CHAPTER 3

AN AIDED GENETIC APPROACH FOR MULTIPROCESSOR SCHEDULING

3.1 INTRODUCTION

Genetic algorithms are based on the mechanism of natural selection and natural genetics. Genetic algorithms (Goldberg 1989; Srinivasa and Patnaik 1994; Tang et al 1996; Gen and Cheng 1997) are powerful and broadly applicable to stochastic search and optimization techniques based on principles from evolution theory. They differ from conventional search techniques, starting with an initial set of random solutions called initial population. Genetic algorithms perform a multiple search by maintaining a population of potential solutions. Each individual in the population is called a chromosome, representing a solution to the problem. A chromosome is a string of symbols. The chromosomes evolve through successive iterations called generations. During each generation, the chromosomes are evaluated, using some measures of fitness. To create the next generation, new chromosomes, called offsprings, are formed by either merging two chromosomes from current generation using crossover operator, or modifying a chromosome using a mutation operator.

At the beginning of a genetic search, there is widely random and diverse population and crossover operator tends to perform a widespread search for exploring all solution space. As the high fitness solutions develop, the crossover operator provides exploration in the neighborhood of each of them.

A new generation is formed by
i. selecting, according to the fitness values, some of the parents and offspring
ii. rejecting others so as to keep the population size constant.
Fitter chromosomes have higher probabilities of being selected. After several generations, the algorithm converges to the best chromosome, which hopefully represents the optimal or sub-optimal solution to the problem. Genetic algorithms combine elements of directed and stochastic search which can make a balance between exploration and exploitation of the search space.

Genetic algorithms differ from conventional optimization and search procedures as they

i. work with coding of solution set, not the solutions themselves.
ii. search from a population of solutions, not a single solution.
iii. use fitness function.
iv. use probabilistic transition rules, not deterministic rules.

The genetic approach for parallel machine scheduling problem is summarized in Gen and Cheng (1997), but with certain limitations. The genetic algorithm (GA) for multiprocessor scheduling proposed by Hou et al (1994) schedules the nodes of a precedence-constrained task graph representing a parallel program onto the nodes of the processor graph such that the completion time is minimized. But their approach assumes negligible intertask communication and the multiprocessor is assumed to be homogeneous (all processors have the same service rate, memory capacity, link capacities, etc.). Most importantly, this genetic algorithm does not consider the problem of contention (which arises in the context of the usage of communication links of a multiprocessor system) and does not handle the incompletely connected multiprocessor system.

In the current work, a genetic algorithm is proposed to schedule the tasks of the deterministic directed acyclic task graph with non-negligible intertask communication onto the message-passing multicomputer system with heterogeneous processors and heterogeneous link capacities in which processors assume any type of interconnection topology.
Though genetic algorithms have proved to be a versatile and effective approach for solving optimization problems, there are many situations in which the simple genetic algorithm does not perform particularly well. Various methods of hybridization (Gen and Cheng, 1997), and parallel genetic algorithms (Srinivasa and Patnaik, 1994) have been proposed to improve the performance of the pure genetic algorithm.

Parallel genetic algorithms reduce the run time of the genetic algorithm and get near optimal solutions as it explores the search space in parallel by having different initial population placed in each processor. It does selection, crossover and mutation operations in parallel as there are several processors running the algorithm.

The genetic algorithm for multiprocessor scheduling faces the local extrema problem due to the limitation in the input population size and its quality which would make the resulting schedules sub-optimal. To minimize this problem, an Aided Genetic Algorithm (AGA) is proposed in this thesis, in which a member of the initial population of the Genetic Algorithm is obtained from a heuristic pre-scheduler. The AGA combines the concepts of list scheduling, heuristic and genetic to get reduced schedule length. It also finds near-optimal schedule that is not priority based.

It is found that the AGA achieves the required convergence in (a) lesser number of iterations and (b) lesser number of trials in obtaining the near-optimal solution compared to the conventional genetic algorithm.

3.2 GENETIC ALGORITHM WITH INTERTASK COMMUNICATION

The GA for multiprocessor scheduling consists of schedules represented as strings (genes) in the search space, a set of genetic operators for generating the new schedules, a fitness function to evaluate the generated schedules, and a stochastic assignment to control the genetic operators. This algorithm passes "good" genes (strings) to the next generation by making use of the fact "survival of the fittest", and explores new search points by combining different strings.
The steps involved in the GA are:

1) find the height of each task and generate schedules (strings) to form the initial population (to choose the best from) through population generation.

