8-30-1991

Efficient multiprocessor scheduling based on genetic algorithms

Hong Ren
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

Title of Thesis: Efficient Multiprocessor Sheduling
Based on Genetic Algorithms

Hong Ren, Master of Science in Electrical Engineering, 1991
Department of Electrical and Computer Engineering

Thesis directed by: Dr. Edwin S. H. Hou
Assistant Professor
Department of Electrical and Computer Engineering

The problem of multiprocessor scheduling can be stated as finding a schedule for a general task graph to be executed on a multiprocessor system so that the schedule length can be minimized. This scheduling problem in known to be NP-hard and methods based on heuristic search have been proposed to obtain optimal and sub-optimal solutions to the problem. Genetic algorithms have recently received much attention as robust stochastic searching algorithms for various optimization problems. In this thesis, we propose an efficient method based on genetic algorithms to solve the multiprocessor scheduling problem. The representation of the search node will be based on the schedule of the tasks in each individual processor. The genetic operator proposed is based on the precedence relations between the tasks in the task graph. The proposed genetic algorithms will be applied to the problem of scheduling the robot inverse dynamics computations and randomly generated task graphs.

EFFICIENT MULTIPROCESSOR SCHEDULING BASED ON GENETIC ALGORITHMS

by

Hong Ren

Thesis submitted to the Faculty of the Graduate School of

the New Jersey Institute of Technology

in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

August 1991

APPROVAL SHEET

Title of Thesis: Efficient Multiprocessor Scheduling
Based on Genetic Algorithms

Candidate: Hong Ren
Master of Science in Electrical Engineering, 1991

Thesis and Abstract Approved by the Examining Committee:

Dr. Edwin S. H. Hou, Advisor — Date
Assistant Professor
Department of Electrical and Computer Engineerig

Dr. Nirwan Ansari — Date
Assistant Professor
Department of Electrical and Computer Engineerig

Dr. S. Ziavras — Date
Assistant Professor
Department of Electrical and Computer Engineerig

New Jersey Institute of Technology, Newark, New Jersey

VITA

Hong Ren

Date of Birth

Place of Birth

Education

1990-1991	**New Jersey Institute of Technology**	MSEE
1979-1984	**Shanghai Tongji University**	BSEE

ACKNOWLEDGMENT

I am very grateful to my graduate advisor Dr. Edwin S. H. Hou for his valuable guidance, support, inspiration and encouragement during the entire course of this thesis.

I would also like to thank other committee members, Dr. Nirwan Ansari and Dr. S. Ziavras for serving on the examining committee and for evaluating this research.

This research was supported by a grant from the New Jersey Department of Higher Education through NJIT Separately Budget Research.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Problem Overview

Parallel processing systems or multiprocessor systems have been recently employed in a wide variety of computer applications such as the control of robots and real-time high-speed simulation of dynamical systems etc. However, exploiting the full potential of multiprocessor systems requires scheduling the computational tasks onto the multiprocessor system.

The problem of multiprocessor scheduling can be stated as finding a schedule for a general task graph to be executed on a multiprocessor system so that the schedule length can be minimized. This problem is known to be NP-hard [1]. The difficulty of the problem depends heavily on the topology of the task graph representing the precedence relations among the computation tasks, the topology of the multiprocessor system, the number of parallel processors, the uniformity of the task processing time, and the objective function chosen. Due to this computational complexity issue, heuristic algorithms have been proposed to obtain optimal and sub-optimal solution to various scheduling problems.

In this thesis we propose an efficient method based on genetic algorithms to solve the multiprocessor scheduling problem. Genetic algorithms are different from other optimization and search techniques in the following ways:

1. Genetic algorithms work with a coding of the parameter set.

2. Genetic algorithms search from a population of points.

3. Genetic algorithms use objective function information, instead of derivatives or other auxiliary knowledge.

4. Genetic algorithms use probabilistic transition rules, instead of deterministic rules.

The representation of the search node will be based on the schedule of the tasks in each individual processor. The genetic operator proposed is based on the precedence relations between the tasks in the task graph.

The system considered in this thesis is assumed to consist of a multiprocessor, a set of identical processors capable of independent operation on independent tasks. A job or a compound task submitted to the system consists of a set of tasks and a partial ordering of these tasks. This partial ordering may be represented by a precedence graph in which each vertex represents a task and precedence relationships between tasks are represented by edges. The dependency graph clearly gives the precedence relationship indicated by the partial ordering. Given a computation graph and a multiprocessor system, the problem is to determine a schedule for executing the tasks. The schedule must not violate any of the precedence relationships or the requirement

that no more than one processor can be assigned to a task at any time. It is assumed that the number of processors available to the program is always constant.

The proposed genetic algorithm will be applied to the problems of scheduling the robot inverse dynamics computations for the Stanford manipulator and Elbow Manipulator. We also test it on randomly generated task graphs.

1.2 Literature Review

The problem of multiprocessor scheduling has been widely studied [2]-[4].

Various approaches to the general multiprocessor scheduling problem have been proposed [2]-[5]. We will review some of them.

Tomas L. Adam, K. M. Chandy and J. R. Dickson [3] have proposed a dynamic programming solution for the case in which execution times are random variables.

Kashara and Narita [4] proposed an heuristic algorithm (critical path/most immediate successors first) and an optimization/approximation algorithm (depth first/implicit heuristic search). The critical path/most immediate successors first) method is one of the efficient heuristic method to solve the scheduling problem. The CP/MISF method consists of the following steps. *Step* 1: Determine the level l_i for each task. *Step* 2: Construct the priority list in the descending order of l_i. *Step* 3: Execute list scheduling on the basis of the

priority list.

Chen et al. [5] developed a state-space search algorithm (A*) coupled with an heuristic derived from the Fernandez and Bussell bound to solve the multiprocessor scheduling problem [5].

List schedules are a class of implementable schedules in which tasks are assigned priorities and placed in a list ordered in decreasing magnitude of priority. Wherever executable tasks contend for processors, the selection of task to be immediately processed is done on the basis of priority with the higher priority tasks executable being assigned processors first [6].

1.3 Thesis Organization

The contents of this thesis are organized as following:

In Chapter 2, we present the model and some related definitions of the task graph.

In Chapter 3, we introduce the basic theory of genetic algorithms, and the application of genetic algorithms in scheduling problems.

In Chapter 4, we present the experimental results based on the problem of scheduling the robot inverse dynamics computations, and four randomly generated task graphs.

- In Chapter 5, we conclud this thesis.

Chapter 2

MODEL AND DEFINITIONS

A task system can be represented by a directed acyclic task graph, $TG = (V, E)$, consisting of a finite nonempty set of vertices, V, and a set of finite directed edges, E, connecting the vertices. The collection of vertices, $V = \{T_1, T_2 ..... T_m\}$, represents a set of computation tasks to be executed and the directed edges, $E = \{e_{i,j}\}$, ($e_{i,j}$ represents a directed edge from vertex T_i to T_j) implies a partial ordering or precedence relation, exists between the tasks. The following definitions are used when discussing task graphs:

Definition 1:

A task T_i is a predecessor of T_j, written as $T_i \ll T_j$, if there is an edge $e_{i,j}$ from T_i to T_j . That is, if $T_i \ll T_j$, then T_i must be completed before T_j can be started.

Definition 2:

The set of predecessors of T_i, written as $Pred(T_i)$, is defined as

$$Pred(T_i) = \{T_j | T_j \ll T_i\}$$

In other words, task T_i can be started if and only if all the tasks in $Pred(T_i)$ have been completed.

Definition 3:

The set of successors of T_i, written as $Succ(T_i)$, is defined as

$$Succ(T_i) = \{T_j | T_i \ll T_j\}$$

No task in the set $Succ(T_i)$ can be started until task T_i is completed.

Definition 4:

The number of predecessors of T_i. is defined as $predN(T_i)$.

Definition 5:

The number of successors of T_i. is defined as $succN(T_i)$.

Definition 6:

The execution time of task T_i is defined as $et(T_i)$.

Definition 7:

$$height(T_i) = \begin{cases} 0, & \text{if } Pred(T_i) = \emptyset \\ 1 + \underset{T_j \in Pred(T_i)}{Max} height(T_j), & \text{otherwise.} \end{cases}$$

The *height* function in a way conveys the precedence relations between the tasks. In fact, if $height(T_i) < height(T_j)$, and there is a path from task T_i to task T_j (that is, there exists a series of directed edges leading from task T_i to T_j), then T_i must be executed before T_j. However, if there is no path between the two tasks, then the order of execution between the two tasks can be arbitrary.

Definition 8:

$$level(T_i) = \begin{cases} et(T_i), & \text{if } Pred(T_i) = \emptyset \\ et(T_i) + \underset{T_j \in Pred(T_i)}{Max} level(T_j), & \text{otherwise.} \end{cases}$$

The *level* function incorporate both execution time of a task and the precedence relation with the other tasks. It indicates the earliest time the task can be completed.

Definition 9:

The path with the longest execution time in task graph is defined as the *critical path*. Its value is equal to

Note that the *critical path* is the minimum execution time required to complete the task graph.

Definition 10:

The finishing time of a schedule is defined as the time required to complete all the tasks according to their order in the schedule.

We can randomly generate a task graph with the following algorithm.

The input parameters for this algorithm are:

- $taskN$ — number of tasks in TG.
- *max-et* — maximum execution time for each task.

