Parallel Genetic Algorithm for a Flow-Shop Problem with Multiprocessor Tasks

C. Oguz[1], Yu-Fai Fung[2], M. Fikret Ercan[3], and X.T. Qi[1]

[1] Dept. of Management, [2] Dept. of Electrical Eng., The Hong Kong Polytechnic, University, Hong Kong SAR
{msceyda, eeyffung}@polyu.edu.hk
[3] School of Electrical and Electronic Eng., Singapore Polytechnic, Singapore
mfercan@sp.edu.sg

Abstract. Machine scheduling problems belong to the most difficult deterministic combinatorial optimization problems. Since most scheduling problems are NP-hard, it is impossible to find the optimal schedule in reasonable time. In this paper, we consider a flow-shop scheduling problem with multiprocessor tasks. A parallel genetic algorithm using multithreaded programming technique is developed to obtain a quick but good solution to the problem. The performance of the parallel genetic algorithm under various conditions and parameters are studied and presented.

Keywords: Genetic algorithms, parallel architectures, parallel computing

1 Introduction

Multiprocessor task scheduling is one of the challenging problems in computer and manufacturing processes. The general problem of multiprocessor task scheduling can be stated as scheduling a set of independent and simultaneously available tasks onto a set of parallel identical processors so that a given performance criterion is optimized. This type of scheduling problems is known to be intractable even with the simplest assumption [12]. An extensive survey on multiprocessor tasks scheduling can be found in [7]. This survey reveals that a single-stage setting for the processor environment is assumed in most of the multiprocessor task scheduling studies. Although this kind of assumption may be meaningful for some problems, there are many other practical problems that require jobs to go through more than one stage where each stage has several parallel processors. This type of environment is known as flow-shops in scheduling theory [9].

In this paper, we consider a multiprocessor task (MPT) scheduling problem in a flow-shop environment, which can be described formally as follows: There is a set J of n independent and simultaneously available MPTs to be processed in a two-stage flow-shop, where stage j consists of m_j identical parallel processors $(j = 1,2)$. Each $MPT_i \in J$ should be processed on p_{ij} identical processors simultaneously at stage j without interruption for a period of T_{ij} $(i = 1,2,...,n \ \ and \ \ j = 1,2)$. Hence, each

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2659, pp. 548–559, 2003.

$MPT_i \in J$ is characterized by its processing time, T_{ij}, and its processor requirement, p_{ij} $(i = 1,2,...,n \ \ and \ \ j = 1,2)$. All the processors are continuously available from time 0 onwards and each processor can handle no more than one MPT at a time. Tasks flow from stage 1 to stage 2 by utilizing any of the processors while satisfying the flow-shop and the MPT constraints. The objective is to find an optimal schedule to minimize the maximum completion time of all tasks.

The motivation for this problem comes from machine vision systems developed to perform real-time image understanding [3,8]. These systems utilize multiple layers of multiprocessor computing platforms where data have to pass through from one layer to another. Algorithms on parallel identical processors of each layer process image data. These systems can be analyzed from a scheduling perspective since they resemble the multi-stage flow-shop environment with MPT s, where data represent the incoming MPT s and algorithms applied, such as, feature detection, feature grouping, or object recognition define their operations at respective stages. Another application where the above MPT scheduling problem encountered is in diagnosable microprocessor systems [11], where a number of processors must process a task to detect faults. Other applications arise when a task requires more than one processor, tool or workforce simultaneously (see for instance, power system transient stability computations [16]).

Despite many practical applications that may involve MPT s in multi-stage settings, majority of research in this area has focused on MPT scheduling in a single-stage; little attention has been given to MPT scheduling in multi-stage settings. Extensive surveys on scheduling MPT s can be found in [7]. Another limitation of previous studies is that they mainly addressed the computational complexity issues of the problems (see for instance [2]). Most recently, Oguz et al. [14] provided approximation algorithms for the MPT scheduling problems in a flow-shop environment.