2) perform reproduction and evaluate the fitness value of each schedule.

3) allow the members of the population to crossover and produce offspring.

4) randomly introduce changes in the offspring through mutation.

5) repeat steps 2 to 4 until the algorithm converges.

3.2.1 String Representation and Initial Population

The task graph and the processor graph models are similar to the models presented in Chapter 2 under the DTH technique except for the data token in the task graph. Typical task graph and processor Graph are shown in Figure 3.1a and 3.1b respectively. A schedule that satisfies the precedence relations among the tasks is a legal string (search node) in the multiprocessor scheduling problem. Legal strings are unique (a task should be present only once), complete (all the tasks should be present), and executable.
The GA for multiprocessor scheduling follows integer list representation of schedules. A schedule (string) is built by several lists, where each list contains the tasks allocated to a processor, and in which, each task appears only once. The list representation not only facilitates easy evaluation of the fitness of each string but also takes care of problem constraints such as task precedence relationships.

The precedence relation between the tasks is indirectly conveyed using height function. The height of the first task in the task graph is zero and the heights of other tasks are calculated as follows:

Height $HT_i = 1 + \max(\text{height of predecessor task})$.

The executable strings are obtained when the tasks in the sub-strings are in the non-decreasing order of their heights, and their ordering indicates the order of execution. The population is made as random as possible avoiding illegal strings.

Figure 3.2 shows two legal schedules (String A and String B) for a multiprocessor system having 3 processors. In this figure each string consists of 3 lists. Each list corresponds to the tasks allocated to a processor.
The genetic algorithm searches many nodes in the search space in parallel. This however requires random generation of an initial population of the search nodes. The population size mainly depends on problem size and the objective function. The size of the input population is experimentally determined by varying the population size from 3 till 25. The number of iteration needed to get the minimum makespan is obtained in each case. It is found that the population size of 10 suffices in most of the multiprocessor scheduling problems.

The following steps in the population generation procedure randomly generate the initial population of schedules in list representation for a multiprocessor system with M processors.

Population generation

Begin
{
  Repeat for all tasks from 0 through L
    {Compute height of tasks
    } end for
  Let MAXHT= Maximum height of the task graph.
  Repeat for all heights from 0 through MAXHT
    {Let NHT(h) = Number of tasks in height(h)
    } end for
  Let NSTR= Number of members in the initial population.
  Repeat for all strings from 1 through NSTR
    { Repeat for all heights from 0 through MAXHT
      { Repeat for all tasks in the current height (NHT(h))
        { Randomly generate a number r between 1 and M
          Assign the current task to processor r
        } end for
      } end for
    } end for
}
end
3.2.2 Finishing Time of a Schedule

The *finishing time algorithm* proposed in this thesis evaluates the completion time of a task graph with non-negligible intertask communication scheduled onto a message-passing multicomputer system with heterogeneous processors and heterogeneous link capacities.

Let \( T_1, T_2, \ldots, T_q, \ldots, T_M \) denote a set of tasks allocated to processors \( 1, 2, \ldots, q, \ldots, M \). Let the completion time of task \( i \) on processor \( q \), be \( C_i(T_1, T_2, \ldots, T_q \cup i, \ldots, T_M) \). Task \( 'i' \) cannot be taken up for execution on processor \( 'q' \) until the data from the predecessors of task \( 'i' \) are available at processor \( 'q' \), and also until processor \( 'q' \) completes its execution of the last task \( 't' \), in the ordered set \( T_q \). Therefore the completion time of task \( 'i' \) is given by

\[
C_i(T_1, T_2, \ldots, T_q \cup i, \ldots, T_M) = \max \{D_{iq}, C_i(T_1, T_2, T_q, \ldots, T_M)\} + (s_i / \mu_q)
\]

where \( D_{iq} \) denotes the earliest time at which data from all the parents of task \( i \) become available at processor \( q \), and \( t \) is the last task in the ordered set \( T_q \). To compute \( D_{iq} \) it is assumed that the data communication from parent tasks takes place in the order in which they are completed and \( s_i / \mu_q \) is the service time of task \( i \) on processor \( q \). When processors are not directly connected, first the shortest path from the source processor to the destination processor is determined for transferring the predecessor task data. Through this shortest path the predecessor task data arrival time is evaluated. The finishing time of each schedule is calculated using the *finishing time algorithm*. The completion time of the last task \( L \) in the task graph is taken as the finishing time of the schedule.

