CHAPTER 2

LOAD BALANCING IN PARALLEL COMPUTING

In parallel computing, a program is partitioned into tasks that can be executed concurrently and the tasks are then assigned to the computing elements in a multiprocessor system. The performance of a parallel computing system depends on the effective utilization of all the processors. Load balancing algorithms are designed to distribute the load on processors equally and maximize their utilization while minimizing the total task execution time. This chapter deals with hybrid GA based load balancing schemes which combines the local search methods with genetic algorithm.

2.1 INTRODUCTION

Load balancing mechanism should guarantee minimum difference between the heaviest-loaded and the lightest-loaded processors (Albert Y. Zomaya et al 2001). The execution of the load-balancing algorithm should be fast enough to make rapid task assignments.

Load-balancing algorithms can be broadly categorized as centralized or decentralized, dynamic or static (Casavant et al 1988; Niranjan G. Shivratri 1992). In a centralized load-balancing algorithm, the central scheduler will make the load balancing decisions based on the global load information sent from other processors. In decentralized load balancing, each processor in the system will broadcast its load information to the rest of the processors so that
the load information tables that are maintained locally can be updated. As every processor in the system keeps track of the global load information, load balancing decisions can be made on any processor.

A centralized algorithm can support large number of processors as it imposes fewer overheads on the system than the decentralized (distributed) algorithm. However, a centralized algorithm exhibits lower reliability since the failure of the central scheduler will result in the dysfunction of the load balancing policy. Despite its ability to support smaller systems, a decentralized algorithm is still easier to implement.

In static load-balancing, tasks will be allocated only once during compile time according to the apriori knowledge and will not be affected by the state of the system at the time. This static load balancing can be effective only for computations that have predictable runtime behaviors. On the other hand, for computations whose run time behavior is non-deterministic, it is better to perform dynamic load balancing which maps the tasks more than once or periodically during runtime according to the state of the system. A near-optimal schedule must be determined on the fly such that the tasks scheduled can be completed in the shortest time possible. As redistribution of tasks has to take place during runtime, dynamic load-balancing mechanisms are usually harder to implement. However, they tend to give better performance compared to static ones. This chapter deals with a dynamic centralized load balancing scheme for multiprocessor system using hybrid genetic algorithms.

2.2 MAJOR ISSUES IN LOAD BALANCING

In general, any load-balancing mechanism involves issues such as information policy, initiation policy, information exchange rule, and load balancing operation.
2.2.1 Information policy

Information policy covers the issues related to the load information necessary for making load balancing decisions. Load information is represented by a non-negative quantity called load index which takes a zero value for an idle processor and non-zero values for loaded processors. Load index may be based on the total number of resident tasks or the response time of the resident tasks in a processor or the task completion time and CPU utilization. As load measurement will occur frequently, the parameters chosen should not impose more overheads.

2.2.2 Initiation policy

Initiation policy makes a decision as to when and how often the load balancing mechanisms have to be initiated. Load balancing actions are initiated only when the expected profit exceeds the overheads. Threshold policy is used to indicate whether a node is lightly or heavily loaded. There are two types of threshold policies. The first policy is based on the difference between the largest and smallest load on the processors. If the difference is above a preset threshold, the profit is assumed to exceed the cost. The second policy uses two threshold values, one for indicating heavy load and the other for light load (Alonso et al. 1988; Pradeep K Sinha 2002). If there exist one heavily loaded processor and at least one or more lightly loaded processors, then the profit will exceed the cost. The threshold values can either be fixed or adaptive. The load balancing actions could be initiated either by a sender or a receiver or by both after determining the profitability. In the case of sender initiation, the profitability is determined when a new task arrives while in the case of receiver initiation by a departure of a task in execution. In the hybrid initiation policy, sender initiated policy is used when the system load is low and receiver initiated policy is used when the system load is high.
2.2.3 Information Exchange rule

This rule explains how to collect and maintain the workload necessary for making load-balancing decisions. There are three possible approaches governing information exchange. In periodic approach, individual processors will send their load information to other processors periodically even though it may not be needed at that moment. Too short period may result in increased overhead while a large period will make load information obsolete. In on-demand approach, load information is collected on demand just before making the load balancing decision. Though this approach minimizes the overhead, it causes an additional delay in load balancing. The on-change approach notifies its load information to other processors only when its load status changes.

2.2.4 Load balancing operation

The load balancing operation is defined by three policies: the location policy, the distribution policy, and the selection policy. The location policy specifies the load balancing domains which could be local, global or of variable size. Distribution policy determines how to redistribute the load among the processors in the balancing domain. Selection policy determines the tasks in a processor that are suitable to migrate. Preemptive selection policy allows the executing tasks to be migrated while non preemptive selection policy allows the tasks to be migrated if only they are newly created.

2.3 META HEURISTICS

This section explains the fundamentals of Genetic Algorithms, Simulated Annealing, Memetic Algorithms, and local search heuristics employed in this work.
2.3.1 Genetic Algorithms

A Genetic Algorithm (GA) is a search algorithm based on the principles of evolution and natural genetics. Genetic Algorithms combine the exploitation of past results with the exploration of new areas of the search space. By using survival of the fittest techniques combined with a structured yet randomized information exchange, a GA can mimic some of the innovative flair of a human search (David E. Goldberg 1989).

Genetic algorithms in contrast to the conventional search techniques start with an initial set of random solutions called population. Each individual in the population called a chromosome represents a solution to the problem at hand. A chromosome is a string of symbols (each symbol is called a gene); it is usually but not necessarily, a binary string (David E. Goldberg 1989).

At the beginning of genetic search, there is a widely random and diverse population and crossover operator tends to perform widespread search for exploring the entire solution space. The next generation is evolved using new chromosomes, called offspring, formed by either merging two chromosomes (parent) from current generation using a crossover operator or modifying a chromosome using a mutation operator. A new generation is formed by selecting chromosomes according to the fitness values. Fitter chromosomes have higher probabilities of being selected.

As the high fitness solutions develop, the crossover operator provides exploration in the neighborhood of each of them. In other words, the kind of search (exploitation or exploration) a crossover performs would be determined by the environment of the genetic system but not by the operator itself. In addition, simple genetic operators are designed as general-purpose
search methods (the domain independent search methods). After several
generations, the algorithm converges to the best chromosome which hopefully
represents the optimum or sub optimal solution to the problem.

