A Modified Genetic Algorithm for Task Scheduling in Multiprocessor Systems

Yi-Hsuan Lee and Cheng Chen
Department of Computer Science and Information Engineering
National Chiao Tung University, Hsinchu, Taiwan, R.O.C.
{yslee, cchen}@csie.nctu.edu.tw

Abstract

The impressive proliferation in the use of multiprocessor systems these days in a great variety of applications is the result of many breakthroughs over the last two decade. In these multiprocessor systems, an efficient scheduling of a parallel program onto the processors that minimizes the entire execution time is vital for achieving a high performance. This problem is exactly known very hard to solve, so many heuristic methods are designed to obtain near-optimal solutions. Genetic Algorithms are widely used to solve this problem, which are quite effective but not efficient enough. Therefore, we propose a modified genetic algorithm to overcome this drawback and construct a simulation and evaluation environment to evaluate it. Our method is called Partitioned Genetic Algorithm (PGA), *which integrates the concept of* Divide-and-Conquer *mechanism to partition the entire problem into subgroups and solve them individually. According to our experimental results, PGA can not only dramatically decrease the time doing scheduling, but also obtain similar performances as original genetic algorithms, sometimes it is even better.*

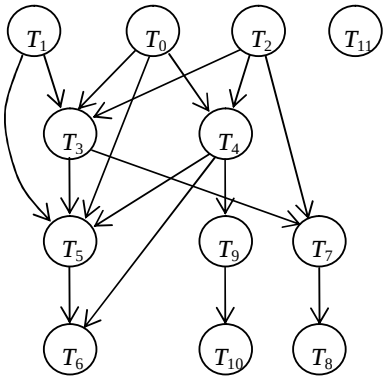
1 Introduction

With many breakthroughs such as device technology, theory, computer architectures, and software tools, multi- processor systems are used in a great variety of applications [1]. In these multiprocessor systems, scheduling is a major issue in its operation, which is also an important problem in other areas such as manufacturing, process control, economics, operation research, etc. [2]. Basically, scheduling is to simply allocate a set of tasks to resources such that the optimum performance is obtained. However, it is known to be NP-complete for the general case and even for many restricted cases [3]. Therefore, scheduling is usually handled by heuristic methods which provide reasonable solutions of the problem.

Multiprocessor scheduling methods can be divided into list heuristics and meta-heuristics [4-5]. List heuristics assign each task a priority and sort them in decreasing order [8-9]. As processors become available, the task with the highest priority is selected and allocated to the most suited processor. Most of them are efficient but often can't obtain reasonable solutions in all situations.

Meta-heuristics, known as *Genetic Algorithms*, is a guided random search method which mimics the principles of evolution and natural genetics [7]. Because genetic algorithms search optimal solutions from entire solution space, they often can obtain reasonable solutions in all situations. Nevertheless, their main drawback is to spend much time doing scheduling. Hence, we propose a modified genetic algorithm to overcome this drawback in this paper.

Our method is named *Partitioned Genetic Algorithm* (*PGA*), which integrates the concept of



n_i	t_i	e_{ij}	c_{ij}	e_{ij}	c_{ij}
T_0	15	$E_{0\,3}$	3	$E_{4\,9}$	2
T_1	6	$E_{0\,4}$	1	$E_{5\,6}$	1
T_2	2	$E_{0\,5}$	1	$E_{7\,8}$	3
T_3	3	$E_{1\,3}$	2	$E_{9\,10}$	2
T_4	9	$E_{1\,5}$	2		
T_5	4	$E_{2\,3}$	2		
T_6	1	$E_{2\,4}$	2		
T_7	9	$E_{2\,7}$	3		
T_8	4	$E_{3\,5}$	2		
T_9	9	$E_{3\,7}$	1		
T_{10}	2	$E_{4\,5}$	1		
T_{11}	10	$E_{4\,6}$	1		

Fig. 1. Task graph.