In GA the finishing time of the schedule can be obtained in two ways. In the first method, the completion time of a task depends on the completion of its predecessor tasks irrespective of its height. In the second method the completion time of each task is evaluated based on its height and its position in the processor. The second method allows the predecessor data routing based on height ordering to minimize link contention. The *finishing time algorithm* given below is based on second method.
Finishing time algorithm

Begin
♦ {
For S = 1 to NSTR
{Do
{Repeat for all heights HT
{Repeat for all tasks in the current height (NHT(h))
Take task i
If (Completed) then go to the next task in that processor
with same height
else
{Let q be the processor on which task i is placed
Let \{\beta_i\} represents the set of predecessor of task i
Let C_{i|j} be the set of completion time of j^{th} predecessor of i^{th} task
where 1 \leq j \leq |\beta_i|
Arrange C_{i|j} in non decreasing order
For j = 1 to |\beta_i|
Do
Find the processor p on which j^{th} predecessor is allocated
If p \neq q then
Let k_0 = p
Find the shortest path between p and q
Represent the processors in the shortest path
as (k_0, k_1, ..., k_r, q)
Let A_{mn} be the available time of the link connecting
processors m and n
Let A_{k_0k_1} = max [A_{k_0k_1}, C_{i|j}] + (v_{ij} / c_{k_0k_1})
where v_{ij} is the volume of data transfer
between tasks i and j
c_{k_0k_1} is the capacity of the link connecting
processors k_0 and k_1
Let k_{r+1} = q
For x = 1 to r
Do
A_{k_xk_{x+1}} := max [A_{k_xk_{x+1}}, A_{k_xk_{x+1}}] + (v_{ij}k_x / c_{k_xk_{x+1}})
end do
The data arrival time of j^{th} predecessor is
\[ d_{i|j} = A_{k_xk_{x+1}} \]
else
\[ d_{i|j} = C_{i|j} \] (no predecessor task data arrival time)
3.2.3 Fitness Function

The fitness function in the genetic algorithm is the objective function that has to be optimized. It is used to evaluate the search nodes and also control the genetic operators. For the multiprocessor scheduling problem the fitness function is based on the finishing time of the schedule. The finishing time $FT(S)$ of a schedule $S$ is $C_L$, where $C_L$ is the completion time of the last task in the task graph. The genetic operator, reproduction, tries to maximize the fitness function. So, the finishing time is converted into maximization form by defining the fitness value of a schedule $S$ as follows (Hou et al 1994).

$$FV(S) = C_{max} - FT(S)$$

where $C_{max}$ is the maximum finishing time observed so far. The higher the fitness, the better is the string and hence the lesser the finishing time of the schedule. The strings which have higher fitness values have a greater chance of survival. Thus, the optimal schedule has the smallest finishing time and a fitness value larger than the other schedules.
3.2.4 Genetic Operators

Reproduction, crossover and mutation are the genetic operators used in the GA. Reproduction evaluates the fitness of each schedule. During reproduction, a biased roulette wheel is constructed in which 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 choose the strings for the next generations. All those strings that withstand this test are passed to the next generation. On these strings, the GA applies the crossover and mutation operators.

Reproduction

The reproduction operation is performed on a population of strings STR, in turn generating a new population of strings NEWSTR. The following code outlines the reproduction operation:

```
begin
{
    Repeat for all strings from 1 through NSTR
    {Find Finishing time FT[S], and Fitness value FV[S] of all the strings using the finishing time algorithm
    }end for
    Let FITSUM=0
    For S=1 to NSTR
    {FITSUM=FITSUM+FV[S];
    }end for
    Repeat for all strings from 1 through NSTR-1
    {Generate a random number r between 0 and FITSUM;
    Identify the slot and hence the string by indexing r into the slot;
    Add this string to NEWSTR;
    }end for
}
end
```

Crossover

Crossover is the main genetic operator and it produces the two offspring solutions from two parents. A simple way to achieve crossover would be to choose a
random cut-point and generate the offspring by combining the segment of one parent to the left of the cut point with the segment of the other parent to the right of the cut-point.