- $max\text{-}succN$ — maximum number of successors for each task.
- $max\text{-}predN$ — maximum number of predecessors for each task.

Task Graph Randomly Generated Algorithm:

This algorithm randomly generates a task graph in the size by definition.

1 [Initialize.] Define the set $DONE$ as the tasks which have been considered, abd the set $READY$ as the tasks which have not yet considered. $DONE \leftarrow \emptyset$; $READY \leftarrow$ *all the tasks in* TG.

2 [Execution time.] Randomly generate the execution time for tasks in TG. The execution times are less than the maximum value defined. than maximum value in definition.

3 [Predecessors.] Randomly generate the number of predecessors, $predN(T_i) \leq max\text{-}predN$. Do step 4 $predN(T_i)$ times.

4 [Pick up one of Predecessors.] Randomly generate a task T_j, $T_j \in DONE$.

5 [Check the number of successors.] If $succN(T_j) \geq max\text{-}succN$ do step 4, else $DONE \leftarrow \{T_i\}$. $READY \leftarrow READY - \{T_i\}$.

6 [Generate all tasks in TG.] Do step 3 until $READY = \emptyset$.

Figure 2.1 illustrate a task graph with 15 tasks. The number inside the node denotes the the task number; The number associated with each node denotes respectively the *execution time*, *height*, and *level*.

The problem of multiprocessor scheduling considered in this thesis is based on the deterministic model, that is, the execution time and the precedence relations between the computational tasks are known. We assume that the multiprocessors system is uniform and non preemption, that is the processors are identical and that a processor completes the current task before executing a new one. Further more, the communication time between processors is assumed to be negligible.

The problem of optimal scheduling a task graph onto multiprocessor system is to assign the computational tasks to the processors so that the precedence relations are maintained and that all the tasks are completed in the shortest possible time. Figure 2.2 illustrates a schedule displayed as a Gantt chart for the example task graph TG using two processors. The finishing time for the schedule is 240ms. Notice that this schedule is not very efficient on processor P2. P2 is idle for the first $76ms$. The idle periods in the schedule are also referred to as holes.

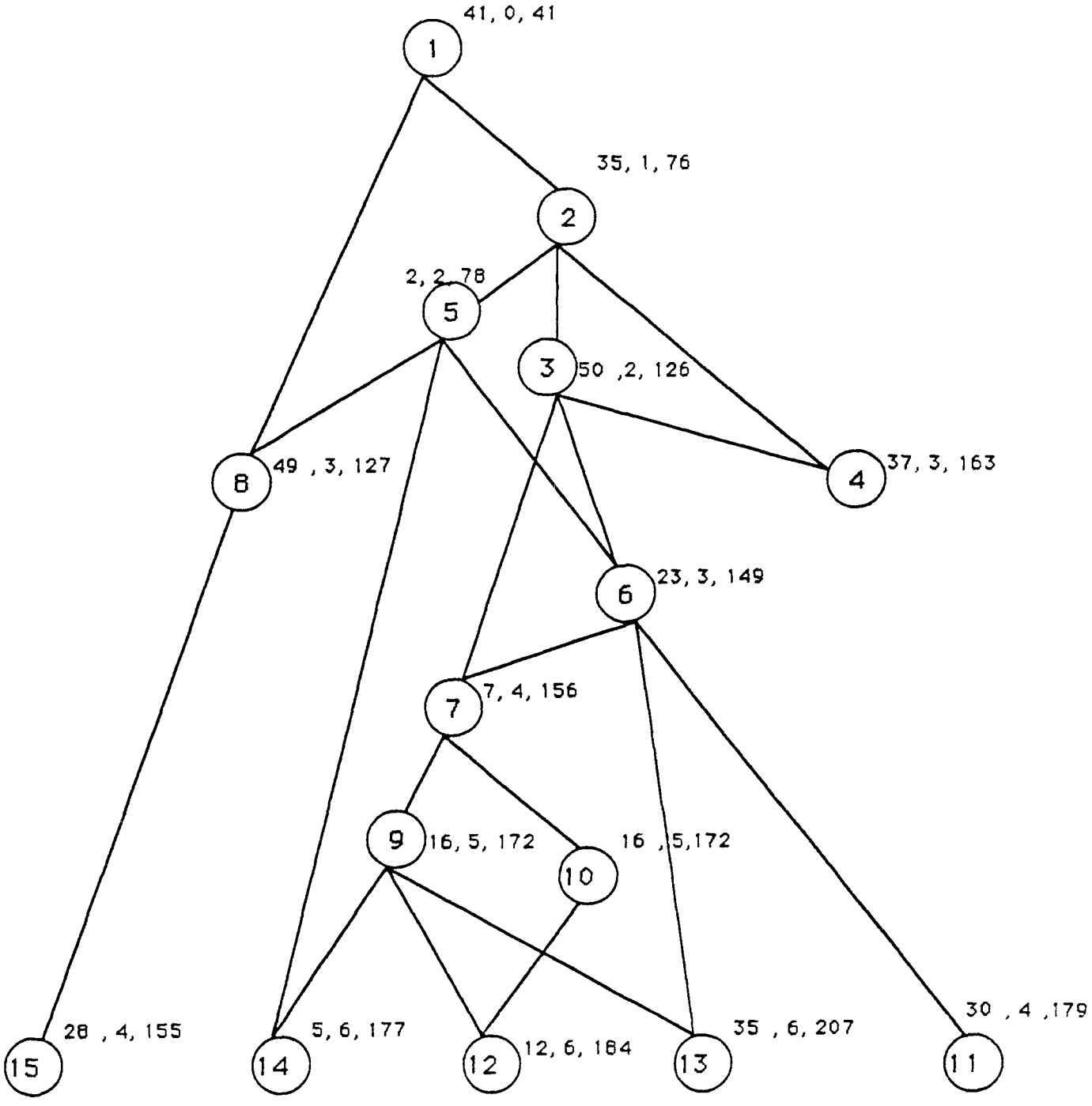

Figure 2.1: A task graph TG

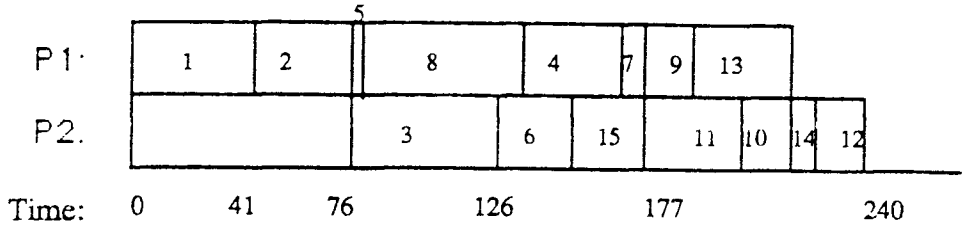

Figure 2.2: Schedule for two processors

Chapter 3

GENETIC ALGORITHMS AND APPLICATION IN MULTIPROCESSOR SCHEDULING

3.1 Overview of Genetic Algorithms

In this chapter, we will introduce genetic algorithms and their application in multiprocessor scheduling.

Genetic algorithms, developed by John Holland at the University of Michigan, are search algorithms based on the mechanics of natural selection and natural genetics. They combine the notion of survival of the fittest with a structured but randomized information exchange to form a search algorithm. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. While randomized, genetic algorithms are not simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance.

A genetic algorithm consists of a string representation of the nodes in the search space, a set of genetic operators for generating new search nodes, a fitness function to evaluate the search nodes, and a stochastic assignment to control the genetic operators.

Typically, a genetic algorithm consists of the following steps:

1. Initialization– an initial population of the search nodes are randomly generated.

2. Evaluation of fitness function–the fitness value of each node is calculated according to the fitness function (objective function).

3. Genetic operations–new search nodes are generated randomly by examining the fitness value of the search nodes and applying the genetic operators to the search nodes.

4. Repeat step 2 and 3 until convergent.

From the above description, we can see that genetic algorithms utilize the notion of survival of the fittest; passing "good" genes to the next generation of strings, and combining different strings to explore new search points. The construction of a genetic algorithm for any problem can be separated into four distinct and yet related tasks:

(1) the choice of the representation of the strings,

(2) the design of the genetic operators,

(3) the determination of the fitness function, and

(4) the determination of the probabilities controlling the genetic operators.

Each of the above components will greatly affect the solution obtained and the performance of the genetic algorithm.

In the following sections, we will examine each of the above components for the problem of multiprocessor scheduling.

3.2 String Representation

One of the main criteria in selecting the string representation for the search nodes is that the new string generated from the application of the genetic operators must represents a legal search node. For the multiprocessor scheduling problem, a legal search node should satisfy the same rule as the scheduling which is (1) the precedence relations among the tasks and (2) every task is present and appears only once in the schedule. The string representation used in this thesis is based on the schedule of the tasks in each individual processor. This representation eliminates the need to consider the precedence relations between the tasks scheduled to different processors. However, the precedence relations within the processor must still be maintained.

The representation of the schedule for a genetic algorithm must accommodate the precedence relations between the computational tasks. This is resolved by representing the schedule as several lists of computation tasks. Each list represents the computational tasks executed on a processor and the

order of the tasks in the list indicates the order execution (see Fig. 3.1). This ordering allows us to maintain the precedence relations for the tasks executed in a processor (intraprocessor precedence relation) and ignore the precedence relations between tasks executed in different processors (interprocessor precedence relation). This is due to the fact that the interprocessor precedence relation does not come into play until we actually calculate the finishing time of the schedule.

