Observations on Using Genetic Algorithms for Dynamic Load-Balancing

Albert Y. Zomaya, *Senior Member*, *IEEE*, and Yee-Hwei Teh

Abstract—Load-balancing problems arise in many applications, but, most importantly, they play a special role in the operation of parallel and distributed computing systems. Load-balancing deals with partitioning a program into smaller tasks that can be executed concurrently and mapping each of these tasks to a computational resource such a processor (e.g., in a multiprocessor system) or a computer (e.g., in a computer network). By developing strategies that can map these tasks to processors in a way that balances out the load, the total processing time will be reduced with improved processor utilization. Most of the research on load-balancing focused on static scenarios that, in most of the cases, employ heuristic methods. However, genetic algorithms have gained immense popularity over the last few years as a robust and easily adaptable search technique. The work proposed here investigates how a genetic algorithm can be employed to solve the *dynamic* load-balancing problem. A dynamic load-balancing algorithm is developed whereby optimal or near-optimal task allocations can "evolve" during the operation of the parallel computing system. The algorithm considers other load-balancing issues such as threshold policies, information exchange criteria, and interprocessor communication. The effects of these and other issues on the success of the genetic-based load-balancing algorithm as compared with the first-fit heuristic are outlined.

Index Terms—Genetic algorithms, heuristics, load-balancing, parallel processing, scheduling.

1 Introduction

Load-balancing algorithms are designed essentially to equally spread the load on processors and maximize their utilization while minimizing the total task execution time [1], [18], [20]. In order to achieve these goals, the load-balancing mechanism should be "fair" in distributing the load across the processors. This implies that the difference between the heaviest-loaded and the lightest-loaded processors should be minimized. Therefore, the load information on each processor must be updated constantly so that the load-balancing mechanism can be more effective. Moreover, the execution of the dynamic load-balancing algorithm should not take long to arrive at a decision to make rapid task assignments [18]. In general, load-balancing algorithms can be broadly categorized as centralized or decentralized, dynamic or static, periodic or nonperiodic, and those with thresholds or without thresholds [2], [12], [14].

In a *centralized* load-balancing algorithm, the global load information is collected at a single processor, called the central scheduler. This scheduler will make all the load-balancing decisions based on the information that is 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 locally maintained load information tables 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 a larger system as it imposes fewer overheads on the system than the decentralized (distributed) algorithm. However, a centralized algorithm has 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.

Moreover, for *static* load-balancing problems, all information governing load-balancing decisions is known in advance. Tasks will be allocated during compile time according to a priori knowledge and will not be affected by the state of the system at the time. On the other hand, a *dynamic* load-balancing mechanism has to allocate tasks to the processors dynamically as they arrive. 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 have better performance in comparison to static ones.

2 Key Issues in Dynamic Load-Balancing

Today, *load sharing* and *task migration* are some of the widely researched issues in dynamic load-balancing algorithms [18], [19]. In a situation whereby newly created tasks arrive randomly into the system, processors can become heavily loaded while others are idle or lightly loaded. Therefore, the main objective of load sharing is to develop task assignment algorithms to transfer or migrate tasks from heavily to lightly loaded processors so that no processors are idle while there are other tasks waiting to be processed. In

- *A.Y. Zomaya is with the Parallel Computing Research Laboratory Malaysia, Department of Electrical and Electronic Engineering, University of Western Australia, Western Australia 6907. E-mail: zomaya@ee.uwa.edu.au.*
- *Y.-H. Teh is with Oracle Corporation, Level 38, Menara Citibank, 165 Jalan Ampang, 50450 Kuala Lumpur, Malaysia.*

Manuscript received 11 Jan. 2001; accepted 26 Mar. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 113453.

general, a dynamic load-balancing algorithm consists of four major components: the load measurement rule, the information exchange rule, the initiation rule, and the load-balancing operation [19].

2.1 Load Measurement

Load information is typically quantifiable by a load index set to zero when the processor is not loaded and increases, as the processor load gets heavier. As load measurement occurs frequently, the calculations must be very efficient, which rules out any exhaustive use of too many parameters. Simple parameters, such as the response time or completion time (of all tasks), are sufficient in most cases.

2.2 Information Exchange

This rule specifies how to collect and maintain the workload information necessary for making load-balancing decisions. This is a balancing act between the cost of collecting global load information and maintaining the accurate state of the system. There are three basic information exchange rules. In *periodical* information exchange, individual processors will report their load information to each other periodically at a predetermined time interval, even though this information is not needed at the moment. This exchange rule is good in the sense that the load-balancing operation can be initiated based on the well-maintained workload information without any delay. However, the main problem lies in setting the interval for information exchange. An interval that is too long would mean inaccuracy in decision making while a short time interval may incur heavy communication overhead.

Next, the *on-demand* information exchange in which load information will only be collected just before making a load-balancing decision. This rule is good for minimizing the number of communication messages, but it comes with the cost of extra delay for load-balancing operations. In the case of *on-state* information exchange, a processor will broadcast its load information to the rest in the distributed system whenever its load status changes. While maintaining a global view of the system state, the large overhead in communication makes this rule impractical for large systems. Hence, a centralized rule can be used in which a dedicated processor will collect and maintain the system's load information. It is also noted that, for all the cases mentioned above, new load-balancing decisions can be made based on current workload information as well as feedback from previous decisions.

Finally, the *initiation* rule is used to decide on when to begin load-balancing operations. The invocation decision must weigh its overhead cost against its expected performance benefits (that is, whether the balancing operation will be profitable). Load-balancing operations are initiated using several common policies [8], [13], [17].

2.3 Load-Balancing Operation

The load-balancing process can be defined by three rules: the location rule, the distribution rule, and the selection rule. The *location* rule determines which processors will be involved in the balancing operation. Load-balancing domains can be either global or local. A global domain allows the balancing operation to transfer load from one processor to any of the processors in the system, while a local domain only allows balancing operations to be performed within the set of nearest-neighbor processors.

The *distribution* rule determines how to redistribute the workload among processors in the balancing domain. This rule depends on the balancing domain that is determined by the location rule, while the *selection* rule decides on whether the load-balancing operation can be performed preemptively or nonpreemptively. The former may be transferred to other processors in the middle of execution while, in the latter, tasks can only be transferred if they are newly created.

3 Genetic Algorithms

A Genetic Algorithm (GA) is a search algorithm based on the principles of evolution and natural genetics. GAs 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.

A generation is a collection of artificial creatures (strings). In every new generation, a set of strings is created using information from the previous ones. Occasionally, a new part is tried for good measure. GAs are randomized, but they are not simple random walks. They efficiently exploit historical information to speculate on new search points with expected improvement [5], [9], [11], [15].

The majority of optimization methods move from a single point in the decision space to the next using some transition rule to determine the next point. This point-to-point method is dangerous as it can locate false peaks in multimodal (many-peaked) search spaces. By contrast, GAs work from a database of points simultaneously (a population of strings), climbing many peaks in parallel. The probability of finding a false peak is reduced compared to methods that go point to point.