In the current work, a random crossover point is generated between 0 and the maximum height of the task graph. This height (for eg. height 2) divides schedule A and schedule B in Figure 3.3 into two parts each. New schedules are formed by swapping those parts of the parent strings demarcated by the crossover point. Figure 3.3 shows the new strings created after the crossover operation. For eg., tasks 0, 2, 4, and 7 are allocated to processor P1 before crossover, and tasks 0, 2, 4, and 8 are allocated to processor P1 after crossover operation in string A.

Figure 3.3 Crossover

Following steps perform crossover operation.

begin

\{Repeat for all strings 1 through NSTR
\{randomly select two strings A and B;
randomly generate a number h between 0 and MAXHT of the task graph;
\{Repeat for all processors 1 through M
\{Repeat for all tasks 1 through L
\{Choose crossover sites-
Find the task i in processor p with height h:
Let \( k \) be the task following \( i \) and \( h < HT[k] \);

Repeat for all processors 1 through \( M \)

\{ Exchange the bottom halves of the strings beyond crossover sites; \}

end

Through crossover alone the number of tasks in the sub-string is changed. Although the crossover operation is powerful and random in nature, it may eliminate the optimal solution. Application of crossover operation is controlled by a crossover probability, which is evaluated experimentally and is kept as 0.9 for this problem.

The crossover rate is defined as the ratio of the number of offspring produced in each generation to the population size (NSTR). A higher crossover rate allows exploration of more solution space and reduces the chances of settling for a false optimum. But if this rate is too high, it results in the wastage of a lot of computation time in exploring unpromising regions of the solution space.

**Mutation**

Mutation is a background operator which produces spontaneous random changes in various chromosomes. A simple way to achieve mutation would be to alter one or more genes. In the current work the mutation operator is applied on a string by randomly exchanging two tasks with the same height. The following code performs the mutation operation on a string and generates a new string.

begin

\{ Repeat for all strings 1 through NSTR 
  \{ Randomly pick a task \( i \); 
    Search the string for task \( j \) with the same height; 
    Exchange tasks; 
    Form a new string; 
  \} end for

end
The frequency of applying the mutation operator is controlled by a mutation probability whose value is determined as 0.1 experimentally. The probability of mutation is kept low, as mutation is more often than not a destructive process.

The mutation rate is defined as the percentage of the total number of genes in the population. The mutation rate controls the rate at which new genes are introduced into the population for trial. If it is too low, many genes that would have been useful are never tried out, but if it is too high there will be much random perturbation, the offspring will start losing the resemblance to the parent, and the algorithm will lose the ability to learn from the history of the search.

3.2.5 Genetic Algorithm for Multiprocessor Scheduling

All the individual algorithms are now combined to form the GA for multiprocessor scheduling

GA for multiprocessor scheduling

1. Call Population generate and store the strings in STR
2. Do steps 3 to 6 for fixed number of iterations.
3. Compute the Fitness value of each string in the population through the finishing time algorithm.
4. Call Reproduction and create NEWSTR.
   Copy the best-string of STR to NEWSTR.
5. Call Crossover.
6. Call Mutation

When the GA starts functioning, the first operator to be used on the initial population is reproduction to choose the strings for next generation. The crossover and mutation operators are applied on these strings to produce new strings.
The probabilities of crossover and mutation can also be made adaptive by changing their values for every iteration bearing in mind that the probability of mutation must be low and that of crossover must be high to get good results.

The best string is preserved by including it in the next generation. Fitness values of the resultant strings are evaluated, and the process is repeated for a fixed number of iterations (say 1000).

In multiprocessor scheduling, with non-negligible intertask communication, the optimal solution is not explicitly known beforehand, and hence it is not possible to define a rigorously derived error criterion. Also, the relationship between the required number of generations before termination and the problem size (processor and task sizes) is not particularly evident (Chockalingam and Arun Kumar, 1992).

Hence, the criterion used in this thesis is based on the observation of the schedule length for a large number of consecutive iterations, and, if it remains constant, it will be accepted as the minimum schedule length. It may be noted that only such heuristically derived convergence criteria are possible for the above problem.

