Application centric load balancing for distributed systems using genetic algorithm scheduling

Sheng Bai
University of Windsor

Application Centric Load Balancing for Distributed Systems Using Genetic Algorithm Scheduling

By

Sheng Bai

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
The Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2008

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-42253-3
Our file Notre référence
ISBN: 978-0-494-42253-3

Abstract

This thesis proposes a GA based scheduling algorithm for a heterogeneous distributed computing environment. It uses the application centric load balancing system. The proposed system removes all those network delay assumptions and considers the allocation problem for both computing resources and network resources. It addresses the generalized workload i.e. direct acyclic graph (DAG) kind of workload that is composed of sub jobs with internal dependencies.

Rather than allocate applications simply according to their arrival time, we introduce GA scheduling strategy into our load balancing system to find the proper applications allocating schedule, which uses the available resources more efficiently. With introduction of GA scheduling into both application level and process level, certain improvements on the practicability, accuracy and performance are expected.

Instead of using constant GA parameters, our proposed algorithm dynamically adjusts the key parameters, such as crossover rate and mutation rate, adapting them to the quality of generations. Later, we will implement more new ideas, such as gender assignment, fertility rate and ageing into our GA algorithm to achieve better performance.

Keywords:

Load balancing, network delay, workload simulator, Adaptive Genetic Algorithm scheduling etc.

Dedication

To

My grandma

Acknowledgements

I wish to express my deep sense of gratitude to Dr. Akshai Aggarwal, my advisor, for his inspiring ideas, persistent guidance, and valuable assistance through the course of this thesis. His encouragement has assured the quality of the work.

I would like to express my appreciation to Dr. Fritz Rieger, Dr. Robert Kent, and Dr. Jessica Chen for being on my committee and offering me valuable advice and comments concerning this thesis.

I am also grateful to all the people who have given me useful suggestions and unrelenting or continual support during the research.

Table of Content

List of Tables

List of Figures

1. Introduction

Since Grid Computing came into computer world's view, the resource allocation and scheduling algorithms, which are used to allocate available resources for processing applications, have become an important area of research. A grid [9, 10] seamlessly integrates geographically spread out compute and data resources and sensors. It provides resources on demand to users. In order to satisfy the QoS requirements of applications on the grid, scheduling algorithms are introduced to ensure optimum utilization of available resources and to provide the required service to users.

The general motivation of a scheduling algorithm is to allocate the workload to each compute node according to its processing capacity. From the application perspective, it minimizes the processing time for a single job or for a set of jobs. From the resource perspective, it will try to keep all the nodes busy during a given session. In the optimum circumstance, the allocated jobs on each node should be done almost at the same time.

Most of the existing scheduling algorithms build on certain assumptions either related to workload or the distributed computing environment. Some load balancing algorithms consider all the compute-nodes in the distributed environment as homogeneous with the same processing capacity. In other algorithms, the distributed environment is idealized as a network composed of network segments with the same data transfer rate. There are certain algorithms which do consider the diversity of both the communication network segments and the compute-nodes. But these algorithms can only process divisible workload, which can be split into several child jobs, without considering any dependency.

In this thesis, we focus our research on a scheduling algorithm that deals with generalized workload i.e. direct acyclic graph (DAG) kind of workload composed of child jobs with dependencies. Following the rule in [7], we denote a job as an application and child jobs as processes of the application. A typical DAG type of application is shown in Figure1 [11-page9].

From the network respective, we are considering the workload allocation problem of compute-nodes, spread over a wide area network, with bus topology. Even though, the Internet has acquired a very complex topology, the research grids of today seems to follow a bus structure. We remove the constant data rate assumption, and consider the network as heterogeneous.

Rather than allocate applications simply according to their arrival time, we introduce GA scheduling strategy into our load balancing system to find the proper applications allocating schedule, which uses the available resources more efficiently. Instead of using constant GA parameters, our proposed algorithm dynamically adjusts the key parameters, such as crossover rate and mutation rate, adapting them to the quality of generations. Later, we will implement more new ideas, such as gender assignment, fertility rate and ageing into our GA algorithm to achieve better performance.

The objective of this thesis is to propose a general scheduling algorithm using the application centric load balancer that works on workload with dependencies. It will consider the allocation problem both on the compute resource and network. The final aim is to improve the system throughput in a given session.

The rest of the thesis is organized as follows. In chapter 2, we describe the existing grid scheduling algorithms. Chapter 3 explains our adaptive algorithm based on GA. In Chapter 4, we present the experiments and validations. The last chapter presents the conclusions from our study and some ideas for future work

2. Literature Review

Many of the grids scheduling algorithms are based on the large body of research in the area of load balancing for clusters of computers. The load balancing problems in a distributed system can be broadly divided into two areas: Application centric load balancing and communication network load balancing. Application centric load balancing focuses on allocating the workload to the compute resources in the system, while communication load balancing works on reducing network delays introduced by data transfers from the parent processes to the child processes. For getting near-optimum performances of a grid scheduler, we have to consider both the load balancing systems.