The mechanics of a simple GA are surprisingly simple, involving nothing more complex than copying strings and swapping partial strings. Simplicity of operation and power of effect are two main attractions of the GA approach. The effectiveness of the GA depends upon an appropriate mix of *exploration* and *exploitation*. Three operators to achieve this are: *selection*, *crossover*, and *mutation*.

Selection according to fitness is the source of exploitation. The mutation and crossover operators are the sources of exploration. In order to explore, they must disrupt some of the strings on which they operate. The trade-off of exploration and exploitation is clearest with mutation. As the mutation rate increases, mutation becomes more disruptive until the exploitative effects of selection are completely overwhelmed. More information is provided on these operators in [5].

4 Proposed Method

A fixed number of tasks, each having a task number and a size, is randomly generated and placed in a task pool from which tasks are assigned to processors. As load-balancing is performed by the centralized GA-based method, the first

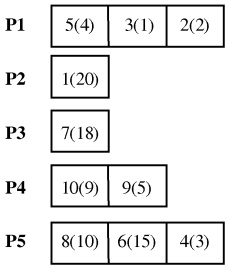


Fig. 1. Two-dimensional representation of a string.

thing to do is to initialize a population of possible solutions [16]. This can be achieved by using the *sliding-window* technique.

4.1 Sliding-Window Technique

Since the objective is to determine task schedules "on the fly" as the tasks arrive, the strings in the population are used to represent the list of tasks known to the system at the time. However, as there may be too many tasks waiting to be assigned at a time, the sliding-window technique is used so that only tasks that are within the window are considered for execution each time [7].

The window size is fixed, with the number of elements in each string equal to the size of the window. Then, permutations of these tasks will form lists that represent the different orders in which these tasks can be scheduled for execution. When the GA 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.

4.2 Encoding Mechanism

Normal binary encoding does not work very well for this problem as the strings may become too long in order to incorporate all the information that is needed. Therefore, the strings are encoded using decimal numbers. Each element in the strings has two decimal values, one to represent the task number and another (in brackets) to represent the size of the task [21].

Two-dimensional strings provide a better representation for the load-balancing problem. One dimension represents the numbers of the processors in the system while the second dimension shows the schedule of tasks on each individual processor (i.e., processor queues) [21]. An example of a two-dimensional string is shown in Fig. 1. The tasks are executed in a left to right order.

As the crossover operator cannot work with two-dimensional strings, these strings must be mapped to one-dimensional strings first. This is basically achieved by connecting the two-dimensional array end-to-end to form one long string, as shown in Fig. 2.

4.3 Objective and Fitness Functions

An objective function is the most important component of any optimization method, including a GA, as it is used to evaluate the quality of the solutions. 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. Then, the objective function is incorporated into the fitness function of the GA. This fitness function will then be used to measure the performance of the strings in relation to the objectives of the algorithm.

4.4 Maxspan

The first objective function for the proposed algorithm is the maxspan of the task schedule [11], represented by the string. Maxspan is basically the largest task completion time among all the processors in the system. For example, based on the string in Fig. 2, the maxspan can easily be calculated as shown in Fig. 3. The numbers in the boxes are indicative of the sizes of the tasks.

Assuming that all the processors are idle when this string is evaluated, then the total time for executing tasks 5, 3, and 2 on processor 1 is $4 + 1 + 2 = 7$ time units. Similarly, task 1 will be executed on processor 2 for a total of 20 time units. The rest of the completion times for processors 3, 4, and 5 are calculated to be 18, 14, and 28 time units, respectively. In this example, the maximum completion time is 28 time units for processor 5. Therefore, the maxspan for this task schedule is 28.

However, as the processors may not always be idle when the string is evaluated, it is inaccurate to calculate maxspan by considering the sizes of these new tasks alone. The current existing load on individual processors must also be taken into account in order to get a more accurate maxspan value (Fig. 4). Therefore, if the current loads on processors 1 to 5 are 3, 4, 17, 13, and 8 time units, respectively, and the same schedule in Fig. 3 is evaluated again, then the total task completion times of each processor can be calculated using the following formula:

$$P_X(\textit{completion time}) = \textit{current load of } P_X + \textit{ new load assigned to } P_X.$$

This leads to the following completion times for the different processors:

$$\mathrm{P}_1 = 3 + 7 = 10, \qquad \mathrm{P}_2 = 4 + 20 = 24,$$
$$\mathrm{P}_3 = 17 + 18 = 35, \qquad \mathrm{P}_4 = 13 + 14 = 27,$$
$$\mathrm{P}_5 = 8 + 28 = 36.$$

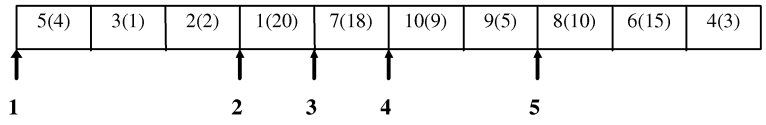


Fig. 2. Mapping a two-dimensional string to a one-dimensional string.

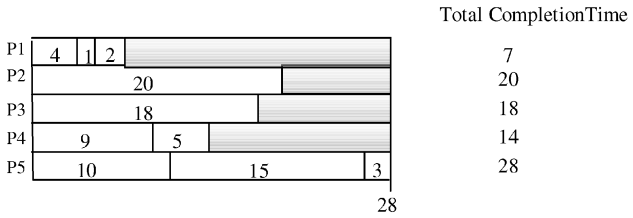


Fig. 3. Maxspan with no current system load.

Hence, the maxspan in this case is 36 time units. The smaller the maxspan, the better the task schedule is as this means that the assigned tasks will be completed in a shorter time. Therefore, minimizing the maxspan optimizes this objective function.

4.5 Average Utilization

The next detrimental 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. By keeping the processors highly utilized, the total execution time should be reduced.

Once again, the task schedule in Fig. 4 is used here to support the arguments. The expected utilization of each processor based on the given task assignment must be calculated. This is achieved by dividing the task completion times of each processor by the maxspan value. The utilization of the individual processors can be given by:

$$P_X(utilization) = P_X(completion\ time)/maxspan$$

This will lead to

$$\begin{aligned} &\mathrm{P}_1 = 10/36 = 0.2777, \qquad \mathrm{P}_2 = 24/36 = 0.6667,\\ &\mathrm{P}_3 = 35/36 = 0.9722, \qquad \mathrm{P}_3 = 27/36 = 0.750,\\ &\mathrm{P}_4 = 36/36 = 1.000. \end{aligned}$$

Then, dividing the sum of all the processors utilization by the total number of processors will give the average processor utilization (APU), which in this case is ≈ 0.73. By optimizing the average processor utilization, the probability of processors being idle for a long time will be reduced.

4.6 Number of Acceptable Processor Queues

The overall task assignment being evaluated may have a small maxspan and high average processor utilization. However, assigning these tasks to the processors may still overload some of the processors. Therefore, the third objective is to optimize the number of acceptable processor queues. Each processor queue is checked individually to see if assigning all the tasks on the processor queues will overload or underload the processors.

