Dynamic Task Scheduling in Multiprocessor Real Time Systems Using Genetic Algorithms

Gheni Ahmed Ali [1]

Abstract:

The objective of the scheduling algorithm is to dynamically schedule as many tasks as possible such that each task meets its execution deadline while minimizing the total delay time of all of the tasks. The problem of scheduling of real-time tasks in multiprocessor systems is to determine ***when*** and on ***which processor*** a given task executes. In this paper we suggest a genetic algorithm for dynamic scheduling of real time tasks in multiprocessors system.

The algorithm based on the use of a fixed size chromosome and repeatedly applying specific crossover (single point or double point) and mutation procedures with variable mutation rates (0.05 – 0.1) until all tasks are successfully scheduled.

[1] **Al-Rafidain University College / Computer Communication Dept.**

Dynamic Task Scheduling in Multiprocessor Real Time Systems Using Genetic Algorithms

1. Introduction

Multiprocessors systems have emerged as a powerful computing means for real-time applications such as those found in nuclear plants and process control [1] because of their capability for high performance and reliability [2]. The problem of scheduling of real-time tasks in multiprocessor systems is to determine ***when*** and on ***which processor*** a given task executes [3]. Real-time systems use sophisticated scheduling algorithms to maximize the number of real-time tasks that can be processed without violating timing constraints. The performance of a scheduling algorithm is measured by its ability to generate a feasible schedule for a set of real-time tasks. A schedule for assigning tasks to one or more processors is said to be ***feasible*** if the execution of each task in the set can be completed before its deadline [4].

2. Real Time Systems

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. If the timing constraints of the system are not met, system failure is said to have occurred. Hence, it is essential that the timing constraints of the system are guaranteed to be met. Guaranteeing timing behavior requires that the system be predictable. Predictability means that when a task is activated it should be possible to determine its completion time with certainty. It is also desirable that the system attain a high degree of utilization while satisfying the timing constraints of the system [5].

Each task occurring in a real-time system has some timing properties. These timing properties should be considered when scheduling tasks on a real-time system. The timing properties of a given task refer to the following items [6]:

(a) Release time (or ready time): Time at which the task is ready for processing.

(b) Deadline: Time by which execution of the task should be completed, after the task is released.

(c) Minimum delay: Minimum amount of time that must elapse before the execution of the task is started, after the task is released.

(d) Maximum delay: Maximum permitted amount of time that elapses before the execution of the task is started, after the task is released.

(e) Worst case execution time: Maximum time taken to complete the task, after the task is released. The worst case execution time is also referred to as the worst case response time.

(f) Run time: Time taken without interruption to complete the task, after the task is released.

(g) Weight (or priority): Relative urgency of the task.

3. Taxonomy of Real Time Tasks

A real-time application is normally composed of multiple tasks with different levels of criticality. Although missing deadlines is not desirable in a real-time system, *soft real-time tasks* could miss some deadlines and the system could still work correctly. However, missing some deadlines for soft real-time tasks will lead to paying penalties. On the other hand, *hard real time tasks* cannot miss any deadline; otherwise, undesirable or fatal results will be produced in the system. There exists another group of real-time tasks, namely *firm real-time tasks*, which are such that the sooner they finish their computations before their deadlines, the more rewards they gain [5].

We can formally define a real-time system as follows.

Consider a system consisting of a set of tasks, $T = \{T1, T2, \ldots\ldots, Tn\}$, where the worst case execution time of each task $Ti \in T$ is Wi.. The system is said to be real-time if there exists at least one task $Ti \in T$, which falls into one of the following categories:

(1) Task Ti is a hard real-time task. That is, the execution of the task Ti should be completed by a given deadline Di; i.e., $Wi \leq Di$.

(2) Task Ti is a soft real-time task. That is, the later the task Ti finishes its computation after a given deadline Di, the more penalty it pays. A penalty function P (Ti) is defined for the task. If $Wi \leq Di$, the penalty function P (Ti) is zero. Otherwise P(Ti) > 0. The value of P(Ti) is an increasing function of $Wi - Di$.

.

(3) Task Ti is a firm real-time task. That is, the earlier the task Ti finishes its computation before a given deadline Di, the more rewards it gains. A reward function R(Ti) is defined for the task. If $Wi \geq Di$, the reward function R(Ti) is zero. Otherwise R(Ti)> 0. The value of R(Ti) is an increasing function of Di-Wi.

The set of real-time tasks $T = \{T1, T2, \ldots\ldots, Tn\}$ can be a combination of hard, firm, and soft real-time tasks.

4 Multiprocessor Scheduling Algorithms of Real -Time Systems

The goals for real-time scheduling are completing tasks within specific time constraints and preventing from simultaneous access to shared resources and devices [7]. Although system resource utilization is of interest, it is not a primary driver. In fact, predictability and temporal

correctness are the principal concerns. The algorithms used, or proposed for use; in real-time scheduling vary from relatively simple to extremely complex.

