reported scheduling algorithms and their characteristics. These are, the modified critical path (MCP) algorithm [2], the dominant sequence clustering (DSC) algorithm [3], and the mobility directed (MD) algorithm [2].

3.1. Modified critical path (MCP) algorithm

The MCP algorithm is designed based on an attribute called the latest possible start time of a node, as shown in Table 1. A node's latest possible start time is determined through the as-late-as-possible (ALAP) binding. This is done by traversing the task graph upward from the exit nodes to the entry nodes and by pulling the nodes start times downwards as much as possible. The latest possible start time of the node itself is followed by a decreasing order of the latest possible start time of its successor nodes. The MCP algorithm then constructs a list of nodes in an increasing

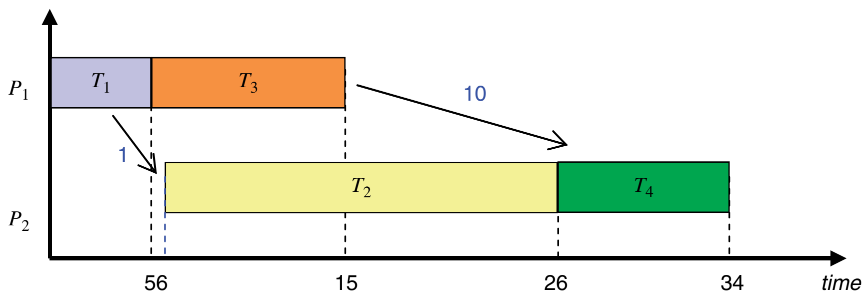


Fig. 4. Gantt Chart: the schedule generated by DSC algorithm.

lexicographical order of the latest possible start times' lists. At each scheduling step the first node is removed from the list and scheduled to a processor that allows for the earliest start time. The MCP algorithm was originally designed for a bounded number of processors.

The MCP algorithm assigns higher priorities to nodes which have the smaller latest possible start times. However, the MCP algorithm does not necessarily schedule nodes on the CP first. For example, consider the task graph in Fig. 2. There, the MCP algorithm schedules a node in the same order as the HLFET algorithm and hence generates the same schedule as shown in Fig. 3. The MCP algorithm does not assign node priorities accurately even though it takes communication among nodes into account when computing the priorities.

3.2. Dominant sequence clustering (DSC) algorithm

The DSC algorithm is based on an attribute called the dominant sequence, which is essentially the critical path of the partially scheduled task graph at each step. At each step, the DSC algorithm checks whether the highest CP node is a ready node. If the highest CP node is a ready node, the DSC algorithm schedules it to a processor that allows the minimum start time. Such a minimum start time may be achieved by rescheduling some of the node's predecessors to the same processor. On the other hand, if the highest CP node is not a ready node, the DSC algorithm does not select it for scheduling. Instead, the DSC algorithm selects the highest node that lies on a path reaching the CP for scheduling. The DSC algorithm schedules it to the processor that allows the minimum start time of the node, provided that such a processor selection will not delay the start time of an unscheduled CP node. The delay scheduling of the CP nodes allows the DSC algorithm to incrementally determine the next highest CP node.

Although the DSC algorithm can identify the most important node at each scheduling step, it does not schedule a CP node if it is not a ready node. However, delaying the scheduling of a CP node may prevent it from occupying an earlier idle time slot in the subsequent scheduling steps. Another deficiency of the DSC algorithm is that it uses more processors than necessary. It schedules a node to a new processor if its start time cannot be reduced by scheduling to any processor already in use. However, it is possible to save processors by scheduling nodes to processors already in use without degrading the schedule length. For the task graph in Fig. 2, the DSC algorithm generates the schedule shown in Fig. 4. The deficiencies mentioned above are not revealed by this example.

3.3. Mobility directed (MD) algorithm

The MD algorithm selects a node at each step for scheduling based on an attribute called relative mobility. Mobility of a node is defined as the difference between a node's earliest start time and latest start time. Similar to the ALAP binding mentioned in MCP algorithm, the earliest possible start time is assigned to each node through the as-soon-as-possible (ASAP) binding, which is done by traversing the task graph downward from the entry nodes to the exit nodes and by pulling the nodes upward as much as possible. Moreover, relative mobility is obtained by dividing the mobility with the node's computation cost. Essentially, a node with zero mobility is a node on the CP; at each step, the MD algorithm schedules the node with the smallest mobility to the first processor which has a large enough time slot to accommodate the node without considering the minimization of the node's start time. After a node is scheduled, all the relative mobility is updated.

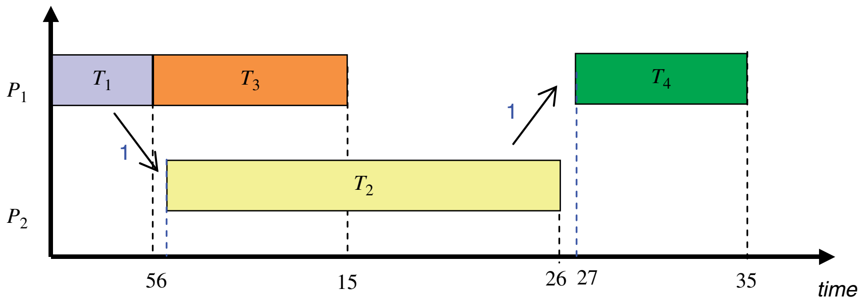


Fig. 5. Gantt Chart: the schedule generated by MD algorithm.

As opposed to the MCP algorithm, the MD algorithm determines node priorities dynamically. Although the MD algorithm can correctly identify the CP nodes for scheduling at each step, the selection of a suitable time slot and a processor are not done properly. The major problem with the MD algorithm is that it pushes scheduled nodes downwards to create a large enough time slot to accommodate a new node without paying any regard to the degradation of the schedule length. It may occur that pushing down the nodes might increase the final schedule length. Another problem with the MD algorithm is that it inserts a node into an idle time slot on a processor without considering whether the descendants of that node can be scheduled in a timely manner. The schedule generated by the MD algorithm for the task graph in Fig. 2 is shown in Fig. 5. When node T_4 is considered, it is found that there is a time slot on P_1 large enough to accommodate it. The MD algorithm schedules T_4 to P_1 without considering other processors. As a result, a longer schedule length is obtained.