Whether a processor queue is acceptable or not is determined by the light and heavy thresholds used. If the task completion time of a processor (by adding the current system load and those contributed by the new tasks) is within the light and heavy thresholds, this processor queue will be acceptable. If it is above the heavy threshold or below the light-threshold, then it is unacceptable (more discussion on threshold policy in the following section).

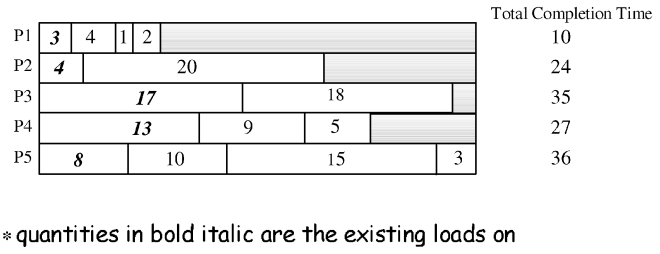


Fig. 4. Maxspan—when current system load is considered.

For example, if the heavy threshold is set at 30 and the light threshold at 22, then, based on the situation in Fig. 4, processor 1 will be underloaded and processors 3 and 5 will be overloaded. Therefore, only processor queues 2 and 4 are acceptable. In order to optimize this objective, the percentage of acceptable processor queues is calculated. This is achieved by dividing the number of acceptable processor queues by the total number of processors in the system. The higher the percentage, the better this schedule is in terms of its load-balancing potential.

4.7 Combined Fitness Function

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

$$fitness = \frac{1}{maxspan} \times average\ utilization \times \frac{\#\ acceptable\ queues}{\#\ processors}.$$

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

4.8 Selection, Crossover, and Mutation

The selection technique used in this paper is based on the roulette wheel method [5]. 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 surviving for 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 (see Table 1).

TABLE 1
Fitness Values for Different Strings

String	Fitness Value	Probability	Slot
1	0.6332	0.20738	0.20738
2	0.3452	0.11305	0.32043
3	0.689	0.22565	0.54608
4	0.986	0.32292	0.86900
5	0.4	0.13100	1.00000
Total	3.0534	1.00000	

After defining the slots, random numbers between zero and one are generated. The numbers obtained will determine which strings 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.

After completing the selection process, the fitter strings should be left in the pool. These strings must then be converted to 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 GA as using this method may cause some tasks to be assigned more than once while some are not assigned at all. Therefore, in order to ensure that each task is only assigned once, the cycle crossover method is adopted in this work [21].

For the sake of completeness, a brief discription of the cycle crossover is given. With this crossover method, the recombination is performed under the constraint that each task comes from one parent or the other in order to show what this means and how a cycle crossover operation works. For example, two parent strings, A and B, are used in this example. At the outset, strings A and B have the following structure:

$$\begin{array}{lcccccccccc} \mathrm{A} = & 8 & 6 & 4 & 10 & 9 & 7 & 1 & 5 & 3 & 2 \\ \mathrm{B} = & 10 & 2 & 3 & 5 & 6 & 9 & 8 & 7 & 4 & 1. \end{array}$$

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

$$\begin{array}{lcccccccccc} \mathrm{A}' = & 8 & - & - & - & - & - & - & - & - & - \\ \mathrm{B}' = & 10 & - & - & - & - & - & - & - & - & -. \end{array}$$

As mentioned earlier, every task has to be taken from one of the two parents. Therefore, with the first choice being task 8 from position 1 of string A, we must now get task 10 from string A because this is the task at the equivalent position in string B.

$$\begin{array}{lcccccccccc} \mathrm{A}' = & 8 & - & - & 10 & - & - & - & - & - & - \\ \mathrm{B}' = & 10 & - & - & 5 & - & - & - & - & - & -. \end{array}$$

This selection in turn requires that we select task 5 from string A next because task 5 is situated at the same position (position 4) in string B. Using a similar selection pattern, this process continues until the following pattern is generated:

$$\begin{array}{lcccccccccc} \mathrm{A}' = & 8 & 6 & - & 10 & 9 & 7 & 1 & 5 & - & 2 \\ \mathrm{B}' = & 10 & 2 & - & 5 & 6 & 9 & 8 & 7 & - & 1 \end{array}$$

This process terminates when task 1 is selected from string A. This is because the selection of task 1 results in having to choose task 8 next from string A. However, this is impossible because task 8 have been selected as the first task earlier. Hence, the process has to terminate at this point. The fact that we eventually return to the original task completes a cycle, thus giving the operator its name. Following the completion of 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, completing the example and performing the complementary cross yields the following children strings in the end:

$$\begin{array}{lcccccccccc} \mathrm{A}' = & 8 & 6 & 3 & 10 & 9 & 7 & 1 & 5 & 4 & 2 \\ \mathrm{B}' = & 10 & 2 & 4 & 5 & 6 & 9 & 8 & 7 & 3 & 1. \end{array}$$

In order to implement this crossover operator, first of all, the strings in the population have to be separated into two groups. Then, one string is chosen randomly from each group. This is to ensure that the two strings chosen randomly are not the same.

The two strings chosen are stored as Parent string 1 (P1) and Parent string 2 (P2). To initiate the crossover operation, a starting point must be selected. This is done by randomly selecting a number between zero and the length of the string (or the size of the window). Once this is accomplished, the crossover operation begins.

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 is the task position in that string. Let $\mathrm{T}_{1\mathrm{S}}$ be the task at position S (i.e., the starting point that was selected) in P1. This task is marked as "completed." Then, the task at the same position in P2 ($\mathrm{T}_{2\mathrm{S}}$) is also marked off as "completed." Then, the task in P1 that matches $\mathrm{T}_{2\mathrm{S}}$ must be located. When this is found to be at position X in P1, this task ($\mathrm{T}_{1\mathrm{X}}$) 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 ($\mathrm{T}_{1\mathrm{S}}$) is reached again. Once this happens, all the tasks that are marked off stay unchanged, while the rest are swapped between strings. After crossover is completed, the strings will be reordered/converted back to its two dimensional form to enable the calculation of their new fitness values.

In this work, swapp mutation is adopted. This mutation technique works by randomly selecting two tasks and then swapping them. The swap mutation technique used in this project randomly selects a processor and then chooses a task randomly on that processor. Similarly, a second processor is randomly selected. However, as the second processor may be the same as the first, there is a possible chance that the second task selected is 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.

4.9 Evaluation of Strings

After completing the mutation process, the population of strings will be evaluated using the fitness function. Each individual string will have a new fitness value and a new probability of surviving into the next generation. These values will then be used to define the slots of the roulette wheel in the next GA cycle. The total and average fitness values of the strings in the pool are calculated after every cycle to see if the new pool of strings is fitter than the ones in the old pool.