The scheduling of real-time systems has been much studied, particularly upon uniprocessor platforms, that is, upon machines in which there is exactly one shared processor available, and all the jobs in the system are required to execute on this single shared processor. In multiprocessor platforms there are several processors available upon which these jobs may execute.

The following assumptions may be made to design a multiprocessor scheduling algorithm:

(a) Job preemption is permitted

That is, a job executing on a processor may be preempted prior to completing execution, and its execution may be resumed later. We may assume that there is no penalty associated with such preemption.

(b) Job migration is permitted

That is, a job that has been preempted on a particular processor may resume execution on a different processor. Once again, we may assume that there is no penalty associated with such migration.

(c) Job parallelism is forbidden

That is, each job may execute on at most one processor at any given instant in time.

Real-time scheduling theorists have extensively studied uniprocessor real-time scheduling algorithms. Recently, steps have been taken towards obtaining a better understanding of multiprocessors real-time scheduling.

Multiprocessor scheduling techniques fall into two general category:

(1) Global Scheduling Algorithms

Global scheduling algorithms store the tasks that have arrived but not finished their execution in one queue which is shared among all processors. Suppose there exist m processors. At every moment the n highest priority tasks of the queue are selected for execution on the n processors using preemption and migration if necessary [8].

(2) Partitioning Scheduling Algorithms

Partitioning scheduling algorithms partition the set of tasks such that all tasks in a partition are assigned to the same processor. Tasks are not allowed to migrate; hence the multiprocessor scheduling problem is transformed to many uniprocessor scheduling problems [8].

5. Genetic Algorithms [9]

Genetic algorithms are search methods that employ processes found in natural biological evolution. These algorithms search or operate on a given population of potential solutions to find those that approach

some specification or criteria. To do this, the algorithm applies the principle of survival of the fittest to find better and better approximations.

At each generation, a new set of approximations is created by the process of selecting individual potential solutions (individuals) according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural genetics. This process leads to the evolution of populations of individuals that are better suited to their environment than the individuals that they were created from, just as in natural adaptation.

The GA will generally include the three fundamental genetic operations of selection, crossover and mutation. These operations are used to modify the chosen solutions and select the most appropriate offspring to pass on to succeeding generations. GAs consider many points in the search space simultaneously and have been found to provide a rapid convergence to a near optimum solution in many types of problems; in other words, they usually exhibit a reduced chance of converging to local minima. GAs suffers from the problem of excessive complexity if used on problems that are too large.

Figure 1 shows the structure of a simple genetic algorithm. Genetic algorithms work on populations of individuals rather than single solutions, allowing for parallel processing to be performed when finding solutions to the more large and complex problems. They are an iterative procedure that consists of a constant-sized population of individuals, each one represented by a finite linear string of symbols, known as the chromosome, encoding a possible solution in a given problem space. This space, referred to as the search space or state space, comprises all possible solutions to the optimization problem at hand. Standard genetic algorithms are implemented where the initial population of individuals is generated at random. At every evolutionary step, also known as generation, the individuals in the current population are decoded and evaluated according to a fitness function set for a given problem. The expected number of times an individual is chosen is approximately proportional to its relative performance in the population. Crossover is performed between two selected individuals by exchanging part of their genomes to form new individuals. The mutation operator is introduced to prevent premature convergence.

