A Genetic Algorithm for Multiprocessor Scheduling

Edwin S. H. Hou, *Member, IEEE*, Nirwan Ansari, *Member, IEEE*, and Hong Ren

***Abstract*— 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 is known to be *NP*-hard, and methods based on heuristic search have been proposed to obtain optimal and suboptimal solutions. Genetic algorithms have recently received much attention as a class of robust stochastic search algorithms for various optimization problems. In this paper, an efficient method based on genetic algorithms is developed to solve the multiprocessor scheduling problem. The representation of the search node is based on the order of the tasks being executed in each individual processor. The genetic operator proposed is based on the precedence relations between the tasks in the task graph. Simulation results comparing the proposed genetic algorithm, the list scheduling algorithm, and the optimal schedule using random task graphs, and a robot inverse dynamics computational task graph for various are presented.**

***Index Terms*—Direct acyclic graph, genetic algorithms, genetic operators, multiprocessor scheduling, NP-hard, optimization, stochastic search algorithms**

I. Introduction

MULTIPROCESSOR scheduling has been a source of challenging problems for researchers in the area of computer engineering. The general problem of multiprocessor scheduling can be stated as scheduling a set of partially ordered computational tasks onto a multiprocessor system so that a set of performance criteria will be optimized. The difficulty of the problem depends heavily on the topology of the task graph representing the precedence relations among the computational tasks, the topology of the multiprocessor system, the number of parallel processors, the uniformity of the task processing time, and the performance criteria chosen. In general, the multiprocessor scheduling problem is computationally intractable even under simplified assumptions [1]. Because of this computational complexity issue, heuristic algorithms have been proposed to obtain optimal and suboptimal solutions to various scheduling problems.

Various approaches to the multiprocessor scheduling problem have been proposed [2]–[8]. Because of the intractability of the problem, heuristic approaches have been developed to solve the problem. Kashara and Narita [5], [6] proposed a heuristic algorithm (critical path/most immediate successors first) and an optimization/approximation algorithm (depth first/implicit heuristic search). Chen *et al.* [7] developed a state-space search algorithm (A*) coupled with a heuristic derived from the Fernandez and Bussell bound to solve the multiprocessor scheduling problem. Hellstrom and Kanal [8] map the multiprocessor problem into a neural network model, asymmetric mean-field network. In this paper, we present a genetic algorithm approach to the multiprocessor scheduling problem.

The multiprocessor scheduling problem considered in this paper is based on the deterministic model; that is, the execution time and the relationship between the computational tasks are known. The precedence relationship among the tasks is represented by an acyclic directed graph, and the task execution time can be nonuniform. We assume that the multiprocessor system is uniform and nonpreemptive; that is, the processors are identical, and a processor completes the current task before executing a new one. This paper presents an efficient method based on genetic algorithms to solve the multiprocessor scheduling problem. Genetic algorithms have recently received much attention as robust stochastic searching algorithms for various optimization problems [9]–[12]. This class of methods is based on the principles of natural selection and natural genetics that combine the notion of survival of the fittest, random and yet structured search, and parallel evaluation of nodes in the search space.

This paper is organized as follows. First, we present the model for multiprocessor scheduling and some related definitions. Next a brief introduction of genetic algorithms is given. The representation of the search nodes and a method for generating initial population are presented next, followed by a discussion on the fitness function, the construction of three genetic operators: crossover, reproduction, and mutation. Finally, we present the genetic algorithm for multiprocessor scheduling and the simulation results.

II. Model and Definitions

A set of partially ordered computational tasks 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, $V = \{T_1, T_2, \cdots, T_m\}$, represents the set of computational tasks to be executed and the directed edges, $E = \{e_{ij}\}$, (e_{ij} denotes a directed edge from vertex T_i to T_j) implies a partial ordering or precedence relation, $\gg$, exists between the tasks. That is, if $T_i \gg T_j$, then task T_i must be completed before T_j can be initiated. A simple task graph, TG, with 8 tasks is illustrated in Fig. 1.

Manuscript received July 5, 1991; revised February 18, 1993. This work was supported in part by the New Jersey Department of Higher Education through NJIT Separately Budgeted Research.

E. S. H. Hou and N. Ansari are with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102.

H. Ren was with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102. She is now with Penril Datability Networks, Carlstadt, NJ 07072.

IEEE Log Number 9214463.