In this paper, we combine two areas, namely scheduling and parallel computing by parallel implementing a scheduling algorithm for real-time machine vision systems. Since MPT scheduling is intractable even in its simplest forms [1,12], we focus on efficient approximate algorithms to find a near optimal solution. We also concentrate on the fact that the real-life problems, like machine vision systems, require a quick but a good solution. It is thus of interest how a fast approximation algorithm could be developed for machine vision systems, which can be modeled as a MPT flow-shop scheduling problem.

Considering the success of the genetic algorithms (GA) developed for scheduling problems [4, 6, 15], we choose to use this local search method to provide a good solution to our problem. GAs are introduced by Holland [10]. A typical GA starts with an initial population of possible solutions to the problem (chromosomes). Each chromosome is characterized by its fitness, which is determined by the associated value of the objective function. The fittest chromosomes in the population (parents) will be selected for the generation of new solutions (children). This generation will take place according to a genetic operation, such as mutation (introducing of variations into the chromosomes) and crossover (taking the best features of each parent and mixing the remaining features). Hence, the new chromosomes will be somewhat different than their parents. In the new generation, the fitness of the children is evaluated in a similar fashion as their parents, and the worst fitted chromosomes will die to maintain the desired population. The birth and death processes will define the population size but usually it remains con-

stant from one generation to the next. This procedure is repeated until a desired termination criterion is reached. The output of this simulated evolution process will be the best chromosome in the final population, which can be a highly evolved solution to the problem.

Yet, excessive computation time in finding this highly evolved solution is a disadvantage of the GAs [5]. Hence, we present a parallel GA (PGA) in this paper. However, the benefit of the PGA is expected to be not only a speed-up in the computation time but also a better solution, that is shorter makespan for the scheduling problem, compared to a sequential GA)[5].

In the following section, we describe the design of the PGA. Next, the computational study is presented. We then report and discuss the computational results. Conclusions are given in the last section.

2 Parallel Genetic Algorithm

There are various kinds of implementation of PGAs that can be classified into three categories: global, migration and diffusion. These categories are mainly based on the structure of the population. The global PGA, often known as the worker/farmer model, treats the whole population as a single unit. Each chromosome can mate with any other chromosome in the entire population. The migration PGA, which is more similar to the natural evolution than the global PGA, divides the whole population into several subgroups. A chromosome can only mate within the subgroup and migrations may happen among the subgroups. The migration PGA is also called the coarse-grained model or island model. The diffusion PGA regards each chromosome as a separate unit. One chromosome can mate with another one in its neighborhood. The use of local neighbor leads to a continuous diffusion of chromosomes over the whole population. The diffusion PGA is also known as fine-grained, neighborhood, or cellular model.

In our PGA, the whole population is divided into G subgroups, each of which has s chromosomes and is processed by a sequential sub-GA. The sub-GAs run concurrently with some migrations among them. An epoch is the number of generations between two occurrences of migration. The effects of the number of subgroups, the value of s, the number of migrated chromosomes, and the length of an epoch on the performance of the PGA are analyzed in Section 3.

2.1 Hardware and Software Environment

The PGAs can be implemented on various parallel computing hardware and software environments, from networked PCs to mainframes. We implemented our PGA on SUN servers with multithreaded programming. With the emergence of the shared memory symmetric multiprocessor (SMP) computing systems, multithreaded programming provides the right programming paradigm to make maximum use of these new machines. The PGA algorithm is based on running several sequential GAs (SGA). Multithreaded programming is selected in the implementation so that each independent thread processes an SGA, since multithreading technique allows one program to execute multiple tasks concurrently. In the following, we will use the terms thread and subgroup interchangeably.

2.2 The Sequential Genetic Algorithm

Here, we will briefly describe the structure of the SGAs used in our PGA.

Chromosome Design: We define a chromosome as a string of $2n$ bytes. The first half of a chromosome is a permutation of $1,2,...,n$ representing the task list at stage 1. Similarly, the second half of a chromosome is a permutation of $1,2,...,n$, representing the task list at stage 2. A chromosome is decoded to a schedule by assigning the first unscheduled task in the task list to the machines at each stage.