2.3.2 Simulated Annealing

Simulated Annealing (SA), like genetic algorithm, is an
optimization procedure that performs randomized search in large, complex
and multimodal search space for providing a near optimal solution
(Kirkpatrick et al 1983). Annealing is a metallurgical process where the
ground state behavior of a metal is studied by gradually changing the
substance from a molten state to the lowest energy state.

In simulated annealing, a problem state is defined by the values of a
number of parameters. The state transition is done by changing the values of
the parameters using the Boltzmann distribution function in thermodynamics.
The objective is to minimize or maximize the value of some objective
function. At each state transition, the temperature of the system is reduced by
a small amount. The temperature schedule is so designed that the state of the
system freezes after hundreds of transitions. A logarithmically decreasing
temperature is found useful for convergence without getting stuck to a local
minimum state.

But it takes time to cool down the system to the equilibrium state. In
particular, simulated annealing knows little about whether a region of the
search space has been explored or whether a region is better for searching by
the use of statistical distribution function. An increasing and very fast
optimization procedure can be developed along with genetic algorithm
framework.
The SA algorithm begins with a randomly generated initial solution. This initial solution is said to be the current solution. A neighbour of this current solution is then generated. If the neighbour is found to be better than the current solution, it is unconditionally accepted to be the next current solution. On the other hand, if the neighbour is found as worse, it is not rejected outrightly, but accepted with a certain probability. To begin with, the probability of accepting a worse solution is kept high (thereby reducing the chance of SA algorithm getting trapped in a local optimum). As the number of iterations increases, this probability is reduced according to a specific distribution. The control parameter is known as the temperature (analogous to that of the physical process). The temperature is fixed at a very high value and is then brought down according to a schedule known as the cooling / annealing schedule. This annealing schedule determines how the probability of accepting a worse solution decreases.

2.3.3 Memetic Algorithms

A Memetic algorithm (MA) is a population-based heuristic search approach for solving combinatorial optimization problems based on cultural evolution (David Corne et al 1999; Radcliffe et al 1994). Meme is defined as a unit of information that reproduces itself while people exchange ideas. In contrast to genes, memes are typically adapted by the people who transmit them before they are passed on to the next generation. Genetic local search approach is a special case of a memetic algorithm which has been shown as a very effective approach for several combinatorial optimization problems such as the Traveling Salesman Problem (TSP) etc.

In contrast to hybrid evolutionary algorithms that use local refinement techniques as additional operators, MAs are designed to search in the space of locally optimal solutions instead of searching in the space of all candidate solutions (Merz 1997). This is achieved by applying local search after each of the genetic operators. Crossover and mutation operators are
applied to randomly chosen individuals for a predefined number of times. To maintain local optimality, the local search procedure is applied to the newly created individuals resulting from the crossover or mutation operator and the new individuals are added to the population afterwards. Before a new generation is processed, some individuals are selected for survival in order to keep the population size constant. The main aim of an adequate selection strategy is to keep the search goal-oriented and also to maintain the diversity of the population. Thus, in any generation, the population of individuals consists solely of local optima. The general template of the memetic algorithm used in this work is shown in Figure 2.1.

**procedure GLS / MA;**

begin
  Initialize population $P$;
  for each individual $i \in P$ do
    $i := \text{Local-Search}(i)$;
  end for;
  repeat
    for $i := 1$ to $\#\text{crossovers}$ do
      select two parents $i_a, i_b \in P$ randomly;
      $i_c := \text{Crossover}(i_a, i_b)$;
      $i_c := \text{Local-Search}(i_c)$;
      add individual $i_c$ to $P$;
    end for;
    for $i := 1$ to $\#\text{mutations}$ do
      select an individual $i \in P$ randomly;
      $i_m := \text{Mutation}(i)$;
      $i_m := \text{Local-Search}(i_m)$;
      add individual $i_m$ to $P$;
    end for;
    $P := \text{select}(P)$;
    until converged;
  end;

**Figure 2.1 GLS / MA Algorithm**
The creation of the initial population of candidate solutions for a given optimization problem is done in two steps. First, the desired number of feasible solutions is generated and then a local search procedure is applied to obtain local optima. An obvious way to generate initial solutions is to construct them in a purely random fashion and make sure that the feasibility constraints are satisfied but problem-dependent heuristics can be used alternatively.

To realize such a hybrid approach successfully, it is crucial that the interaction of the GA and LS leads to a search algorithm that is better than both of its components (Radcliffe et al 1994). An important question is how to design genetic operators that yield better (locally optimal) individuals than those produced by a multi-start local search. The recombination/crossover and mutation operators are supposed to explore the search space by “jumping” to new regions which are in turn searched by the local search procedure efficiently. The crossover operator enables us to define new starting points for a local search based on the information contained in the current population. The region between the new starting points may contain one or more local optima with better fitness, depending on the structure of the problem.

Assuming that good solutions lie relatively close to each other in the search space, one can “travel” from one solution to another in order to get close to the optimum. This approach is taken by many heuristics implicitly, but they differ in how they climb “uphill” to move from a locally optimal solution to a better one. Therefore, the aim of the crossover is to determine the regions of the search space where better solutions are most likely to be found by the local search procedure. The assumption of several solutions lying close to each other in the search space implies some kind of a distance criterion. The notion of a distance between solutions helps the crossover (in conjunction with the selection strategy) to decide where to guide the search.
On the other hand, the mutation operator should attempt to focus the search on randomly chosen regions, so that the algorithm is able to identify solutions that are hard to find by the distance-controlled crossover with its regional limitations. The different properties of the crossover and the mutation operators of the GA lead to a robust search algorithm.

2.4 PROPOSED LOAD BALANCING SCHEMES

In this section, two hybrid algorithms are proposed to provide an optimal solution for the task mapping in load balancing problem. First method improves the searching ability of simulated annealing by blending the search properties of Genetic Algorithm (GA) and SA to develop a hybrid genetic algorithm (HGA) which is equally applicable and has a better searching ability and power to reach a near optimal solution. Second method combines GA and Local Search heuristics to develop a genetic local search (GLS) algorithm also known as Memetic Algorithm (MA). Genetic local search is a hybrid heuristic that combines the advantages of population based search and local optimization. The proposed load balancing scheme has the following assumptions.

1. There are \( m \) identical parallel processors and \( n \) independent tasks \( (m < n) \).
2. The available tasks can be processed by any one of the \( m \) processors.
3. No task can be preempted once its processing has begun.
4. A fixed number of tasks, each having a task number and a size, is randomly generated and placed in a task pool in the central scheduler from where tasks are assigned to processors.
5. The strings in the population are used to represent the list of tasks known to the system at that point of time.
6. Dynamic load balancing scheme is adopted in a global balancing domain where an overloaded processor sends excess tasks to an under loaded processor during execution.