Divide-and- Conquer mechanism to partition the entire problem into subgroups and solve them individually. Like the essential advantage of *Divide-and-Conquer* mechanism, experimental results show that PGA can dramatically decrease the time doing scheduling. Meanwhile, our results also indicate that PGA can obtain similar performance as original genetic algorithms, sometimes it is even better.

The remaining of this paper is organized as follows. Section 2 contains some preliminaries. Design issues and principles of our PGA are introduced in Section 3. Section 4 gives experimental results to demonstrate features and merits of our PGA. Finally, some conclusions are given in Section 5.

2 Preliminaries

In this Section we formally define the multiprocessor scheduling problem. Principles of genetic algorithm are also introduced.

2.1 Multiprocessor Scheduling [4-6]

A *homogeneous multiprocessor system* is composed of m identical processors $P_0 \ldots P_{m-1}$. They are connected by a complete communication network, where all links are identical. Each processor can execute only one task at a time and task preemption is not allowed.

The *parallel program* is described by a *Directed Acyclic Graph* (*DAG*) $G = (V, E, T, C)$ where V is the set of task nodes, E is the set of communication edges. The value $t_i \in T$ is the execution time of node $n_i \in V$. The value $c_{ij} \in C$ is the communication cost incurred along the edge $e_{ij} = (n_i, n_j) \in E$, which is zero if both n_i and n_j are assigned to the same processor. In this case, n_i is said to be an *immediate predecessor* of n_j, and n_j itself is said to be an *immediate successor* of n_i. A task without any predecessor is called *entry* task and *exit* task is a task without any successor. An example of task graph is shown in Fig. 1.

Tlevel(n_i) is defined to be the length of the longest path in the task graph from an entry task to n_i, excluding the computation cost of n_i. Symmetrically, *blevel*(n_i) is the length of the longest path from n_i to an exit task, including the computation cost of n_i. Formula (2.1) and (2.2) are formal definitions of *tlevel*(n_i) and *blevel*(n_i). Notice that we consider communication costs

while calculating values *tlevel* and *blevel*.

$$tlevel(n_i) = \max_{n_j \in pred(n_i)} \{tlevel(n_j) + t_j + c_{ji}\} \quad (2.1)$$

$$blevel(n_i) = t_i + \max_{n_j \in succ(n_i)} \{c_{ij} + blevel(n_j)\} \quad (2.2)$$

Given a parallel program to be executed on a multiprocessor system, the scheduling problem consists of finding a *task schedule* that minimizes the entire execution time. The execution time yielded by a schedule is usually called *makespan*. A solution to a scheduling problem is an assignment for each task of a starting time and a processor. Optimizing allocation under time and precedence constraints in a multiprocessor system is an NP-hard problem in general [4-5, 10].

2.2 Genetic Algorithm [4-5, 7, 10]

Genetic algorithm is a guided random search algorithm based on the principles of evolution and natural genetics. It combines the exploitation of past results with the exploration of new areas of the search space. By using *survival of the fittest* techniques and a structured yet randomized information exchange, genetic algorithm can mimic some of the innovative flair of human search. Genetic algorithm is randomized but not simple random walks. It exploits historical information efficiently to speculate on new search points with expected improvement.

Genetic algorithm maintains a *population* of candidate solutions that evolves over time and ultimately converges. Individuals in the population are represented with *chromosomes*. Each individual has a numeric *fitness* value that measures how well this solution solves the problem. Genetic algorithm contains three operators. The *selection* operator selects the fittest individuals of the current population to serve as parents of the next generation. The *crossover* operator chooses randomly a pair of individuals and exchanges some part of the information. The *mutation* operator takes an individual randomly and alters it. As natural genetics, the probability of applying mutation is very low while that of crossover is usually high. The population evolves iteratively (in the genetic algorithm terminology, through *generations*) in order to improve the fitness of its individuals.

The structure of genetic algorithm is a loop composed of a selection followed by a sequence of crossovers and mutations. Probabilities of crossover and mutation are constants and fixed in the beginning. Finally, genetic algorithm is executed until some termination condition is achieved, such as the number of iterations, execution time, results stability, etc.