Selection and Fitness: The fitness of chromosome x_k, i.e. the probability of chromosome x_k being selected to be a parent, is given by:

$$F(x_k) = \frac{f_{\max} - f(x_k)}{\sum_{i=1}^{N_{pop}} (f_{\max} - f(x_i))} ,$$

where $f(x_k)$ is the makespan of the schedule decoded from x_k, and $f_{\max}$ is the maximum $f(x_k)$ in the current generation, and N_{pop} is the population size. We use the well-known and commonly used roulette-wheel method as the selection operation [5]. Other selection methods, such as tournament selection, are also considered, but no improvement is observed from experimental results.

Crossover Operation: The crossover is an operation to generate two children chromosomes from two parent chromosomes selected. Three crossover operators are considered in this study: the one-point crossover (c1), the two-point crossover (c2) and the uniform crossover (c3). Since, in our study, a chromosome is composed of two parts, how to apply the crossover to these two parts is a problem. In our computational experiments, we find that it is better to crossover the two parts in the same position. The details are as follows:

(1) One-point crossover. For two parents, x_1 and x_2, a crossover position r, $r < n$, is randomly generated. The first r bits of the first child are the same as the first r bits of the parent x_1, and the bits from $r+1$ to n of the first child are in the same order as they are in parent x_2. The second half of the first child chromosome is generated in the same way. The child has the same absolute task sequence as parent x_1 and the same relative task sequence as parent x_2. Another child is generated with the same absolute sequence as parent x_2 and with the relative sequence as parent x_1. For example, if $x_1 = [(1,2,3,5,4,6)(1,2,5,3,4,6)]$, $x_2 = [(2,1,4,3,5,6)(2,4,1,3,5,6)]$ and the crossover position is 3, then the two children will be $C_1 = [(1,2,3,4,5,6)(1,2,5,4,3,6)]$, $C_2 = [(2,1,4,3,5,6)(2,4,1,5,3,6)]$.

(2) Two-point crossover. We randomly generate two positions r and s, $r < s$. One child will have the same absolute sequence as x_1 in the bits from 1 to r and from $s+1$ to n, and other bits have the same relative sequence as x_2. Another child is generated

correspondingly. For the above example, if $r = 2$ and $s = 4$, then the two children will be $C_1 = [(1,2,3,5,4,6)(1,2,3,5,4,6)], C_2 = [(2,1,3,4,5,6)(2,4,1,3,5,6)]$.

(3) Uniform crossover. Uniform crossover can be regarded as a multiple-point crossover. First a 0-1 string, called the mask string, is randomly generated. Then one child is generated with the same absolute sequence as x_1 where the corresponding mask string bit is 1, and other bits have the same relative sequence as x_2. For the above example, if the mask string is $(0,1,1,0,1,1)$, then $C_1 = [(1,2,3,5,4,6)(1,2,5,3,4,6)]$, $C_2 = [(2,1,3,4,5,6)(2,4,1,3,5,6)]$.

The crossover rate, that is, the probability of applying the crossover operator to the parents, is often considered to be 1 in scheduling problems [4, 13]. We also found from our computational study that the crossover rate of 1 is better.

Mutation Operation: The mutation operation modifies a chromosome. The arbitrary two-bit exchange mechanism is applied in our algorithm. In the arbitrary two-bit exchange, two positions are randomly selected and exchanged. After each mutation operation, we compare the new chromosome with the original one. If the fitness of the new chromosome is greater than that of the original, the new chromosome replaces the original. Otherwise, the original chromosome is kept in the population. This procedure can be regarded as a GA combined with a stochastic neighborhood search [6].

The mutation rate, that is the probability of the mutation to be applied to a chromosome, is reported to be large for scheduling problems. For instance, an initial mutation rate of 0.8 is used in [15], which is decreased by 0.01% at each generation. In [13], it is reported that the mutation rate of 1 is the best. A large mutation rate in scheduling problems may be due to the combinatorial character of the scheduling problems. Large mutation rate might help the GA to search the neighborhood of a schedule. Based on our computational study, we chose a mutation rate of 1.