Instead of waiting for the GA to converge, it will be allowed to run for a fixed number of k cycles ($k = 10$ in this

paper). The decision was made because solutions generated in less than k generations may not be good enough. On the other hand, running the GA for more than k generations may not be very feasible, as too much time will be devoted to genetic operations. When the GA is terminated after k cycles, the fittest string in the pool will be decoded and used as the task schedule.

As the GA is designed for the purpose of load-balancing, it is important that the task assignments do not result in any idle processors. Hence, if the fittest assignment determined by the GA results in any processors being idle, the assignment will be rejected. The GA will run for another 10 cycles again to find another task assignment that utilizes all processors.

5 Initiation of the Load-Balancing Mechanism

The load-balancing mechanism involves more than developing optimal task mappings. Other issues, such as the initiation rule, information exchange rule, and threshold policy, also play important roles in the development of this load-balancing mechanism. However, as the performance of this mechanism relies mainly on the GA for good task mappings, we will consider the initiation of the GA as the initiation of the balancing mechanism.

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 paper only schedules tasks without task migration, running the load-balancing too often may overload the processors. The central scheduler may keep assigning more tasks to the processors when 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 this work, the load-balancing mechanism is initiated whenever the central scheduler detects that a processor has finished processing all the tasks in its queue. This is more feasible than waiting for all the processors to finish executing all their tasks before the GA operation is repeated again. With this initiation rule, the processors will unlikely be idle because, every time a processor executes all its tasks, another set of tasks will be allocated to it. It is also unlikely that the processors will become overloaded because a task allocation constraint is used as well (more in Section 6).

5.1 Load Information Update

In this paper, 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. However, it is not necessary for the local processors to constantly send messages to the central scheduler after every time unit as this will result in high communication costs. Instead, information will only be collected from all the processors whenever a set of tasks is sent to the destination processors based on the task mapping determined by the GA. This will update the global load information of the system.

Once a set of tasks is assigned, 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 the completion of tasks by any of the processors. If the tasks are still executing, 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 processed by each processor. This process will continue until the scheduler detects the completion of a task at the processors.

5.2 Threshold Policy

Fixed (or static) threshold values do not work very well with this algorithm since the state of the system is changing all the time. For example, if only 15 tasks are left in the system, the fixed threshold value determined earlier for a system with 100 tasks will now be too large for the load-balancing mechanism to be effective. Hence, an adaptive threshold policy [4], [21] is used so that the thresholds are adjusted as the global system load changes.

5.3 Average Load

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 GA). In order to determine proper thresholds, the average load must be determined first. This is achieved by summing the current load in the system (at the time when the load-balancing function is called) and the new load contributed by the new set of tasks within the sliding-window and then dividing the total by the number of processors in the system:

$$L_{ave} = \frac{\sum_{i=1}^{N}(CTT_i + QT_i)\sum_{k=1}^{M} TT_k}{N},$$

where

- L_{ave} = 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 queue i,
- TT_k = execution time of individual tasks within the sliding-window,
- N = number of processors in the system, and
- M = number of tasks in the sliding-window.

If allocating the set of new tasks results in all processor loads being the same as 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 slightly relax the requirements for load-balancing. Therefore, heavy and light thresholds are used to add more flexibility to the assignment process.

5.4 Heavy and Light Thresholds

Threshold is a value that is used to indicate whether a processor is heavily or lightly loaded. In a system without a threshold policy, the arrival or completion of tasks for every processor will be reported directly to the central scheduler. There will be no indication of the load status of individual processors.

However, a system with threshold policy will usually have two workload thresholds. Processors with loads that are less than the light threshold are referred to as light-loaded processors, whereas those with loads higher than the heavy threshold are categorized as heavy-loaded processors [12].

Finally, those with loads between the heavy and light thresholds are normal-loaded processors. Each individual processor has a load status and information will only be sent to the central scheduler when there is a change in the load status instead of when a task arrives or finishes. Having a threshold policy in the load-balancing algorithm will reduce the traffic overhead significantly.

It is important to determine appropriate thresholds for a good load-balancing algorithm. If the threshold is set 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. The new thresholds will then be broadcast to the rest of the processors via the central scheduler. Based on the average load value calculated earlier, the heavy and light thresholds are derived as follows:

$$T_H = H \times L_{ave},$$
$$T_L = L \times L_{ave},$$

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. Further, H is greater than one and it determines the amount that the average workload can be exceeded by before the processor becomes heavily loaded. Conversely, L is less than one and it shows how much the processor loads can be short of 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 are set to 1.2 and 0.8, respectively. This means that processor loads that are 20 percent above or below the average value will be considered acceptable.

5.5 Task Allocation

For efficiency, tasks are not allocated one at a time. Instead, all the tasks on a processor queue are allocated in one shot if the task allocation criteria are met. It is important to note that, even though the fittest task mapping shows good load-balancing and processor utilization, it is not always feasible to assign tasks to all processors queues. Therefore, a constraint had been placed on the assignment so that only those processor queues that will not overload the processors will be allocated. Then, the list of tasks assigned in this round must be recorded for reference later.

5.6 Updating the Window

5.6.1 Complete Window

After all the appropriate tasks have been assigned, the window has to be filled up again by sliding along the subsequent tasks waiting in the task queue and using these to replace the assigned tasks. In order to do so, first, the scheduler will have to search for tasks in the window that match the ones that have been assigned. Then, each matched task will be removed from the window and replaced with a new task taken from the system task queue. With the window full again, genetic operations can be performed on the new set of tasks when the load-balancing function is invoked again

5.6.2 Incomplete Window

In the process described above, it was assumed that there would always be tasks waiting at the system task queue. However, as the simulator only generates a finite number of tasks, there will come a moment in time when there are not enough tasks to fill up the window. When this occurs, another rule has to be used to assign these tasks since it is no longer feasible to run the GA on a small number of tasks. To finish allocating the remaining tasks in the system, the central scheduler will search for the processor with the lightest load. When this is found, the first of the remaining tasks will be assigned to it and the load on this processor will be updated. After this is completed, the next task can be assigned by repeating the same process again. This will continue until all the tasks in the system are assigned.

6 A Simple Example

An example is shown below to illustrate how the proposed load-balancing mechanism works. All the steps, starting from the time the load-balancing mechanism is initiated until the assignment of tasks, are discussed in detail.

Step 1: Current State of the System. Table 2 holds the information at time t. Since processor 3 is idle, a new set of tasks has to be allocated.