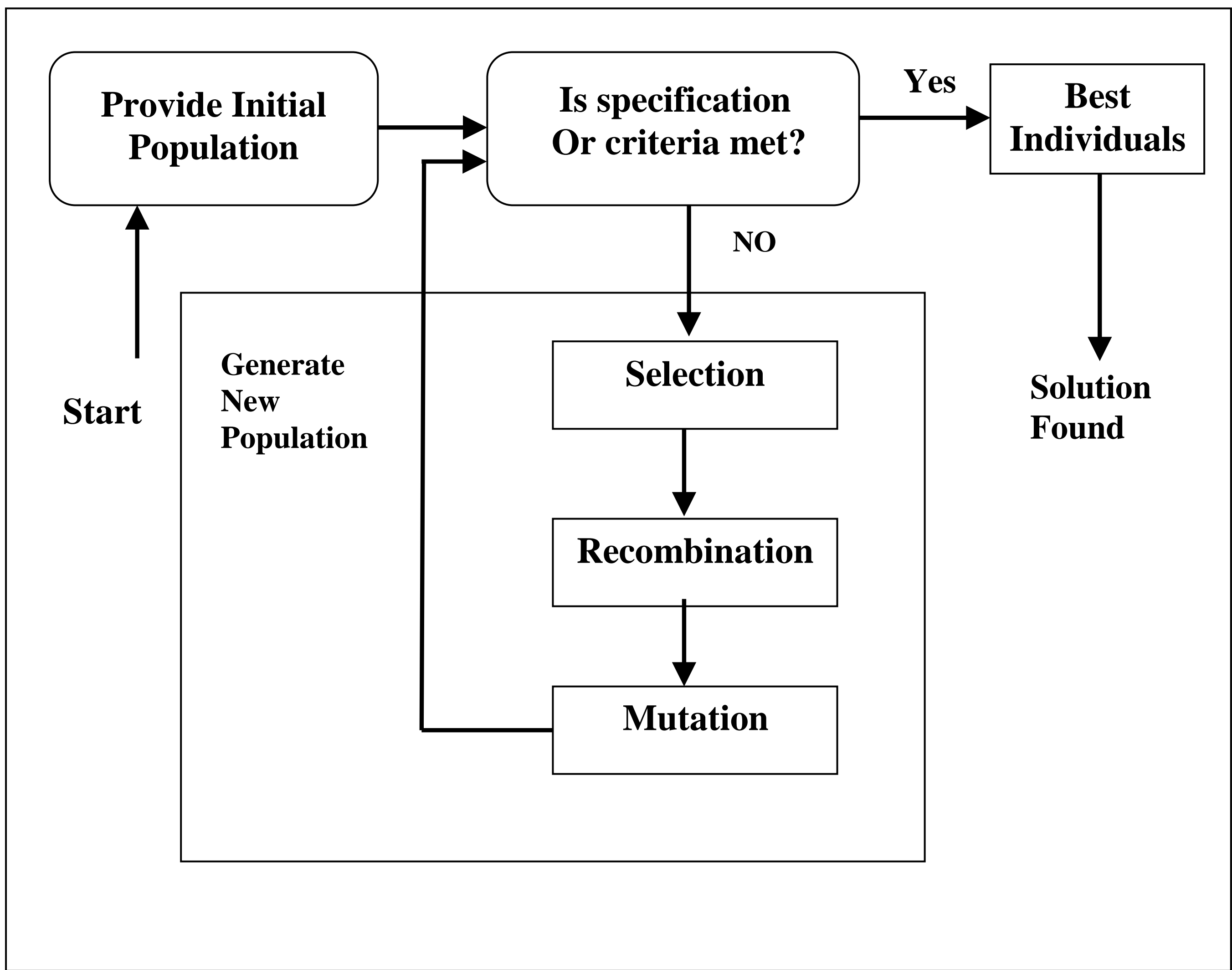

Figure 1, Structure of a single population evolutionary algorithm

6. Task model [10]

We assume a multiprocessor system consists of n identical processors; the system is processing m tasks. All processors are connected through shared medium.

Any task Ti ($1 \le i \le m$) may be assigned to any one of the processors Pj ($1 \le j \le n$). Each task is characterized by the followings:

Ai: arrival time of the task Ti.
Ri: ready time of the task Ti.
Wi: Worst case computation time of the task Ti.
Di: Deadline of the task Ti.
St(Ti): starting time of the task Ti.
Ft(Ti): finish time of the task Ti.

The task Ti meets its deadlines if and only if:

$Ri \le St(Ti) \le Di-Wi$ and
$Ri + Wi \le Ft(Ti) \le Di$.

7. Suggested Genetic Scheduling Algorithm.

Generally some important considerations must be taken into account in task scheduling of multiprocessing systems. Some important ones are: ***dependency*** and ***resource sharing.*** Dependent tasks will scheduled to be executed on the same processor (to eliminate communication cost) by ordering these tasks in ascending order (tasks with earlier deadlines first). Some tasks may share some resources (variables, data structures…) with other tasks. The resource access will be given firstly to the task with higher priority.

In our work global scheduling algorithms basis is used, in which the tasks that have arrived but not finished their execution are stored in one queue which is shared among all processors. At every moment the suggested scheduler must take n highest priority tasks of the queue for execution on the n processors. Each processor associates with its own dispatch task queue.

7-1 Chromosome Format

In tasks scheduling algorithms, the three main elements that must be included in the chromosome format are:

- The list of the tasks to be scheduled.
- The order in which these tasks should execute on a given processor.
- The list of the processor which these tasks should be assigned to.

A two dimensional array is used to represent the chromosome. Figure 2 below shows an example for chromosome format.

Gene

Task	3	4	1	2	5
Processor	1	2	3	1	4

Figure 2, Chromosome format

Each column in the array (chromosome) represents a gene. It is clear that task 3 will be executed on processor 1; task 4 will be executed on processor 2 and so on. The sequence of tasks execution is 3, 4, 1, 2 and 5.

7-2 Crossover operation

In this operation all the members of the population are grouped (randomly) into subsets of two chromosomes per set. After a randomly selected point the processor part of the gene in the chromosomes pair is swapped. This is equivalent to assigning some tasks to different processors. This operation called **single point crossover**. Figure 3 shows a single point crossover operation between pair of chromosomes after the 2nd gene.

Parent 1	Task	3	4	1	2	5
	Processor	1	2	4	3	2

Parent 2	Task	4	6	5	3	7
	Processor	2	4	3	5	6

(a) Before crossover

Child 1	Task	3	4	5	3	7
	Processor	1	2	4	3	2

Child 2	Task	4	6	1	2	5
	Processor	2	4	3	5	6

(b) After crossover

Figure 3, Single Point Crossover operation

There are many alternative crossover methods available, the most well known one is **two points crossover.** Rather than select one crossover point and swap the tail of the two chromosomes, two point crossover selects two random points in the chromosomes and swaps the information in between. Figure 4 shows an example of two points crossover.