Other Factors: The SGA uses a population size, which will be determined based on computational results given in Sect. 3.1, and an elitist strategy for reproduction, which is to remove the worst chromosome from the current population and include the best one from the previous population. The termination criterion is set to a limited number of generations, which is 500. While many researchers generate the initial population randomly, others favor a good solution as the initial "seed". In our problem, it is reasonable to say that in an optimal schedule, if an MPT is processed early at stage 1, probably it will be processed early at stage 2, too. Hence, the task sequence in two stages will be almost identical. Therefore, we use a ratio of 3:1 for initial chromosomes with the same task list in both stages and initial chromosomes with random task list.

2.3 Design of Migration

The design of migration concerns two aspects: the route of migration and the communication method among subgroups or threads. For the route of migration, we generate a migration table $(r_1, r_2, ..., r_n)$, which is a permutation of $(1,2,...,n)$. According to the migration table, the emigrants of a subgroup s will go to the destination subgroup r_s. Two kinds of migration routes are tested: fixed route and random route. In the fixed route, the migration table of each epoch is defined as $(2,3,...,n,1)$. In the random route,

the migration routes are randomly generated for each epoch. Computational results show that the random route is better and the details are presented in Section 3.2.

The communication method between the subgroups will influence the computation time. One easy implementation is to synchronize all the threads for each epoch. In this way, threads will pause after one epoch to wait for the completion of other threads. When all threads have completed for an epoch, another independent thread (we use the main thread to save computer resource) will be in charge of dealing with migration, and then each thread continues. For example, in the fixed migration route, the main thread will first save the emigrants of thread 1 in a buffer, and then copy the emigrants of thread $s+1$ to replace the emigrants of thread s, $s=1,2,...,n-1$. Finally it will copy the emigrants in the buffer to replace the emigrants of thread n.

Another approach is the asynchronous method which is more complex but efficient. For each thread, a buffer is allocated to hold the emigrants coming from a different subgroup. A thread can place its emigrants in the buffer of the destination thread without waiting for the destination thread to finish. After reading the emigrants in its buffer, a thread can proceed with its next epoch. Since the buffer is a shared memory block, which can be accessed by different threads, it must be protected by locks. The mutual exclusion lock (mutex) is applied in our PGA. Each buffer has two mutex locks, namely read-mutex-lock and write-mutex-lock, each of which can be locked only by one thread at any time. Only by locking its own read-mutex-lock, a thread can read from its buffer. Similarly, only by locking the write-mutex-lock of a buffer, a thread can write to the buffer of the destination thread. The procedure of communication for a thread can be explained by the following pseudo codes:

Pseudo code for migration of one thread.

```
// Procedure of writing emigration
Determine the destination thread from the migration table;
Lock the write-mutex-lock of the destination thread;              (O1)
Copy the emigrants to the buffer of the destination thread;       (O2)
Unlock read-mutex-lock of the destination thread;                 (O3)
// Procedure of reading immigration
Lock its own read-mutex-lock;                                     (O4)
Read from its buffer the immigrants and replace the emigrants;    (O5)
Unlock its own write-mutex-lock;                                  (O6)
```

Vector (r,w,b) represents the state of the buffer, where $r=0$ (or $w=0$) means the read-mutex-lock (or write-mutex-lock) is unlocked, $r=1$ (or $w=1$) means the read-mutex-lock (write-mutex-lock) is locked, $b=0$ means the buffer is empty and $b=1$ means the buffer is full. The following example explains the state of the buffer after each operation in the above code, where the symbol "?" represents an undetermined state: Consider operation (O1) at state $(1,?,?)$, which means that the read-mutex-lock is locked, and the write-mutex-lock is undetermined. If $w=1$, which means the buffer of the destination thread cannot be written to, then operation (O1) is blocked to wait for w to become 0; if $w=0$ (now b must be 0), then operation (O1) can continue. By operation (O1), the thread locks the buffer of the destination thread and changes w to 1 and the state of the destination thread becomes $(1,1,0)$. Similarly, before operation (O4), the state of a thread is undetermined and when (O4) finishes, it must become $(1,1,1)$. For the destination thread, we have $(1,?,?)\Rightarrow(1)\Rightarrow(1,1,0)\Rightarrow 2\Rightarrow(1,1,1)\Rightarrow(0,1,1)$. For the thread itself, we have