Step 2: New Tasks to Be Scheduled. The new set of tasks within the sliding-window is shown in Table 3.

$$\text{Total time units} = 9 + 8 + 7 + 9 + 17 + 18 + 4 + 11 + 17 + 5 = 105.$$

Step 3: Fittest Task Mapping Generated by the GA. Based on the set of tasks in Step 2, the fittest task mapping generated using the GA is as follows:
Generated task mapping:

TABLE 2
State of the System at the Outset

PROC #	CURRENT TASK	PROCESSOR QUEUE	TOTAL QUEUE LENGTH (TIME UNITS)
1	6(1)	5(2) 3(4)	7
2	8(6)		6
3	7(0)		0
4	1(8)	9(5)	13
5	4(8)		8
Total load in the system			34

TABLE 3
New Tasks Inserted into the Sliding Window

11(9)	12(8)	13(7)	14(9)	15(17)	16(18)	17(4)	18(11)	19(17)	20(5)

$$\begin{aligned}
&\mathrm{P1}: 15(17)\ \ 20(5)\\
&\mathrm{P2}: 18(11)\ \ 11(9)\\
&\mathrm{P3}: 19(17)\ \ 13(7)\\
&\mathrm{P4}: 12(8)\ \ \ 17(4)\\
&\mathrm{P5}: 14(9)\ \ \ 16(18).
\end{aligned}$$

Load (time units) added to processors:

$$\begin{aligned}
&\mathrm{P1}: 22\\
&\mathrm{P2}: 20\\
&\mathrm{P3}: 24\\
&\mathrm{P4}: 12\\
&\mathrm{P5}: 27.
\end{aligned}$$

Step 4: Determine the New Thresholds. The next step is to determine new thresholds based on the current system load (given in step 1) and new tasks to be scheduled (Step 2):

$$\begin{aligned}
L_{ave} &= \frac{(7+6+0+13+6)+105}{5}\\
&= 27.4\\
T_H &= 1.2 \times 27.4\\
&\approx 32.88\\
T_L &= 0.8 \times 27.4\\
&\approx 21.92.
\end{aligned}$$

Hence, the new heavy threshold is 33 while the light threshold is 22.

Step 5: Find Acceptable Load Sizes for Each Processor. The acceptable load that each processor can accept in this round can be calculated by subtracting the current processor queues (Step 1) from the heavy threshold determined in Step 4. For example, for processor 1:

$$\begin{aligned}
\mathit{Acceptable\ load} &\leq \mathit{heavy\ threshold}\\
&\quad - \mathit{total\ queue\ length\ of\ processor}\ 1\\
&\leq 33 - 7\\
&\leq 26.
\end{aligned}$$

Similarly, processors 2-5 can receive processor queues that do not exceed the acceptable loads of 27, 33, 20, and 25, respectively.

Step 6: Task Assignment Decision. Now that we have the acceptable loads for each processor, we can go back to the task assignment generated in Step 3 and see if the individual total processor queues will exceed these values. As the heavy load multiplier used is 1.2, this means that the decision already has a 20 percent flexibility from the ideal load-balancing scenario. Therefore, if processor queues that exceed the acceptable load sizes are still assigned, this will cause a more severe load imbalance situation.

Hence, by inspecting the task assignment in Step 3, we can see that the new set of tasks for processor 1 can be assigned since its load is less than the acceptable load (i.e., 2,226). The same applies to the tasks on processors 2-4. However, the new tasks for processor 5 will not be assigned since its load of 27 exceeds the acceptable load of 25. Therefore, only tasks for processors 1-4 in the generated task mapping will be assigned to the respective processors.

Step 7. New State of the System. After the assignment of tasks, the load information table at time $t+1$ is as shown in Table 4. From the above table, processor 1 is detected to be idle. Therefore, another list of tasks have to be assigned. In order to do this, the sliding-window must be filled up with new tasks again.

Step 8: Updating the Sliding-Window. Previously (in Step 2), the sliding-window consisted of the tasks in Table 5.

However, as tasks 11, 12, 13, 15, 17, 18, 19, and 20 were assigned to the processors in step 7, these tasks in the sliding-window have to be replaced with eight new tasks from the system task queue. Hence, the new sliding-window will have the tasks shown in Table 6.

$$\begin{aligned}
\text{Total time units} &= 4+7+8+9+10+18+2+15+12+9\\
&= 94.
\end{aligned}$$

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

TABLE 4
Load Information at Time $t + 1$

PROC #	CURRENT TASK	PROCESSOR QUEUE	TOTAL QUEUE LENGTH (TIME UNITS)
1	6(0)	5(2) 3(4) 15(17) 20(5)	28
2	8(5)	18(11) 11(9)	25
3	19(17)	13(7)	24
4	1(7)	9(5) 12(8) 17(4)	24
5	4(7)		7
Total load in the system			108

TABLE 5
Contents of Old Sliding Window

11(9)	12(8)	13(7)	14(9)	15(17)	16(18)	17(4)	18(11)	19(17)	20(5)

TABLE 6
Contents of New Sliding Window

21(4)	22(7)	23(8)	14(9)	24(10)	16(18)	25(2)	26(15)	27(12)	28(9)

7 Results

7.1 Test Parameters

The performance of the proposed method was compared to the First Fit (FF) algorithm [10] and a random allocation scheme [21]. However, as random allocation does not provide consistent results that can be used for meaningful analysis, it is more important to focus on the results for the GA and the FF algorithm in most cases.

Initially, the test runs were based on a set of default values: number of tasks $= 100$, number of processors $= 5$, window size $= 10$, number of generation cycles $= 5$, population size $= 10$, maximum size of each task $= 20$, H multiplier $= 1.2$, and L multiplier $= 0.8$.

The measurement of performance of these algorithms was based on two metrics: total completion time and average processor utilization. The default parameters were varied and the results collected from test runs were used to study the effects of changing these parameters.

Only one parameter was changed each time so that any changes in performance would be based solely on this parameter. However, in some instances (trivial cases), more than one parameter was varied in order for the GA to work. For example, consider the situation when there are 20 tasks to be allocated and there are 20 processors in the system. The GA, or any other algorithm for that matter, will not work very well in this case since it is hard to find a task mapping that can equally balance out the load of 20 tasks over 20 processors (and have exactly one task assigned to each processor). Therefore, the examples chosen must lead to results that are meaningful and valid for analysis.

In some cases, the performance of the GA was not compared to that of the FF algorithm and the random allocation as some of the parameters were only relevant to the GA. For example, changing the window size, population size, number of generation cycles, and the H and L constants will not have any effect on the FF algorithm and random allocation. These tests were only performed to observe the effects on the GA. Six tests were produced for each set of parameters to allow a rough average and standard deviation to be obtained.

7.2 Changing the Number of Tasks

Default values were used for all the parameters except for the number of tasks to be assigned. The number of tasks was varied from 20-1000 and the effects on the total completion time and average processor utilization are given below.

7.2.1 Completion Time

Fig. 5 shows that the total time taken for all three algorithms increased linearly as the number of tasks was increased. This is expected as the more tasks to be scheduled, the longer it takes to complete all the tasks. It was also noted that the GA performed better than the random allocation and the FF algorithm in all cases.

When comparing the results of the GA and the FF algorithm, one can observe that the gap between these two curves was widening as the number of tasks was increased. This shows that the GA actually reduced the total completion time by a considerable amount (greater speedup) in comparison to the FF algorithm as the number of tasks increased. This also indicates reliable performance of the GA when the number of tasks increases. This is very valuable from a practical point of view.