Table 3.1

<table>
<thead>
<tr>
<th>Task</th>
<th>Ht</th>
<th>2 Processor System</th>
<th>3 Processor System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Processor Completion time (UT)</td>
<td>Processor Completion time (UT)*</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

* UT – Unit Times
Table 3.1 presents the genetic scheduling results, which include intertask communication overhead for the task graph shown in Figure 3.1a. In this table, the completion times obtained using 2 processor system and 3 processor system are presented. In this table ‘Ht’ represents the height of the task. The critical path for this task graph is given by 1→3→6→8. The critical length is 8 time units. The completion time of the task graph is 10 time units in both 2 and 3 processor systems. The communication overhead is 25%.

3.3 THE AIDED GENETIC APPROACH

Inadequacies in GA

The essential issue in the multiprocessor scheduling problem using GA is the permutation and combination for tasks and processors. The optimum solution is feasible in GA by global exploration of the search space. Thus, a schedule for L tasks and M processors is a permutation of L numbers with M cycles. When the task graph and processor graph sizes increase, the need for considering different parent type combinations in the input population demands the increase in the population size and number of generations.

It is also observed that increase in population size and number of generations causes disproportionate increase in the execution time of GA without substantial improvement in the quality of the schedules generated. Moreover, large population size encounters memory limitation problem.

An ideal genetic algorithm should maintain a high degree of diversity within the population as it iterates from one generation to the next. Otherwise, the population may converge prematurely before the desired solution is found.

If there is not much scope for different parent-type combination in the available population then the genetic algorithm has to be run again with different population to get an acceptable near optimal solution. For each run, the genetic scheduler
performs 1000 iterations, with each trial costing about 15 seconds approximately on a Pentium machine (for a typical scheduling scenario in which task graph has 20 nodes and processor graph has processor graph has 4 nodes.)

In the genetic algorithm, when the schedule length does not change with the number of generations, local maxima for fitness value and local minima for finishing time result. It is observed that the local minima problem occurs due to absence of tasks at certain heights on all processors in the initial population leading to poor quality of input population. Crossover is not possible at those heights. So, the length of the string will not vary at all. Of course, mutation will exhaust all the possibilities at this length of the schedule as the number of iterations are increased. But, further progress is hampered.

In summary, inadequate population size, low degree of population diversity and the poor quality of the input population are the main factors causing local extrema problems, resulting in an increased schedule length and premature convergence in GA for multiprocessor scheduling problem.

In multiprocessor scheduling, there exists a tradeoff between the quality of scheduling and the time taken to obtain it. To get good quality solution in a reasonable amount of time, pure genetic algorithm requires a good population to start with.

Solutions

Local extrema problems are solved by several hybrid approaches (Gen and Cheng 1997). One of the most common forms of hybrid genetic algorithms is to incorporate local optimization as an add-on extra to the simple genetic algorithm. Their empirical results show that hybridization methods improve pure GA’s in solution quality, speed of convergence, and consistency on a variety of problems.

Features of AGA

In this thesis, a new approach named the Aided Genetic Algorithm (AGA) is proposed to alleviate the local extrema problem. The AGA solves this problem by
providing a member of the initial population from the heuristic pre-scheduler. The two stage heuristic algorithm based on conventional list scheduling heuristic (CH), proposed by Pattipati et al. (1990) takes the intertask communication into account while scheduling, and therefore it is used as the pre-scheduler in the present work. The rationale behind the hybridization of GA and CH is to exploit the desirable directed search properties of CH, while maintaining the population approach and the recombinative power of GA. It is found that the proposed AGA, further improves the solution quality and achieves the required convergence in lesser number of trials and iterations, compared to the conventional genetic algorithm.

The proposed AGA has the following advantages:

- Minimises the local extrema problem.
- Overcomes the limitations in the heuristic scheduler in which the allocation is priority based.
- Reduces the need for considering large population size.
- Reduces the number of generations and the number of trials to get minimum completion time.
- Allows heterogeneous processors and heterogeneous link capacities in the processor graph.
- Considers intertask communication while scheduling.
- Handles incompletely connected processor topology.
- Considers link contention problem.