$(?,?,?) \Rightarrow (4) \Rightarrow (1,1,1) \Rightarrow 5 \Rightarrow (1,1,0) \Rightarrow (1,0,0)$. The initial state of each thread is $(1,1,0)$, i.e., the read-mutex-lock is locked and the write-mutex-lock is unlocked. To avoid deadlock, a thread is not assigned as its own destination.

2.4 Property of SGA

Since each subgroup is processed with a sequential sub-GA, the final result of the PGA will be affected from the application of different crossover operations. As mentioned in Section 2.2, we consider three different crossover operations. By applying them to the sub-GAs, we obtain three PGAs, denoted by PGA-c1, PGA-c2, PGA-c3, respectively. In addition, we employ a combined structure of the sub-GAs such that the crossover operations for some sub-GAs are different from others. For example, half of the sub-GAs use the one-point crossover and half of the sub-GAs use the two-point crossover, which is denoted by PGA-c12. Similarly, we have PGA-c13 and PGA-c23. In PGA-c123, each of the three crossover operations is used in one third of all the sub-GAs. The combined structure simulates the natural evolution environment in which each subgroup evolves in different conditions.

3 Experimental Results

The performance of the PGA under different hardware platforms and parameters are studied. We focused on the properties of the parallel computation, including the speed-up of computation time, the number of sub-groups, the migration and different crossover operations. The program is coded in C++ and run on SUN servers. All the results are the means of running 50 problems, where the number of machine is 16 for both stages. Unless explicitly specified, the PGA is the PGA-c12 with random migration route and has a migration size of two individuals. The number of epoch is 25 and the epoch length is 20 generations, which means a total of 500 generations. The number of subgroups is 15 and the size of a subgroup is 16.

The speed-up ratio of computation time of the PGA under different hardware platforms and processor workloads is reported. The speed-up ratio is defined as the ratio of computation time with a single thread to computation time with multiple threads and different workload. Experimental results show that the speed-up depends on the hardware configuration and the workload of the computers. Hence, three different SUN servers, S1, S2 and S3, are tested. S1 is a SUN SPARCserver 1000E with 4 CPUs and the CPU usage is less than 50% in normal conditions. The other two machines, S2 and S3, are logically derived from a SUN Ultra Enterprise (UE 10000) server with 16 CPUs. S2 and S3 have 4 and 8 CPUs, respectively, while the CPU usage for both machines is almost 100% in normal conditions. We tested the problems with 50, 100, 150, 200 and 300 jobs and with different number of threads, G =1, 4, 8, 10, 15, 20, 30, and 60. In each case, the population size of a subgroup is $s = 240/G$. Thus the total computation requirements are identical for different number of threads. The speed-up ratios obtained under different machines and conditions are listed in Tables 1, 2 and 3.

Table 1. Speed-up ratios for S1: 4 CPUs, not busy.

Threads	50 jobs	100 jobs	150 jobs	200 jobs	300 jobs
4	3.69	3.67	3.70	3.76	3.77
8	3.79	3.74	3.72	3.84	3.84
10	3.82	3.80	3.83	3.84	3.85
15	3.78	3.83	3.83	3.85	3.85
20	3.74	3.82	3.85	3.84	3.86
30	3.72	3.80	3.83	3.83	3.85
60	3.72	3.68	3.69	3.73	3.75

Table 2. Speed-up ratios for S1: 4 CPUs, busy.