7.2.2 Average Processor Utilization

In most cases, the GA out performed the FF algorithm in terms of processor utilization. The processor utilization using the GA was in the range of 85-99 percent while the FF algorithm only had a utilization of 80-91 percent. It was also noticed that the processor utilization of the GA actually approached 99 percent as the number of tasks was increased, thus indicating that a GA works better in the case of more tasks. This is quite consistent with other results in the literature [21] (see Fig. 6).

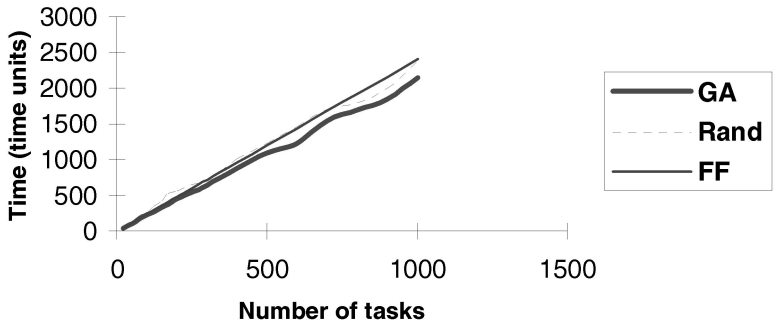


Fig. 5. Total completion times.

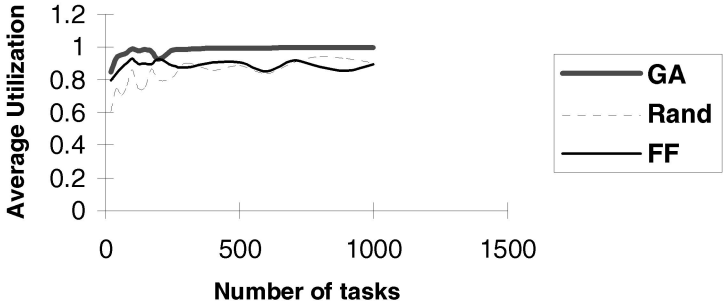


Fig. 6. Average processor utilization.

7.3 Varying the Number of Processors as Window Size Increases

In this section, the size of the window and the number of processors were varied to observe their effect on the GA's performance. It is important to note that the number of processors was changed according to the size of the window because an inappropriate number of processors used may prevent the GA from working properly. 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.

It was mentioned earlier that if the fittest task assignment results in any processors being idle, it will be rejected and the GA will be initiated again to generate another assignment. Therefore, when there is a situation whereby 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 GA will be reexecuted over and over again, but a task mapping that works for this load-balancing mechanism will never be generated. Hence, it is crucial that the window size correlates with the number of processors in the system.

Taking into account all the issues mentioned above, the number of processors and window sizes used for the test runs are shown in Table 7. The effects of changing the window size (and the number of processors) from 10 to 50 are shown in Fig. 7 and Fig. 8. The results of the FF algorithm were not shown in these figures as changing the window sizes does not affect this algorithm.

7.3.1 Completion Time

Fig. 7 shows that the completion time improved when the window size was increased. However, this improvement was mainly caused by the fact that more processors were used for larger windows. Therefore, even though there were more tasks to be scheduled in each round, the extra load was easily handled by the additional processors.

7.3.2 Average Processor Utilization

Unlike the performance improvement in terms of total completion time, Fig. 8 shows that the average processor utilization deteriorated as the size of the window increased. This shows that load-balancing is harder when more tasks are to be balanced out across a larger system. Hence, improving the processing time by increasing the window size and number of processors accordingly comes at a cost of lower processor utilization.

7.4 Number of Generation Cycles

The number of generation cycles for the GA was changed to observe the effects of this parameter on the performance of the algorithm.

7.4.1 Completion Time

Fig. 9 shows that the total completion time was significantly reduced as the number of generations was increased from 5

TABLE 7
Window Sizes and Number of Processors

WINDOW SIZE	NUMBER OF PROCESSORS USED
10	5
15	7
20	10
25	12
30	15
35	17
40	20
50	25

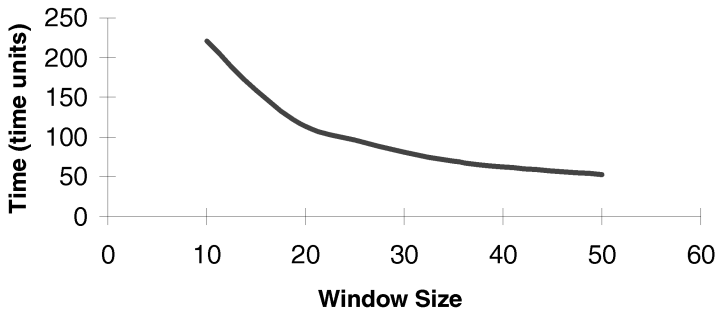


Fig. 7. Total completion time.

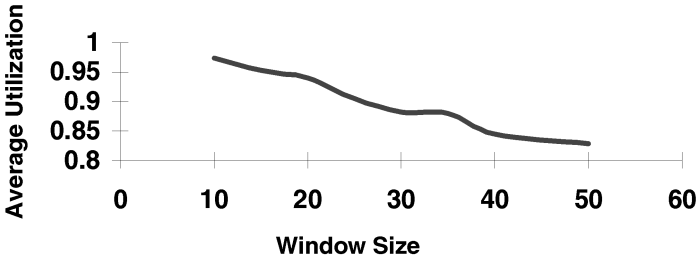


Fig. 8. Average processor utilization.

to 20. This is due to the fact that the quality of the generated task assignment improves after each generation. However, after 20 generations, it was observed that running the GA does not seem to improve performance much. The results suggest that running the GA for more generations will be superfluous as the improvement will only be marginal.

7.4.2 Average Processor Utilization

Besides reducing the total completion time, the average processor utilization also improved as the GA was executed for more generations (Fig. 10). However, once again, the improvement achieved slowed down as the solution converged. An important point worth noting is that every test run (for a different number of generations) was based on different sets of tasks. Running for more generations does not mean that the same population of solutions was being operated on further. Therefore, the results may be slightly different if the same set of solutions was being optimized.

7.5 Changing the Population Size

Different population sizes were used to generate the results. The population sizes ranged from 5 to 40.

7.5.1 Completion Time

It is observed that increasing the population size does not increase the performance by much. Even though an improvement is shown in Fig. 11, the improvement is only minimal. Despite that, choosing an appropriate population size is still very important as one that is too small may cause the GA to converge too quickly, with insufficient processing of too few schemata, while one that is too large results in long waiting times for significant improvement [6]. Note that the results shown in Fig. 11 were only based on the load of the tasks and not the actual processing time. Therefore, if the population pool was larger, the computations involved should be more intensive, hence resulting in high processing costs. So, using a population size that is too large will not be very feasible [3].

7.5.2 Average Processor Utilization

Despite minimal improvement in terms of total completion time, increasing the population size had a positive effect on the processor utilization. The processors were better utilized since a larger search space will give the GA a more opportunity to find better and fitter task mappings (see Fig. 12).