3 Partitioned Genetic Algorithm

As mentioned before, the main drawback of genetic algorithms is to spend much scheduling time. Obviously, its scheduling time directly depends on the number of tasks being scheduled. Hence, we present a *Partitioned Genetic Algorithm* (*PGA*), which integrates the concept of *Divide-and-Conquer* mechanism to decrease the number of tasks being scheduled at a time.

3.1 Blevel Partition Algorithm

Main steps of *Divide-and-Conquer* algorithm are to divide the problem into subgroups, solve them individually, and merge them to form the final solution. In PGA we present a *Blevel Partition Algorithm* to partition the original task graph according to the *blevel* value of every task. Steps of *Blevel Partition Algorithm* are shown below:

I. Calculate the *blevel* of each task.

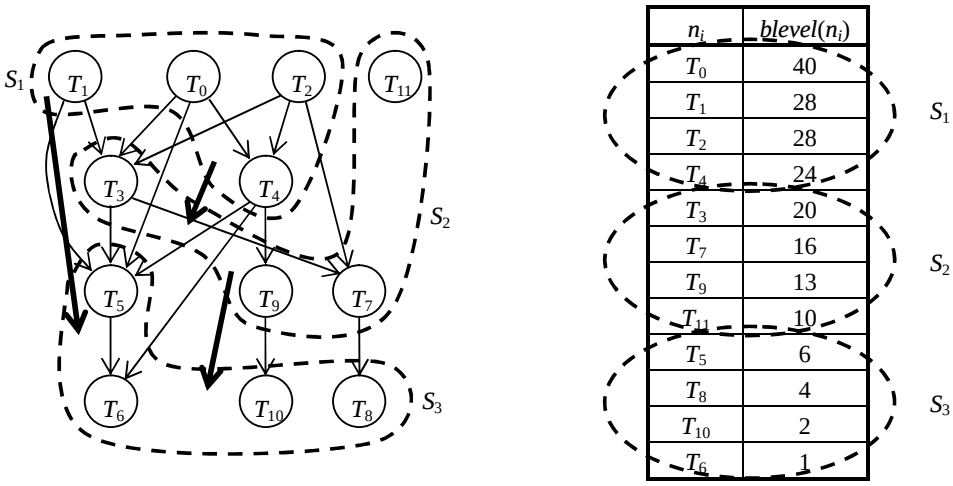


Fig. 2. Partition result.

II. Sort tasks in decreasing order according to their *blevel*. Tie- breaking is done randomly.

III. Partition tasks into subgroups evenly in sequence.

Fig. 2 is the result of partitioning task graph in Fig. 1 into three subgroups and black arrows represent precedence constraints among subgroups. Notice that after partitioning, these precedence constraints cannot form any cycle. A partition result is *legal* if its precedence constraints don't contain any cycle. Fortunately, the following Lemma proves that *Blevel Partition Algorithm* always generate legal partition result.

Lemma *Blevel Partition Algorithm* can always generate legal partition results.

Proof: Assume that $S_1 \ldots S_n$ are subgroups generated by *Blevel Partition Algorithm* and tasks $n_i \in S_i$, $n_j \in S_j$, for $i < j$. Because *Blevel Partition Algorithm* sorts and partitions tasks in sequence, it is obvious that $blevel(n_i) \geq blevel(n_j)$. From the definition of *blevel*, n_i cannot be a successor of n_j. Thus, tasks in S_i will not depend on any task in S_j and the partition result is always legal. □

3.2 Genetic Algorithm

After partitioning, all subgroups are scheduled using standard genetic algorithms individually in sequence. Complete time of every processor in subgroup S_i is transferred to subgroup S_{i+1} as the ready time of corresponded processor. In other words, all processors can start executing tasks at different time except for the first subgroup. This transferring step is a key point of PGA, which can make the final task schedule much compact.