Threads	50 jobs	100 jobs	150 jobs	200 jobs	300 jobs
4	2.26	2.58	2.62	2.63	3.10
8	2.83	3.42	3.50	3.51	3.50
10	2.87	3.61	3.68	3.71	3.72
15	3.10	4.01	4.32	4.41	4.43
20	3.46	4.09	4.55	4.45	4.43
30	3.52	4.30	4.70	4.69	4.70
60	3.18	4.20	4.57	4.57	4.58

Table 3. Speed-up ratios for S1: 8 CPUs, busy.

Threads	50 jobs	100 jobs	150 jobs	200 jobs	300 jobs
4	2.46	2.53	2.62	3.14	3.15
8	3.46	4.30	4.31	4.65	4.89
10	3.82	5.14	5.10	5.26	5.57
15	3.89	6.01	6.11	6.93	6.95
20	3.95	6.54	6.60	7.62	7.60
30	4.19	6.96	6.97	8.67	8.68
60	4.10	6.52	6.87	8.28	8.31

Table 4. Processing time (in sec.) by a single thread under the three different servers.

Computers	50 jobs	100 jobs	150 jobs	200 jobs	300 jobs
S1	238.4	563.5	278.4	1069	2859
S2	207.3	435.1	606.8	1001	1652
S3	99.30	292.4	486.5	922.5	1565

From the results, we can examine the relationship between the speed-up ratio and the number of jobs of the problem as well as the number of threads. The processing times of the PGA using a single thread are presented in Table 4 and these reflect the complexity of the PGA algorithm.

The parallel implementation of the GA will create certain overheads in the processing time such as the time required to create the threads and to synchronize two threads. These times are in the scale of microseconds and can be ignored when compared to the processing time demanded by the algorithm (see Table 4). Moreover, the synchronization between threads only takes place during migrations and these occur in a very limited number of times (25 in our tests) for the complete process.

As our algorithms are implemented on general-purpose servers, during run time, they are competing with other users' processes for the available system resources (CPUs). Usually, the Operating System, Solaris in our case, is responsible for the fair allocation of resources to users' processes. If there are more jobs than the available number of CPUs, then the jobs will form a queue and share the CPUs based on the round-robin mechanism. This is the major source of overhead induced in the computation time.

When a problem has more jobs, the computing requirement increases. Hence, the effect of the overhead is reduced and a better speed-up ratio is obtained. This is substantiated by the results presented in Tables 1, 2, and 3. Based on the results, we can observe that if the number of threads is fixed, the speed-up ratio has a tendency to increase for cases with more jobs. There are, however, some exceptional cases. These may be caused by workloads submitted by other users, or by tasks performed by the operating system while the PGA program is being executed.

The speed-up ratio increases with the increase in the number of threads because this increases the share of PGA on the system. However, the speed-up ratio begins to decrease if there are too many threads, for example, 60 threads. The total processing time of the PGA is equal to the total duration of all threads, i.e., it is determined by the last thread terminated. The duration of a thread is the combination of the total time when it is served by a CPU (t_s) and the time when it is waiting for available resources (t_w). When the number of thread increases, the term t_s decreases since the workload assigned to a thread is reduced. On the other hand, the term t_w increases as there are more threads waiting in the queue. A speed-up is obtained when the total duration ($t_s + t_w$) of the last terminating thread is reduced compared to the single thread case. When both terms are minimized then the speed-up ratio will be optimized. In our experiments, the optimal result is obtained when 30 threads are created. The speed-up ratios decrease in the 60-thread case because the total duration is increased due to the term t_w.

3.1 Number of Subgroups

The advantage of PGA is not only the speed-up of computation time, but also the improvement of the solution. We can find a better solution by dividing a large population into several subgroups. Table 5 depicts such a result. We have a whole population of 240 individuals, which means that the total computing requirements are fixed. With different number of subgroups G, the subgroup size is $s = 240/G$. To compare the solutions for different subgroups, the solution obtained from the single thread (group) case is used as a reference. The ratio of the solutions obtained from using different number of subgroups to the single group case are evaluated and listed in Table 5.