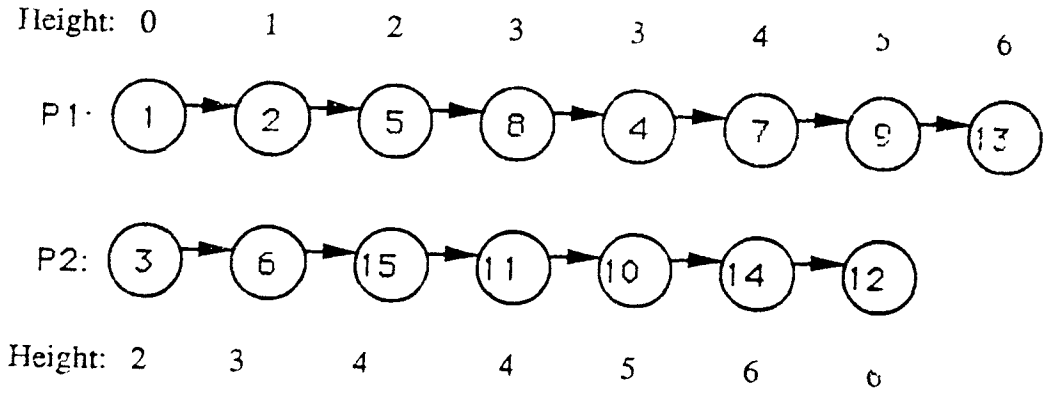

Figure 3.1: String representation of the schedule in Fig. 2.2

The *height* function in a way conveys the precedence relations between the tasks. In order to simplify the construction of the genetic operator, we will impose the following *ordering condition* on the schedules we generate.

The list of tasks within each processor of the schedule is ordered in ascending order of their height.

For example, consider the first processor P1 in Figure 3.1, we have

$$height(1) < height(2) < height(5) < height(8)$$

$$\leq height(4) < height(7) < height(9) < height(13)$$

One major drawback of having this condition is that the optimal schedule may not satisfy it. To reduce the likelihood of this happening, we can modify the definition of *height* as following.

If the difference between the *height* of a task T_i, and its successors is greater than 1, then we can define the new *height* of the task (*height'*) to be a random integer between $height(T_i)$ and $max(height(T_j)) - 1$, over all $T_j \in Succ(T_i)$; otherwise, its *height* is unchanged.

3.3 Initialization

One of the merits of genetic algorithms is that it search many nodes in the search space in parallel. This requires us to randomly generate an initial population of the search nodes. The population size is typically problem dependent and has to be determined experimentally The Initial Population Algorithm are following:

Initial Population Algorithm:

This algorithm randomly generates a schedule of the task graph TG for a multiprocessor system with $procN$ processors.

1 [Initialize.] Compute height and level for every task in TG.

2 [Separate the tasks according to their height.] Partition the tasks in TG into different sets, $G(h)$ ($G(h)$ defined as the set of tasks with *height* h), according to the value of *height*.

3 [Loop $procN - 1$ times.] For first $procN - 1$ processors, do step 4 and 5.

4 [Loop for each set of tasks.] For each set, $G(h)$, $NG(h)$ (number of tasks in $G(h)$ and do step 5.

5 [Form a schedule for a processor.] Randomly generate a number ,r, between 0 and $NG(h)$. Pick r tasks from $G(h)$ in order of level descent, and assign it to the current processor.

6 [Last processor.] Assign the remaining tasks in the sets $G(h)$.

The quality of the schedules generated in the initial population will affect the performance and result obtained by the genetic algorithm.

3.4 Fitness Function

The fitness function is essentially the objective function for the problem. It provides a mean to evaluate the search nodes and also controls the reproduction process. For the multiprocessor scheduling problem, we can consider factors such as throughput, finishing time, and processor utilization for the fitness function. The fitness function used for our algorithm is based on the finishing time of the schedule. To compute the finishing time of a schedule using our representation, we need to calculate two sets of values, *ftt* and *ftp*, defined as

ftt(i) - finishing time of task T_i.

ftp(j) - finishing time for processor P_j. The finishing time of a schedule is then

$$FT = \max_{T_j} ftp(j)$$

To calculate the finishing time of each processor, we can apply the following algorithm.

Finishing Time Algorithm:

This algorithm calculates the finishing time of a schedule.

1 [Initialize.] For each processorP_j, $ftp(j) \leftarrow 0$.

2 [Loop for every task T_i.] For each task in processorP_j do steps 3-5.

3 [Task T_i has no predecessor.] If $Pred(T_i) = 0$, $ftt(i) \leftarrow et(i) + ftp(j)$.

4 [Task T_i has predecessors.] otherwise, $ftt(i) \leftarrow et(i) + max\ \{ftt(k), ftp(j)\}$ where $T_k \in Pred(T_i)$.

5 [Update the finishing time of processor P_j.] $ftp(j) \leftarrow ftp(j) + ftt(i)$.

6 [Calculate the finishing time of the schedule.] $FT \leftarrow max\ ftp(j)$ over all P_j.

The fitness function is then $1/FT$.

Intuitively, we can think of the fitness function as some measure of profit, utility, or goodness that we want to maximize. Copying string according to their fitness values means that strings with a higher value will have a higher

probability of contributing one or more offspring in the next generation. This operator, of course, is an artificial version of natural selection, a Darwinian survival of the fittest among the strings. In natural population fitness is determined by a creature's ability to survive predators. In our unabashedly artificial setting, the objective function is the final arbiter of the string-creature's life or death.

3.5 Genetic Operator

The function of the genetic operator is to create new search nodes based on the current population of search nodes. New search nodes are typically constructed by combining or rearranging parts of the old search nodes. The idea (as in genetics) is that with a proper chosen string representation of the search nodes, certain structures in the representation would represent the "goodness" of that search node. Thus, by combining the good structures of two search nodes, we may result in an even better one. Relating this idea to multiprocessing scheduling, certain portions of a schedule may belong to the optimal scheduling. By combining several of the "optimal" parts, we can find the optimal schedule efficiently.

One important criterion in constructing the genetic operator is that the number of illegal strings (strings that do not represent any nodes in the search space) generated must be relatively small compared to the legal ones. If this in not the case, then computation time will be wasted in generating the illegal strings and determining the legality of the new strings. For the multiprocessor scheduling problem, the genetic operators used must enforce the intraprocessor precedence relations. This would ensure that the new strings generated

will always represent legal search nodes. We will develop a genetic operator for the scheduling problem based on reproduction, crossover and mutation.

3.5.1 Reproduction

The reproduction process is typically based on the fitness value of the strings. The principle is that the strings with higher fitness value will have a higher chance of surviving into the next generation. The reproduction operator may be implemented in algorithmic form in a number of ways. The easiest way is to implement it as a biased roulette wheel where each string in the population has a slot sized proportional to its fitness value. Suppose the size of the population of strings is $popN$.

By summing the value over all strings in the population, we obtain a total fitness value, $NSUM$. We then construct a roulette wheel which is divided into $popN$ slots. Each slot corresponds to a string. The length of each slot is $f_i/NSUM$. To obtain the new population, we generate a random number and use it as an index into the roulette wheel to select the string into the roulette wheel to select a string. Once a string has been selected for reproduction, an exact replica of the string is made. This string is then entered into a mating pool, a tentative new population, for further genetic operation.

We can make a slight modification to the basic reproduction operation to increase the performance of the genetic algorithm. This is done always passing the best string in the current generation to the next generation. The reproduction operation is summarized in the following algorithm.

Reproduction Algorithm:

This algorithm performs reproduction on a population of strings POP and generates a new population of strings $NEWPOP$.

1 [Initialize.] Let $popN \leftarrow$ number of strings in POP.

2 [Construct the roulette wheel.] $NSUM \leftarrow$ sum of all the fitness value f_i of the strings in POP; form $popN$ slots with slot length equal to $f_i/NSUM$ and assign string to the slots according to the fitness value of the string.

3 [Loop $NPOP - 1$ time.] Do step 4 $NPOP - 1$ times.

4 [Pick a string.] Generate a random number between 0 and 1 and use it to index into the slots to find the corresponding string; add this string to $NEWPOP$.

5 [Add the best string.] Add the string with the highest fitness value in POP to $NEWPOP$.

3.5.2 Crossover

The mechanics of reproduction and crossover are simple. It involves random number generation, string copies, and some partial string exchanges. Nonetheless, the combined emphasis of reproduction and the structured, though randomized, information exchange of crossover give genetic algorithms much of their power.

For example, consider the two strings(schedules) shown in Fig. 3.2. We can create new strings by exchanging portions of the two strings using the following method.