← Crossover points →

Parent 1	Task	3	4	1	2	5
	Processor	1	2	4	3	2

Parent 2	Task	4	6	5	3	7
	Processor	2	4	3	5	6

(a) Before crossover

Child 1	Task	3	6	5	3	5
	Processor	1	2	4	3	2

Child 2	Task	4	4	1	2	7
	Processor	2	4	3	5	6

(b) After crossover

Figure 4, Two Points Crossover operation

7-3 Mutation

In real evolution, the genetic material can by changed randomly by erroneous reproduction or other deformations of genes, e.g. by gamma radiation. In genetic algorithms, mutation can be realized as a random deformation of the strings with a certain probability. The positive effect is preservation of genetic diversity and, as an effect, that local maxima can be avoided [11].

The mutation rate indicates the probability that a cell will be changed. As a result, the expected number of mutations per individual is equal to the mutation rate multiplied by the length of an individual. If a cell is selected to be mutated, then either the task number or the processor number of that cell will be randomly changed [12]. Figure 5 shows an example of simple mutation process, note that chromosome x and gene 3 of this chromosome is randomly selected for mutation

Chromosome x	Task	5	4	5	2	3
	Processor	1	2	4	3	2

(a) Chromosome x before mutation

Chromosome x	Task	5	4	1	2	3
	Processor	1	2	4	3	2

(b) Chromosome x after mutation

Figure 5, Simple mutation process

7-4 Fitness Function

A fitness function, which measures the quality of each candidate solution according to the given optimization objective, is used to help determine which chromosomes are retained in the population as successive generations evolve. The fitness function suggested in this work is calculated for each chromosome, and is determined by determining the number of tasks in the chromosome that meets their deadlines.

8- The Complete Algorithm

The suggested algorithm is written using turbo Pascal and implemented on a computer with 1.6 GHz, dual core Pentium 4 processor. Figure 6 shows the flow chart of the complete algorithm.