4. The proposed genetic algorithm

In general, multiprocessor and heuristic algorithms have been proposed to obtain optimal and sub optimal solutions to the multiprocessor scheduling problem. To develop powerful algorithms for difficult optimization problems there has been increasing interest in imitating human beings since the 1960s. A term now in common use to refer to such techniques is evolutionary computation. The best known algorithms include genetic algorithms that are stochastic search algorithms, which mimic the process of natural selection and genetics. Developed by John Holland at the University of Michigan, the original goals were to study the adaptive behavior of natural systems and to design software systems with adaptive behavior. Recently, genetic algorithms have received growing interest as effective tools in problem solving and optimization [11,15].

In general GAs, string is a candidate solution for a problem. Therefore, a string should represent a complete schedule. A string should be feasible, all the tasks in a DAG should be scheduled and the schedule should satisfy the precedence relations. In order that normal binary coding would not be suited for this GA of multiprocessor scheduling problem with communication costs. It has a set of rules which must be obeyed:

1. A task's predecessors must have completed their execution before they can initialize executing.
2. All tasks within the task graph must execute at least once. Then this representation eliminates the need to consider the precedence relations between the computational tasks.

We have to design a proper string representation coupled with a crossover operation and mutation operation. Therefore, we design a new chromosome that not only represents a complete schedule but also represents assigning processors, which we label as the PMC.

4.1. Priority-based multi-chromosome (PMC)

For the multiprocessor scheduling problem, there were several approaches [8,10,16] that solved the multiprocessor scheduling problem by using genetic algorithms, which have some weakness with coding methods. How to encode a solution of the problem into a chromosome is a key issue for the genetic algorithm. It has been investigated from

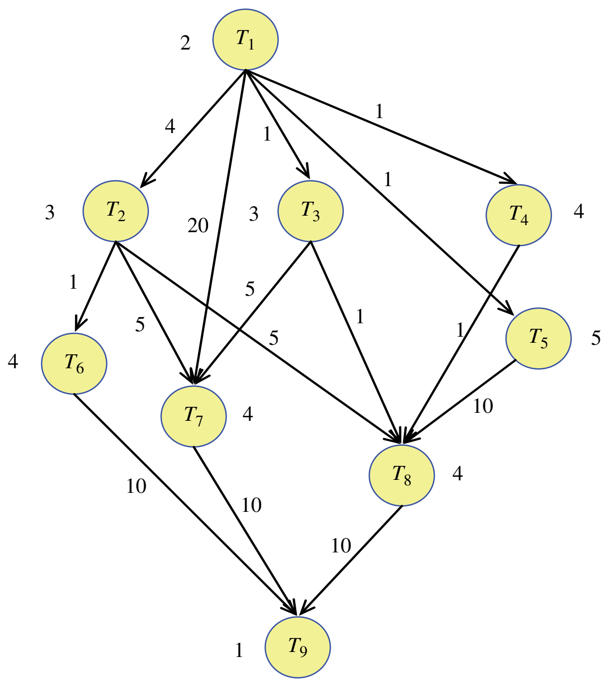


Fig. 6. Example DAG with nine tasks.

different angles, such as mapping characters from genotype space to phenotype space when individuals are decoded into solutions and metamorphosis properties when individuals are manipulated by genetic operators.

Attempting to solve the numerous problems in the industrial engineering sphere, it is nearly impossible to layout the pattern of solutions with binary encoding. During the last 10 years, various encoding methods have been created for particular problems in order to effectively implement genetic algorithms.

Normal binary encoding, as mentioned earlier, does not work effectively for the multiprocessor scheduling problem where connection was arbitrarily executed. These are some difficulties:

1. In the case of consideration for multiprocessors, there will be representation of a lot of chromosomes for one schedule.
2. In the case of a large number of tasks, the crossover and mutation operators will be difficult to work with precedence relations among tasks.

To overcome these difficulties we proposed the extension of the PMC that strings a present task priority of task nodes with the mapping processor simultaneously. For example, Fig. 7 illustrates one of the simple PMC that represent nine tasks and two processors with DAG (Fig. 6). The number 2 presents two types; first it represents the priority of task nodes that have higher priority to be assigned to the processor. Second, the number 2 presents the allocation for a target processor. Here, we consider two processors so that we are able to acquire the value 1 or 2; if we get the value 1, it will assign to processor 1; and, if we get the value 2, it will assign to processor 2 (Figs. 6 and 7).

4.2. Priority-based encoding/decoding

How to make an encoding which can treat the precedence constraint efficiently is a critical step and conditions all subsequent steps. The proposed encoding method is based on the priority of the task. The following procedure explains the generation of the initial PMC:

First, we input the scope numbers randomly which depend on the processor number. This encoding example uses two processors with the simple DAG (Fig. 6). The next step, randomly select two nodes and swap them as $n/2$ times. Then we can obtain one of the simple chromosomes as shown in Fig. 8.

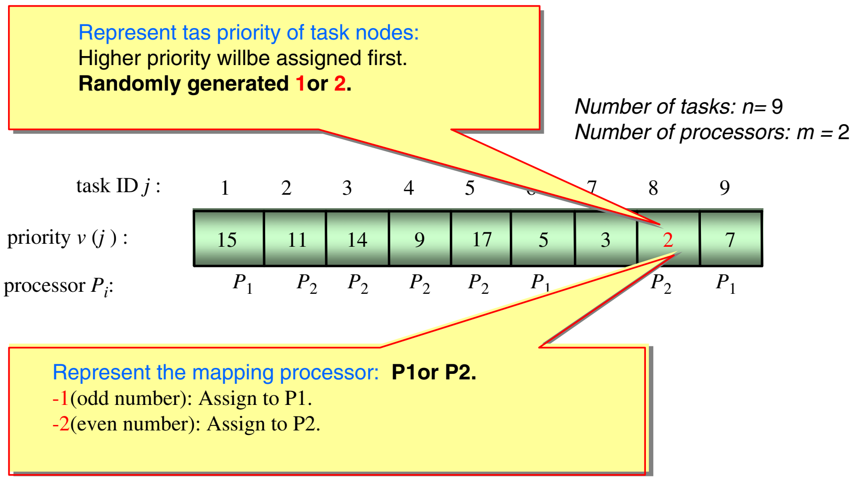

Represent tas priority of task nodes:
Higher priority willbe assigned first.
Randomly generated 1or 2.
Number of tasks: n= 9
Number of processors: m = 2
task ID j : 1 2 3 4 5 6 7 8 9
priority v (j) : 15 11 14 9 17 5 3 2 7
processor P_i: P1 P2 P2 P2 P2 P1 P2 P1
Represent the mapping processor: P1or P2.
-1(odd number): Assign to P1.
-2(even number): Assign to P2.