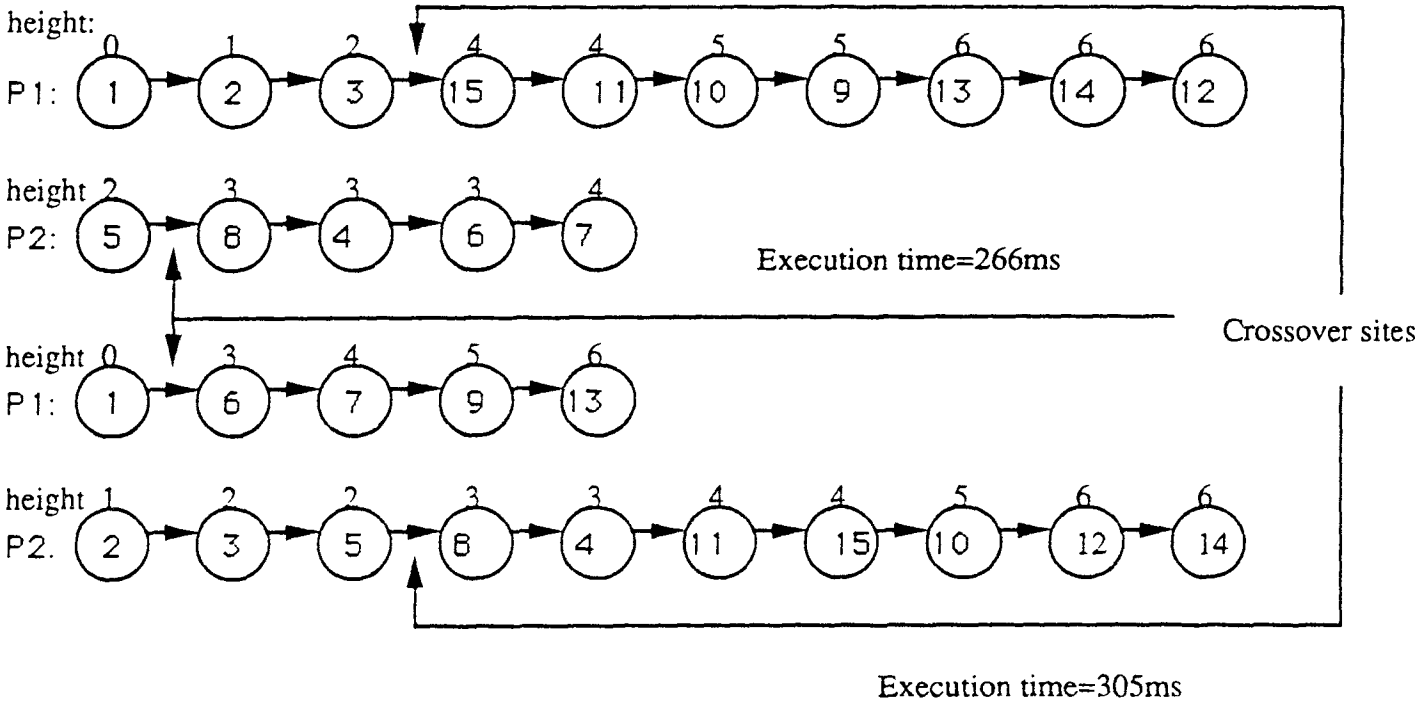

Figure 3.2: Two different strings before crossover

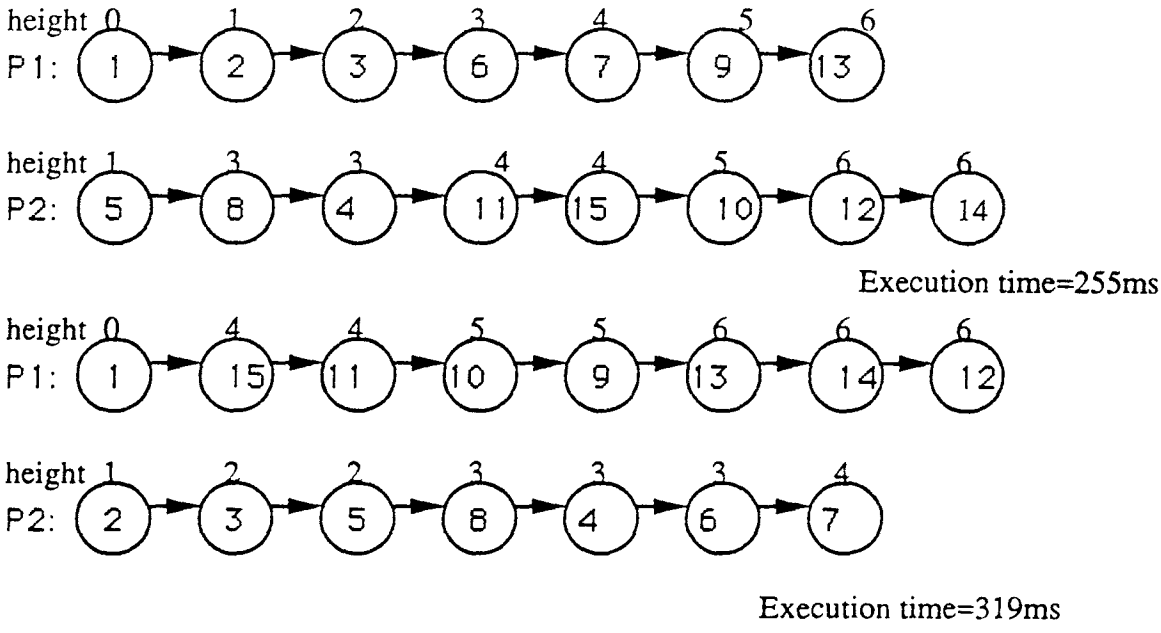

Figure 3.3: Two new strings after crossover

1. Select sites (crossover sites) where we can cut the lists into two halves(see Fig. 3.2).

2. Exchange the bottom halves of P1 in string A and string B.

3. Exchange the bottom halves of P2 in string A and string B.

The new strings created are shown in Fig. 3.3. Notice that one of the new strings has a smaller finishing time than the previous two strings. The operation described above can be easily extended to m processors and appears to be quite effective. However, we still have to find a method to select the crossover sites and show that the new strings generated are legal.

Notice that the crossover sites used in the above example always lie between tasks with two different heights. and the result is two legal strings. It appears that the selection of the crossover sites is strong related to the legality of the strings generated. It can be easily shown that if the crossover sites are chosen in such a way that (1) the height of the tasks next to the crossover site are different in heights and (2) the heights of the all the tasks in front of the crossover site are the same or smaller then the new strings generated will always be legal schedules. The crossover operation is summarized in the following algorithm.

Crossover Algorithm:

This algorithm performs the crossover operation on two schedules and generate two new schedules, A and B.

1 [Select crossover site.] Randomly generate a number, c, between 0 and $heightN$(maximum height value of TG).

2 [Loop for every processor.] For each processor in both strings, do step 3.

3 [Find the crossover sites.] Find consecutive tasks T_j^i and T_k^i in processor P_i such that $height(T_j^i) \leq c < height(T_k^i)$, $height(T_k^i)$ are the same for all i.

4 [Loop for every processor.] For each processor P_i in string A and string B do step 5.

5 [Crossover] Exchange the bottom halves of the tasks in string A and B for each processor P_i.

Although the crossover operation is powerful, it may not be desirable to perform it all the time. This is because the crossover operation is random in nature and may possibly eliminate the optimal solution. Typically ,its operation is controlled by a crossover probability. The value of this crossover probability is usually determined experimentally.

3.5.3 Mutation

The mutation operator plays a secondary role in the operation of genetic algorithm. Mutation is needed because, even though reproduction and crossover effectively search and recombine extent notions, occasionally they may become overzealous and lose some potentially useful genetic material. In artificial genetic systems, the mutation operator protects against such an irrecoverable loss, and avoid local minimum.

Mutation Algorithm:

This algorithm performs the mutation operator

1 [Find mutation position.] Find the height, h, of the task in processor P_i which will mutated.

2 [Find the new position.] In processor P_{i+1} find the consecutive tasks T_j and T_k such that $height(T_j) \leq h < height(T_k)$.

3 [Mutation.] Move the mutation task from P_i to P_{i+1}, and insert between T_j and T_k.

After mutation, the resultant strings will also satisfy the ordering condition.

3.6 Complete Algorithm

We can now combined all the algorithms discussed above to form the genetic algorithm for multiprocessor scheduling.

Find-Schedule:

This algorithm attempts to solve the multiprocessor scheduling problem.

1 [Initialize.] Call Generate-Schedule N times and store the strings created in POP.

2 [Repeat until convergent.] Do steps 3-7 until convergent.

3 [Compute fitness values.] Compute the fitness value of each string in POP.

4 [Perform Reproduction.] Call Reproduction.

5 [Perform Crossover.] Do step 6 $NPOP/2 - 1$ times.

6 [Crossover.] Pick two strings from $NEWPOP$ and call Crossover with a probability P_c. If Crossover is performed, put the new strings in TMP; otherwise put the two strings picked in TMP.

7 [Perform Mutation.] For each task of string in TMP, call Mutation with a probability of P_m. If mutation is performed, put the new string in POP; otherwise put the string picked in POP.

Chapter 4

SIMULATION RESULTS

The proposed genetic algorithm was tested on randomly generated task graphs and the task graph for Newton-Euler inverse dynamic equation for the Stanford manipulator and the Elbow Manipulator [12]. The simulation program was tested in a DEC system 5500.

Four random task graphs are used and they are listed in Table 4.1-4.4

Case 1:

The total number of tasks are 40. The maximum execution time is 50 ms. The maximum number of predecessors is 4. The maximum number of successors is 4. The crossover probability is 0.5. The mutation probability is 0.01. The Population size 20.