The techniques and principles of the proposed load balancing algorithms are discussed below:

2.4.1 Sliding Window Technique

At any point of time, there may be too many tasks waiting to be assigned in the task pool of the central scheduler. In order to limit the number of tasks to be assigned, and to initialize the population of possible solutions, sliding window technique is used. Only the tasks within the window are considered for execution each time.

The window size is fixed and the number of elements in each string is equal to the size of the window. The permutations of these tasks will form lists that represent the different orders in which these tasks can be scheduled for execution. When the proposed HGA/MA arrives at a task schedule, these tasks will be assigned to processors accordingly. Once these tasks have been assigned, the sliding-window will be updated with new tasks by sliding along to the next set of tasks on the task queue and repeating the assignment process.

2.4.2 Encoding Mechanism

There are many encoding methods: Binary encoding, Character encoding and Real-value encoding. In order to enhance the understanding of the problem formulation, the strings are encoded using decimal numbers. Each element in the strings has two values, one to represent the task number and another (in brackets) to represent the load in time units.
Two-dimensional strings provide a better representation of the load-balancing problem. One dimension represents the numbers of the processors in the system while the other dimension shows the schedule of tasks on each individual processor (i.e., CPU queues). An example of a two-dimensional string is shown in Figure 2.2. The tasks in the string are executed in a left to right order. As the crossover operator cannot work with two dimensional strings, these strings are connected end-to-end to form a long one dimensional string as shown in Figure 2.3.

<table>
<thead>
<tr>
<th>P1</th>
<th>5 (9)</th>
<th>3 (6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>2 (7)</td>
<td>7 (3)</td>
</tr>
<tr>
<td>P3</td>
<td>1 (1)</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>4 (8)</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>6 (16)</td>
<td>8 (14)</td>
</tr>
</tbody>
</table>

Figure 2.2 Two dimensional representation of a string

<table>
<thead>
<tr>
<th>5 (9)</th>
<th>3 (6)</th>
<th>2 (7)</th>
<th>7 (3)</th>
<th>9 (16)</th>
<th>1 (1)</th>
<th>4 (8)</th>
<th>6 (16)</th>
<th>8 (14)</th>
<th>10 (19)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.3 Mapping a two dimensional string to a one dimensional string

2.4.3 Objective and Fitness Functions

An objective function is a measure used to evaluate the quality of the solutions (Albert Y. Zomaya et al 2001; Seong hoon Lee 2005). The main objective here is to arrive at task assignments that will achieve minimum execution time, maximum processor utilization, and a well-balanced load
across all processors. The objective function is mapped to the fitness function of the HGA/MA/GA. This fitness function will then be used to measure the performance of the strings in relation to the objectives of the algorithm.

### 2.4.3.1 Makespan

The first objective function for the proposed algorithm is the makespan of the task schedule. *Makespan* is basically the largest task completion time among all the processors in the system (Albert et al 2001; Kidwell et al 1994).

For the string shown in Figure 2.3, the *makespan* is calculated as follows: Assuming that all the processors are idle when this string is evaluated, the total time for executing tasks 5 and 3 on processor 1 is $9 + 6 = 15$ time units. Similarly, tasks 2, 7, and 9 will be executed on processor 2 for a total of 26 time units. The completion times of the remaining processors 3, 4, and 5 are calculated to be 1, 8, and 49 time units, respectively. In this example, the maximum completion time is 49 time units for processor 5. Therefore, the *makespan* for this task schedule is 49.

However as the processors may not always be idle when the string is evaluated, it is inaccurate to calculate *makespan* by considering the sizes of these new tasks alone. The current load existing on individual processors must also be taken into account in order to get a more accurate *makespan* value. Therefore, if the current loads on processors 1 to 5 are 3, 4, 15, 13, and 8 time units, respectively, and the same schedule in Figure 2.3 is evaluated again, then the total task completion times of each processor can be calculated using the following formula:

$$P_i (completion\ time) = current\ load\ of\ P_i + new\ load\ assigned\ to\ P_i$$

(2.1)
The completion times for different processors are now calculated as follows:

\[
P_1 = 3 + 15 = 18 \quad P_2 = 4 + 26 = 30
\]
\[
P_3 = 15 + 1 = 16 \quad P_4 = 13 + 8 = 21
\]
\[
P_5 = 8 + 49 = 57
\]

The makespan in this case is 57 time units. The task schedule is better when the makespan is smaller. Therefore, minimizing the makespan optimizes this objective function.

2.4.3.2 Average Processor Utilization

The next crucial factor to the fitness of a string (task schedule) is the average processor utilization. This is essential since high average processor utilization implies that the load is well balanced across all processors and the total execution time is reduced.

The expected utilization of each processor is calculated based on the given task assignment. The utilization of the individual processors is given by:

\[
P_i(\text{utilization}) = \frac{P_i(\text{completion time})}{\text{makespan}}
\]  \hspace{1cm} (2.2)

This will lead to

\[
P_1 = \frac{18}{57} = 0.3158 \quad P_2 = \frac{30}{57} = 0.5263
\]
\[
P_3 = \frac{16}{57} = 0.2807 \quad P_4 = \frac{21}{57} = 0.3684
\]
\[
P_5 = \frac{57}{57} = 1.0000
\]
The average processor utilization (APU) is given by:

\[
APU = \sum P_i(\text{utilization})/N
\]  

(2.3)

where \( N \) = number of processors in the system. The average processor utilization in this case is \( \approx 0.4982 \). By increasing the average processor utilization, the probability of processors being idle for a long time will be reduced.

### 2.4.3.3 Number of Acceptable Tasks

The overall task assignment being evaluated may have a small makespan and high average processor utilization. However, the task assignment process may still overload some of the processors. Therefore, the third objective is to determine the number of acceptable tasks. Each processor’s task queue is checked individually to find whether new tasks if assigned, will overload or underload the processor.

If the load of a processor (found by adding the current system load and those contributed by the new tasks) is within the light and heavy thresholds, the new tasks are acceptable. If it is above the heavy threshold or below the light-threshold, then it is unacceptable.