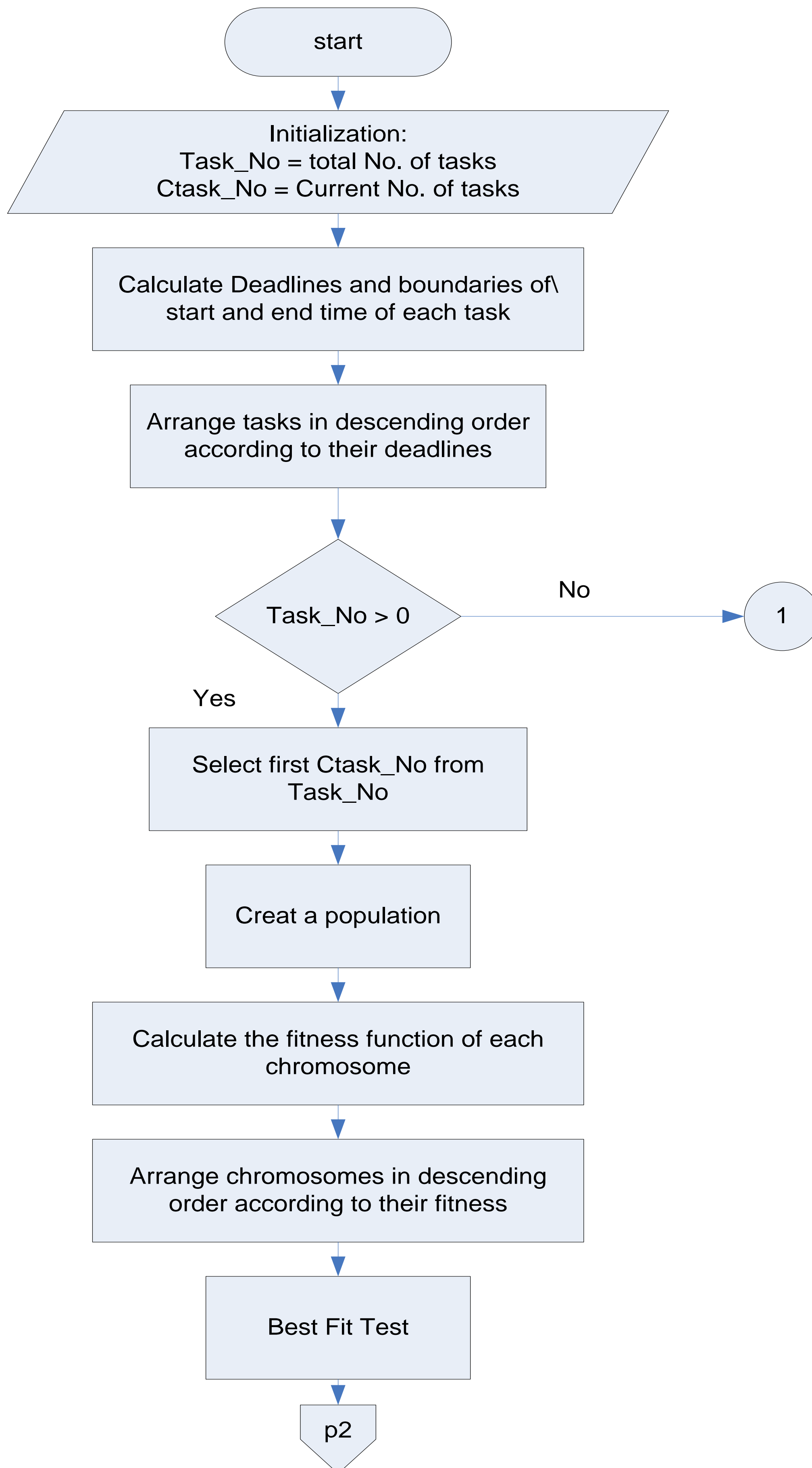

Figure 6, the flow chart of the suggested algorithm

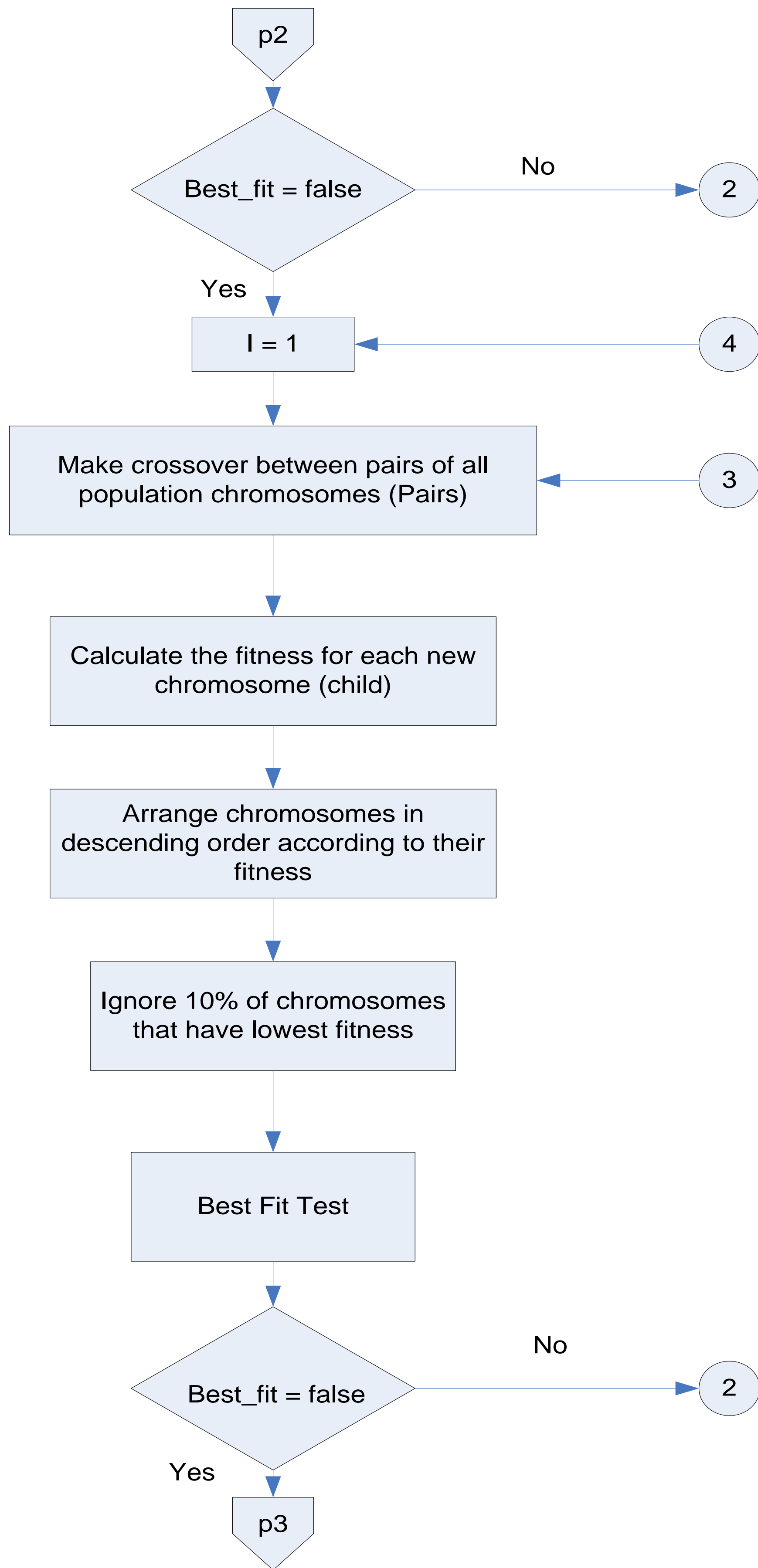

Figure 6, cont.