Table 4.1: Randomly generated tasks for *Case 1*

Task Number	et (ms)	predecessor	successor	Task Number	et (ms)	predecessor	successor
1	41	-	2 3 4 5	21	42	10 11	26 36
2	35	1	3 4 5 6	22	18	19 15 7	25 34 37
3	50	2 1	4 5 6 7	23	31	19 14	32
4	48	3 2 1	8 9 12 13	24	49	16 17	27 37 38
5	23	1 2 3	6 8 11 15	25	1	13 22 9	40
6	16	2 3 5	7 10 11 12	26	48	21 10 18	35
7	5	3 6	8 14 18 22	27	14	24	28 29 35 37
8	25	7 5 4	15	28	49	16 10 27	-
9	13	4	25 36	29	5	27	-
10	29	6	21 26 28	30	14	15	31
11	34	6,5	17 21 32	31	49	30	34
12	6	4 6	14 40	32	41	11 23	33 34
13	10	4	14 16 18 25	33	8	32	-
14	47	7.13,12	18 23	34	11	31 32 22	-
15	20	8 5	19 22 30	35	43	27 26 17	39
16	48	13	19 24 28	36	14	9 20 21	-
17	6	11	24 35 38	37	19	27 24 22	-
18	36	14 13 7	26 38	38	1	24 17 18	-
19	17	15 16	20 22 23	39	42	35	-
20	47	19	36	40	26	25 12	-

Case 2:

The total number of tasks are 50. The maximum execution time is 150 ms. The Maximum number of predecessors is 5. The Maximum number of successors is 5. The crossover probability is 0.5. The mutation probability is 0.01. The population size 20.

Table 4.2: Randomly generated tasks for *Case 2*

Task Number	et (ms)	predecessor	successor	Task Number	et (ms)	predecessor	successor
1	91		2 4 5 6 16	26	4	16 18 23	48
2	35	1	3 4 5 6 23	27	69	11 22 9 20	50
3	100	2	4 5 7 10 11	28	37	22 9	29
4	87	3 2 1	5 12 20 36 42	29	49	19 28	42
5	73	1 2 3 4	8 13 14 15 37	30	105	14 9	31 38 47
6	15	2 1	7 8 18 23 35	31	134	22 30 21	37
7	3	3 6	9 10 11 17 20	32	56	23 13 24	34 38 42 43 45
8	91	6 5	9 11 13 15 17	33	46	22	44
9	5	7 8	22 25 27 28 30	34	7	32 22 18 17	
10	75	7 3	13 16 21 23 24	35	38	6 24 19	39 44
11	66	8 3 7	13 14 24 27	36	116	4	44
12	28	4	15 16	37	93	13 31 24 5	
13	63	5 10 11 8	19 32 37	38	12	32 30 19	46
14	17	5 11	16 19 30 40 41	39	26	35	49
15	62	8 5 12	44 45 47	40	106	14	49
16	19	10 14 1 12	21 26 42 47	41	28	14	
17	89	7 8	18 20 21 22 34	42	79	16 29 4 32	46
18	47	6 17	19 22 26 34	43	140	32	48
19	136	18 13 14	20 29 35 38	44	73	33 35 15 36	
20	99	17 4 7 19	22 25 27	45	85	15 32	
21	56	16 17 10	31	46	85	42 38	
22	33	17 20 18 9	27 28 31 33 34	47	88	30 15 16	
23	16	10 6 2	26 32	48	37	43 26	
24	51	11 10	32 35 37 50	49	149	39 40	
25	111	20 9		50	91	27 24	

Case 3:

The total number of tasks are 60. The maximum execution time is 50 ms. The maximum number of predecessors is 5. The maximum number of successors is 5. The crossover probability is 0.5. The mutation probability is 0.01. The population size 20.

Table 4.3: Randomly generated tasks for *Case 3*

Task Nr.	et (ms)	pred.	succ.	Task Nr.	et (ms)	pred.	succ.
1	41	-	2 4 5 6 16	31	34	22 30 21	37 52
2	35	1	3 4 5 6 23	32	6	23 13 24	34 38 42 43 45
3	50	2	4 5 7 10 11	33	46	22	44 53
4	37	3 2 1	5 12 20 36 42	34	7	32 22 18 17	57 58
5	23	1 2 3 4	8 13 14 15 37	35	38	6 24 19	39 44
6	15	2 1	7 8 18 23 35	36	16	4	44
7	3	3 6	9 10 11 17 20	37	43	13 31 24 5	52
8	41	6 5	9 11 13 15 17	38	12	32 30 19	46
9	5	7 8	22 25 27 28 30	39	26	35	49 53 56
10	25	7 3	13 16 21 23 24	40	6	14	49 60
11	16	8 3 7	13 14 24 27	41	28	14	55
12	28	4	15 16 58	42	29	16 29 4 32	46
13	13	5 10 11 8	19 32 37	43	40	32	48
14	17	5 11	16 19 30 40 41	44	23	33 35 15 36	-
15	12	8 5 12	44 45 47 53	45	35	15 32	-
16	19	10 14 1 12	21 26 42 47 57	46	35	42 38	-
17	39	7 8	18 20 21 22 34	47	38	30 15 16	-
18	47	6 17	19 22 26 34	48	37	43 26	-
19	36	18 13 14	20 29 35 38	49	49	39 40	53 57
20	49	17 4 7 19	22 25 27	50	41	27 24	-
21	6	16 17 10	31	51	4	30 27 26	-
22	33	17 20 18 9	27 28 31 33 34	52	39	37 26 31 28	-
23	16	10 6 2	26 32 54	53	25	49 15 33 39	57 60
24	1	11 10	32 35 37 50	54	28	23	59
25	11	20 9	-	55	39	27 28 30 41	56
26	4	16 18 23	48 51 52 58	56	42	39 55 27	-
27	19	11 22 9 20	50 51 55 56 59	57	35	49 34 16 53	-
28	37	22 9	29 52 55	58	6	12 26 34	-
29	49	19 28	42	59	25	27 54	-
30	5	14 9	31 38 47 51 55	60	15	53 40	-

Case 4:

The total number of tasks are 60. The maximum execution time is 500 ms. The maximum number of predecessors is 5. The Maximum number of successors is 5. The crossover probability is 0.5. The mutation probability is 0.01. The population size 20.

Table 4.4: Randomly generated tasks for Case 4

Task Nr.	et (ms)	pred.	succ.	Task Nr.	et (ms)	pred.	succ.
1	91	-	2 4 5 6 16	31	434	22 30 21	37 52
2	85	1	3 4 5 6 23	32	6	23 13 24	34 38 42 43 45
3	400	2	4 5 7 10 11	33	96	22	44 53
4	387	3 2 1	5 12 20 36 42	34	307	32 22 18 17	57 58
5	223	1 2 3 4	8 13 14 15 37	35	38	6 24 19	39 44
6	465	2 1	7 8 18 23 35	36	316	4	44
7	153	3 6	9 10 11 17 20	37	243	13 31 24 5	52
8	241	6 5	9 11 13 15 17	38	462	32 30 19	46
9	105	7 8	22 25 27 28 30	39	276	35	49 53 56
10	25	7 3	13 16 21 23 24	40	156	14	49 60
11	66	8 3 7	13 14 24 27	41	428	14	55
12	328	4	15 16 58	42	179	16 29 4 32	46
13	263	5 10 11 8	19 32 37	43	440	32	48
14	317	5 11	16 19 30 40 41	44	323	33 35 15 36	-
15	462	8 5 12	44 45 47 53	45	485	15 32	-
16	19	10 14 1 12	21 26 42 47 57	46	485	42 38	-
17	189	7 8	18 20 21 22 34	47	38	30 15 16	-
18	297	6 17	19 22 26 34	48	237	43 26	-
19	486	18 13 14	20 29 35 38	49	449	39 40	53 57
20	99	17 4 7 19	22 25 27	50	341	27 24	-
21	306	16 17 10	31	51	254	30 27 26	-
22	183	17 20 18 9	27 28 31 33 34	52	39/39	37 26 31 28	-
23	466	10 6 2	26 32 54	53	275	49 15 33 39	57 60
24	401	11 10	32 35 37 50	54	228	23	59
25	61	20 9	-	55	139	27 28 30 41	56
26	54	16 18 23	48 51 52 58	56	142	39 55 27	-
27	119	11 22 9 20	50 51 55 56 59	57	435	49 34 16 53	-
28	337	22 9	29 52 55	58	306	12 26 34	-
29	349	19 28	42	59	25	27 54	-
30	105	14 9	31 38 47 51 55	60	165	53 40	-

Table 4.5: Simulation results for randomly generated tasks for *Case 1-4*

Proc. Nr.	Case 1		Case 2		Case 3		Case 4	
	mut. et(μs)	no mut. et(μs)	mut. et(μs)	no mut. et(μs)	mut. et(μs)	no mut. et(μs)	mut. et(μs)	no mut. et(μs)
1	1081	-	3221	-	1579	-	14829	-
2	604	645	1806	1840	870	904	7657	7990
3	476	481	1368	1402	690	714	5652	5758
4	429	439	1233	1293	632	634	4738	4970
5	400	400	1197	1209	586	588	4403	4510
6	400	400	1131	1131	581	581	4031	4031
7	400	400	1131	1131	581	581	4031	4031
CR-path	400	400	1131	1131	581	581	4031	4031

The simulation results for the four cases with mutation and without mutation are listed in Table 4.5.

Figures 4.1-4.3 respectively plots the finishing time versus the number of generations for *Case 3* with 3, 4 and 5 processors. In all cases, a better schedule is found with the mutation operator enabled. Further more, with increase number of processors. the genetic algorithm found a optimal schedule with finishing time equal to the critical path length.

Figure 4.4 illustrated the relationship between the finishing time and the number of processors used for four cases.