Many genetic algorithms designed for DAG scheduling have been proposed. Except for pure genetic algorithms, some knowledge-augmented methods are developed to produce better results. Since each algorithm contains its own characteristics, we choose some famous genetic algorithms and construct a simulator to integrate them [4-5, 8]. Following subsections contain methods we have implemented.

3.2.1 Coding

A schedule is *feasible* if it satisfies the following two conditions:

I. A task's predecessors must have finished their execution before it can start executing.

T_0	T_2	T_4	T_1	T_3	T_{11}	T_9	T_7	T_5	T_{10}	T_8	T_6
2	0	2	1	1	2	0	1	2	0	1	2

Fig. 3. Feasible schedule.

II. All tasks within the task graph must execute at least and only once.

A tricky question is how to represent a schedule in a way suitable for a heuristic algorithm. We decide on the following representation.

Task#	t_1	t_2	…	t_n	← *sequence part*
Proc#	p_1	p_2	…	p_n	← *allocation part*

where a pair t_i, p_i means that task T_{ti} should be executed on processor P_{pi}. There is an explicit ordering among tasks in sequence parts. Chromosomes in many previous studies only interpret explicit task order on each processor. But in our coding, we let tasks ordered globally, which means the starting time of task T_{ti} is less than or equal to T_{ti+1} whether they are allocated to the same or different processors. Fig. 3 is a feasible schedule of task graph in Fig. 1.

An important factor in selecting the string representation is that all possible feasible schedules in the search space can be unique represented. It is also desirable, though not necessary, that the strings are in one-to-one correspondence with the search nodes. This feature can greatly simplify the design of genetic operators. It is obvious that our coding method satisfies this feature, because tasks are ordered globally.

3.2.2 Initial Population

Each individual of the initial population is generated through a random list heuristic. For each iteration, the task to be scheduled is determined by the following two rules:

I. Choose a *ready task*, which all predecessors are already scheduled, at random.
II. Allocate it to a processor randomly.

In previous subsection we have defined that a feasible schedule must satisfy two conditions. Based on the first rule we can always generate feasible schedules in the initial population. On the other hand, the task distribution over processors is uniform since we randomly choose a processor at every iteration.

3.2.3 Fitness Function

Our scheduling goal is to minimize the entire execution time of the task schedule. But in the implementation, we change it to maximization problem. We let the fitness value of a feasible schedule equals to (*max_makespan* – *makespan*) where *makespan* is the entire execution time of this schedule; *max_makespan* is the largest *makespan* of the current population.

3.2.4 Selection

The selection is done using a biased *roulette wheel* principle. Thus, the better the fitness of an individual, the better the odds of it being selected.

3.2.5 Crossover

Crossover takes two individuals as input and generates two new individuals, by crossing the parents' characteristics. Hence, the offsprings keep some of the characteristics of the parents. Let s_1 and s_2 be individuals which should generate offsprings s_1' and s_2'. We implement two crossover operators in PGA. The first one is the classical one-point crossover. S_1' and s_2' are generated by following rules and illustrated in Fig. 4:

I. Keep the sequence parts of s_1 and s_2 to s_1' and

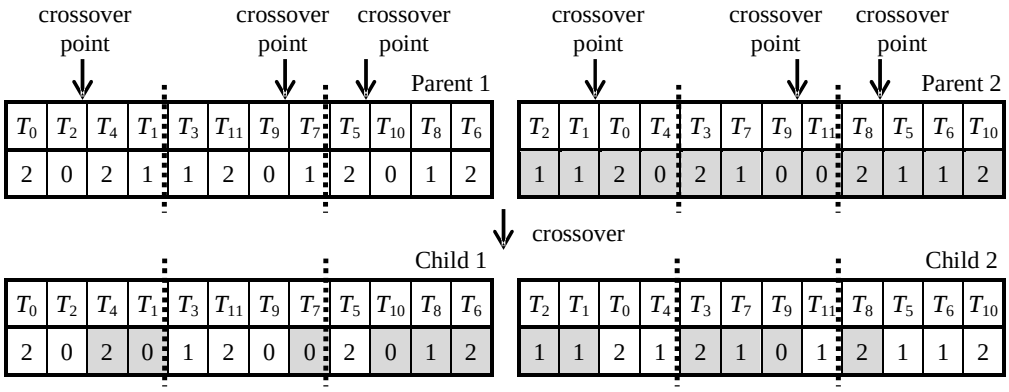


Fig. 4. Crossover example.