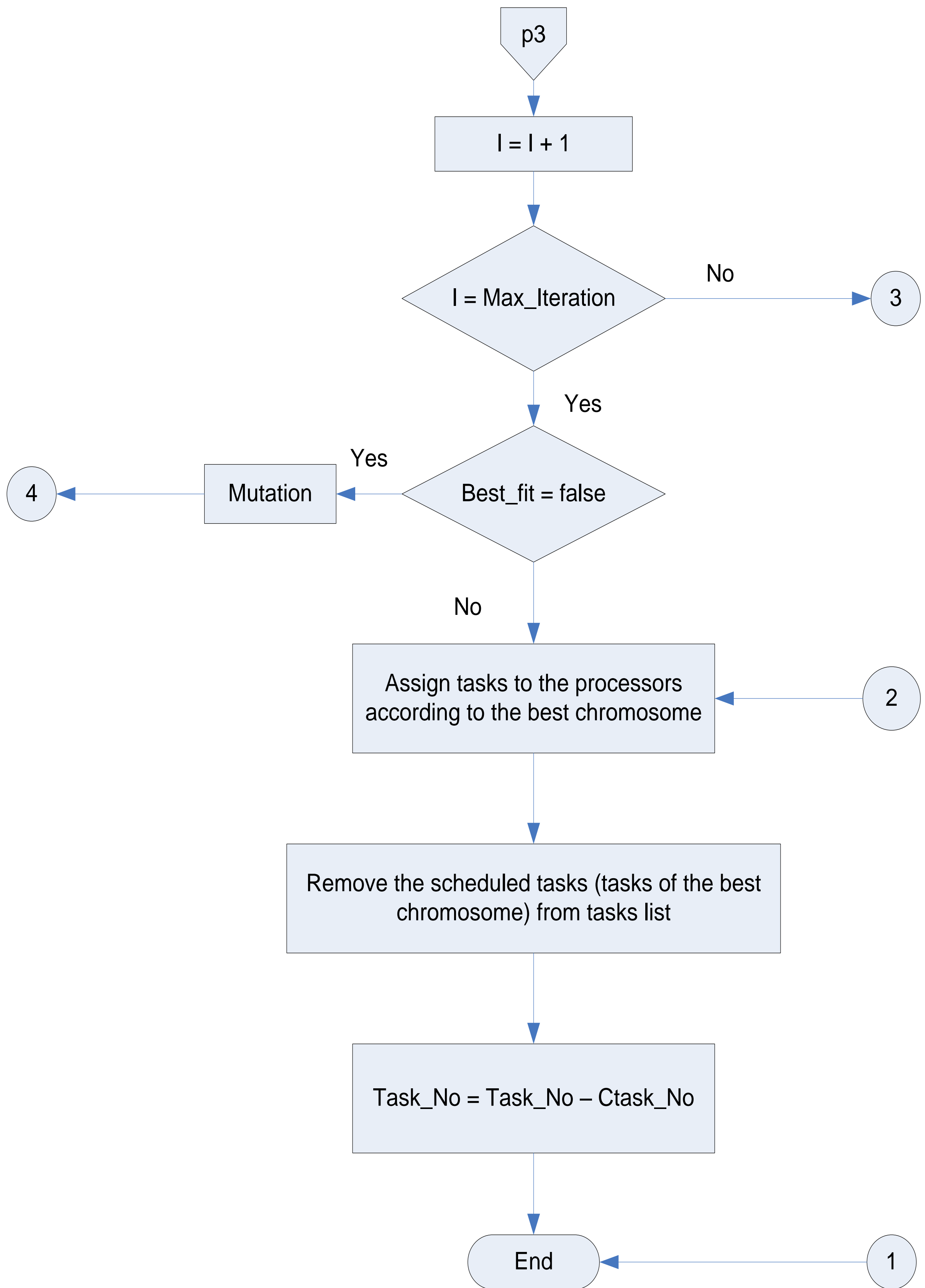

Figure 6, cont.

9 – Results and Analysis

The evaluation of the results of the suggested algorithm is obtained by calculating success ratio, where:

$$Success\ ratio = \frac{number\ of\ \ scheduled\ tasks}{total\ number\ of\ tasks}$$

The algorithm is applied too many times with different chromosome size (5 – 10), crossover method and mutation rates, taking into account the number of iteration required to reach the best fitness of each chromosome. Figure 7 shows the relation between number of generations and success ratio for chromosome size = 5, population size = 20 individuals for two methods of crossover.