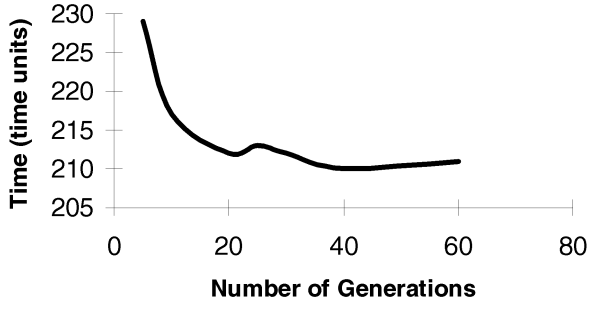


Fig. 9. Total completion time.

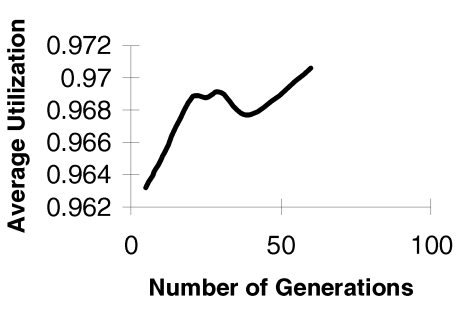


Fig. 10. Average processor utilization.

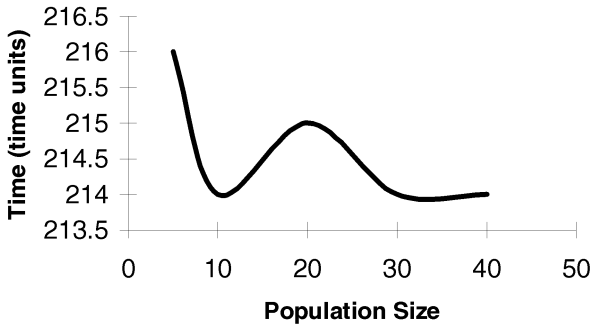


Fig. 11. Total completion time.

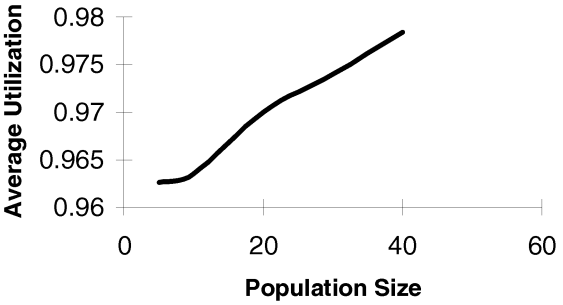


Fig. 12. Average processor utilization.

7.6 Changing the H and L Multipliers

The values of the H and L multipliers used to implement the threshold policy determine the flexibility of the load-balancing algorithm. Therefore, different H and L values were incorporated in the algorithm to see the effects of these parameters on the performance of the whole algorithm (Table 8).

7.6.1 Completion Time

Fig. 13 shows that the completion time was reduced as the flexibility of the load-balancing was increased. However, the performance deteriorated after the 20 percent mark. Therefore, having a load-balancing algorithm that is too restrictive is inefficient as it is unrealistic to keep the load equally balanced.

On the other hand, if the algorithm is too flexible, it becomes ineffective as overloaded processors will not be detected until it is too late. Fig. 13 leads to an observation that this load-balancing mechanism worked best when the flexibility was 20 percent (H = 1.2 and L = 0.8).

TABLE 8
H and L Multipliers

Flexibility of load-balancing algorithm	H	L
5%	1.05	0.95
10%	1.10	0.90
20%	1.20	0.80
30%	1.30	0.70
40%	1.40	0.60

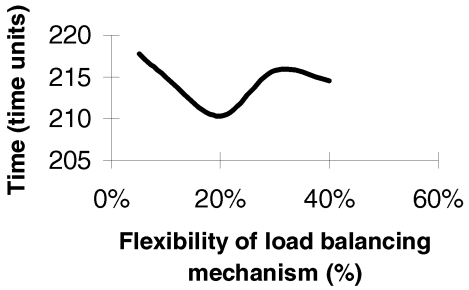


Fig. 13. Total completion time for different H and L multipliers.

7.6.2 Average Processor Utilization

Similarly, to performance in terms of completion time, increasing the flexibility of the algorithm also improved the processor utilization significantly in the beginning. But, it reached its peak performance at the 20 percent point and a drop in average processor utilization was observed thereafter. This is due to the same reason explained earlier. Therefore, it is best to keep the H and L values at 1.2 and 0.8, respectively, for the optimal performance of the algorithm (Fig. 14).

This section summarized the performance of the proposed dynamic load-balancing mechanism using GA as compared to the FF heuristic and the random allocation scheme. Each of the test runs used total completion time and average processor utilization as performance measures.

The performance of the GA and the FF algorithm was compared in many cases by changing the parameters of the algorithm. The parameters varied (one at a time) were the number of tasks to be executed, the number of processors used, the window size, the number of generation cycles, the population size, and the H and L multipliers. It was observed that the GA performed better than the FF algorithm in most cases when the number of tasks and number of processors were changed in separate occasions.

In summary, the above shows that GAs have great potential in handling the dynamic load-balancing problem. The rest of the parameters were changed because identifying an optimal set of parameters for the GA may bring possible improvement for the overall dynamic load-balancing mechanism.

8 Conclusions

The proposed dynamic load-balancing mechanism developed using genetic algorithms has been very effective,

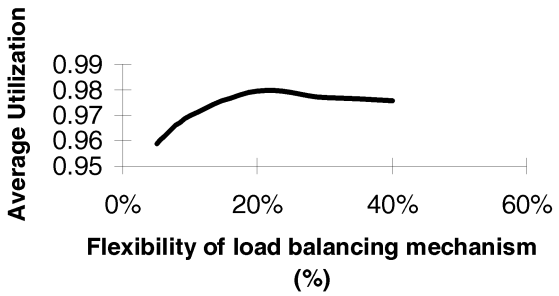


Fig. 14. Average processor utilization.

especially in the case of a large number of tasks. In fact, a GA-based scheme works better when the number of tasks is large and where we observe consistent performance while other heuristics fail.

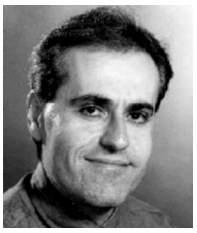
The use of a central scheduler was also effective as it can handle all load-balancing decisions with minimal interprocessor communication. The threshold policy used also provided better performance in comparison to the first fit algorithm that does not have such a mechanism. Thus, the GA-based algorithm worked rather well in terms of achieving the goals of minimum total completion time and maximum processor utilization [22].

ACKNOWLEDGMENTS

The authors would like to thank the three anonymous referees for the suggested corrections that improved the manuscript. Professor Zomaya's work was supported by Australian Research Council large grant A00000902.