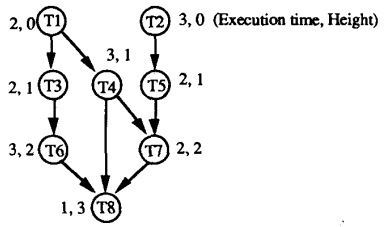


Fig. 1. A task graph TG.

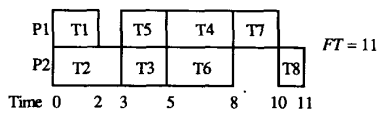


Fig. 2. A schedule for two processors displayes as Gantt chart.

The problem of optimal scheduling a task graph onto a multiprocessor system with p processors is to assign the computational tasks to the processors so that the precedence relations are maintained and all of the tasks are completed in the shortest possible time. The time that the last task is completed is called the *finishing time* (FT) of the schedule. Fig. 2 illustrates a schedule displayed as Gantt chart for the example task graph TG using two processors. This schedule has a finishing time of 11 units of time. An important lower bound for the finishing time of any schedule is the *critical path length*. The critical path length, t_{cp}, of a task graph is defined as the minimum time required to complete all of the tasks in the task graph.

We will adopt the following notations when discussing a task graph $TG = (V, E)$:

T_i is a *predecessor* of T_j and T_j is a successor of T_i if $e_{ij} \epsilon E$.

T_i is an *ancestor* of T_j and T_j is a *child* of T_i if there is a sequence of directed edges leading from T_i to T_j.

PRED(T_i)—the set of predecessors of T_i.

SUCC(T_i)—the set of successors of T_i.

$et(T_i)$—the execution time of T_i.

The height of a task in a task graph is defined as

$$height(T_i) = \begin{cases} 0, & \text{if PRED } (T_i) = \varnothing, \\ 1 + \max\limits_{T_j \in \text{PRED}(T_i)} height(T_j), & \text{otherwise.} \end{cases}$$

This height function indirectly conveys the precedence relations between the tasks. In fact, if task T_i is an ancestor of task T_j (i.e., if T_i must be executed before T_j), then *height* (T_i) < *height* (T_j). If there is no path between the two tasks, however, then there is no precedence relation between the two tasks, and the order of execution of the two tasks can be arbitrary.

III. Fundamentals of Genetic Algorithms

Genetic algorithm was developed by Holland [13] to study the adaptive process of natural systems and to develop artificial systems that mimic the adaptive mechanism of natural systems. Recently, genetic algorithms have been successfully applied to various optimization problems, such as the traveling salesman problem and gas pipeline optimization. Genetic algorithms differ from traditional optimization methods in the following ways [12].

1) Genetic algorithms use a coding of the parameter set rather than the parameters themselves.
2) Genetic algorithms search from a population of search nodes instead of from a single one.
3) Genetic algorithms use probabilistic transition rules.

A genetic algorithm consists of a string representation ("genes") 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 is randomly generated.
2) Evaluation of the 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 steps 2 and 3 until the algorithm converges.

From the above description, we can see that genetic algorithms use the notion of survival of the fittest by passing "good" genes to the next generation of strings and combining different stings 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 four components greatly affects the solution obtained as well as the performance of the genetic algorithm. In the following sections, we examine each of them for the problem of multiprocessor scheduling.

IV. String Representation and Initial Population

This section introduces the string representation used for the multiprocessor scheduling problem and also presents a method to generate an initial population of search nodes.

A. String Representation

An important factor in selecting the string representation for the search nodes is that all of the search nodes in a search space are represented and the representation is unique. It is also desirable, though not necessary, that the strings are in one-to-one correspondence with the search nodes. That is, each string corresponds to a legal search node (see Fig. 3). The design of the genetic operator is greatly simplified if the

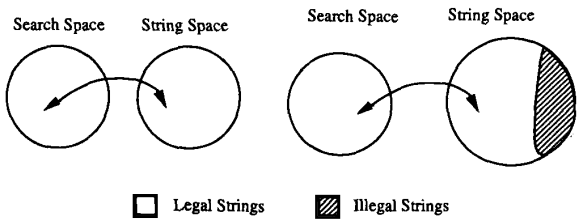


Fig. 3. Mapping between string representation space and search space.

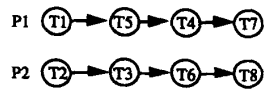


Fig. 4. List representation of schedule.

string representation space and the search space is in one-to-one correspondence. (See Section VI.) Davis [14] considered the problem of finding a representation for genetic algorithms in the problem of job shop scheduling. An intermediary encoded representation of the schedules and a decoder was used that would always yield legal solutions to the problem. The representation is somewhat complicated and is for a different problem.