For the example shown in Figure 2.2, if the heavy threshold is set at 32 and the light threshold at 20, then, processors 1 and 3 will be under-loaded and processor 5 will be overloaded. Therefore, only tasks to processors 2 and 4 are acceptable. The percentage of acceptable tasks is given by:

\[
\% \, \text{of Acceptable tasks} = \frac{\text{Number of acceptable tasks}}{N}
\]  

(2.4)
where $N = \text{number of processors in the system}$. Higher percentage of acceptable tasks indicates the betterness of its load balancing potential.

### 2.4.3.4 Combined Fitness Function

The three objectives discussed above are incorporated into a single fitness function and given by the following equation:

$$
Fitness = \frac{1}{\text{Makespan}} \times \text{APU} \times \frac{\text{No. of processor queues}}{\text{No. of processor}}
$$

(2.5)

The fitness function is used to evaluate the quality of the task assignments.

### 2.4.4 Selection

The selection strategy of GA and the proposed HGA and GLS methods are discussed below:

#### 2.4.4.1 Selection Strategy in GA

The selection technique used is based on the roulette wheel method. In this case, the slots of the roulette wheel must be determined based on the probabilities of individual strings surviving into the next generation. These probabilities are calculated by dividing the fitness values of the individual strings by the sum of the fitness values of the current pool of strings. Adding the probability of the current string to the probability of the previous string creates the slots. For example, if the probabilities of survival of string 1 and string 2 are 0.25 and 0.3, respectively, then slot 1 will range from 0-0.25 while slot 2 will range from 0.26-0.55. The slots will be allocated up to the value of 1. Hence, each string in the population will occupy a slot size that is proportional to its fitness value (David E. Goldberg 1989; Michalewicz 1994).

After defining the slots, random numbers between zero and one are generated. The numbers obtained will determine the strings that will survive
into the next generation. As the fitter strings are represented by larger slots on the wheel, the chances of the randomly generated numbers falling into slots that represent these strings will be greater.

### 2.4.4.2 Selection Strategy in Hybrid GA

In the new method of selection called stochastic selection, a chromosome with a value $x_i$ is considered from a pool $P(g)$ of generation $g$ and is selected based on Boltzmann probability distribution function. Let $f_{max}$ denote the fitness of currently available best string. If the next string has fitness $f(x_i)$, the new string is selected with Boltzmann probability (Pakhira 2003).

$$P = e^{-\frac{f_{max} - f(x_i)}{T}}$$  \hspace{1cm} (2.6)

where $T = T_0 (1 - \alpha)^k; \quad k = \frac{1 + 100g}{G}; \quad g$ - current generation number; $G$-max $(g)$.

The value of $\alpha$ can be chosen from the range $[0, 1]$, and $T_0$ from the range $[5, 100]$. Equation (2.6) shows that the value of $T$ decreases exponentially or logarithmically with increasing value of $g$. Hence, the probability $P$ decreases with every generation. This is significant in terms of convergence. The final state is reached when computation approaches zero value of $T$, i.e., the global solution is achieved at this point.

In the hybrid genetic algorithm shown in Figure 2.4, the probability of selecting the population with best strings is very high. However, elitism is suggested to eliminate the chance of any undesired loss of information during the mutation stage.
procedure HGA;
begin
    $g = 0$;
    Initialize ($T$, $P$ ($g$));
    Evaluate $P$ ($g$) using fitness function;
    $f_{max}$ maximum fitness of $P$ ($g$);
    termination_condition = false;
    while (NOT termination_condition) do
    begin
        $g = g + 1$;
        for $i = 1$ to $N$ do
        begin
            if $f_{max} - f(x_i) \leq 0$ then
                select $x_i$ from $P$ ($g$) and set $f_{max}$ to $f(x_i)$;
                $e^{\left(\frac{(f_{max} - f(x_i))}{T}\right)} > \text{random [0,1]}$ then
                select $x_i$ from $P$ ($g$);
            else
                select $x$ corresponding to $f_{max}$;
            end
            crossover;
            mutation;
        end evaluate $P$ ($g + 1$) using fitness function;
        lower $T$;
    end
end

Figure 2.4 Hybrid Genetic Algorithm

2.4.4.3 Tournament Selection in GLS / MA

In tournament selection, ‘$n$’ number of individuals is selected at random with uniform probability and the best one among them finds its way into the new population (David E. Goldberg 1989; Michalewicz 1994). The
winner can also be chosen probabilistically. The process is then repeated population size times. A widely used value of \( n \) is two. Tournament selection has the advantage that it need not be global so that local tournaments can be held simultaneously in a spatially organized population.

2.4.5 Crossover

After completing the selection process, the fitter strings should be left in the pool. These strings must then be converted into one-dimensional strings before they can be crossed over. Then, the crossover operation is performed on pairs of strings that are picked at random from the population. However, the normally used single point crossover method cannot be applied to this load balancing problem as this method may cause some tasks to be assigned more than once while some may not be assigned at all. Therefore, in order to ensure that each task is assigned only once, the cyclic crossover method is implemented in this work (Kidwell et al 1994) both for GA and HGA approaches.

2.4.5.1 Cyclic crossover for GA / HGA

Consider two Parent strings \( P1 \) and \( P2 \). The tasks in these strings are denoted as \( T_{ab} \) whereby \( a \) represents the string number (i.e., Parent string 1 or 2) and \( b \) the task position in that string. The crossover point is selected randomly anywhere between the first and the last position of the string. Let \( T_{1S} \) be the task at position \( S \) (i.e., the starting point that is selected) in \( P1 \). This task is marked as “completed.” Then, the task at the same position in \( P2 \) (\( T_{2S} \)) is also marked off as “completed.” Then, the task in \( P1 \) that matches \( T_{2S} \) must be located. When this is found to be at position \( X \) in \( P1 \), this task (\( T_{1X} \)) is marked off as “completed.” Then, the task that is at the location \( X \) in \( P2 \) is marked off as “completed” as well. The same process is repeated until the start task (\( T_{1S} \)) is reached again. Once this happens, all the tasks that are
marked off stay unchanged, while the rest are swapped between strings. After
the crossover is completed, the strings will be reordered/converted back to its
two dimensional form to enable the calculation of their new fitness values.