From Table 5, we can see that PGA can obtain better solutions than the sequential GA on a large whole population (the case of one thread). Computational results show that 15 subgroups each with 16 individuals produce the best result. If the subgroup size is too small, the improvement is less significant. In addition, PGA gives a better result if we have more jobs to schedule. For the problems with 50 jobs the maximum improvement is 0.010 while for the problems with 300 jobs the improvement is 0.025.

Generally, when the amount of computation increases, a better solution can be obtained. The size and the number of subgroups determine the amount of computation and

we have shown that a small subgroup is better than a large one. Next we will study the effect of altering the number of subgroups by fixing the size of each subgroup. We concentrate on the 300-job case and we use 16 and 30 as the size of a subgroup for comparison. The results, which are given in Table 6, indicate that increasing the number of subgroups can improve the solutions, but the improvement becomes less significant when the number of subgroups exceeds 20. If the number of subgroups is fixed, the subgroup size of 30 gives better results than 16. However the improvement is not significant.

Table 5. Ratio of solutions found by different number of subgroups to the single group case.

Sub-groups	50 jobs	100 jobs	150 jobs	200 jobs	300 jobs
1	1.000	1.000	1.000	1.000	1.000
4	0.994	0.984	0.981	0.978	0.989
8	0.991	0.982	0.977	0.976	0.984
10	0.991	0.982	0.977	0.975	0.980
15	0.990	0.982	0.976	0.974	0.975
30	0.993	0.983	0.978	0.976	0.980
60	0.994	0.984	0.979	0.977	0.983

Table 6. Ratio of solutions found by different number of subgroups to the single group case.

Subgroup	1	4	8	10	15	20	30	40
16	1.000	0.987	0.983	0.977	0.973	0.971	0.970	0.970
32	0.998	0.98 6	0.982	0.975	0.972	0.970	0.969	0.969

3.2 Design of Migrations

In Table7, we compare the fixed route migration and random route migration mechanism with different number of migration. The problems with 300 jobs are used and the solution of the fixed route mechanism with zero migration is taken as a base. It is easy to observe that the random route migration performs better than the fixed route migration. For the number of individuals for migration, 2 or 3 should be a better choice. We also consider the complete island model in which no migration occurs among the subgroups, that is, when migration number is zero. The result of the complete island model is worse than the result of none-zero migration, which demonstrates the usefulness of migration.

Table 7. Ratio of solutions found by different numbers of migration to the zero migration.

No. of migration	0	1	2	3	4	5
Fixed route	1.000	0.988	0.985	0.985	0.986	0.987
Random route	1.000	0.987	0.984	0.984	0.985	0.987

The frequency of migration, which is represented by the epoch length, also affects the performance of PGA. The short epoch length will weaken the outcome of the island model, which will become similar to the global PGA. The long epoch length will lead to the complete island model. To compare the different epoch lengths, we chose the epoch

length to be 10, 20, 30, 40, 50 and 100, with the corresponding number of epochs of 50, 25, 17, 13, 10 and 5, so that the total number of generations will be almost identical, i.e. 500, or slightly more. The results are provided in Table 8 for problems with 300 jobs, where the problem with the epoch length of 100 is taken as a base. The results are consistent with the above analysis and the epoch length of 20 gives the best result.

Table 8. Ratio of solutions found by different length of epoch.

Length of Epoch	10	20	30	40	50	100
Relative error	0.989	0.987	0.988	0.990	0.995	1.000

3.3 PGAs with Different Crossover Operations

As mentioned in Section 2.3, seven PGAs, namely PGA-c1, PGA-c2, PGA-c3, PGA-c12, PGA-c13, PGA-c23 and PGA-c123, are analyzed. The performances of the seven PGAs for the 300-job case are compared and the results are depicted in Table 9, where PGA-c12 is taken as a base. Results show that PGA-c1 and PGA-c2 have similar performances and are better than PGA-c3. On the other hand, the performances of all these PGAs, with only one kind of crossover operation, are worse than that of the PGAs with combined crossover operations. Among different combinations of the crossover operations, the combination of two operations seems to be sufficient for obtaining reasonable performance since the combination of three operations, PGA-c123, does not produce a better result.