For the multiprocessor scheduling problem, a legal search node (a schedule) is one that satisfies the following conditions.

1) The precedence relations among the tasks are satisfied.
2) Every task is present and appears only once in the schedule (completeness and uniqueness).

The string representation used in this paper 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. The precedence relations within the processor, however, must still be maintained.

The representation of a schedule for genetic algorithms must accommodate the precedence relations between the computational tasks. This is resolved by representing the schedule as several lists of computational tasks. Each list corresponds to the computational tasks executed on a processor, and the order of the tasks in the list indicates the order of execution. Fig. 4 illustrates the list representation of the schedule in Fig. 2. 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 relations do not come into play until we actually calculate the finishing time of the schedule. Each list can be further viewed as a specific permutation of the tasks in the list (allowing the last task to map to the first task). Fig. 5 illustrates the permutation representation of the schedule in Fig. 4. Thus, a schedule for n tasks and p processors is a permutation of n numbers with p cycles. The permutation representation of schedules is useful when we actually implement the genetic algorithm.

$$\begin{pmatrix}1&2&3&4&5&6&7&8\\5&3&6&7&4&8&1&2\end{pmatrix} = \begin{pmatrix}1&5&4&7\\5&4&7&1\end{pmatrix}\begin{pmatrix}2&3&6&8\\3&6&8&2\end{pmatrix}$$

i corresponds to Ti

Fig. 5. Permutation representation of schedule.

Note that not every permutation of n numbers with p cycles corresponds to a legal schedule because of the precedence relations. This representation of schedules falls into the category that the string space is not in one-to-one correspondence with the search space. We must bear this in mind when we design the genetic operators.

B. Initial Population

One of the merits of genetic algorithms is that it searches many nodes in the search space in parallel. This requires us to generate randomly an initial population of the search nodes. The population size is typically problem-dependent and has to be determined experimentally. To facilitate the generation of the schedule and the construction of the genetic operators (see Section VI), we imposed the following height-ordering condition on the schedules generated:

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

For example, in processor P1 of Fig. 4, we have *height*(T1) < *height*(T5) $\leq$ *height*(T4) < *height*(T7).

To guarantee that a schedule satisfying this condition is still a legal schedule, that is, that the precedence relations are not violated, we have the following lemma.

Lemma 1: A schedule satisfying the height-ordering condition is a legal schedule.

Proof: A schedule would be illegal if a task is scheduled to be executed before its ancestor. Suppose that T and T' are tasks assigned to the same processor, and that T' is an ancestor of T. By the definition of height, we have height(T') < height(T). If we order the tasks in ascending order of height, then T' will be executed before T, and the schedule will be legal. □

For example, consider the task graph in Fig. 1. Task T5 (with height 1) is an ancestor of T8 (with height 3). If they are both assigned to the same processor, then T5 will precede T8 according to the height ordering, and this would guarantee that T5 will be executed before T8 in that processor. If there are no precedence relations between two tasks, however, then the height ordering does not have to apply. For example, tasks T6 (with height 2) and T5 are not related, and they can be executed in any order in a processor.

Since the height-ordering condition is only a necessary condition, the optimal schedule may not satisfy it. To reduce the likelihood of this happening, we can modify the definition of height as follows.

Define the new height (*height'*) of a task, T_j, to be a random integer between (max *height*(T_i))+1 and (min *height*(T_k))−1, over all $T_i \in$ PRED(T_j) and $T_k \in$ SUCC(T_j). We can then use the following algorithm to generate the initial population of schedules in list representation:

Algorithm **Generate-Schedule**
[This algorithm randomly generates a schedule of the task graph TG for a multiprocessor system with p processors.]
GS1. [Initialize.] Compute *height'* for every task in TG.
GS2. [Separate the tasks according to their height.] Partition the tasks in TG into different sets, G(h) (G(h) is defined as the set of tasks with height h), according to the value of *height'*.
GS3. [Loop $p-1$ times.] For each of the first $p-1$ processors, do step GS4.
GS4. [Form the schedule for a processor.] For each set, $G(h)$, set $NG(h)$ to be the number of tasks in $G(h)$. Randomly generate a number, r, between 0 and $NG(h)$.
Pick r tasks from $G(h)$, remove them from $G(h)$, and assign them to the current processor.
GS5. [Last processor.] Assign the remaining tasks in the sets to the last processor.

By repeatedly applying the algorithm **Generate-Schedule**, we can generate the initial population of search nodes needed.