The Aided Genetic Algorithm for Multiprocessor Scheduling

[This algorithm gives the task allocation for a multiprocessor system]

Step 1: Determine the level of tasks
Step 2: Generate the priority list
Step 3: Obtain a schedule through heuristic pre-scheduler
Step 4: Arrange the heights of the tasks in each sub-string of the pre-schedule in increasing order
Step 5: Input the pre-scheduler schedule as a member of the initial population in the GA.

Step 6: Run the GA for the fixed number of iterations to get the final schedule.

The list scheduling heuristic algorithm consists of sequence of computational steps which locally minimizes the completion time of the selected task. This belongs to a search strategy in which the search is along the best search direction. The GA, which is a randomized search strategy, performs widespread search for exploring all solution space. The AGA combines directed and stochastic search to get near optimal solutions.

**Pre-scheduler of AGA**

The pre-scheduler used in the AGA has to provide a good initial population to the genetic algorithm. So, any scheduler that gives a schedule for the directed task graph with non-negligible intertask communication in a reasonable amount of time can be used as a pre-scheduler for the AGA.

Though the list scheduling heuristic is used as a pre-scheduler in the present work, there is no restriction on the type of the pre-scheduler used in the AGA.

The CH employs a one-step optimization method to determine the task allocation that locally minimizes the schedule length. So, the resulting schedule of the CH is not only near-optimal but also takes care of the other factors, such as load balancing and processor utilization. As the task allocation in the CH is done according to the priority list, a task whose predecessor tasks have been completed cannot be considered for processor allocation if it has a lower priority than a task whose predecessor tasks have not yet been completed at that point in time. This problem is overcome in the AGA owing to its height based task allocation and effective use of genetic operators.

When the schedule is obtained from CH, the lists are tested for their height ordering. This test is needed since CH does not take into account the height ordering of the tasks while scheduling. So, a legal schedule of CH may not be a legal input
population to GA. Hence, the resulting strings of CH are arranged in increasing order of their heights and fed to GA as the first member of the initial population. The GA is then run for a fixed number of iterations to get the final scheduling results.

The quality of the schedule obtained from the GA is proved to be optimal only when the number of iterations approaches infinity. Moreover, the population size has to be adequate to provide enough scope for different parent-type combinations to take place. In the AGA, the one step optimization phase and the load balancing effect of the CH minimize the problem of local extrema. Hence the AGA provides the near-optimal solution in a lesser number of iterations and trials.

3.4 AN ILLUSTRATIVE EXAMPLE

Consider the task graph and the processor graph shown in Figure 3.4a and Figure 3.4b respectively. The schedules obtained using the CH, GA and AGA are presented in Table 3.2. In this table, for each task the priority, processor on which it is allocated (P), its height and completion time obtained using the CH, GA and AGA are presented.

Table 3.2

<table>
<thead>
<tr>
<th>Task No</th>
<th>Priority</th>
<th>P</th>
<th>Completion time (UT)</th>
<th>Ht</th>
<th>P</th>
<th>Completion time (UT)</th>
<th>P</th>
<th>Completion time (UT)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>2</td>
<td>24</td>
<td>2</td>
<td>1</td>
<td>19</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>16</td>
<td>2</td>
<td>2</td>
<td>16</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
<td>18</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>2</td>
<td>26</td>
<td>3</td>
<td>1</td>
<td>21</td>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>1</td>
<td>22</td>
<td>3</td>
<td>2</td>
<td>20</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>2</td>
<td>29</td>
<td>4</td>
<td>1</td>
<td>24</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1</td>
<td>28</td>
<td>4</td>
<td>2</td>
<td>26</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>1</td>
<td>36</td>
<td>4</td>
<td>1</td>
<td>35</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>1</td>
<td>36</td>
<td>5</td>
<td>1</td>
<td>35</td>
<td>1</td>
<td>31</td>
</tr>
</tbody>
</table>

* UT - Unit time
Figure 3.4a Task Graph

Figure 3.4b Processor Graph
The schedules obtained from CH, GA and AGA approaches are shown in Figure 3.5. The completion times of the schedules obtained using CH approach and GA approach are 36 and 35 time units respectively. The optimal solution obtained using AGA is 31 units. Consider the AGA schedule list representation shown in Figure 3.5. It can be observed that the schedule obtained from CH and AGA differ only in allocation of tasks 4 and 5. It can also be observed that getting such an allocation is not possible in CH approach due to its priority based allocation policy. Thus the AGA minimizes the shortcomings of the popular list scheduling heuristics and also overcomes the drawbacks of the simple genetic algorithm.