2.4.5.1.1 Illustration -I

For example, consider two parent strings A and B with the
following structure:

\[ A = 6 \ 10 \ 7 \ 1 \ 4 \ 8 \ 2 \ 5 \ 9 \ 3 \]
\[ B = 4 \ 8 \ 9 \ 5 \ 3 \ 7 \ 10 \ 6 \ 2 \ 1 \]

To initiate the operation, a starting point must be chosen first. In this
case, the first task from parent string A is chosen as the starting point.

\[ A' = 6 \ 10 \ 7 \ 1 \ 4 \ 8 \ 2 \ 5 \ 9 \ 3 \]
\[ B' = 4 \ 8 \ 9 \ 5 \ 3 \ 7 \ 10 \ 6 \ 2 \ 1 \]

As mentioned earlier, every task has to be taken from any one of the
two parents. Therefore, with the first choice being task 6 from position 1 of
string A, task 4 has to be selected from string B because this task is at the
equivalent position in string B.

\[ A' = 6 \ 10 \ 7 \ 1 \ 4 \ 8 \ 2 \ 5 \ 9 \ 3 \]
\[ B' = 4 \ 8 \ 9 \ 5 \ 3 \ 7 \ 10 \ 6 \ 2 \ 1 \]

The task 4 from string A is selected next. In turn, this selection
requires task 3 to be selected from string B next because task 3 is situated at
the same position (position 5) in string B. Using a similar selection pattern,
this process continues until the following pattern is generated:
This process terminates when task 5 is selected from string A. This is because the selection of task 5 results in choosing task 6 next from string A. However, this is impossible because task 6 has been selected as the first task earlier. Hence, the process has to terminate at this point. When the algorithm returns to the original task, it completes a cycle thus giving the operator its name. After completing this cycle, the remaining gaps in the parent strings are filled with tasks from the other string. This means that the tasks from the parent strings at these gaps are swapped. Therefore, the cyclic crossover yields the following child strings in the end:

\[
A' = 6 \quad - \quad - \quad 1 \quad 4 \quad - \quad - \quad 5 \quad - \quad 3 \\
B' = 4 \quad - \quad - \quad 5 \quad 3 \quad - \quad - \quad 6 \quad - \quad 1
\]

2.4.5.2 Sub-schedule Preserving Crossover for MA

The sub-schedule means a complete schedule for one machine/processor. Sub-schedule preserving crossover operator is chosen because a sub-schedule is considered to be the natural building block (Runwei Cheng et al 1997). The proposed crossover takes two parents and creates a single offspring by propagating the overall partitioning structure and a sub-schedule into offspring from one parent and then completing the offspring with remaining jobs derived from another parent. The main steps are as follows:
1. Select any one processor from one parent randomly.

2. Select the sub-schedule of that processor from the same parent.

3. Get remaining jobs from the other parent by marking a left-to-right scan.

### 2.4.5.2.1 Illustration- II

Consider the parent strings A and B which denote the task schedule of the three processors, P1, P2 and P3.

\[
A = 6 \ 10 \ 7 \ * \ 1 \ 4 \ 8 \ * \ 2 \ 5 \ 9 \ 3 \\
B = 4 \ 8 \ 9 \ 5 \ * \ 3 \ 7 \ 10 \ * \ 6 \ 2 \ 1
\]

The sub-schedule of each parent string is separated by an asterisk. It represents the tasks scheduled within the processor. In the sub-schedule preservation crossover, the middle sub-schedule from the parent A is selected and transmitted to the child C. The other sub-schedules are obtained from the other parent B by making a left to right scan.

\[
A = 6 \ 10 \ 7 \ * \ 1 \ 4 \ 8 \ * \ 2 \ 5 \ 9 \ 3 \\
C = 9 \ 5 \ 3 \ * \ 1 \ 4 \ 8 \ * \ 7 \ 10 \ 6 \ 2 \\
B = 4 \ 8 \ 9 \ 5 \ * \ 3 \ 7 \ 10 \ * \ 6 \ 2 \ 1
\]

### 2.4.6 Mutation

The mutation technique works by selecting two tasks randomly and swapping them. The swap mutation technique used in this work selects a processor randomly and a task from that processor. Similarly, a second processor and a task in that processor are randomly selected and the tasks are
swapped. However, as the second processor may be the same as the first, there is a possibility of the second task selected being the same as the first task. When this happens, the mutation process will be redundant and hence, limits the search space. Therefore, the swap mutation operation must ensure that both the processors and tasks selected are different before these tasks can be swapped over. After mutating the strings, the population will be decoded to find the new fitness values.

2.4.7 Local Search Heuristic in MA

Local search heuristic employed in the MA is the 2-opt heuristic (David Corne et al 1999; Merz et al 1997). The 2-opt neighborhood is defined as the set of all solutions that can be reached from the current solution by swapping two elements in the string/chromosome. Fitness is calculated for the entire neighborhood of that particular string. If any one solution in the neighborhood improves the fitness then the corresponding string replaces the original string.

2.4.8 Phases of load balancing mechanism

The load-balancing mechanism involves initiation rule, information exchange rule, and task mapping. It is important to decide when and how often to initiate the load-balancing mechanism in order to achieve maximum profits from it. As the mechanism used in this work schedules tasks without task migration, running load-balancing too often may overload the processors. The central scheduler may assign the tasks continuously to the processors while only a few of the previously assigned tasks have been completed. On the other hand, initiating the mechanism infrequently may result in some processors being idle for a long time before a new set of tasks is allocated to these processors again. This will result in low processor utilization.
Thus, in the proposed work, the load-balancing mechanism is initiated whenever the central scheduler detects a processor that has finished processing all the tasks in its queue. This is more feasible than waiting for all the processors to finish their tasks before the task mapping is repeated. This initiation rule ensures that the processors are neither idle nor overloaded.

The overall load in the system is updated by using both the global and automatic update rules. As the central scheduler makes the load-balancing decisions, it is important that the information in the load information table is recent and updated. Load information is collected from the processors only when a set of new tasks is sent to the destination processors based on the task mapping.

Once the load information is collected, the scheduler will update the global load information table in the central scheduler automatically. After every time unit, the scheduler will check the load information table to detect whether any processor has completed the tasks. If the tasks are still under execution, the scheduler will update the load information table automatically by deducting one unit of time from the remaining times of the tasks that are currently being processed by each processor. This process will continue until the scheduler detects the completion of a task by a processor.