V. Fitness Function

The fitness function in genetic algorithms is typically the objective function that we want to optimize in the problem. It is used to evaluate the search nodes and also controls the genetic operators. 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 genetic algorithm is based on the finishing time of the schedule. The finishing time of a schedule, S, is defined as follows:

$$FT(S) = \max_{P_j} ftp(P_j),$$

where $ftp(P_j)$ is the finishing time for the last task in processor P_j.

Since one of the genetic operators (reproduction) will try to maximize the fitness function, we need to convert the finishing time into maximization form. This can be done by defining the fitness value of a schedule, S, as follows:

$$C\max - FT(S),$$

where $C_{\max}$ is the maximum finishing time observed so far. Thus, the optimal schedule would be the smallest finishing time and a fitness value larger than the other schedules.

VI. Genetic Operators

One of the functions of the genetic operators 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, it may result in an even better one. Relating this idea to multiprocessor scheduling, certain portions of a schedule may belong to the optimal schedule. By combining several of these "optimal" parts, we can find the optimal schedule efficiently.

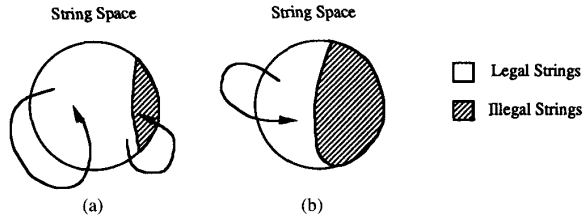


Fig. 6. Genetic operator that genetates: (a) Both legal and illegal strings. (b) Only legal strings.

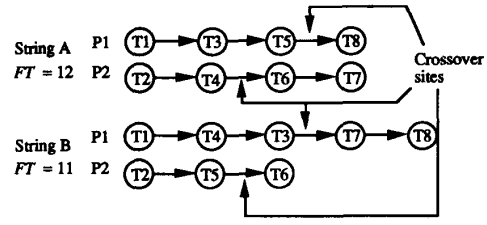


Fig. 7. Two strings of crossover operation.

If the string representation space and the search space is not in one-to-one correspondence, then we must design the genetic operator carefully. If the number of strings that corresponds to illegal search nodes is relatively small, then it may be acceptable to allow the genetic operator to generate illegal strings (see Fig. 6(a)) and discard them later with a legality test. If the number of strings that corresponds to illegal search nodes is comparable to those representing legal search nodes, however, then too much computational time will be wasted in generating the illegal strings and checking them. In this case, a "good" genetic operator would be one that always generates a string representing a legal search node (see Fig. 6(b)). This may not be achievable, however, because of factors such as the difficulty of implementation and the high computational cost of the operation.

For the multiprocessor scheduling problem, the genetic operators used must enforce the intraprocessor precedence relations, as well as the completeness and uniqueness of the tasks in the schedule as discussed in Section IV. This would ensure that the new strings generated will always represent legal search nodes. We develop a genetic operator for the multiprocessor scheduling problem based on the notion of crossover [12].

A. Crossover

Consider the two strings (schedules) shown in Fig. 7. We can create new strings by exchanging portions of the two strings by using the following method.

1) Select sites (crossover sites) where we can cut the lists into two halves. (See Fig. 7).
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.

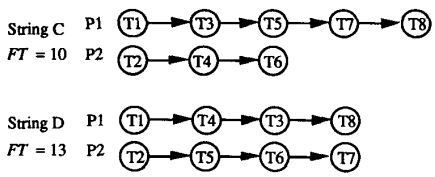


Fig. 8. The two new strings generated.

The new strings created are shown in Fig. 8. Notice that one of the new string, C, has a smaller finishing time than the previous two strings. In fact, this is the optimal finishing time for the task graph TG using two processors. The operation described above can be easily extended to p processors and appears to be quite effective. We still have to define the method for selecting the crossover sites, however, and show that the new strings generated are legal.

Undoubtedly, the legality of the new strings generated are strongly related to the selection of the crossover sites. Notice that the crossover sites used in the above example always lie between tasks with two different heights (*height*(T5) $\neq$ *height*(T8), *height*(T4) $\neq$ *height*(T6), etc.). In fact, we can prove the following theorem.

Theorem 1: If the crossover sites are chosen so that the following conditions exist.

1) The height of the tasks next to the crossover sites are different.
2) The height of all the tasks immediately in front of the crossover sites are the same, thus, the new strings generated will always be legal.