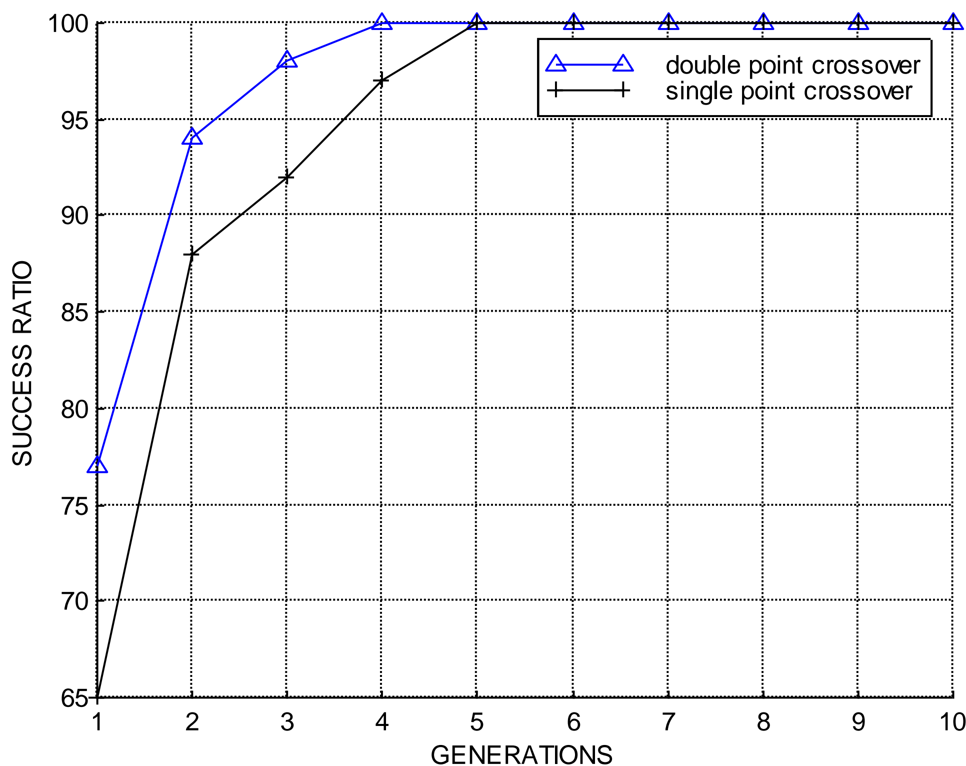

Figure 7, Success ratio versus number of generations for chromosome size = 5, in a system with 5 processors.

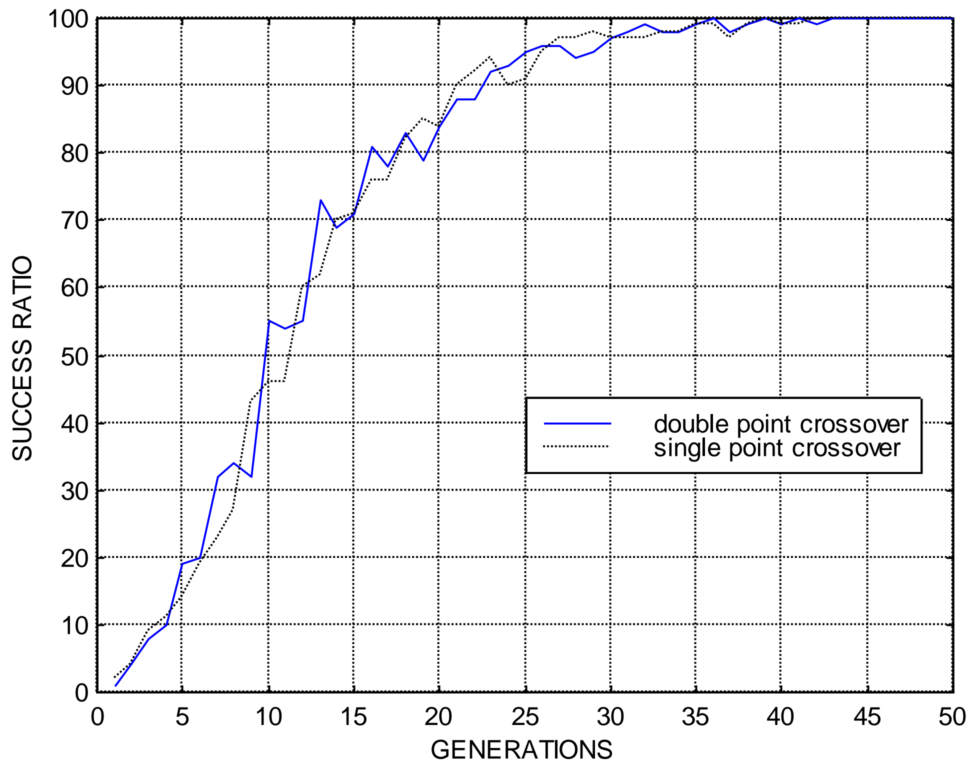

Figure 8, Success ratio versus number of generations for chromosome size = 10, in a system with 10 processors.

It is clear from the figure that the algorithm gives best fit (all tasks are successfully scheduled) at the 4th generation if double points crossover is applied, while the same algorithm needs 5 generations to give the same result if single point crossover is used.

Figure 8 shows the success ratio versus number of generations for chromosome size of 10, and population size of 10 individuals. Each point in the performance curves of the figures above is the average of twenty simulation runs with 100 tasks in each run. The mutation rates used in the suggested algorithm is variable and ranges from 0.05 and 0.1.

The use of double point crossover helps in reaching the best solution in less no of iterations. In the case of chromosome of size 5, if single point crossover is applied, all tasks are successfully scheduled with an average number of iterations of about 51, while in double point crossover the same results are obtained with about 38 iterations only.