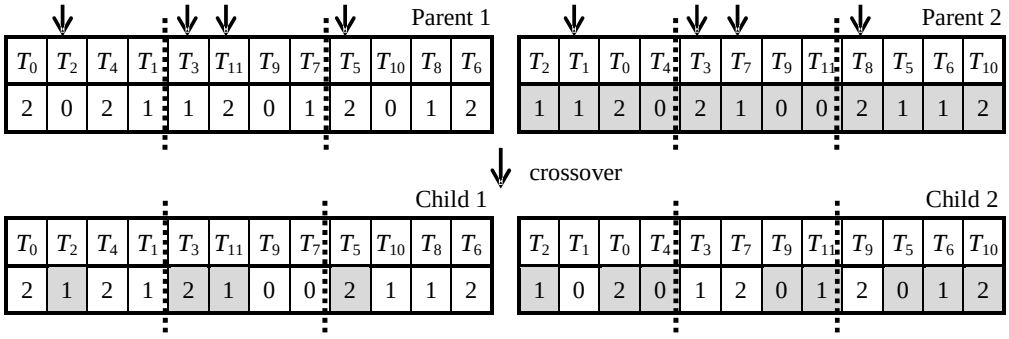


Fig. 5. Crossover example.

s_2' directly.

II. Choose a crossover point randomly to separate the allocation parts of s_1 and s_2.

III. Exchange the allocation parts of s_1 and s_2 alter the crossover point.

The second one uses the following rules and is illustrated in Fig. 5:

I. Keep the sequence parts of s_1 and s_2 to s_1' and s_2' directly.

II. Select a set of tasks randomly.

III. Exchange the allocation parts of s_1 and s_2 of the selected tasks.

In these two crossover mechanisms, we never change the sequence parts. Hence, s_1' and s_2' generated by them are always feasible. This feature makes us skip the design of check and repair algorithm, which is the most complex part in genetic algorithm design.

3.2.6 Mutation

Mutation ensures that the probability of finding the optimal solution is never zero. It also acts as a safety net to recover good genetic material that may be lost through selection and crossover. We implement two mutation operators in PGA. The first one selects two tasks randomly and swaps their allocation parts. The second one selects a task and alters its allocation part at random. These operators can always generate feasible offspring, too.

3.3 Conquer Algorithm

Since *Blevel Partition Algorithm* always generates legal partition result, subgroups can be

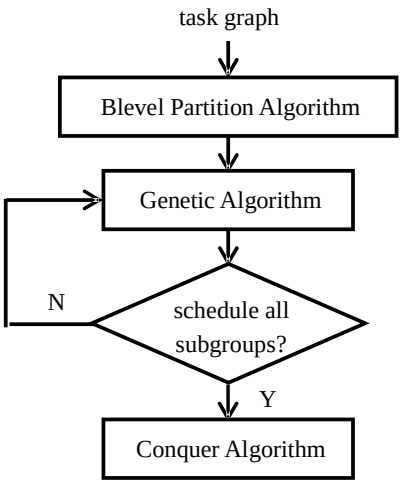


Fig. 6. Flowchart of PGA.

cascaded in sequence to form the global schedule. However, because precedence constraints between subgroups are not yet considered, the entire makespan currently maintained is not precise. Therefore, after scheduling all subgroups, we need an additional conquer algorithm to cascade all local schedules and recalculate the entire makespan. This conquer algorithm is quite simple. Final schedule are directly combined form all local schedules, and the makespan recalculating process is the same as before.

Finally, Fig. 6 shows the complete PGA flowchart.