Fig. 7. Example priority-based multi-chromosome (PMC).

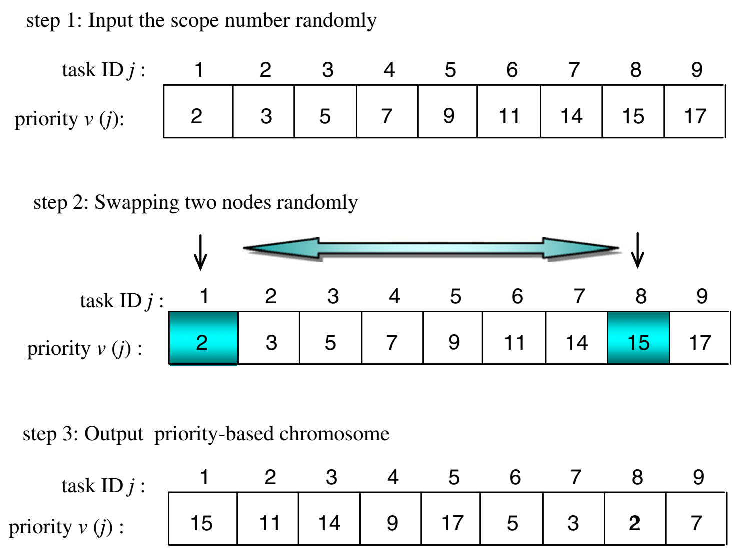

step 1: Input the scope number randomly
task ID j : 1 2 3 4 5 6 7 8 9
priority v (j): 2 3 5 7 9 11 14 15 17
step 2: Swapping two nodes randomly
task ID j : 1 2 3 4 5 6 7 8 9
priority v (j) : 2 3 5 7 9 11 14 15 17
step 3: Output priority-based chromosome
task ID j : 1 2 3 4 5 6 7 8 9
priority v (j) : 15 11 14 9 17 5 3 2 7

Fig. 8. Priority-based encoding procedure.

```
procedure 1: priority-basedencoding
input: number of processors $m$, number of tasks $n$
output: chromosome $v(j)$
begin
    for $j = 1$ to $n$
        $v(j) \leftarrow m * j$-random$[0, m-1]$;
    end
    for $i = 1$ to $\lfloor n/2 \rfloor$
        $j \leftarrow$ random$[1, n]$;
        $k \leftarrow$ random$[1, n]$;
        if $j \neq k$ then
```

```
            swap $\{v(j), v_k(k)\}$;
end
        output: the chromosome $v(j)$;
end
```

Suppose we want to assign n tasks to m processors by using the PMC (Fig. 8) then we will use the simple DAG as Fig. 6. Firstly, we try to find the first schedule. T_1 is the just eligible task for this position. And T_2, T_3, T_4 and T_5 are eligible nodes for the position, which are suitable for the next node. The priorities of them are 11, 14, 9 and 17, respectively. Then the task T_5 has the highest priority and is put into the task sequence *TS*. The third possible nodes are T_2, T_3 and T_4 and they have an 11, 14 and 9, priority, respectively, then we put T_3 into the task sequence *TS*. Finally, we repeat these steps until the task sequence: $TS = (T_1, T_5, T_3, T_2, T_4, T_6, T_7, T_8, T_9)$ is completed as in following the decoding procedure.

```
procedure 2: priority-based decoding (one task sequence growth)
input: number of tasks $n$, chromosome $v(j)$, the set of nodes $\bar{S}$
output: task sequence TS
begin
    $\bar{S} \leftarrow \emptyset, TS \leftarrow \emptyset$;
    $n \leftarrow 0, j \leftarrow 0$;
    while $(j \leqslant n)$ do
        $\bar{S} \leftarrow \bar{S} \cup suc_j$;
        $j^* \leftarrow \mathbf{arg\,max}\{v(j) | j \in \bar{S}\}$;
        $\bar{S} \leftarrow \bar{S} \setminus j^*$;
        $TS \leftarrow TS \cup j^*$;
        $j \leftarrow j^*$;
end
    output: task sequence TS;
end
```

In the next step, we assign tasks to processors from the above task sequence *TS*, which also corresponds to the value of priority $v(_j)$. In the example, just two processors are used, so we can check if the $v(_j)$ is the even number or the odd number. It assigns to processor P_2 and another will be mapped to P_1, if the value of the priority has the even number. Then we obtain the schedule for two processors as follows:

$$S = \{P_1 : (T_1, T_5, T_2, T_4, T_6, T_7, T_9), P_2 : (T_3, T_8)\}.$$

Note: In assigning this step we must check the precedence relationship and add in the communication delay (τ_{jk}) among tasks that are assigned to a different processor. Table 2(a) and (b) describes the trace table for an overall coding procedure. It can be generated through one of the feasible schedules shown in Fig. 9.

```
procedure 3: priority-based decoding (assigning tasks to processors)
input: processing time $p_k$, task sequence TS,
chromosome $v(j)$, the communication delay $\tau_{jk}$
output: makespan f, schedule S
begin
   $P_i \leftarrow 0, i = 1, 2, \ldots, m$;
   $t_k \leftarrow 0, k = 1, 2, \ldots, n$;
   $s \leftarrow 0$;
   for $j = 1$ to $n$
   $s \leftarrow TS_i$;
   $P_i \leftarrow v(s)\%m$;
   if $P_i = 0$ then $P_i \leftarrow m$;
```