Genetic algorithm was also tested with the problem of scheduling the robot inverse dynamics computation for the Stanford manipulator and the elbow manipulator. The tasks for the Stanford manipulator and elbow manipulator are respectively listed in Table 4.6 and Table 4.7. Table 4.8 comparers the simulation results with the optimal solution for the Stanford manipulator [10]. It can be seen that there is a maximum error between the solution found by genetic algorithms and the optimal sulotion is 17%. Figure 4.5 shows the best schedule obtained by genetic algorithm for the stanford manipulator with 2 processors. Figures 4.6-4.10 plot the finishing time obtained by the genetic algorithms versus the number of generations. Table 4.9 compares the simulation results with the optimal solution for the elbow manipulator [12]. The maximum error between the solution found by genetic algorithm and optimal sulotion is 14%.

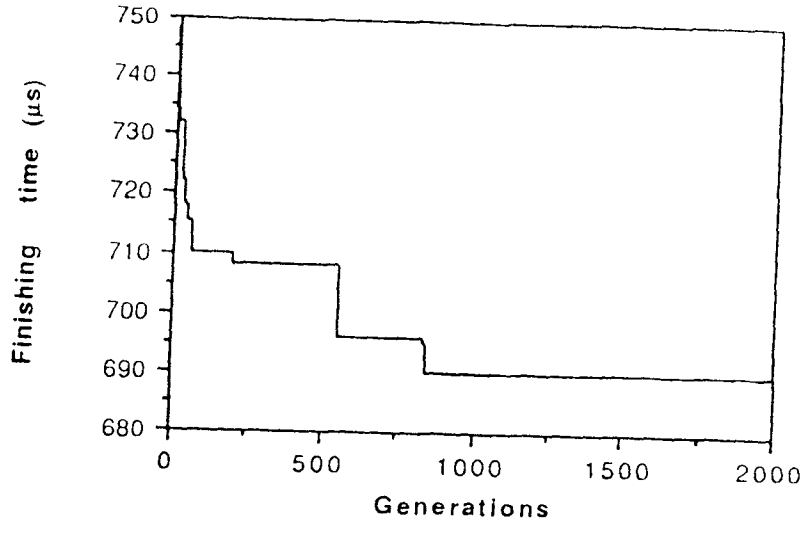

Figure 4.1: Finishing time vs number of generations for *Case 3* with three processors

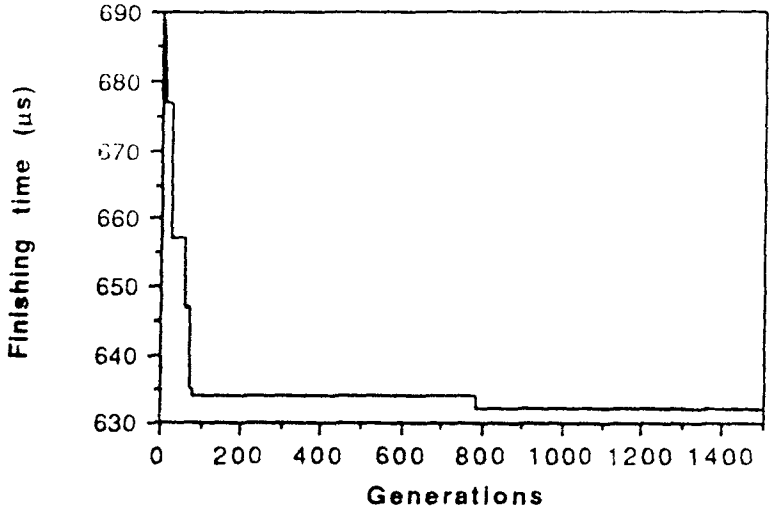

Figure 4.2: Finishing time vs number of generations for *Case 3* with four processors

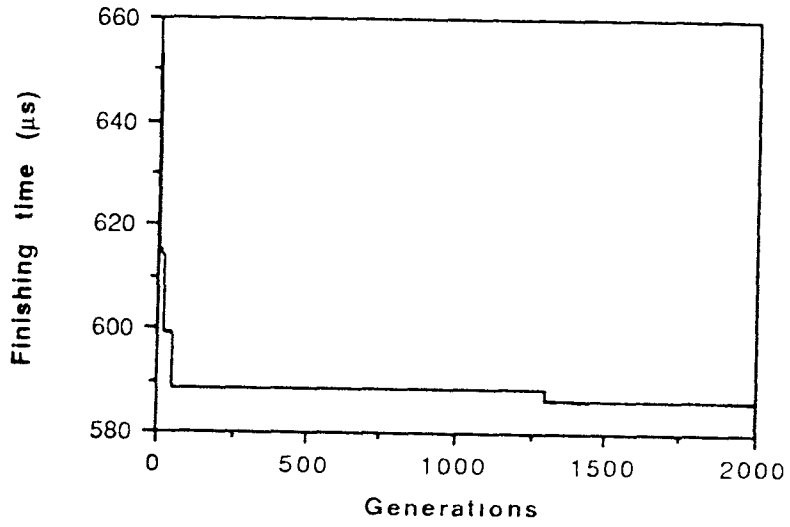

Figure 4.3: Finishing time vs number of generations for *Case 3* with five processors

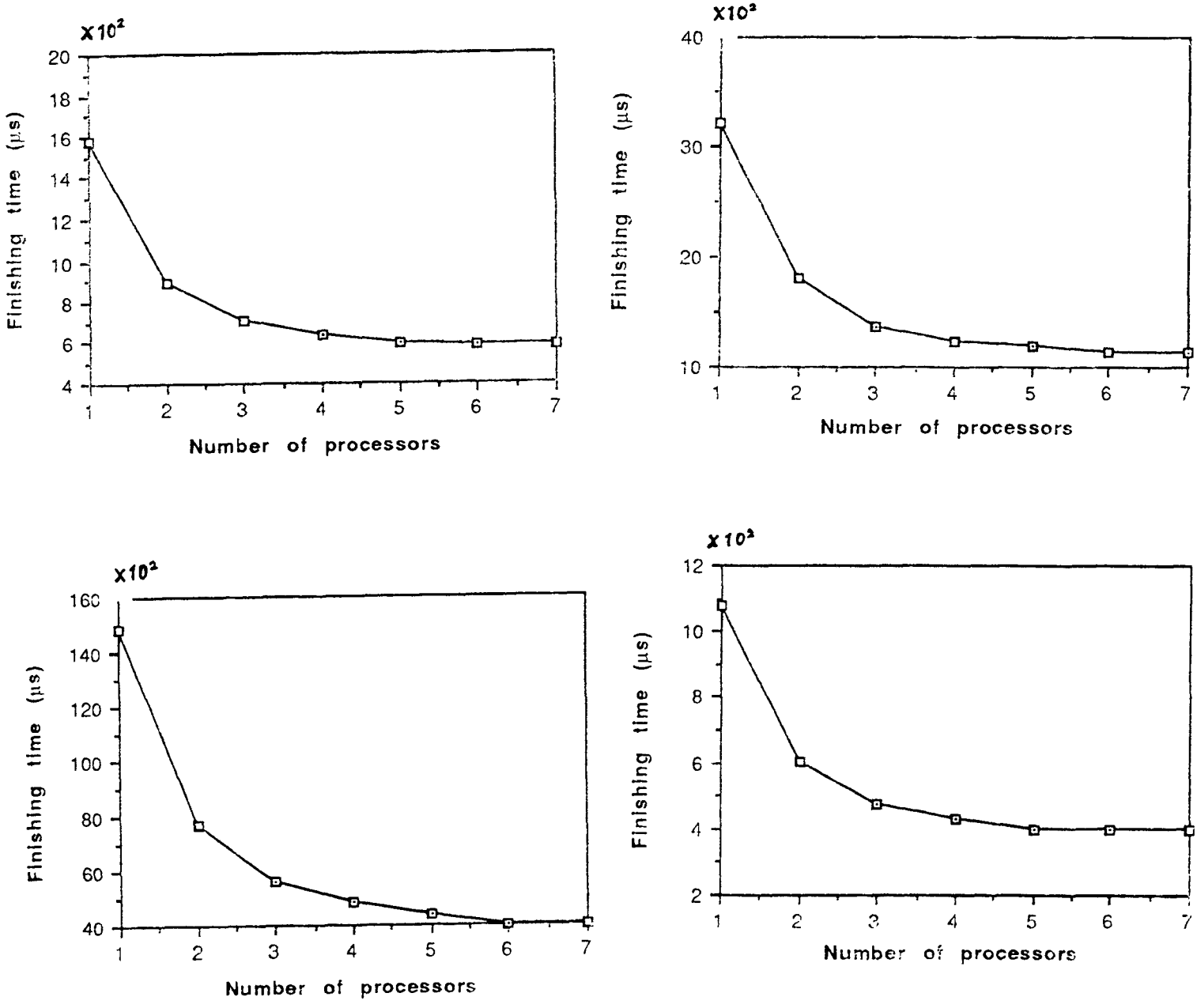

Figure 4.4: Plots illustrating the relationship between finishing time and the number of processors in cases 1-4