Proof: We need to show that the precedence relation is not violated and that the completeness and uniqueness of the tasks still holds. Consider the situation depicted in Fig. 9(a). We have the following conditions:

$$height(T_i) < height(T_j), height(T_{i'}) < height(T_{j'}),$$
$$height(T_i) = height(T_{i'}).$$

Since all of the tasks with height greater than *height*(T_i) are exchanged between the two strings, no task is deleted or duplicated. This means that completeness and uniqueness is preserved.

After the crossover operation (see Fig. 9(b)), the following relations are valid:

$$height(T_i) < height(T_{j'}), height(T_{i'}) < height(T_j).$$

Therefore, the new strings generated still satisfy the height-ordering condition. It follows from Lemma 1 that the new strings generated are legal schedules. □

The crossover operation uses the above fact and selects the crossover sites so that conditions 1) and 2) are always satisfied. It is summarized in the following algorithm:

Algorithm **Crossover**.
[This algorithm performs the crossover operation on two strings (A and B) and generates two new strings.]
C1. [Select crossover sites.] Randomly generate a number, c, between 0 and the maximum height of the task graph.

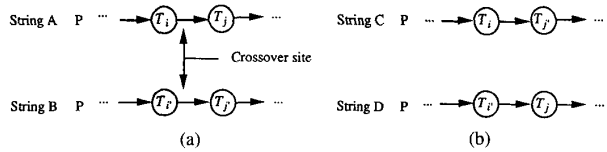


Fig. 9. (a) Strings A and B for crossover. (b) New strings C and D generated.

C2. [Loop for every processor.] For each processor P_i in string A and string B, do step C3.
C3. [Find the crossover sites.] Find the last task T_{ji} in processor P_i that has height c, and T_{ki} is the task following T_{ji}. That is, c =*height'* (T_{ji}) < *height'* (T_{ki}) and *height'* (T_{ji}) are the same for all i.
C4. [Loop for every processor.] For each processor P_i in string A and string B, do step C5.
C5. [Crossover.] Using the crossover sites selected in step C3, exchange the bottom halves of strings A and B for each processor P_i.

Although the crossover operation is powerful, it is random in nature and may eliminate the optimal solution. Typically, its application is controlled by a crossover probability whose value is determined experimentally. Furthermore, we can always preserve the best solution found by including it in the next generation.

B. Reproduction

A commonly used genetic operator is reproduction. The reproduction process forms a new population of strings by selecting strings in the old population based on their fitness values. The selection criterion is that strings with higher fitness value should have a higher chance of surviving to the next generation. The rationale here is that "good" strings have high fitness value and therefore should be preserved into the next generation. Typically, a biased roulette wheel is used to implement reproduction where each string in the population occupies a slot size proportional to its fitness value. Random numbers are generated and used as an index into the roulette wheel to determine which string will be passed to the next generation. Because strings with higher fitness value will have larger slots, they are more likely to be selected and passed to the next generation.

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

Algorithm **Reproduction**.
[This algorithm performs the reproduction operation on a population of strings POP and generates a new population of strings *NEWPOP*.]
R1. [Initialize.] Let *NPOP* ← number of strings in *POP*.
R2. [Construct the roulette wheel.] *NSUM*, sum of all of the fitness value of the strings in *POP*; form *NSUM* slots and assign string to the slots according to the fitness value of the string.

R3. [Loop *NPOP* -1 times.] Do step R4 *NPOP* -1 times.
R4. [Pick a string.] Generate a random number between 1 and *NSUM*, and use it to index into the slots to find the corresponding string; add this string to *NEWPOP*.
R5. [Add the best string.] Add the string with the highest fitness value in *POP* to *NEWPOP*.

C. Mutation

Mutation can be considered as an occasional (with small probability) random alternation of the value of a string. One can think of mutation as an escape mechanism for premature convergence. For the multiprocessor scheduling problem, mutation is applied by randomly exchanging two tasks with the same height. The mutation operation is summarized in the following algorithm:

Algorithm **Mutation**.
[This algorithm performs the mutation operation on a string and generates a new string.]
M1. [Pick a task]. Randomly pick a task, T_i.
M2. [Match height.] Search the string for a task, T_j, with the same height.
M3. [Exchange tasks.] Form a new string by exchanging the two tasks, T_i and T_j, in the schedule.

Typically, the frequency of applying the mutation operator is controlled by a mutation probability whose value is determined experimentally.