3.5 COMPUTATIONAL COMPLEXITY AND PERFORMANCE

The complexity of the GA is \( O(|E| + \varepsilon L) \varepsilon I \), where \( \varepsilon \) is the size of the population, and \( I \) is the number of iterations to get near-optimal solution.
The complexity of the AGA approach is $O(|E| + |L| + \psi L) + |L|$, which is lesser than that of GA, since the value of $\psi$ lies between 0.1 to 0.25.

The convergence characteristics of GA and AGA are presented in Figure 3.6 for the illustrative example. It can be observed that due to the good input population, there is reduction in search space and the optimal solution is obtained in a much shorter time in AGA compared to GA. The convergence performances of the GA and the AGA obtained for different task graphs prove that the minimum finishing time and the number of iterations to get the minimum finishing time is lesser in the case of AGA.


The quality of the schedules obtained from the AGA is tested on various standard task graphs for which near-optimal schedules are already known, and the results are presented in Table 3.3. In this table, the makespans obtained through other Conventional Heuristic (OCH), Conventional Heuristic (CH), GA, AGA and the critical lengths are presented for various task graphs and processor graphs. The critical length gives the lower bound on the makespan without considering the interprocessor communication overhead.

It is found that the makespans of these task graphs obtained from the AGA are lesser than or equal to the makespans obtained through the other scheduling algorithms. Hence, it is inferred that the performance of the AGA in terms of makespan is better than the other existing scheduling algorithms.
## Table 3.3
Makespans of different DAGs under different scheduling - comparison

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Task Graph Details</th>
<th>Processor Graph Details</th>
<th>Makespan Critical length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of Nodes</td>
<td>No. of Processors</td>
<td>Conne ctivity</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>4</td>
<td>Ring</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>3</td>
<td>Ring</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>2</td>
<td>Pipe</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>2</td>
<td>Pipe</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>3</td>
<td>Ring</td>
</tr>
</tbody>
</table>

*OCH - Other Conventional Heuristic

Based on the results of Table 3.3, one may be tempted to conclude that the reduction in the schedule length is only marginal in the AGA. However, it must be noted that the results of the AGA are obtained in one trial, whereas, the results of the GA listed in this table are those corresponding to the best schedule from 10 trials.

## Table 3.4
Performance comparison

<table>
<thead>
<tr>
<th>S. No.</th>
<th>No of Nodes</th>
<th>No of Processors</th>
<th>Run Time</th>
<th>Makespan (Unit times)</th>
<th>Critical length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>CH (msec)</td>
<td>GA (Sec)</td>
<td>AGA (Sec)</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>3</td>
<td>15</td>
<td>6</td>
<td>1.55</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>4</td>
<td>50</td>
<td>15</td>
<td>3.82</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>7</td>
<td>57</td>
<td>20</td>
<td>5.1</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>4</td>
<td>60</td>
<td>22</td>
<td>5.57</td>
</tr>
<tr>
<td>5</td>
<td>39</td>
<td>8</td>
<td>110</td>
<td>50</td>
<td>12.63</td>
</tr>
<tr>
<td>6</td>
<td>51</td>
<td>11</td>
<td>160</td>
<td>73</td>
<td>18.43</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>12</td>
<td>250</td>
<td>78</td>
<td>19.8</td>
</tr>
<tr>
<td>8</td>
<td>75</td>
<td>12</td>
<td>280</td>
<td>123</td>
<td>31.05</td>
</tr>
</tbody>
</table>
The CH, the GA and the AGA scheduling techniques are implemented on a Pentium PC with 100 MHz clock using the C programming language and tested on various task graphs with known near-optimal schedules. The run times of these scheduling algorithms and the makespans of various task graphs obtained through these algorithms are presented in Table 3.4. It can be seen that the makespans for various task graphs presented in this table are minimum in the AGA consistently. Though the run time of the AGA is higher than that of the CH, it is less than the run time of the GA by a factor of 4 in most of the cases.