2.1 Application centric load balancing

The objective of an application centric load balancer is to run as many applications in a given session as possible. Generally, research in this area can be classified in several ways. The common strategies include mathematical programming [23, 18], and heuristic methods [24, 21]. There are also some researches focusing on artificial intelligence [19] or fuzzy logic [12, 13].

The advantage of using mathematical or dynamic programming methods for load balancing strategies is that the process is fast and it may lead to a truly optimum solution. However, in a heterogeneous distributed environment, it is not possible for these algorithms to provide a closed form solution [14]. In order to solve these problems, many heuristic based algorithms have been developed. But most of them have been design for divisible workload systems without considering any dependencies. Thus Beaumont et.al [3] considers each application to be composed of a set of independent, same-size tasks. Similarly while Tian and Chandy [4] introduce the competitive market concept from microeconomics into load balancing, the work assumes, "each application consists of a single processing unit instead of a graph of interacting processing units" [4-page3]. Such simplifying assumptions reduce the complexity of calculations. But applications, with dependencies, cannot be managed by such schedulers.

In [11], DAG-type jobs have been considered. The proposed algorithm keeps a "node allocation graph in the future domain" [11-page 3]. Once an application is ready to allocate, the algorithm will reference the node allocation graph, get resource availability prediction for each node, and then assign each process of the application to a proper node. The accuracy of this prediction will directly affect the entire session throughput.

In [11], three normally classified workloads: compute bound; I/O bound and hybrid, [22] have been considered. The main algorithm in [11] is as follows:

1. It applies the normalization for each node and every new application.
2. It implements the critical branch heuristics when the application is received.
3. It carries out the queuing algorithm when the application is initiated for execution.
4. It applies the class allocation algorithm, branch delay algorithm and prediction algorithm when application arrives in Ready-to-run queue.
5. Queue shuffling and prioritization algorithms are implemented as well when required.
6. Eager scheduling is applied when the session approaches the end.

2.1.1 Normalization

Based on the algorithm in reference [16], nodes are sorted into different "classes" [16], according to their resources, such as CPU power, I/O power, memory size, etc. All the classes are based on a comparison with a standard machine. This classification helps create "Normalized Lookup Tables (NLT)" [16].

The users supply the estimated executing time for each process of their applications on the node in standard class. A learning system is implemented to adjust that information synchronously. The executing time in the other classes will be calculated by way of referencing the information in NLT.

2.1.2 Critical Branch Heuristics

A DAG kind of application is composed of various processes with interdependencies. In figure 1 & 2 [11-page9], you can see that an application can be divided into several branches starting from the first process to the last one. When an application arrives, it applies the "Critical Branch Heuristics" [11-page3] as below for the preferred class. Once the critical branch is found, we can calculate the allowable process delay, "T_{apd}" [11-page3], for each process. According to T_{apd}, we can delay certain process, which is not in the critical branch, to release the node for another process if required, without delaying the execution time of the whole application.

1. Identify all the branches starting from the first process to the last process.
2. Calculate the total time of each branch, ib, as

$$\mathrm{Tbranch(ib)} = \sum_{\mathbf{j=1}}^{\textbf{Number of processes in the branch ib}} \mathrm{Tpd\ (j,ib)}$$

where Tpd = Duration of process in preferred class.

3. Calculate the critical branch time

$$\mathbf{Tc\ = Max\ [\ Tbranch(ib),\ ib\ equals\ 1\ to\ Number\ of\ branches]}$$

Tc is the lower bound on the time required by the application in the optimized condition. Processes of other independent branches or part of it can be delayed until Tc without any problem.

4. Calculate the probable process start time Ts and allowable process delay Tapd for each process j of the application as

$$\mathrm{Ts(j) = Max\ [\ \sum\ Tpd\ (k,ib)\ )]}$$

k = number of previous processes in branch ib

$$\mathrm{Tapd(j) = [\ Min\ (\ Tc - \sum Tpd(k,ib)\ ) - Ts(j)\ ]}$$

k = number of subsequent processes in branch ib

where ib denotes branches having the process as a leaf.

From Algorithm1 of [11-page3]

Recent research, [2], has introduced a more sophisticated way to generate the allocation schedule of the processes.

1. First, it decomposes the application into many simpler connected "bipartite dags" [2-page7]. Each bipartite dag is simply composed of "sources" and "sinks" [2-page8]
2. Secondly, it tries to find the optimal schedule for each sub-block, and make sure such schedules can always execute all sources of the graph before any sink.