VII. Complete Algorithm

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

Algorithm **Find-Schedule**.
[This algorithm attempts to solve the multiprocessor scheduling problem.]
FS1. [Initialize.] Call **Generate-Schedule** N times, and store the strings created in *POP*.
FS2. [Repeat until convergent.] Do steps FS3–FS8 until the algorithm is convergent.
FS3. [Compute fitness values.] Compute the fitness value of each string in *POP*.
FS4. [Perform Reproduction.] Call **Reproduction**. *BEST-STRING* $\leftarrow$ string in *POP* with the highest fitness value.
FS5. [Perform Crossover.] Do step FS6 *NPOP/2* times.
FS6. [Crossover.] Pick two strings from *NEWPOP*, and call **Crossover** with a probability *PROB-CROSSOVER*. If crossover is performed, put the new strings in *TMP* ; otherwise, put the two strings picked in *TMP*.
FS7. [Mutation.] For each of the string in *TMP*, call **Mutation** with a probability *PROB-MUTATION*. If mutation is performed, put the new string in *POP*; otherwise, put the string picked in *POP*.
FS8. [Preserve the best string]. Replace the string in *POP* with the smallest fitness value by *BEST-STRING*.

TABLE I
Comparison of the Optimal Schedule, the Genetic Alogrithm, and the List Algorithm for Various Random Task Graphs Using Two Processors

No. of Task Nodes	Optimal Schedule (OPT)	Genetic Algorithm (GA)	List Algorithm	$\frac{GA-OPT}{OPT}$ %
30	392	395	416	0.8
35	410	436	457	6.3
41	490	508	522	3.7
51	653	662	674	1.4
61	768	783	822	2.0

TABLE II
Comparison of the Optimal Schedule, the Genetic Algorithm, and the List Algorithm for Various Random Task Graphs Using Three Processors

No. of Task Nodes	Optimal Schedule (OPT)	Genetic Algorithm (GA)	List Algorithm	$\frac{GA-OPT}{OPT}$ %
31	260	266	280	2.3
36	295	305	366	3.3
42	352	378	393	6.9
53	434	451	454	3.8
68	561	584	608	3.9
81	667	707	789	5.7

The algorithm terminates when it meets the convergent criterion. Typically, this criterion can be that the best solution in the population obtained does not change after a specific number of generations.

VIII. Simulation Results

The genetic algorithm discussed in the previous section was implemented and tested on random task graphs with known optimal schedules. The random schedules generated have task numbers ranging from 20 to 90. The number of successors that each task node is allowed is a random number between 1 and 4, and the execution time for each task is a random number between 1 and 50. The task graphs are also tested on a list scheduling algorithm [15]. The random task graphs are non–trivially constructed, but in such a way that the optimal schedule is known [16]. The genetic algorithm used the following parameters throughout the simulations:

- population size $= 10$
- crossover probability $= 1.0$
- mutation probability $= 0.05$
- maximum number of iterations $= 1500$.

Tables I through IV compare the finishing time of the genetic algorithm and the list scheduling algorithm, along with the optimal schedule for the random task graphs, by using different multiprocessor configurations. The simulations were performed on a SUN 4/490, and typically run-time is 1 s to 2 s. The genetic algorithm converges to a solution in less than 1000 generations in all cases. From Tables I through V, the solution obtained by the genetic algorithm is consistently better than the list scheduling algorithm and is within 10% of the optimal schedule.

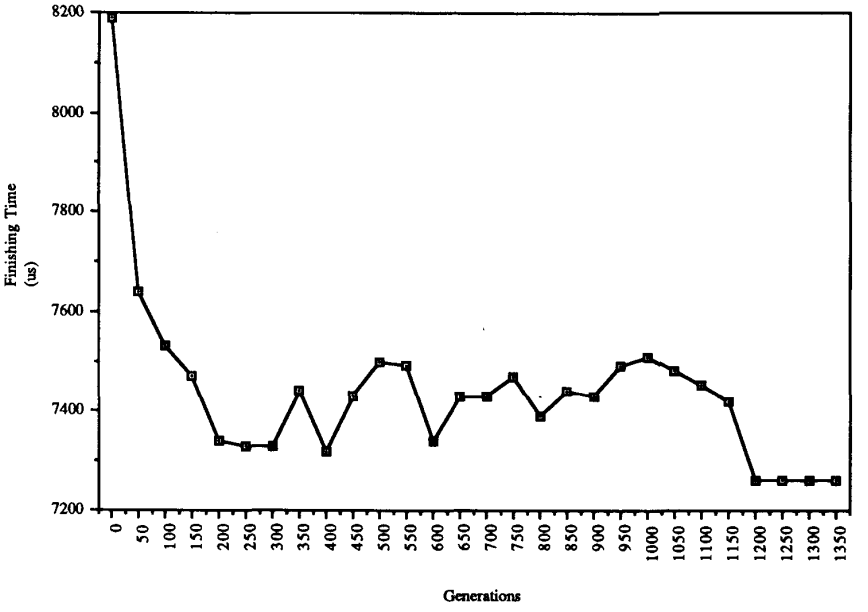


Fig. 10. The finishing times obtained by the genetic algorithm at different generations for the elbow manipulator task graph using four processors.