Fixed (or static) threshold values are not suitable for this dynamic load balancing scheme since the state of the system changes with time. For example, the fixed threshold value for a system with initial load of 100 tasks will not be suitable for a current load of 10 tasks. Hence, in this work an adaptive threshold policy is used in which the thresholds are adjusted as per the change in global system load.
2.4.8.1 Average Load Calculation

Since thresholds are detrimental to the fitness of the generated task mappings, they must be adjusted each time before the tasks within the sliding-window are assigned (via the HGA / MA). In order to determine proper thresholds, the average load must be determined first. The average load is calculated as shown below:

\[
L_{avg} = \frac{\sum_{i=1}^{N}(CTT_i + QT_i) \times \sum_{k=1}^{M} TT_k}{N}
\]  

(2.7)

where

- \( L_{avg} \) = average load
- \( CTT_i \) = remaining execution time of the task currently being processed by processor \( i \)
- \( QT_i \) = total execution time of all the tasks waiting at processor \( i \)'s task queue
- \( TT_k \) = execution time of individual tasks within the sliding-window
- \( N \) = number of processors in the system
- \( M \) = number of tasks in the sliding-window

If allocation of the set of new tasks results in all processor loads being equal to the calculated average load value, the system will be well balanced. However, using this value as a threshold will be very restrictive as a global balanced state is hard to achieve, especially in a large system. Sometimes, it is more realistic to relax the requirements slightly for load-balancing. Therefore, heavy and light thresholds are used to add more flexibility to the assignment process.
2.4.8.2 Double threshold policy

Threshold is a value that is used to indicate whether a processor is heavily or lightly loaded. A system with a double threshold policy will usually have two workload thresholds (Alonso et al 1988; Pradeep K Sinha 2002). Processors with loads that are less than the light threshold are referred to as lightly loaded processors, whereas those with loads higher than the heavy threshold are categorized as heavily loaded processors.

If the load of a processor lies between the heavy and light thresholds, it is said to be normally loaded. Double threshold policy will reduce the traffic overhead significantly. It is important to determine appropriate thresholds for a good load-balancing algorithm. If the threshold is too low, excessive load-balancing will occur thus causing thrashing and degradation in performance. However, if the threshold is set too high, the load-balancing mechanism will not be very effective.

Two kinds of threshold policies can be considered for a load-balancing algorithm, a fixed threshold policy or an adaptive threshold policy. As the name suggests, a fixed threshold policy has predetermined thresholds that will not change when the system load changes. On the other hand, an adaptive threshold policy has thresholds that are adjusted as the system load changes. Based on the average load value calculated in equation (2.7), the heavy and light thresholds are derived as follows:

\[ T_H = H \times L_{\text{avg}} \]  \hspace{1cm} (2.8)

\[ T_L = L \times L_{\text{avg}} \]  \hspace{1cm} (2.9)

where \( T_H \) is the heavy threshold, \( T_L \) is the light threshold, and \( H \) and \( L \) are constant multipliers that determine the flexibility of the load-balancing
mechanism. $H$ value is always greater than one as it determines the amount of average workload that can be exceeded before the processor becomes heavily loaded. Conversely, $L$ value is always less than one as it shows how short the processor loads can be from the average before the processor becomes lightly loaded. These two values determine the flexibility and the effectiveness of the mapping mechanism. The $H$ and $L$ values used for this work using HGA are set to 1.2 and 0.8 respectively and for MA they are set to 1.1 and 0.9 respectively. This means that processor loads that are 20 percent and 10 percent above or below the average value will be considered acceptable for HGA and MA correspondingly.

2.4.8.3 Task Allocation

Even though the fittest task mapping shows good load balancing and processor utilization, it is not always feasible to assign tasks to all the processors. Therefore, the task assignment is constrained such that only those tasks that will not overload the processor will be allocated. The list of tasks assigned in this round is recorded for later use.

2.4.8.4 Updating the Sliding Window

After the schedulable tasks in the sliding window have been assigned by the HGA/MA/GA, the window has to be filled up again by sliding along the subsequent tasks waiting in the task queue. Hence, the scheduler will remove the tasks assigned in the previous round from the window and replace them with new tasks taken from the system task queue. When the window is full again, the load-balancing function is invoked when necessary.

In the process described above, it is assumed that there would be a continuous inflow of tasks to the system task queue always. However, as the
simulator generates only a finite number of tasks at some point of time there may not be enough tasks to fill up the window. In that case it will not be feasible to run the task mapping procedure on a small number of tasks. Hence the central scheduler will allocate the tasks one by one to the lightly loaded processors in a round robin fashion till all the tasks in the window are assigned.

The load balancing schemes are explained in the following illustration.

2.4.9 Illustration - III

Table 2.1 shows the initial state of a distributed system with 5 processors and Table 2.2 shows the new tasks to be assigned. All the steps involved in the assignment of tasks are discussed in detail for both the schemes.

<table>
<thead>
<tr>
<th>Processor Number</th>
<th>Current Task</th>
<th>Tasks Waiting to be Processed</th>
<th>Total Queue Length (Time Units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6(1)</td>
<td>5(2)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>8(0)</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>7(1)</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1(2)</td>
<td>3(1)</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4(1)</td>
<td>9(1)</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total Load in the System</strong></td>
<td></td>
<td></td>
<td><strong>9</strong></td>
</tr>
</tbody>
</table>
Step 1: Current State of the System

Table 2.1 shows that processor 2 is idle and load balancing operation has to be initiated.

Step 2: New Tasks to Be Scheduled

The new set of tasks to be scheduled is admitted into the sliding-window as shown in Table 2.2.

Table 2.2 New Tasks inserted into the Sliding Window

| 11 (1) | 12 (7) | 13 (6) | 14 (8) | 15 (9) | 16 (16) | 17 (3) | 18 (14) | 19 (16) | 20 (19) |

Total time units taken to complete the new tasks = 1 + 7 + 6 + 8 + 9 + 16 + 3 + 14 + 16 + 19 = 99

Step 3: Fittest Task Mapping Generation

The new task mapping using the proposed HGA and MA are given below.

a) HGA

P1: 18(14) 15(9)
P2: 17(3) 20(19)
P3: 12(7) 16(16)
P4: 14(8) 13(6)
P5: 11(1) 19(16)
Load (time units) added to processors:

- P1: 23
- P2: 22
- P3: 23
- P4: 14
- P5: 17

b) MA

- P1: 16(16) 17(3)
- P2: 13(6) 19(16)
- P3: 18(14) 12(7)
- P4: 15(9) 11(1) 14(8)
- P5: 20(19)

Load (time units) added to processors:

- P1: 19
- P2: 22
- P3: 21
- P4: 18
- P5: 19

Step 4: Determine the New Thresholds

Since the adaptive threshold policy is followed, the next step is to determine new thresholds based on the current system load (Table 2.1) and new tasks to be scheduled (Table 2.2):

\[ L_{\text{avg}} = \frac{(3 + 0 + 1 + 3 + 2) + 99}{5} = 21.6 \]
a) HGA

\[ T_H = 1.2 \times 21.6 \approx 25.92 \]
\[ T_L = 0.8 \times 21.6 \approx 17.28 \]

Hence, the new heavy threshold is 26 while the light threshold is 17.

b) MA

\[ T_H = 1.1 \times 21.6 \approx 23.76 \]
\[ T_L = 0.9 \times 21.6 \approx 19.44 \]

Hence, the new heavy threshold is 24 while the light threshold is 19.