Table 4.6: List of tasks for the Newton-Euler method for Stanford manipulator

Task Nr	et (μs)	pred	succ.	Task Nr.	et (μs)	pred	succ.
1	1	-	3 5 4 6 7 8	45	32	35	53 54 55
2	1	-	16 17	46	66	35	56
3	5	1	88	47	15	41	52
4	5	1	6	48	24	41	56
5	10	1	9 10 11 12	49	39	32 37	70
6	28	1 4	9 13 14 15	50	28	40	56 57
7	5	1	16	51	40	41 43	58 59 60
8	10	1	16	52	12	47 44	69
9	1	5 6	18 20 21 22 23 24 25 26 27	53	111	45	62
10	57	5	28	54	84	45	63
11	66	5	28	55	38	45	61
12	38	5	19	56	39	48 46 50	67 68
13	15	6	29	57	28	50	61
14	24	6	29	58	69	51	62
15	10	6	19	59	42	51	63
16	15	2 7 8	81 87	60	10	51	61
17	10	2	19	61	24	60 55 57	63
18	10	9	20	62	12	58 53	65
19	24	12 15 17	29 30	63	39	59 54 61	64
20	40	9 18	32 33 41	64	42	63	65
21	32	9	34 35 36 37	65	12	62 64	66
22	57	9	38	66	28	65	69
23	15	9	38	67	24	56	69
24	10	9	39	68	40	56	72
25	38	9	39	69	24	52 66 67	71
26	53	9	31	70	24	49	75
27	10	9	31	71	28	69	75
28	12	10 11	84	72	40	68	73 74
29	39	13 14 19	82	73	40	39 72	79 80
30	4	19	31	74	38	72	76
31	24	26 27 30	39 40	75	24	42 71 70	76
32	24	20	49	76	37	74 75	78
33	15	20	42	77	10	39	78
34	10	21	41	78	24	38 76 77	83
35	32	21	43 44 45 46	79	10	73	83
36	57	21	42	80	12	73	81 85
37	66	21	49	81	40	16 80	-
38	12	22 23	78	82	24	29	84
39	39	24 25 31	73 77	83	8	78 79	84
40	28	31	50	84	24	28 82 83	86
41	40	20 34	47 48 51	85	24	80	86
42	12	33 36	75	86	36	84 85	88
43	10	35	51	87	24	16	88
44	57	35	52	88	24	3 86 87	-

Table 4.7: List of tasks for the elbow manipulator

Task Nr	et (ms)	pred	succ.	Task Nr.	et (ms)	pred.	succ.
1	10	-	2 7 36 60	53	350	35 41 47	77 91
2	100	1	3 8 24 31 37 61	54	100	12	66
3	320	2	4 9 25 32 38 62	55	150	13	67
4	320	3	5 10 26 33 39 63	56	150	14	68
5	320	4	6 11 40 64	57	150	15	69
6	320	5	41 65	58	150	16	70
7	100	1	13	59	150	17	71
8	100	2	14	60	240	1	66
9	100	3	15	61	570	2	67
10	100	4	16	62	570	3	68
11	100	5	17	63	570	4	69
12	10		13 42 54	64	570	5	70
13	360	7 12	14 27 43 55	65	570	6	71
14	400	8 13	15 28 44 56	66	10	54 60	92
15	400	9 14	16 29 45 57	67	120	55 61	93
16	400	10 15	17 46 58	68	120	56 62	94
17	400	11 16	47 59	69	120	57 63	95
18	10	-	30	70	120	58 64	96
19	280	30	31	71	120	59 65	97
20	280	31	32	72	400	48 73	
21	280	32	33	73	400	49 74	72
22	280	33	34	74	400	50 75	73 78
23	280	34	35	75	400	51 76	74 79
24	100	2	31	76	400	52 77	75 80
25	100	3	32	77	10	53	76
26	100	4	33	78	240	74	82
27	100	13	31	79	420	75	83
28	100	14	32	80	420	76	84
29	100	15	33	81	280	93	92
30	10	18	19 48	82	400	78 94	92
31	400	2 19 24 27	20 49	83	400	79 95	94
32	440	3 20 25 28	21 50	84	400	80 96	95
33	440	4 21 26 29	22 51	85	280	97	96
34	10	22	23 52	86	100	48	92
35	10	23	53	87	140	49	93
36	150	1	48	88	140	50	94
37	340	2	49	89	140	51	95
38	340	3	50	90	100	52	96
39	340	4	51	91	100	53	97
40	340	5	52	92	200	66 81 86	98
41	340	6	53	93	200	67 82 87	81 99
42	50	12	48	94	200	68 83 88	82 100
43	100	13	49	95	200	69 84 89	83 101
44	100	14	50	96	200	70 85 90	84 102
45	100	15	51	97	80	71 91	85 103
46	100	16	52	98	180	92	-
47	100	17	53	99	40	93	-
48	230	30 36 42	72 86	100	40	94	-
49	350	31 37 43	73 87	101	180	95	-
50	350	32 38 44	74 88	102	180	96	-
51	350	33 39 45	75 89	103	40	97	-
52	350	34 40 46	76 90	-	-	-	-

Table 4.8: Simulation results for the Newton-Euler method for Stanford manipulator

Number of Processors	Exec. time(μs) of GA solution	Exec. Time(μs) of Optimal solution	Error %
1	2480	2480	0
2	1310	1242	5
3	937	843	11
4	774	659	17
5	679	586	15
6	627	573	9
7	609	569	7

Table 4.9: Simulation results for elbow manipulator

Number of Processors	Exec. Time(ms) of GA solution	Exec. Time(ms) of Optimal solution	Error %
1	23.42	23.42	0
2	12.34	11.71	5
3	8.94	7.81	14
4	7.26	6.63	9
5	6.83	6.63	3
6	6.63	6.63	0

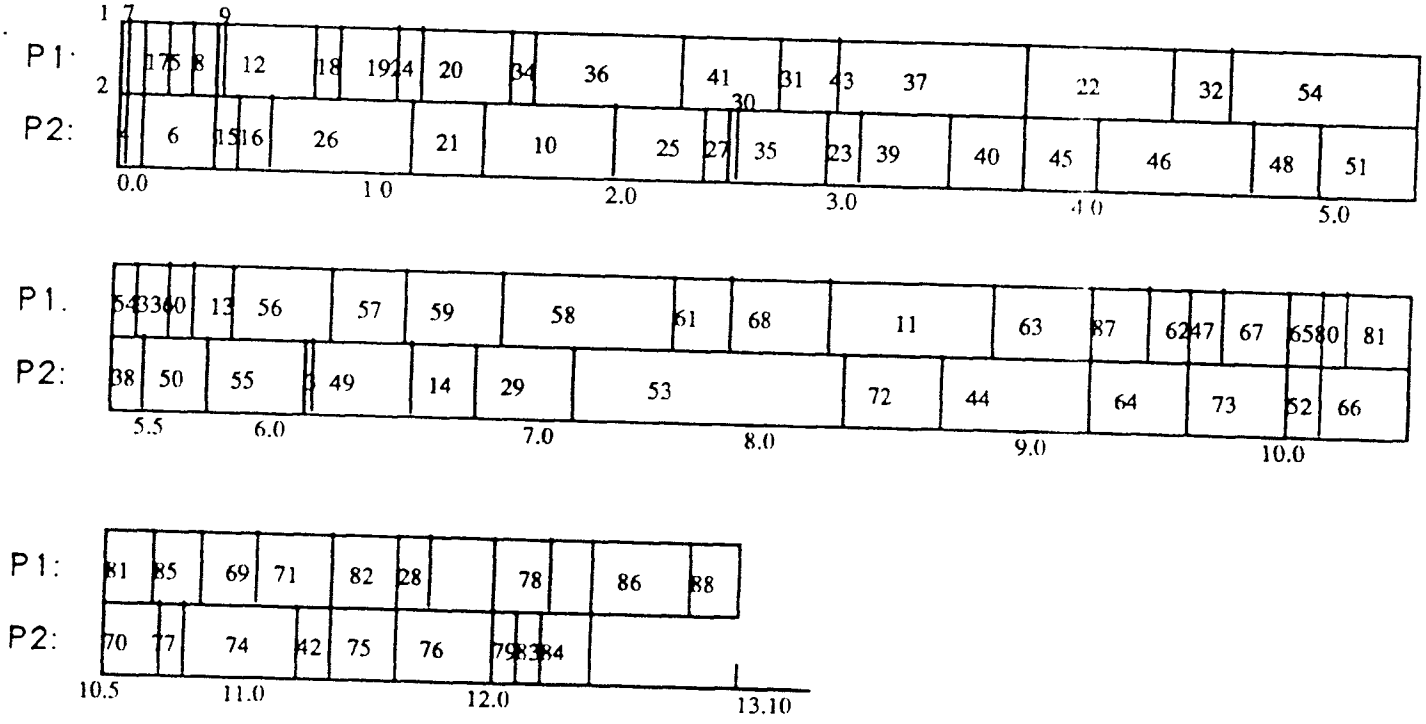

Figure 4.5: Schedule of two processors for Sanford manipulator

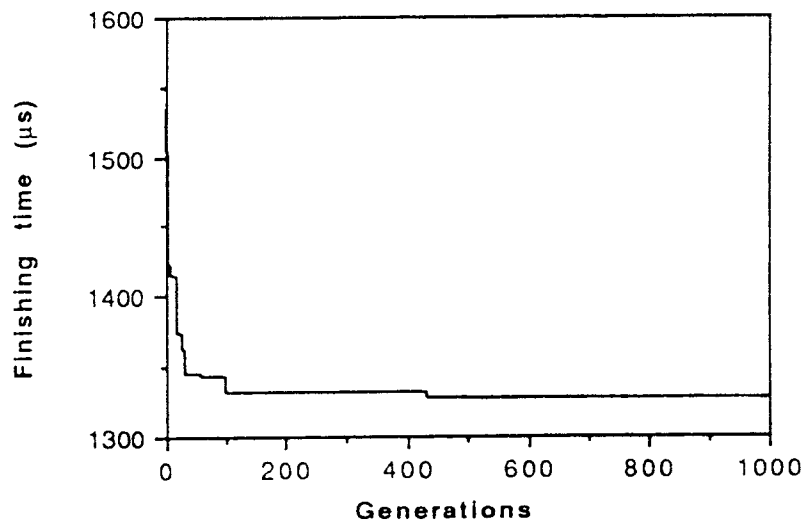

Figure 4.6: Finishing time vs generations for the Newton-Euler method with two processors