TABLE III
COMPARISON OF OPTIMAL SCHEDULE, THE GENETIC ALOGORITHM, AND THE LIST ALGORITHM FOR VARIOUS RANDOM TASK GRAPHS USING FOUR PROCESSORS

No. of Task Nodes	Optimal Schedule (OPT)	Genetic Algorithm (GA)	List Algorithm	$\frac{GA-OPT}{OPT}\%$
28	190	198	237	4.0
41	267	285	291	6.3
57	372	385	400	3.4
64	394	434	484	9.2
75	458	467	511	1.9
87	542	561	574	3.4

TABLE IV
COMPARISON OF THE OPTIMAL SCHEDULE, THE GENETIC ALGORITHM, AND THE LIST ALGORITHM FOR VARIOUS RANDOM TASK GRAPHS USING FIVE PROCESSORS

No. of Task Nodes	Optimal Schedule (OPT)	Genetic Algorithm (GA)	List Algorithm	$\frac{GA-OPT}{OPT}\%$
29	147	153	186	3.9
42	220	232	268	5.2
56	280	305	329	8.2
67	346	357	363	3.1
77	383	407	421	5.9
87	438	455	475	3.7

The genetic algorithm was also tested on the Newton–Euler inverse dynamics equations task graphs for the Stanford manipulator and elbow manipulator [6]. The Stanford manipulator task graph consists of 88 tasks, and task execution time ranges from 1 to 111 μs. The elbow manipulator task graph has 103 tasks, and task processing time ranges from 10 to 570 μs. The genetic algorithm used the following parameters throughout the simulations:

- population size = 20
- crossover probability = 0.5
- mutation probability = 0.005
- maximum number of iterations = 2000.

TABLE V
COMPARISON OF THE OPTIMAL SCHEDULE AND THE GENETICAL ALGORITHM FOR THE STANFORD MANIPULATOR TASK GRAPH

No. of Processors	Optimal Schedule (OPT)	Genetic Algorithm (GA)	$\frac{GA-OPT}{OPT}\%$
2	1242	1249	0.6
3	879	938	6.7
4	659	774	17.5
5	586	679	15.9
6	573	627	9.4
7	570	609	6.8
8	570	570	0
9	570	570	0

Table V summarizes the solution obtained from the genetic algorithm and the optimal solution for the Stanford manipulator task graph with various numbers of processors. The genetic algorithm typically converges to a solution in 1500 generations and requires less than 5 s of CPU time on a VAX 7580. Table VI summarizes the solution found by the genetic algorithm for the elbow manipulator task graph. Fig. 10 shows the finishing time of the best schedule (for four processors) found by the genetic algorithm at different generations, with and without preserving the best schedule in each generation.

IX. CONCLUSION

In this paper, we considered the multiprocessor scheduling problem. A stochastic search method based on the genetic al-

TABLE VI
COMPARISON OF THE OPTIMAL SCHEDULE AND THE GENETIC ALGORITHM FOR THE ELBOW MANIPULATOR TASK GRAPH

No. of Processors	Optimal Schedule (OPT)	Genetic Algorithm (GA)	$\frac{GA-OPT}{OPT}\%$
2	11710	12340	5.4
3	7819	8940	14.3
4	6630	7260	9.5
5	6630	6980	5.3
6	6630	6630	0
7	6630	6630	0

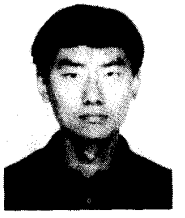
gorithm approach is proposed. The representation of the search nodes (schedules) used are in the form of lists of computational tasks. This eliminates the need to consider the precedence relations between tasks in different processors and allows us to construct an efficient crossover operator. The crossover operator developed takes into account the precedence relations of the tasks and guarantees that the new strings generated are legal. The proposed genetic algorithm was tested with random task graphs and the Newton-Euler inverse dynamic equations task graphs for the Stanford manipulator and elbow manipulator.

ACKNOWLEDGMENT

We would like to thank the anonymous referees for their helpful comments and suggestions that have improved the quality of this manuscript.