Table 2
Trace table of example for DAG (Fig. 6)

(a)

j	$\bar{S}$	$v(j)$	j^*	TS
0	{1}	$v(1)=15$	1	$S=\{1\}$
2	{2, 3, 4, 5}	$v(2)=11$, $v(3)=14$, $v(4)=9$, $v(5)=17$	5	$S=\{1,5\}$
1	{2, 3, 4}	$v(2)=11$, $v(3)=14$, $v(4)=9$	3	$S=\{1,5,3\}$
3	{2, 4}	$v(2)=11$, $v(4)=9$	2	$S=\{1,5,3,2\}$
5	{4, 6, 7}	$v(4)=9$, $v(6)=5$, $v(7)=3$	4	$S=\{1,5,3,2,4\}$
4	{6, 7, 8}	$v(6)=5$, $v(7)=3$, $v(8)=2$	6	$S=\{1,5,3,2,4,6\}$
7	{7, 8}	$v(7)=3$, $v(8)=2$	7	$S=\{1,5,3,2,4,6,7\}$
9	{8}	$v(8)=2$	8	$S=\{1,5,3,2,4,6,7,8\}$
6	{9}	$v(9)=7$	9	$S=\{1,5\,3,2,4,6,7,8,9\}$

(b)

j^*	S	P_i	$t_j=e_j+p_j$
1	$P_1=\{1\}$, $P_2=\{\}$	1	$t_1=0+2=2$
5	$P_1=\{1,5\}$, $P_2=\{\}$	1	$t_5=2+5=7$
3	$P_1=\{1,5\}$, $P_2=\{3\}$	2	$t_3=3+3=6$
2	$P_1=\{1,5,2\}$, $P_2=\{3\}$	1	$t_2=7+3=10$
4	$P_1=\{1,5,2,4\}$, $P_2=\{3\}$	1	$t_4=10+4=14$
6	$P_1=\{1,5,2,4,6\}$, $P_2=\{3\}$	1	$t_6=14+4=18$
7	$P_1=\{1,5,2,4,6,7\}$, $P_2=\{3\}$	1	$t_7=18+4=22$
8	$P_1=\{1,5,2,4,6,7\}$, $P_2=\{3,8\}$	2	$t_8=17+4=21$
9	$P_1=\{1,5,2,4,6,7,9\}$, $P_2=\{3,8\}$	1	$t_9=31+1=32$

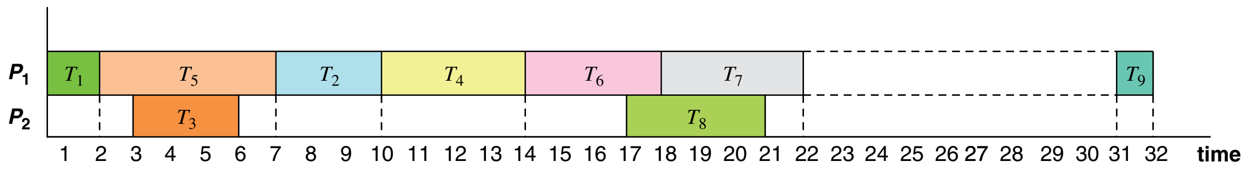


Fig. 9. Gantt Chart: the schedule generated by example of DAG (Fig. 6).

```
  **if** assigned task $T_j \prec T_k$ **then**
    **if** $P_i \neq P_l$ **then**
      $e_k \leftarrow \max\{t_j | j \in \text{pre}(k)\} + \tau_{jk}$;
      $t_k \leftarrow e_k + p_k$;
      $S \leftarrow S \cup S_k\{P_i; e_k + p_k\}$;
    else $e_k \leftarrow \max\{t_j | j \in pre(k)\}$;
      $t_k \leftarrow e_k + p_k$;
      $S \leftarrow S \cup S_k\{P_i; e_k + p_k\}$;
    else $e_k \leftarrow \max\{t_j | j \in \text{pre}(k)\}$;
      $t_k \leftarrow e_k + p_k$;
      $S \leftarrow S \cup S_k\{P_i; e_k + p_k\}$;
**end**
**output:** schedule $S$
    makespan $f \leftarrow \mathbf{max}\{t_k, k = 1, 2, \ldots, n\}$;
**end**
```

4.3. Evaluation function

The evaluation function is essentially the objective function for the problem. It provides a means of evaluating the search nodes, and it also controls the selection process. For the multiprocessor scheduling problem we can consider factors, such as throughput, finishing time and processor utilization for the evaluation function. Here, the calculation of the evaluation function is quite simple. First, the Gantt Chart of each string is calculated. The length of each processor string is measured to find the total finishing time of the schedule. The evaluation function used for our algorithm is based on the makespan (*f*) of the schedule. We convert the minimization problem to the maximization problem, that is, the applicable evaluation function is as follows:

$$eval(v_k) = 1/f^k, \quad k = 1, \ldots, popSize,$$

where f^k the makespan of the *k*th chromosome.