**Step 5: Find Acceptable Load Sizes for Each Processor**

The load that each processor can accept in this round can be calculated by subtracting the current load value (Step 1) from the heavy threshold determined in Step 4.

a) HGA

For example, for processor 1:

\[ \text{Acceptable load} \leq \text{heavy threshold} - \text{total load of processor 1} \]
\[ \leq 26 - 3 \]
\[ \leq 23 \]

Similarly, processors 2 to 5 can receive loads that will not exceed the acceptable loads of 26, 25, 23 and 24 respectively.

b) MA

For example, for processor 3:

\[ \text{Acceptable load} \leq \text{heavy threshold} - \text{total queue length of processor 3} \]
\[ \leq 24 - 1 \leq 23 \]

Similarly, processors 1, 2, 4, and 5 can receive loads that will not exceed the acceptable loads of 21, 24, 21 and 22 respectively.

**Step 6: Task Assignment Decision**

The next step is to check if the task assignment generated in Step 3 is feasible. The heavy load multiplier used is 1.2, 1.1 for HGA and MA respectively. This means that the decision already has 20%, 10% flexibility from the ideal load-balancing scenario for HGA and MA correspondingly. If new tasks that increase the load beyond the acceptable load value are assigned, it will cause a severe load imbalance situation. Those new tasks will not be assigned in the current round instead they will be considered for task mapping in the next round.

In this example, it is observed that the new set of tasks for all the processors can be assigned since the load after the task mapping is less than the acceptable load.

**Step 7: New State of the System**

a) **HGA**

After the assignment of tasks using HGA, the load information table at time \( t + 1 \) is as shown in Table 2.3.
Table 2.3  Load Information at time t +1 after mapping the tasks using HGA

<table>
<thead>
<tr>
<th>Processor Number</th>
<th>Current Task</th>
<th>Tasks Waiting to be Processed</th>
<th>Total Queue Length (Time Units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6(0)</td>
<td>5(2) 18(14) 15(9)</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>17(3)</td>
<td>20(19)</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>7(0)</td>
<td>12(7) 16(16)</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>1(1)</td>
<td>3(1) 14(8) 13(6)</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>4(0)</td>
<td>9(1) 11(1) 19(16)</td>
<td>18</td>
</tr>
<tr>
<td><strong>Total Load in the System</strong></td>
<td></td>
<td></td>
<td><strong>104</strong></td>
</tr>
</tbody>
</table>

b) MA

After the assignment of tasks using the memetic algorithm (MA), the load information table at time t + 1 is shown in Table 2.4.

Table 2.4  Load Information at time t +1 after mapping the tasks using MA

<table>
<thead>
<tr>
<th>Processor Number</th>
<th>Current Task</th>
<th>Tasks Waiting to be Processed</th>
<th>Total Queue Length (Time Units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6(0)</td>
<td>5(2) 16(16) 17(3)</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>13(6)</td>
<td>19(16)</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>7(0)</td>
<td>18(14) 12(7)</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>1(1)</td>
<td>3(1) 15(9) 11(1) 14(8)</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>4(0)</td>
<td>9(1) 20(19)</td>
<td>20</td>
</tr>
<tr>
<td><strong>Total Load in the System</strong></td>
<td></td>
<td></td>
<td><strong>104</strong></td>
</tr>
</tbody>
</table>
Step 8: Updating the Sliding-Window

In Step 2, the sliding-window consisted of the tasks in Table 2.2. Since all the tasks from 11-20 are assigned to the processors in step 7, these tasks in the sliding-window have to be replaced with ten new tasks. Hence, the new sliding window will have the tasks shown in Table 2.5.

Table 2.5 Contents of New Sliding Window

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>(5)</td>
<td>(8)</td>
<td>(9)</td>
<td>(6)</td>
<td>(14)</td>
<td>(4)</td>
<td>(6)</td>
<td>(17)</td>
<td>(12)</td>
</tr>
</tbody>
</table>

Total time units = 5 + 8 + 9 + 6 + 14 + 4 + 6 + 17 + 12 + 13 = 94

With this new set of tasks to assign, all the above steps will be repeated.

2.5 SIMULATION RESULTS

The proposed load-balancing mechanism employing HGA, MA has been simulated and implemented in C on a Pentium IV PC (1.5 GHz) machine.

2.5.1 Test Parameters

The performance of the proposed methods is compared with the GA based load balancing scheme. The algorithms are evaluated based on two metrics: total completion time and average processor utilization. The default parameters are varied one at a time to study the effect of these parameters on the performance.

However, in certain situations it is necessary to vary more than one parameter. For example, consider the situation when there are 20 tasks to be allocated and there are 20 processors in the system. The HGA/MA or any other algorithm will not be efficient in this case since it is difficult to find a
task mapping that can balance out the load of 20 tasks over 20 processors equally and have exactly one task assigned to each processor. Therefore, the examples chosen must lead to meaningful results that are valid for analysis. Six tests are produced for each set of parameters and average of the best three tests is considered for analysis. Synthetic load distributions comprising the following parameters are used:

- Number of tasks : Maximum =1000, Minimum = 100
- Number of processors : Maximum = 25, Minimum = 5
- Window size : Maximum = 50, Minimum =10
- Number of generation cycles : Maximum = 60, Minimum =10
- Population size : Maximum = 40, Minimum =5
- Maximum completion time of each task: 20
- H multiplier = 1.2, and L multiplier = 0.8 for HGA
- H multiplier = 1.1, and L multiplier = 0.9 for MA

### 2.5.2 Effect of Number of Tasks

The number of tasks are varied from 20-1000 and the effect of number of tasks over the makespan and average processor utilization are studied. Default values are used for the other parameters.