REFERENCES

[1] M.R. Garey and D.S. Johnson, *Computers and Intractability*. New York: W.H. Freeman, 1979.
[2] C.V. Ramamoorthy *et al.*, "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. Comput.*, vol. C-21, pp. 137–146, Feb. 1972.
[3] T.L. Adams *et al.*, "A comparison of list schedules for parallel processing systems," *Comm. Assoc. Computing Machinery*, vol. 17, pp. 685–690, Dec. 1974.
[4] M.J. Gonzalez, "Deterministic processor scheduling," *Computing Surveys*, vol. 9, no. 3, pp. 173–204, Sept. 1977.
[5] H. Kasahara and S. Narita, "Practical multiprocessing scheduling algorithms for efficient parallel processing," *IEEE Trans. Comput.*, vol. C-33, no. 11, pp. 1023–1029, Nov. 1984.
[6] H. Kasahara and S. Narita, "Parallel processing of robot-arm control computation on a multimicroprocessor system," *IEEE J. Robotics Automation*, vol. RA-1, no. 2, pp. 104–113, June 1985.
[7] C.L. Chen, C.S.G. Lee and E.S.H. Hou, "Efficient scheduling algorithms for robot inverse dynamics computation on a multiprocessor system," *IEEE Trans. Syst., Man, Cybernetics*, vol. 18, pp. 729–743, Dec. 1988.
[8] B. Hellstrom and L. Kanal, "Asymmetric mean-field neural networks for multiprocessor scheduling," *Neural Networks*, vol. 5, pp. 671–686, 1992.
[9] *Proc. 1st Int. Conf. Genetic Algorithms and Their Applications*, July 24–26, 1985, Carnegie-Mellon University, Pittsburgh, PA.
[10] *Proc. 2nd Int. Conf. Genetic Algorithms and Their Applications*, July 28–31, 1987, MIT, Cambridge, MA.
[11] *Proc. 3rd Int. Conf. Genetic Algorithms*, June 4–7, 1989, George Mason Univ., Washington, DC.
[12] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.
[13] J. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975.
[14] L. Davis, "Job shop scheduling with genetic algorithms," *Proc. 1st Int. Conf. Genetic Algorithms and Their Applications*, July 24–26, 1985, Carnegie-Mellon University, Pittsburgh, PA, pp. 136–140.
[15] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw Hill, 1984.
[16] N. Desni, "Generating random task graphs with known optimal schedule for multiprocessing scheduling," Master's Project Rep., NJIT, Newark, NJ, 1993.



E.S.H. Hou (S'83–M'89) received two B.S. degrees (*magna cum laude*), in electrical engineering and computer engineering, from the University of Michigan, Ann Arbor, in 1982; he received the M.S. degree in computer science from Stanford University, Stanford, CA, in 1984; and he received the Ph.D. degree in electrical engineering from Purdue University, W. Lafayette, IN, in 1989.

He is an Assistant Professor in the Department of Electrical and Computer Engineering, as well as Assistant Director in the Electronic Imaging Center, at the New Jersey Institute of Technology, Newark, NJ. His research interests include infrared imaging, genetic algorithms, scheduling, and neural networks.

Dr. Hou was the Local Arrangement Chairman for the 1993 IEEE Regional Conference on Control Systems.

N. Ansari (S'81–M'88) received the B.S.E.E. from the New Jersey Institute of Technology in 1982, the M.S.E.E. from the University of Michigan in 1983, and the Ph.D. degree from Purdue University in 1988.

Since August 1988, he has been with the Department of Electrical and Computer Engineering at the New Jersey Institute of Technology, Newark, NJ, currently as an Associate Professor. His research interests include neural computing, pattern recognition, nonlinear signal processing, computer vision, and intelligent networks. He has been serving as a frequent referee, as a session chair/organizer, and as a technical representative for various major journals, conferences, and federal agencies.

Dr. Ansari has published his research findings regularly. He has co-edited a book, *Neural Network Applications for Telecommunications*, published by Kluwer in 1993.

H. Ren received the B.S.E.E. degree in Electronic Instruments and Measurements from Tong Ji University, Shanghai, China, in 1984, and the M.S.E.E. from the New Jersey Institute of Technology (NJIT), Newark, NJ, in 1991.

Since October 1991, she has been a Software Engineer at Penril Datability Networks, Carlstadt, NJ. From 1990 to 1991, she was a Research Assistant in the Department of Electrical and Computer Engineering at NJIT.