Table 9. Ratio of solutions found by different crossover operations.

PGA-c1	PGA-c2	PGA-c3	PGA-c12	PGA-c13	PGA-c23	PGA-c123
1.0010	1.0008	1.0147	1.0000	1.0003	1.0002	1.0005

4 Conclusions

The purpose of this study is to provide a quick but a good solution for MPT scheduling in flow-shops. To achieve this objective, we developed a PGA. In the paper, we introduced the design of a SGA, which is the basic element of the PGA, together with the different characteristics of the PGA. The algorithm was implemented by multi-threaded programming on SUN servers. It was observed that if the workload of the server is high, by creating more threads, a better speed-up ratio could be obtained. We found that a relatively small size of a subgroup, about 16, and medium number of subgroups, about 15, are suitable. We compared different methods of migration among subgroups and found that random migration with 2 or 3 individuals is a better choice. In conclusion, the parallel implementation of the genetic algorithm can achieve both speed-up of computation time and the improvement of the near optimal solutions. As a prototype, this parallel genetic algorithm can be used to solve other complex scheduling problems.

Acknowledgement.
The work described in this paper was partially supported by a grant from The Hong Kong Polytechnic University (Project no. G-S551).

References

1. Blazewicz J., Drabowski M., Weglarz J.: Scheduling Multiprocessor Tasks to Minimize Schedule Length, IEEE Trans. Computers **C-35/5** (1986) 389-393
2. Brucker P.: Scheduling Algorithms, Springer, Berlin (1995)
3. Cantoni V., Ferretti M.: Pyramidal Architectures for Computer Vision, Plennium Press, New York (1994)
4. Chen C.L., Vempati V.S., Aljaber N.: An Application of Genetic Algorithms for Flow Shop Problems, European Journal of Operational Research **80** (1995) 389-396
5. Chipperfield A., Fleming P.: Parallel Genetic Algorithms, In: Zomaya, A.Y. (ed.): Parallel and Distributed Computing Handbook, McGraw-Hill (1996)
6. Dorndorf U., Pesch E.: Evolution Based Learning in a Job Shop Scheduling Environment, Comp. Opns. Res. **22** (1995) 25-40
7. Drozdowski M.: Scheduling Multiprocessor Tasks - An Overview, European Journal of Operational Research **94** (1996) 215-230
8. Ercan M.F., Fung Y.F.: The Design and Evaluation of a Multiprocessor System for Computer Vision, Microprocessors and Microsystems **24** (2000) 365-377
9. Gupta J.N.D., Hariri A.M.A., Potts C.N.: Scheduling a Two-stage Hybrid Flow Shop with Parallel Machines at the First Stage, Ann. Oper. Res. **69** (1997) 171-191
10. Holland H.: Adaptation in Natural and Artificial Systems. Ann Arbor, The University of Michigan Press (1975)
11. Krawczyk H., Kubale M.: An Approximation Algorithm for Diagnostic Test Scheduling in Multicomputer Systems, IEEE Trans. Comput. **34/9** (1985) 869-872
12. Lloyd E.L.: Concurrent Task Systems. Opns Res. **29** (1981) 189-201
13. Murata T., Ishibuchi H., Tanaka H.: Genetic Algorithms for Flowshop Scheduling, Comp. Ind. Engng, **30** (1996) 1061-1071
14. Oguz C., Ercan M.F., Cheng T.C.E., Fung Y.F.: Multiprocessor Task Scheduling in Multi Layer Computer Systems, in print European Journal of Operations Research.
15. Reeves C.R.: A Genetic Algorithm for Flowshop Sequencing, Comp. Opns. Res. **22** (1995) 5-13
16. Scala M. L., Bose A., Tylavsky J., Chai J. S.: A Highly Parallel Method for Transient Stability Analysis, IEEE Trans. Power Systems **5** (1990) 1439-1446