5. Genetic operators

5.1. Crossover

Crossover procedures produce new ‘individuals’ that have portions of both of the parent’s genetic material. Here we design the new crossover method, the position-based crossover operator, where the (WMX) is suitable for the PMC. It can be viewed as a two-point crossover of binary string and as a remapping by referring to the parent binary string, as is done in the following procedure and shown in Fig. 10.

```
procedure 4: Weight mapping crossover (WMX)
Input: parent: $v_1$, $v_2$, number of tasks $n$
output: offspring: $v'_1$, $v'_2$
begin
    $s \leftarrow$ random$[1, n-1]$;
    $t \leftarrow$ random$[s+1, n]$;
    $l \leftarrow t - s$;
    for $i = 1$ to $l$
        $s_1[i] \leftarrow v_1[s+i]$;
        $s_2[i] \leftarrow v_2[s+i]$;
    $s_1[.] \leftarrow$ sorting$(s_1[.])$;
    $s_2[.] \leftarrow$ sorting$(s_2)[.])$;
    $v'_1 \leftarrow v_1[1 : s-1]//v_2[s : t]//v_1[t+1 : n]$;
    $v'_2 \leftarrow v_2[1 : s-1]//v_1[s : t]//v_2[t+1 : n]$;
    for $i = 1$ to $l$
        for $j = 1$ to $l$
            if $v'_1[s+i] = s_2[j]$ then
                $v'_1[s+i] \leftarrow s_1[j]$;
        for $j = 1$ to $l$
            if $v'_2[s+i] = s_1[j]$ then
                $v'_2[s+i] \leftarrow s_2[j]$;
    output: offspring: $v'_1$, $v'_2$;
end
```

First, we randomly select two cutting points as a substring and then check the value of their contents and order. For instance, a selected substring of parent v_1 has the order of weight (higher value) as: 1-2-3-4. This order of weight is used to change a selected substring for parent v_2, so substring 6-13-15-11 is changed to be 15-13-11-6. And, parent v_2 also refers to the order of weight in substring parent v_1. WMX does not correspond with a general crossover that merely swaps contents for two selected parents; it changes their order of weight so that it will always represent a

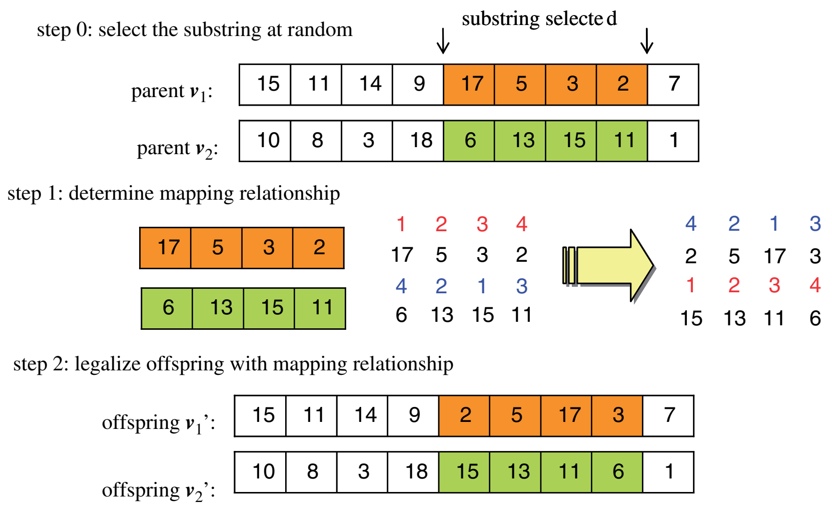


Fig. 10. llustration of the WMX operator.

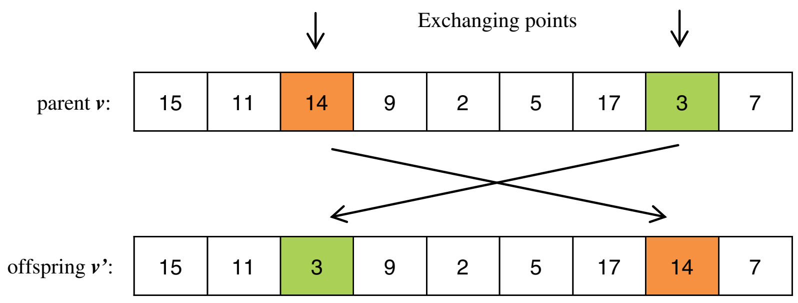


Fig. 11. llustration of the swap mutation operator.

feasible chromosome; (if two parents swap contents of a selected string they will have the same value in the same chromosome).

5.2. Mutation

This is the application of the swap mutation operator, where two positions are selected at random and their contents are swapped as shown in Fig. 11. Random swapping is used as a mutation operation, which is swapping two adjacent operands, or two adjacent operators. Mutation operation performed in this way may also guarantee generation of legal offspring. The overall procedure is shown below:

```
procedure: Swap Mutation
input: chromosome $v$ , $l$
output: chromosome $v'$
begin
    $i \leftarrow$ random$[1 : l - 1]$;
    $j \leftarrow$ random$[i + 1 : l]$;
    $v' \leftarrow v[1 : i - 1]//v[j]//v[i + 1 : j - 1]$
        $//v[i]//v[j + 1 : l]$;
    output: offspring $v'$;
end
```

As shown, applying the swap mutation operator allows the PMC to generate the new offspring that generates the new schedule, and also the new target processor.

5.3. Selection

We use the roulette wheel selection [17] that is a method to reproduce a new generation proportional to the fitness of each individual. In this procedure, the solutions are placed on a roulette wheel where the section of the wheel for a better solution is larger than for a poorer solution.

It is called the roulette wheel selection because the selection technique of the parent selection is that each individual is given a chance to become a parent in proportion to its fitness evaluation; the best chances of selecting a parent can be produced by a spinning roulette wheel with the size of its slots for each parent being proportional to their fitness. Obviously those with the largest fitness (and slot sizes) have more of a chance to be chosen, which is, needless to say, similar to living beings and the notion of the survival of the fittest.

Roulette wheel selection [15] can be constructed as follows:

1. Calculate the fitness value $eval(v_k)$ for each chromosome v_k:

$$eval(v_k) = f(x), \quad k = 1, \ldots, popSize.$$

2. Calculate the total fitness for the population:

$$F = \sum_{k=1}^{popSize} eval(v_k).$$

3. Calculate selection probability p_k for each chromosome v_k:

$$p_k = \frac{eval(v_k)}{F}, \quad k = 1, \ldots, popSize.$$

4. Calculate cumulative probability q_k for each chromosome v_k:

$$q_k = \sum_{j=1}^{k} p_j, \quad k = 1, \ldots, popSize.$$

The selection process begins by spinning the roulette wheel *popSize* times; each time, a single chromosome is selected for a new population in the following procedure:

```
procedure: Roulette wheel selection
step 1: Generate a random number $r$ form the range [0, 1];
step 2: If $r \leqslant q_1$, then select the first chromosome $v_1$;
otherwise, select the kth chromosome $v_k (v_k \leqslant k \leqslant popSize)$ such that $q_{k-1} < r \leqslant q_k$.
```

In other words, the algorithm uses a random number to select one of the sections with a probability equal to its area.