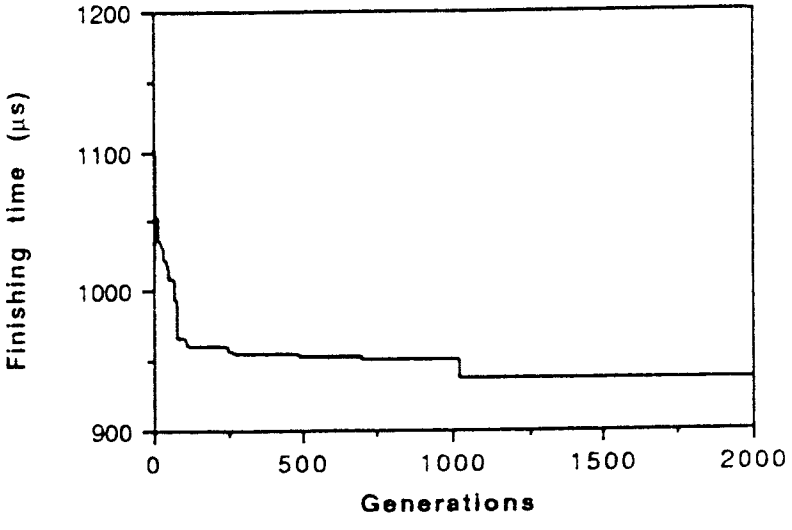

Figure 4.7: Finishing time vs generaions for the Newton-Euler method with three processors

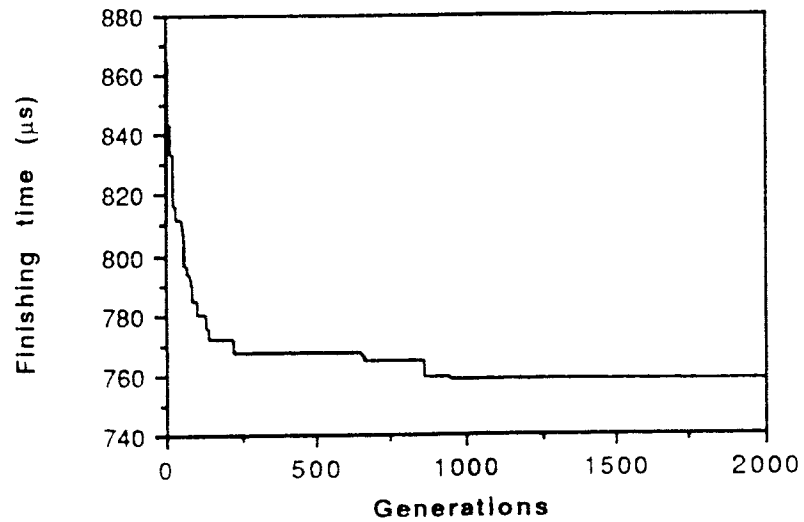

Figure 4.8: Finishing time vs generaions for the Newton-Euler method with four processors

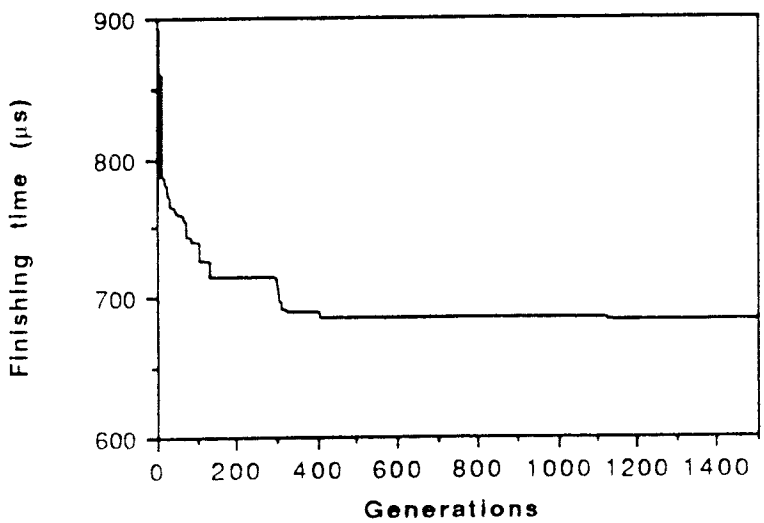

Figure 4.9: Finishing time vs generaions for the Newton-Euler method with five processors

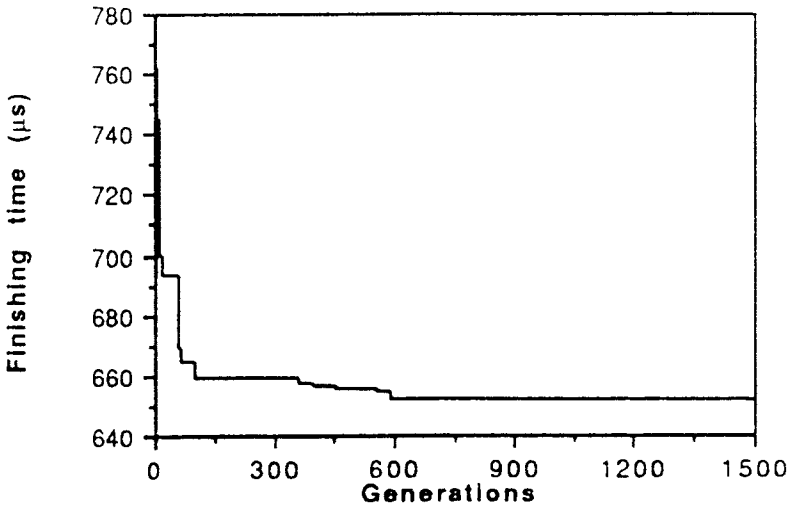

Figure 4.10: Finishing time vs generaions for the Newton-Euler method with six processors

Chapter 5

CONCLUSION

In this thesis, we considered the problem of scheduling a task graph onto a microprocessor system so that the schedule length is minimized. A stochastic search method based on the genetic algorithms is proposed. The representation of the search nodes (schedules) used are in the form of lists of computation tasks. This eliminates the need to consider the precedence relations between tasks in different processors and allow us to construct an efficient crossover genetic operator. The genetic operator developed takes into account the precedence relations of the tasks and guarantee that the new strings generated are legal.

The proposed genetic algorithm was tested on a task graph for the Newton-Euler inverse dynamic equations for the Stanford manipulator and elbow manipulator. This algorithm was also tested on randomly generated task graphs.

References

[1] M. R. Garey and D. S. Johnson, *Computer and intractability. A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[2] C. V. Ramamoorthy et al., " Optimal scheduling strategies in a multiprocessor system, " *IEEE Trans. Computers*, Feb. 1972. vol. C-21, pp. 137-146.

[3] T. L. Adams et al., " A Comparison of list schedules for parallel processing systems, " *Comm. Assoc. Computing Machinery*, Dec. 1974. vol. 17, pp. 685-690.

[4] H. Hasahara and S. Narita, " Practical multiprocessing scheduling algorithms for efficient parallel processing, " *IEEE Transactions on Computers*, November 1984. vol. C-33, no. 11, pp. 1023-1029.

[5] C. L. Chen, C. S. G. Lee, and E. S. H. Hou, " Efficient scheduling algorithms for robot inverse dynamics on a multiprocessor system, " *IEEE Transactions on Systems, Man, and Cybernetics*, December 1988. vol. 18, no. 5 pp. 729-743.

[6] M. J. Gonzalez, " Deterministic Processor Scheduling, " Computing Surveys, September 1977. vol. 9, no. 3, pp. 173-204.

[7] *Proc. of the First Int. Conf. on Genetic Algorithms and Their Applications* (Carnegie-Mellon University, Pittsburgh, PA), July 24-26, 1985.

[8] *Proc. of the Second Int. Conf. on Genetic Algorithms and their Applications* (MIT. Cambridge, MA), July 28-31, 1987.

[9] D. E. Goldberg, *Genetic algorithms and their applications*, Addison Wesley, 1989.

[10] L. Davis, "Job shop scheduling with genetic algorithms, " *Proc. of the First Int. Conf. on Genetic and their Applications* (Carnegie-Mellon University, Pittsburgh, PA), July 24-26, 1985, pp. 136-140.

[11] E. Hou. H. Ren, and N. Ansari, " Multiprocessor scheduling based on genetic algorithms. " *Proc. 16th Annual Conf. IEEE Industrial Electronics Society* (Pacific Grove, CA Nov. 27-30, 1990). Vol. II, pp. 131-1238.

[12] H. Kasahara and S. Narita, " Parallel processing of robot-arm control computation on a multiprocessor system, "*IEEE Journal of Robotics and Automation*, June 1985. vol. RA-1, no. 2, pp. 104-113.