The makespan depends on the number of tasks, task completion time, and the algorithm. Figure 2.5 shows the makespan for all the three algorithms. It is observed that the makespan increases proportionately with respect to the number of tasks. This is because more tasks are to be scheduled and it takes longer time to complete all the tasks. HGA and MA perform better than GA in most of the cases.
Figure 2.5 Number of Tasks Vs. Makespan

In these experiments it is observed that the increase in the number of tasks increases the processor utilization significantly with HGA and MA when compared to GA. The processor utilization using HGA and MA is in the range of 97-99 percent giving a smaller standard deviation whereas the GA has a utilization of 92-98 percent with a higher standard deviation. The effect of average utilization on the number of tasks is shown in Figure 2.6.

Figure 2.6 Number of Tasks Vs. Average Utilization
2.5.3 **Effect of window size**

The size of the window and the number of processors are varied to observe the behavior of the proposed algorithms. It is important to note that the number of processors has to be changed according to the size of the window because an inappropriate number of processors may affect the load balancing schemes. A window size that is too small for the number of processors available may stop the algorithm from working or result in delays in developing the fittest task assignment.

If the fittest task assignment results in any of the processors being idle, it will be rejected and HGA/MA will be initiated again to generate another assignment. Therefore, when there is a situation where there are 25 processors and the size of the window is 10, the chances of having at least one task on each processor (to prevent the processors from being idle) is nil. The HGA/MA will then be re-executed, but a task mapping suitable for this load will never be generated. Hence, it is crucial that the window size correlates with the number of processors in the system.

In the next test run, the completion time is measured for different window sizes ranging from 10 to 50. Table 2.6 shows the number of processors to be used for each window size.

Figure 2.7 shows that the *makespan* is greatly reduced when the window size is increased for all the three approaches. However, this improvement is mainly caused by the fact that more processors are used for larger windows. Therefore, even though there are more tasks to be scheduled in each round, the extra load is easily handled by the additional processors.
Table 2.6  Window Size and Number of Processors

<table>
<thead>
<tr>
<th>Window Size</th>
<th>Number of Processors Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>25</td>
<td>12</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>35</td>
<td>17</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
</tr>
</tbody>
</table>

Figure 2.7  Window Size Vs. Makespan

Unlike the performance improvement in terms of makespan, Figure 2.8 shows that the average utilization deteriorates as the window size increases for GA, HGA, and MA. This shows that load-balancing becomes much difficult if there are more number of tasks to be balanced. Hence,
improving the processing time by increasing the window size and number of processors comes at a cost of lower processor utilization accordingly.

![Figure 2.8 Window Size Vs. Average Utilization](image)

**Figure 2.8 Window Size Vs. Average Utilization**

**2.5.4 Effect of number of Generations**

The number of generations for the GA, HGA and MA is varied to observe their effects on the performance of the algorithm.

Figure 2.9 shows that the *makespan* is significantly reduced as the number of generations is increased from 5 to 30. This is because the quality of the generated task assignment improves after each generation. However, it is observed that running the GA, HGA, and MA does not seem to improve the performance much after 30 generations.
Figure 2.9 Number of Generations Vs. Makespan

Besides reducing the makespan, the number of generations improves the average processor utilization also as seen in Figure 2.10.

Figure 2.10 Number of Generations Vs. Average Utilization

2.5.5 Effect of Population Size

The experiments are repeated to study the effect of population size over the makespan and average processor utilization. The population sizes
ranging from 5 to 40 are used to test the behavior of the HGA, MA and GA based schemes.

It is observed from Figure 2.11 that increasing the population size does not linearly increase the performance. Choosing an appropriate population size is still very important as a very small value will cause the algorithms to converge quickly, with insufficient processing of few schemata while a very large value may result in longer waiting times.

![Figure 2.11 Population Size Vs. Makespan]

**Figure 2.11 Population Size Vs. Makespan**

From the Figure 2.11, it is seen that HGA and MA reduces the makespan when compared to GA, thus improving the performance.

Despite minimal improvement in terms of makespan, increasing the population size has a positive effect on the processor utilization. It is inferred from Figure 2.12 that the processors are better utilized with a larger population as a larger search space will give a better opportunity to find fitter task mappings.
2.5.6 Effect of Flexibility

The values of H and L multipliers in the threshold policy are used to determine the flexibility of the load-balancing algorithm. Therefore, different H and L values are incorporated in the algorithms to study the effects of these parameters on the performance of the whole algorithms. Table 2.7 shows the percentage of flexibility and the corresponding H and L multipliers.

Table 2.7 H and L Multipliers

<table>
<thead>
<tr>
<th>Flexibility of Load Balancing Mechanism</th>
<th>H</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 %</td>
<td>1.05</td>
<td>0.95</td>
</tr>
<tr>
<td>10 %</td>
<td>1.10</td>
<td>0.90</td>
</tr>
<tr>
<td>20 %</td>
<td>1.20</td>
<td>0.80</td>
</tr>
<tr>
<td>30 %</td>
<td>1.30</td>
<td>0.70</td>
</tr>
<tr>
<td>40 %</td>
<td>1.40</td>
<td>0.60</td>
</tr>
</tbody>
</table>

Figure 2.13 shows that the makespan gets reduced as the flexibility of the load-balancing is increased. This leads to an observation that this load-balancing mechanism works best when the flexibility is 20 percent (H = 1.2 and L = 0.8) for GA, HGA and 10 percent (H = 1.1 and L = 0.9) for MA.
Similarly, Figure 2.14 shows that increasing the flexibility of the algorithm initially improves the processor utilization but has very little effect later. Therefore, it is best to keep the H and L values at 1.2 and 0.8 for GA, HGA and 1.1 and 0.9 for MA respectively for the optimal performance of the algorithms.

---

**Figure 2.13 Flexibility of Load balancing mechanism Vs. Makespan**

**Figure 2.14 Flexibility of Load balancing mechanism Vs. Average Utilization**
2.6 SUMMARY

In this work, the load balancing problem has been implemented using HGA, MA and their performance is evaluated against standard GA. Experimental results showed that the proposed load balancing mechanism developed using hybrid genetic algorithm and memetic algorithm has been very effective, especially in the case of a large number of tasks when compared with GA. The new selection strategy based on simulated annealing in the HGA has shown better results in most of the experiments. The proposed HGA and MA based algorithms have achieved minimum makespan and maximum processor utilization for the load balancing problem. The work can be extended further by employing different problem specific genetic operators and local search heuristics. The load balancing problem can also be approached with other optimization methods also such as particle swarm optimization and ant colony optimization.