6. Experimental results

The genetic representation with the method of priority-based coding was discussed in the previous section. We proposed a new representation technique as we labeled it PMC. To evaluate our proposed new priority-based coding method we use a different size of problem in this section: *Experimental problem* 1 with nine task graph (Fig. 6), *Experimental problems* 2 and 3 with use of Gaussian elimination method graphs.

In *Experimental problems* 1 and 2 we compare our priority-based genetic approach with previous list scheduling heuristic algorithms on task graphs (Figs. 6 and 12). We summarize the experimental results [best solution: best result from the 20 times iterations, processor efficiency (%): processor efficiency- actually task execution time/total processing time (used processors*makespan), time: time performance] as shown in Tables 3 and 4. Also, our approach compares with previous genetic algorithms in *Experimental problem* 3 and results are show in Fig. 14.

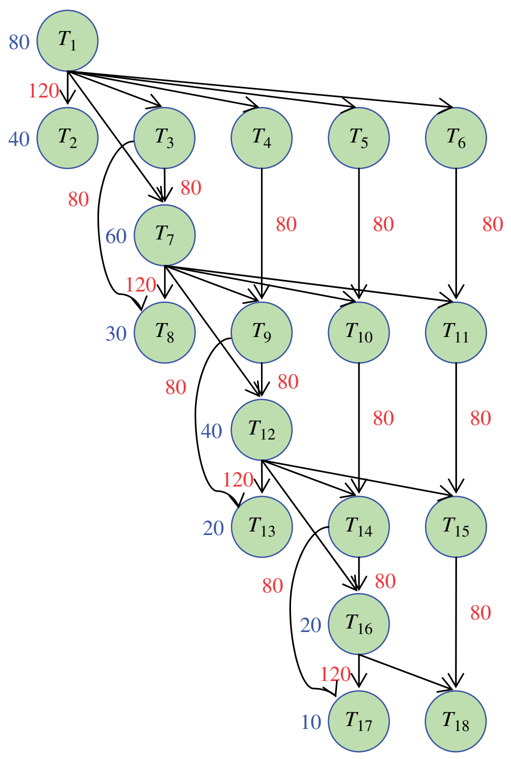


Fig. 12. Example DAG with 18 tasks.

Table 3
Comparative results with nine tasks (Fig. 6)

Algorithms	MCP	DSC	MD	DCP	
No. processor	3	4	2	2	2
Best solution	29	27	32	32	23
Processor efficiency (%)	34.48	27.8	46.9	46.9	65.2
Time (time units)	72	61	81	92	73

Table 4
Comparative results for DAG with 18 tasks (Fig. 12)

Algorithms	MCP	DSC	MD	DCP	
No. processor	4	6	3	3	2
Best solution	520	460	460	440	440
Processor efficiency (%)	28.8	21.7	43.5	45.5	68.2
Time (time units)	662	540	615	732	620

The genetic algorithm applied the following parameters throughout the simulations:

- Population size: $popSize = 100$
- Maximum generation: $maxGen = 1000$
- Crossover probability: $p_C = 0.7$
- Mutation probability: $p_M = 0.3$
- Terminating condition: 100 generations with same fitness.