REFERENCES

[1] S.H. Bokhari, "On the Mapping Problem," *IEEE Trans. Computers,* vol. 30, no. 3, pp. 550-557, Mar. 1981.
[2] F. Bonomi and A. Kumar, "Adaptive Optimal Load-Balancing in a Heterogeneous Multiserver System with a Central Job Scheduler," *IEEE Trans. Computers,* vol. 39, no. 10, pp. 1232-1250, Oct. 1990.
[3] A. Chipperfield and P. Fleming, "Parallel Genetic Algorithms," *Parallel and Distributed Computing Handbook,* first ed., pp. 1118-1193. A. Zomaya, ed., New York: McGraw-Hill, 1996.
[4] M. Deriche, M.K. Huang, and Q.T. Tsai, "Dynamic Load-Balancing in Distributed Heterogeneous Systems under Stationary and Bursty Traffics," *Proc. 32nd Midwest Symp. Circuits and Systems,* vol. 1, pp. 669-672, 1990.
[5] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, Mass.: Addison-Wesley, 1989.
[6] D.E. Goldberg, "Sizing Populations for Serial and Parallel Genetic Algorithms," *Proc. Third Int'l Conf. Genetic Algorithms,* pp. 70-79, 1989.
[7] C.A. Gonzalez Pico and R.L. Wainwright, "Dynamic Scheduling of Computer Tasks Using Genetic Algorithms," *Proc. First IEEE Conf. Evolutionary Computation, IEEE World Congress Computational Intelligence,* vol. 2, pp. 829-833, 1994.
[8] A. Hac and X. Jin, "Dynamic Load-Balancing in a Distributed System Using a Sender-Initiated Algorithm," *Proc. 13th Conf. Local Computer Networks,* pp. 172-180, 1988.
[9] J.H. Holland, *Adaptation in Natural and Artificial Systems.* Univ. of Michigan Press, 1975.
[10] T.C. Hu, *Combinatorial Algorithms.* Addison-Wesley, 1982.
[11] M.D. Kidwell and D.J. Cook, "Genetic Algorithm for Dynamic Task Scheduling," *Proc. IEEE 13th Ann. Int'l Phoenix Conf. Computers and Comm.,* pp. 61-67, 1994.
[12] Y. Lan and T. Yu, "A Dynamic Central Scheduler Load-Balancing Mechanism," *Proc. IEEE 14th Ann. Int'l Phoenix Conf. Computers and Comm.,* pp. 734-740, 1995.
[13] S. Lee, T. Kang, M. Ko, G. Chung, J. Gil, and C. Hwang, "A Genetic Algorithm Method for Sender-Based Dynamic Load-Balancing Algorithm in Distributed Systems," *Proc. 1997 First Int'l Conf. Knowledge-Based Intelligent Electronic Systems,* vol. 1, pp. 302-307, 1997.
[14] H.C. Lin and C.S. Raghavendra, "A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC)," *IEEE Trans. Software Eng.,* vol. 18, no. 2, pp. 148-158, Feb. 1992.
[15] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs,* second ed. Berlin: Springer-Verlag, 1994.
[16] M. Munetomo, Y. Takai, and Y. Sato, "A Genetic Approach to Dynamic Load-Balancing in a Distributed Computing System," *Proc. First Int'l Conf. Evolutionary Computation, IEEE World Congress Computational Intelligence,* vol. 1, pp. 418-421, 1994.
[17] L.M. Ni, C.W. Xu, and T.B. Gendreau, "A Distributed Drafting Algorithm for Load-Balancing," *IEEE Trans. Software Eng.,* vol. 11, no. 10, pp. 1152-1161, Oct. 1985.
[18] S. Salleh and A.Y. Zomaya, *Scheduling in Parallel Computing Systems: Fuzzy and Annealing Techniques.* Kluwer Academic, 1999.
[19] C. Xu and F. Lau, *Load-Balancing in Parallel Computers—Theory and Practice.* Kluwer Academic, 1997.
[20] A.Y. Zomaya, "Parallel and Distributed Computing: The Scene, the Props, the Players," *Parallel and Distributed Computing Handbook,* A.Y. Zomaya, ed., pp. 5-23. New York: McGraw-Hill, 1996.
[21] A.Y. Zomaya, C. Ward, and B. Macey, "Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 8, pp. 795-812, Aug. 1999.
[22] A.Y. Zomaya, F. Ercal, and S. Olariu, *Solutions to Parallel and Distributed Computing Problems: Lessons from Biological Sciences.* New York: Wiley, 2001.



Albert Y. Zomaya received the PhD degree from the Department of Automatic Control and Systems Engineering, Sheffield University, United Kingdom. He is a full professor in the Department of Electrical and Electronic Engineering at the University of Western Australia, where he also leads the Parallel Computing Research Laboratory. Also, he held visiting positions at Waterloo University and the University of Missouri-Rolla. He is the author/coauthor of five books, more than 120 publications in technical journals, collaborative books, and conferences, and the editor of three volumes and three conference proceedings. He is currently an associate editor for the *IEEE Transactions on Parallel and Distributed Systems*, *Journal of Interconnection Networks*, *International Journal on Parallel and Distributed Systems and Networks*, *Journal of Future Generation of Computer Systems*, and *International Journal of Foundations of Computer Science*. He is also the founding editor of the Wiley Book Series on Parallel and Distributed Computing. He is the editor-in-chief of the *Parallel and Distributed Computing Handbook* (McGraw-Hill, 1996). Professor Zomaya is the chair the IEEE Technical Committee on Parallel Processing (1999-current). He is also a board member of the International Federation of Automatic Control (IFAC) Committee on Algorithms and Architectures for Real-Time Control and serves on the executive board of the IEEE Task Force on Cluster Computing. He has been actively involved in the organization of national and international conferences. Professor Zomaya is a chartered engineer and a senior member of the IEEE. He is a member of the ACM, the Institute of Electrical Engineers (United Kingdom), the New York Academy of Sciences, the American Association for the Advancement of Science, and Sigma Xi. He received the 1997 Edgeworth David Medal from the Royal Society of New South Wales for outstanding contributions to Australian science. His research interests are in the areas of high performance computing, parallel and distributed algorithms, scheduling and load-balancing, scientific computing, adaptive systems, mobile computing, data mining, and cluster-, network-, and meta-computing.

Yee-Hwei Teh received the BEng degree in information technology and the BC degree from the University of Western Australia (UWA) in 1998. While completing her degree in Engineering, she developed a keen interest in parallel and distributed computing. This led to her decision to join the Parallel Computing Research Lab with the Department of Electrical and Electronic Engineering at UWA in her final year, where she completed her honors dissertation on using genetic algorithms for dynamic load balancing in a parallel processing environment. Upon graduation, she worked as a software engineer at PDX Infoworld (Kuala Lumpur), where she was involved in the design of a financial application server which serves as a platform for the development of electronic financial services, such as bill presentment and payment. She is currently working as a technical consultant at Oracle Corporation Malaysia, implementing data warehouse and financial applications for the banking industry. She has a strong interest in Java and Internet technology and wishes to use her knowledge in parallel computing to write parallel and distributed programs that will enhance the performance of the software applications that she builds in the future.