The GA and AGA are tested on wide range of randomly generated task graphs. The speedup figures obtained with each of the scheduler while scheduling the random task graphs having 22 nodes and 52 nodes on to the processor graph with star and completely connected (CC) interconnection topologies are presented in Fig 3.7 to 3.18.

Each figure is drawn for a different value of C/E ratio, D, processor topology, number of nodes in the task graph and plots speedup versus number of processors for each task graph for each of the scheduling algorithms as done in chapter 2.

It is observed that the speedup ratio never decreases, while the number of processors available in the multiprocessor system is increased in AGA for CC topology. For other cases it is observed that speedup is not monotonically increasing as the number of processors is increased and it fluctuates before reaching saturation as shown in Figure 3.7-3.18. The fluctuation in speedup is prominent in star topology.

Moreover, it is observed that in Figures 3.9, 3.12, 3.15 and 3.18, when C/E ratio is 2, for star and CC interconnection topologies, for most task graph sizes, GA algorithm fail to obtain a speedup greater than unity, for number of processors is 2. This means, for these graphs, schedules obtained using the GA approach have been worse than sequential schedule.
Figure 3.6 Convergence Performance Comparison
Figure 3.7 Number of Processors vs Speedup

Figure 3.8 Number of Processors vs Speedup

Figure 3.9 Number of Processors vs Speedup
Figure 3.10 Number of Processors vs Speedup

Figure 3.11 Number of Processors vs Speedup

Figure 3.12 Number of Processors vs Speedup
Figure 3.13 Number of Processors vs Speedup

Figure 3.14 Number of Processors vs Speedup

Figure 3.15 Number of Processors vs Speedup
Figure 3.16 Number of Processors vs Speedup

Figure 3.17 Number of Processors vs Speedup

Figure 3.18 Number of Processors vs Speedup
The best and the average percentage improvements in speedup obtained using the AGA approach are summarized in Table 3.5 for different C/E, D, Processor topology and number of tasks in the task graph. It is observed that the average percent improvement is prominent for higher values of C/E values, and in star topology than CC topology.

Table 3.5
AGA’s speedup ratio improvement over GA scheduler

<table>
<thead>
<tr>
<th>C/E</th>
<th>22 Node Task graph, D = 1.8</th>
<th>52 Node Task graph, D = 2.6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Star</td>
<td>CC</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>0.5</td>
<td>36</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>47</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>61</td>
<td>38</td>
</tr>
</tbody>
</table>

% improvement over GA = (AGA speedup - GA speedup) / GA speedup * 100

From Table 3.5 it can be seen that the AGA gives an average improvement in speedup up to a maximum of 38% over GA. The percentage improvement of AGA increases as communication delay increases. This means that AGA can handle communication delays better than GA.

Table 3.6 presents the average speedups obtained in GA and AGA for different C/E, D, Processor topology, number of tasks in the task graph. Average speedup of AGA is always greater than GA. The difference in speedup of GA and AGA increases with increase in C/E for star topology, irrespective of number of nodes in the task graph. Also, difference in speedups is more in star topology than CC topology. This means that AGA can handle missing links in the processor graph better than GA.

It is found that the results of AGA are better than the results of GA in most of the cases. The results of AGA are better than CH in 70% of the cases tested. In other cases AGA and CH give the same result.
The results of AGA are inferior to those of DTH in many cases due to following facts:

(i) AGA considers static shortest path for data routing.

(ii) The concept of having data token in the task graph is not considered in AGA.

### 3.6 CONCLUSION

A genetic scheduler for scheduling a precedence constrained task graph with non-negligible intertask communication onto a message-passing multicomputer system is obtained. An Aided Genetic Algorithm (AGA) to minimize the problem of local extrema is presented. It is found, that the performance of the aided genetic scheduler is better than the genetic scheduler in terms of the schedule length as well as the time to obtain the same.

It is found that the makespans of task graphs obtained from the AGA are less than or equal to the makespans obtained through the existing heuristic algorithms (except DTH). It is also observed, more importantly, that the results arrived at, through the best possible heuristics are attainable now through the AGA. This significant finding makes it possible to infer that the performance of the AGA in terms of makespan is better than other scheduling algorithms (those dealing with regular task graph representation without data tokens).