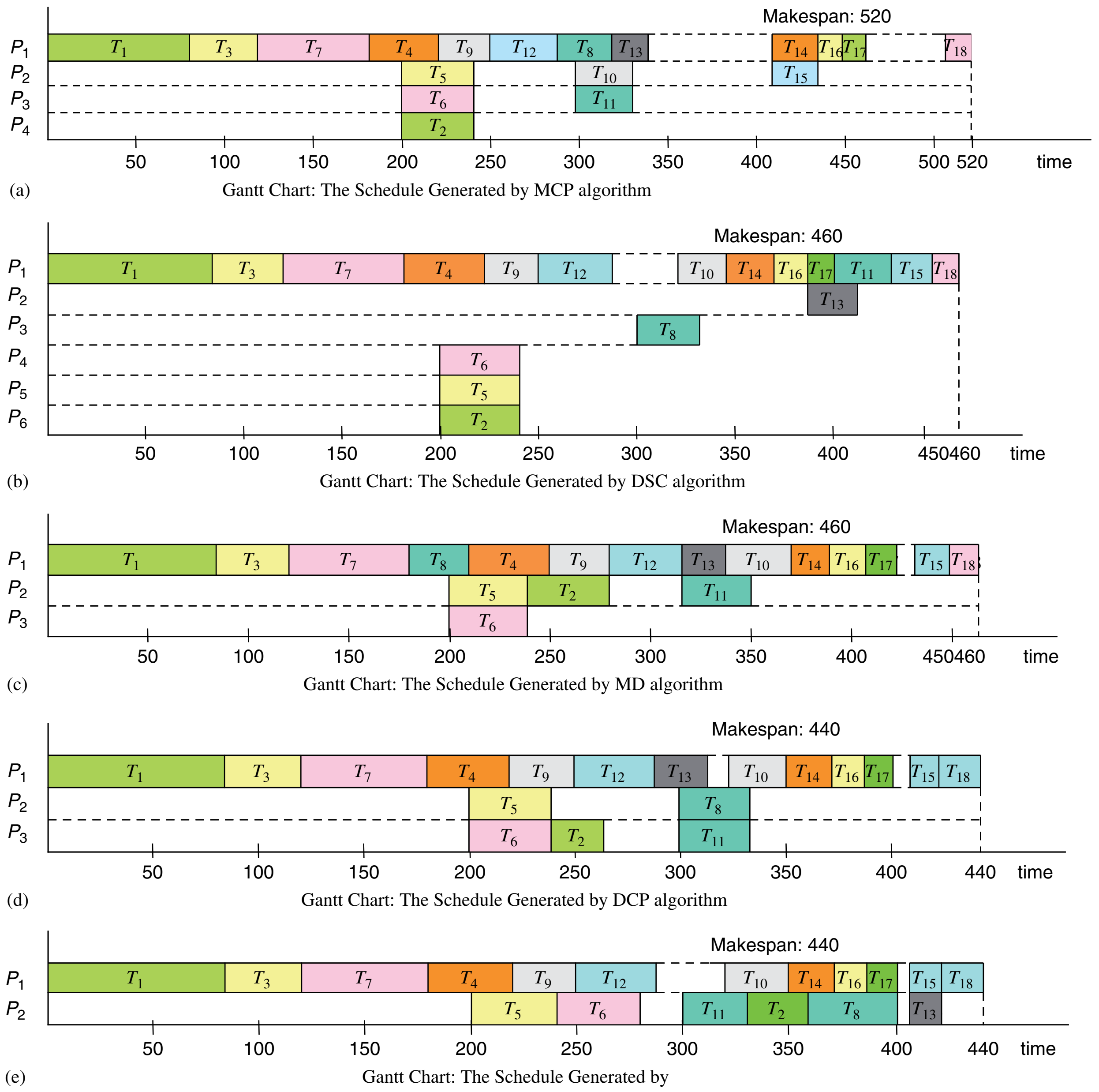


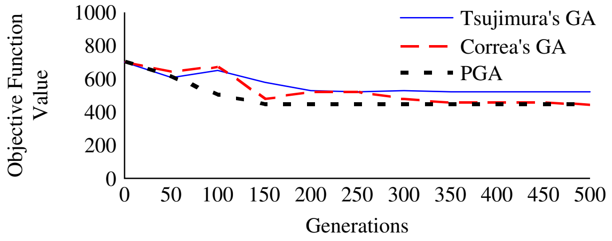
Fig. 13. Gantt chart for checking the result with example DAG with 18 tasks (Fig. 12). (a) Gantt Chart: the schedule generated by MCP algorithm. (b) Gantt Chart: the schedule generated by DSC algorithm. (c) Gantt Chart: the schedule generated by MD algorithm. (d) Gantt Chart: the schedule generated by DCP algorithm. (e) Gantt Chart: the schedule generated by proposed genetic algorithm.

Test algorithms:

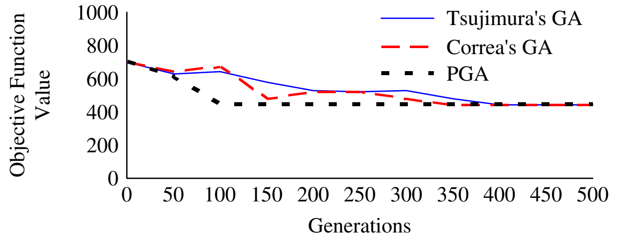
MCP (modified critical path) by Wu and Gajski, DSC (dominant sequence clustering) by Yang and Gerasoulis, MD (mobility directed) by Wu and Gajski, DCP (dynamic critical path) by Kwok and Ahmad, Tsujimura's GA by Tsujimura and Gen, Correa's GA by Correa, Ferreira and Rebreyend and (proposed genetic algorithm).

Experimental problem 1:

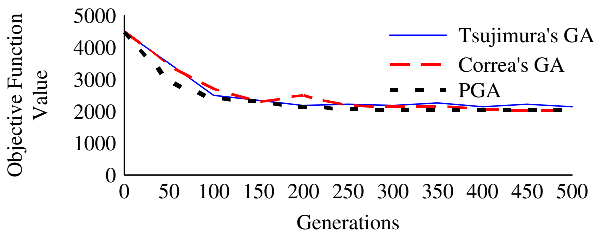
Experimental problem 2: We evaluate the effectiveness of our proposed algorithm for a multiprocessor system. The second example task graph was proposed by Kwok and Ahmad whose data are useful for comparative study. They were compared with some of the list scheduling heuristic algorithms by using DAG with 18 tasks (Fig. 12). For checking the result of schedules, we illustrate the Gantt Chart with three algorithms and our proposed genetic algorithm as Fig. 13.