4 Experimental Results

4.1 Simulation Environment

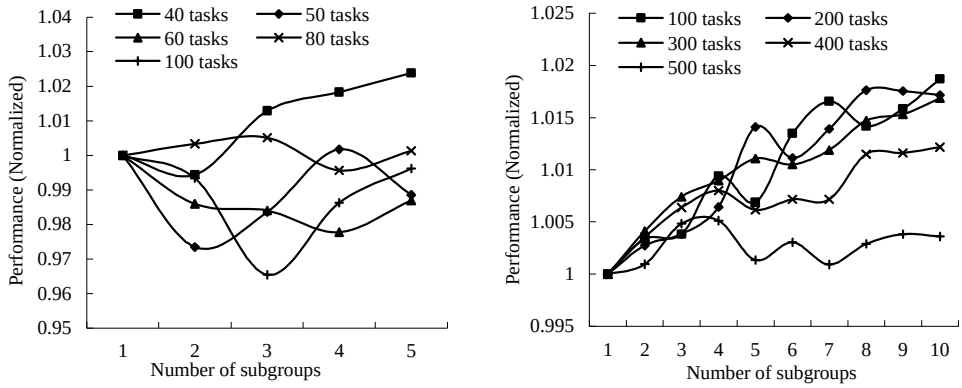
We construct a simulation and evaluation environment to evaluate PGA. Our simulator contains four independent parts which will be executed in sequence. The first part is a task graph generator. Based on the number of tasks inputted by the user, it will generate task graphs randomly. *Blevel Partition Algorithm* is implemented in the second part, which will partition entire task graph into several subgroups assigned by the user. The third part is the most critical one. It applies original genetic algorithm to schedule all subgroups in sequence and generates local schedules. The last part is used to conquer all local schedules and recalculates the entire makespan.

4.2 Results

In order to control the number of tasks in each subgroup, we generate two sets of task graphs to evaluate PGA. The first set contains task graphs with 40 to 100 tasks. They are partitioned into 1~5 subgroups and executed on a system with 4 processors. Task graphs in the second set contain 100 to 500 tasks. We partition them into 1~10 subgroups and use a system with 8 processors to execute them. In the following, experimental results for both task graph sets are shown together. Meanwhile, when scheduling each subgroup, the number of populations is adapted to the number of tasks in that subgroup. Probabilities of crossover and mutation are fixed, and the genetic algorithm will stop when the local makespan is unchanged after some predefined number of generations.

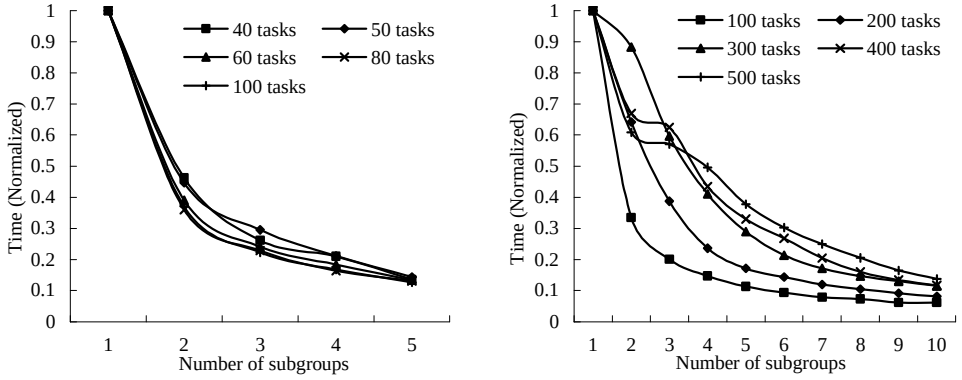
Fig. 7 shows performances of PGA with different number of subgroups. These performances (makespans) are normalized, and PGA with only one subgroup is essentially the same as original genetic algorithm. At here we can see that normalized performances only vary between 0.965 and 1.024. It indicates that the scheduling ability of PGA is similar as original genetic algorithm, sometimes it is even better.