For chromosome of size 10, the scheduling of all tasks is completed in 670 iterations on average if double points crossover is used, While 717 iterations are required on average to schedule all tasks in the case of single point crossover.

10 – Conclusions

The problem of scheduling of tasks to be executed on a multiprocessor system is one of the most challenging problems in parallel computing. Genetic algorithms are well adapted to multiprocessor scheduling problems.

In this paper we describe a proposed genetic algorithm for scheduling tasks in multiprocessor real time systems. The algorithm based on the use of a fixed size chromosome and repeatedly applying specific crossover and mutation procedures with variable mutation rates until all tasks are successfully scheduled.

The analysis of the results obtained from the suggested algorithms reveals that using double points crossover can leads to best scheduling with less iterations. The amount of saving in iterations is about 25% if the chromosome size is 5, and 6.5% for the chromosome size of 10.

References

[1] **"Distributed Computer Systems for Industrial Process Control"** By Schoeffler, J. D., Computer, Vol. 17, February 1984, pp.11-19.

[2] **"An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real-time Systems"**, By Manimaram, G. and Ram Murthy, C. S., IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 3, 1998, pp. 312-319.

[3] **"Scheduling Algorithms and Operating Systems Support for Real-time Systems"**. By Ramamritham, K. and Stankovic, J. A., Proceedings of IEEE, Vol. 82, No. 1, 1994, pp. 55-67.

[4] **"An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real-time Systems"**. By Manimaram, G. and Ram Murthy, C. S., IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 3, 1998, pp. 312-319.

[5] "Real-time Systems Design and Analysis, An Engineer Handbook,", *By* P. A. Laplante, IEEE Computer Society, IEEE Press, 1993.

[6] "Real Time Operating Systems Scheduling Lecturer,", By W. Fornaciari, P. di Milano, www.elet elet.polimi polimi.it/ fornacia it/ fornacia.

[7] "Real-time Operating System Scheduling Algorithms," , K. Frazer, 1997.

[8], "On-line Scheduling on Uniform Multiprocessors," S. Funk, J. Goossens, and S. Baruah , 22nd IEEE Real-Time Systems Symposium (RTSS'01), pp. 183-192, London, England, December, 2001.

[9] " Genetic algorithms – a tutorial". A.A.A. Townsend., http//aa.lasphost.com/tonyart/tomyt/Applets/ Genetic Alg / Genetic%20Algorithm.pdf.

[10] "A Hybrid Genetic Algorithm for Task Scheduling in Multiprocessor Real-Time Systems ", A. Mahmood, www.ici.ro/ici/revista/sic200-3/arto5.html

[11] "Genetic Algorithms: Theory and Applications", Ulrich Bodenhofer, Lecture Notes, Second Edition - WS 2001/2002, Institute of Algebra, Stochastic and mathematic Systems, Johannes Kepler University, A-4040 Linz, Austria.

[12] "**An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling",** Annie S. Wu, Han Yu, Shiyuan Jin, Kuo-Chi Lin, and Guy Schiavone, Member, IEEE, IEEE Transactions on Parallel and Distributed Systems, |Vol. 15, No. 9, September 2004,pp 824-834

الجدولة الديناميكية للمهام في انظمة الزمن الحقيقي متعددة المعالجات باستخدام الخوارزمية الجينية

المستخلص:

تعتبر عملية الجدولة الديناميكية للمهام في أنظمة الزمن الحقيقي متعددة المعالجات من مجالات البحث الغنية والمهمة وذلك لأهمية انظمة الزمن الحقيقي وما توفره من قدرات حسابية كبيرة ووثوقية عالية.

المقصود بجدولة المهام هو تحديد المعالج الذي سينفذ كل مهمة مع الالتزام بالتحديدات الزمنية لكل مهمة ومراعاة المشاركة في الموارد بين المهام ومدى اعتمادية المهام بعضها على بعض. يقدم البحث خوارزمية جينية مقترحة لجدولة المهام في انظمة الزمن الحقيقي متعددة المعالجات. تستند الخوارزمية على تشكيل كروموسومات بأحجام ثابتة باعتبارها مجموعة الحل الابتدائي لعملية الجدولة. بعدها يبدأ تطبيق مراحل الخوارزمية الجينية للوصول الى الحل الأمثل. تم تطبيق الخوارزمية في جدولة (١٠٠ مهمة) وباستخدام طريقتين لتبادل الجينات (crossover) بين الكروموسومات، الأولى هي تبادل الجينات بعد نقطة واحدة (single point crossover) والثانية تبادل الجينات بين نقطتين (double points crossover). كما تضمنت الخوارزمية استخدام عملية تهجين (mutation) للكروموسومات وبنسب تتراوح بين ٠.٠٥ و ٠.١. وقد نجحت الخوارزمية المقترحة في جدولة جميع المهام وبمعدلات تكرار مناسبة.

كلية الرافدين الجامعة / قسم هندسة اتصالات الحاسبات