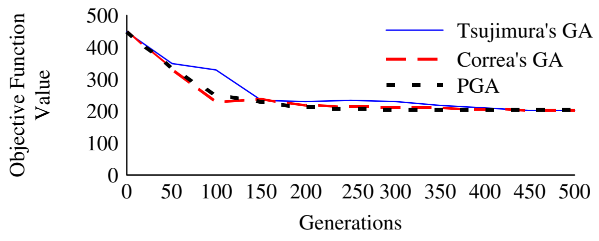
(a) Task: 18, Processor: 2



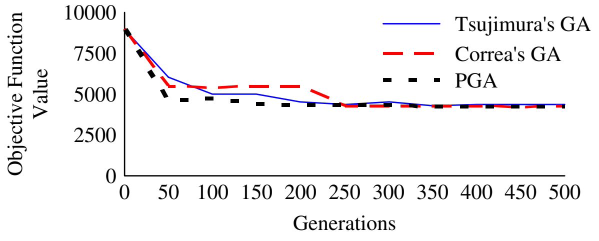
(b) Task: 18, Processor: 5



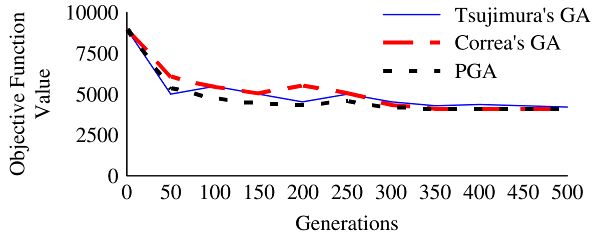
(c) Task: 52, Processor: 3



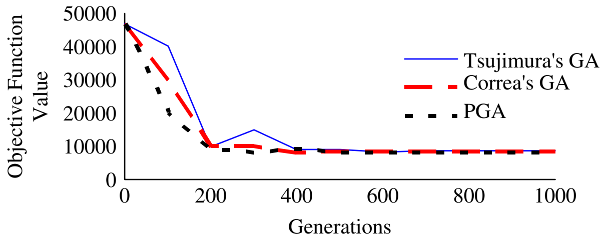
(d) Task: 52, Processor: 6



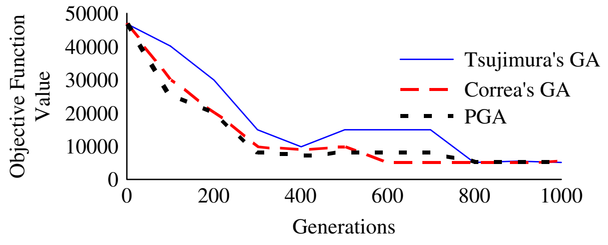
(e) Task: 102, Processor: 4



(f) Task: 102, Processor: 8



(g) Task: 1032, Processor:5



(h) Task: 1032, Processor:10

Fig. 14. Comparison of the three algorithms for the Gaussian elimination graph: (a)–(h). (a) Task: 18, Processor: 2. (b) Task: 18, Processor: 5. (c) Task: 52, Processor: 3. (d) Task: 52, Processor: 6. (e) Task: 102, Processor: 4. (f) Task: 102, Processor: 8. (g) Task: 1032, Processor: 5. (h) Task: 1032, Processor: 10.

Experimental Problem 3: In the last experimental test our proposed algorithm is compared with two of previous genetic algorithms. Fig. 14 presents the results of our experiments. In these figures, the *X*-axis represents the number of generations and the *Y*-axis represents the objective function value as makespan. We use following data set:

No. of tasks	Total task execution time	No. of processors	
18	600	2	5
52	3120	3	6
102	8840	4	8
1032	302 720	5	10

As the figures show, our performs well when the number of tasks and processors are small. And, when the number of tasks and processors are increased our performs as well as, or slightly better than the other two algorithms. This is because our encoding and decoding procedures are simple and can find an optimal solution rapidly.

7. Conclusions

In this paper, we described the multiprocessor scheduling problem based on the deterministic model, that is, the execution time of tasks and the communication costs between tasks. The difficulty of the problem is the assumption that communication costs between processors are not negligible.

We proposed the extension of priority-based coding method by using the priority-based multi-chromosome (PMC), which is key issue of how to encode a problem solution into a chromosome that conditions all subsequent steps in genetic algorithms. Our proposed priority-based encoding will handle the problem how to produce encoding that can treat the precedence constraint efficiently. For the PMC to work efficiently, we designed a new crossover method—weight mapping crossover (WMX).

In a numerical experiment, we used several task graphs for the multiprocessor scheduling problem, and we compared our proposed PMC and WMX relative to a genetic approach with various list scheduling heuristic algorithms and previous GA approaches. From the comparative result it is confirmed that the genetic algorithm can provide workable solutions, and it can find the optimum result rapidly. In comparison with the former GA, our performed as well as, or slightly better than the other two algorithms. The simulation has shown that it may be possible to solve generally formulated problems as well as tightly defined problems. Lastly, and most importantly, it can be applied to other distribution scheduling problems.

References

[1] Adam TL, Chandy KM, Dicksoni JR. A comparison of list schedules for parallel processing systems. Communications of the ACM 1974;17(12):685–90.

[2] Wu MY, Gajski DD. Hypertool: a programming aid for message-passing systems. IEEE Transactions on Parallel and Distributed Systems 1990;1(3):330–43.

[3] Yang T, Gerasoulis A. DSC: scheduling parallel tasks on an unbounded number of processors. IEEE Transactions on Parallel and Distributed Systems 1994;5(9).

[4] Correa RC, Ferreira A, Rebreyend P. Scheduling multiprocessor tasks with genetic algorithms. IEEE Transactions on Parallel and Distributed Systems 1999;10(8):825–37.

[5] Thanalapati T, Dandamudi S. An efficient adaptive scheduling scheme for distributed memory multicomputers. IEEE Transactions on Parallel and Distributed Systems 2001;12(7):758–68.

[6] Nissanke N, Leulseged A, Chillara S. Probabilistic performance analysis in multiprocessor scheduling. Journal of Computing and Control Engineering 2002;13(4):171–9.

[7] Corbalan J, Martorell X, Labarta J. Performance-driven processor allocation. IEEE Transactions on Parallel and Distributed Systems 2005;16(7):599–611.

[8] Hou ESH, Ansari N, Hong R. A genetic algorithm for multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems 1994;5(2):113–20.

[9] Hwang RK, Gen M. Multiprocessor scheduling using genetic algorithm with priority-based coding. Proceedings of IEEJ conference on electronics, information and systems; 2004.

[10] Wu AS, Yu H, Jin S, Lin K-C, Schiavone G. An incremental genetic algorithm approach to multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems 2004;15(9):824–34.

[11] Gen M, Cheng R. Genetic algorithm and engineering optimization. New York: Wiley; 2000.

[12] Hwang JJ, Chow Y-C, Anger FD, Lee C-Y. Scheduling precedence graphs in systems with inter-processor communication times. SIAM Journal on Computing 1989;8(2):244–58.

[13] Kasahara H, Narita S. Parallel processing of robot–arm control computation on a multimicroprocessor system. IEEE Transactions of Robotics and Automation 1985;RA-1(2):104–13.

[14] Kwok YK, Ahmad I. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. IEEE Transactions on Parallel and Distributed Systems 1996;7(5).

[15] Gen M, Cheng R. Genetic algorithms and engineering design. New York: Wiley; 1997.

[16] Tsujimura Y, Gen M. Genetic algorithms for solving multiprocessor scheduling problems. In: Simulated evolution and learning. Heidelberg: Springer; 1995. p. 106–15.

[17] Holland J. Adaptation in natural and artificial systems. Ann Arbor: University of Michigan Press; 1975.

Further reading

[18] Palis MA, Lieu JC. Task clustering and scheduling for distributed memory parallel architectures. IEEE Transactions on Parallel and Distributed Systems 1996;7(1):46–55.
[19] Yang T. Scheduling and code generation for parallel architectures. PhD thesis. Rutgers University, NJ, USA; 1993.