In Fig. 8, it is clear that PGA can dramatically decreases the scheduling time. Relations between scheduling time and number of subgroups are not linear. From our simulation, the decreasing of scheduling time is noticeable only for less number of subgroups. After that the time



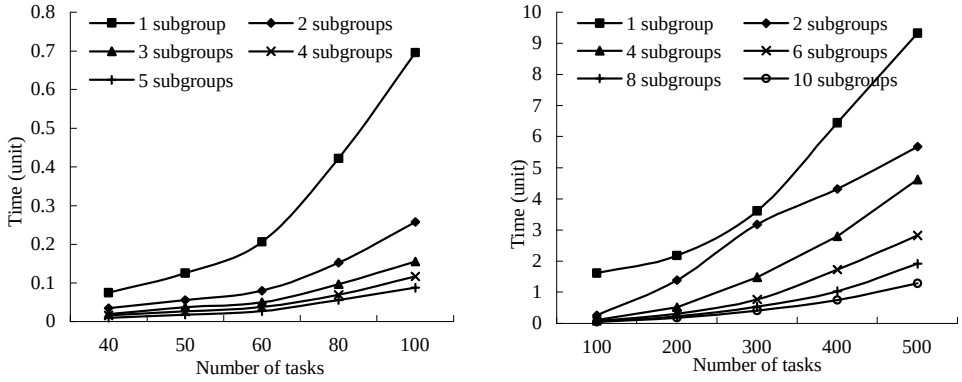
(a) (b)

Fig. 7. Experimental results.



(a) (b)

Fig. 8. Experimental results.



(a) (b)

Fig. 9. Experimental results.

variation is very slight.

Finally, Fig. 9 shows the scheduling time with different pairs of number of tasks and subgroups. It is obvious that if we partition the

task graph into more subgroups, the scheduling time increases much slower when the number of tasks becomes larger. This result indicates PGA is scalable than original genetic algorithm, which can indirectly extend its practicability.

5 Conclusions

In this paper we have proposed a modified genetic algorithm to schedule parallel program on multiprocessor system and constructed a simulation and evaluation environment to evaluate it. Our scheduling goal is to find a schedule that minimizes the entire makespan. Genetic algorithms are powerful but usually suffer from longer scheduling time. Therefore, we present PGA to overcome this drawback. According to our simulation results, PGA can exactly not only obtain similar performance as original genetic algorithm, but also spend less time doing scheduling. This feature also makes PGA more scalable and extends its practicability.

Reference

[1] **Parallel and Distributed Computing Handbook**. A.Y. Zomaya, ed. New York: McGraw- Hill, 1996.

[2] H. El-Rewini, T.T. Lewis, and H.H. Ali. **Task Scheduling in Parallel and Distributed Systems**. New Jersey: Prentice Hall, 1994.

[3] Y. Chow and W.H. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System", *IEEE Transactions on Computers*, Vol. 28, pp. 354-361, 1979.

[4] A.Y. Zomaya, C. Ward, and B. Macey, "Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 8, pp. 795-812, Aug. 1999.

[5] Ricardo C. Correa, Afonso Ferreira, and Pascal Rebreyend, "Scheduling Multiprocessor Tasks with Genetic Algorithms", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 8, pp. 825-837, Aug. 1999.

[6] Andrei Radulescu and Arjan J.C. van Gemund, "Low-cost Task Scheduling for Distributed-memory Machines", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 6, pp. 648-658, June 2002.

[7] D. Goldberg. **Genetic Algorithms in Search, Optimization, and Machine Learning**. Reading, Mass.: Addison-Wesley, 1989.

[8] Yu-Kwong Kwok and Ishfaq Ahmad, "Dynamic Critical-path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 5, pp. 506-521, May 1996.

[9] Tao Yang and Apostolos Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 9, pp. 951-967, Sep. 1994.

[10] M. Lin and L.T. Yang, "Hybrid Genetic Algorithms for Scheduling Partially Ordered Tasks in a Multiprocessor Environment", *Proc. of 6th International Conference on Real-time Computing Systems and Applications*, pp. 382-387, 1999.