3. Then it arranges the "priorities" [2-page9] among the sub-blocks. As an example, assume "building block B_i "has priority over" building block B_j if the schedule that executes all sources of B_i before any of B_j renders sinks eligible at least as fast as does any schedule for executing both building blocks." [2-page9]
4. Finally, the algorithm generates the allocation schedule according to those priorities.

Such an allocation schedule is called "Internet-Computing optimal (IC optimal, for short)" [2-page5]. The objective is to make every step of the computation as efficient as possible.

2.1.3 Queue Algorithm

Basically, the program keeps two queues, "Wait queue" and "Ready-to-run" queue [11-page4]. When a new application arrives, it is placed in the wait queue. A concept of sliding window is introduced for Ready-to-run queue to dynamically manage the session and available resource. This algorithm ensures the system quickly responds to any changes of resources during run time. At the end time of every process, referred to as "sync point" [11-page4], it applies the Queue Algorithm. The pseudo code is as below:

```
If (work load in Ready-to-run queue, $W_{n,}$<= 80 % of its capacity){
    Apply Queue-shuffling algorithm on wait queue;
    If (certain application qualifies the select condition){
        Invoke Class Allocation Algorithm;
```

Algorithm from [11-page4]

2.1.4 Queue Shuffling Algorithm

This algorithm will check each application in the wait queue, and put it into Ready-to-run queue if its priority equals mid, which will be explained in section 2.1.9, or if it qualifies for the conditions as below:

1. Tm – Tc >= 0;
2. Tf – Tp >= 0;

Where "Tm is the maximum free duration in any node till the end of Wn, Tp is total processing time required by the application and Tf is collective free duration in all the nodes within Wn" [11-page4].

For those non qualified applications, it will calculate their time delay: Td = Td + (Tns–T), where "Tns is time of next sync point and T is time of current sync point" [11-page4]. If Td > a predefined allowable delay, it will change the priority of this application to mid to ensure it would be put into "Ready-to-run" queue at next sync point.

2.1.5 Class Allocation Algorithm

This algorithm is used to allocate an application to the resource. First, it will obtain Ts and Tapd for all processes of the application, and arrange the process in ascending order of Ts. Where Ts is the start time of a certain process, and Tapd is the allowable process duration. Then, it will arrange the processes with the same Ts in descending order of Tapd-Tpd, where Tpd is process duration. In the case of multiple processes having the same Tapd-Tpd it will sort in descending order starting from the one with the shortest processing time. Finally it will search the resource to find the free duration on the Node Allocation Graph to allocate all the processes one by one. Please see the sample data in Table 1.

If the required free resource can not be found, it will invoke "Branch Delay algorithm" [11-page6] attempting to release required resource to carry on the current process; if there is still not enough resource, it will invoke "Process Completion Predictor" or even "Application Completion Predictor" [11-page7] trying to allocate the current process.

Please reference the pseudo code as below:

```
a)  Find out the preferred class of each process of the new application
b)  Obtain the processes Ts and Tapd using Critical Branch heuristics
c)  Repeat step d to h for each Ts, where Ts[] represents the start times of the processes of that application only.
d)  Arrange all the processes in the descending order of their Tapd -Tpd at the Ts
e)  If two of more processes have same Tapd -Tpd then schedule the smaller process first.
f)  Repeat step g and h for P[j], j = 1 to n, n = number of processes to be allocated at the Ts from the sorted list.
g)  Find the number of free nodes in the class (nfc) for required duration using the node allocation graph.
h)  If (nfc >= 1){
                Allocate the node, who's $Tfn \approx Tpd$, to the process, where Tfn is free time of the node.
                Update the Node Allocation Graph.
    }
    else {
                Apply the Branch Delay Algorithm to free the required node in the preferred class
                If (succeed){
                        Allocate the node to the process.
                        Update the Node Allocation Graph.
                        Break from loop.
                }
                else {
                        Apply Process Completion Predictor algorithm.
                        If (succeed) break from loop;
                }
    }
    else {
                Apply Application Completion Predictor algorithm
                If (succeed){
                        Allocate the node to the process.
                        Update the Node Allocation Graph.
                }
    }
    else {
                Allocate the process to the earliest free node within Wn in the preferred class.
                Reapply Critical branch heuristic accordingly.
                Update the Node Allocation Graph.
    }
```

From Algorithm 4 of [11-page6]

2.1.6 Branch Delay algorithm

Basically, the "Branch Delay" algorithm tries to release a node for the required duration; Td, for the current process by delaying processes, which have already been allocated on the node allocation graph, but which have not yet run. Since it would be futile to introduce any delay to other applications to allocate the current process, these delayed processes should not be in the critical branches of the other applications.

2.1.7 Process Completion Predictor Algorithm

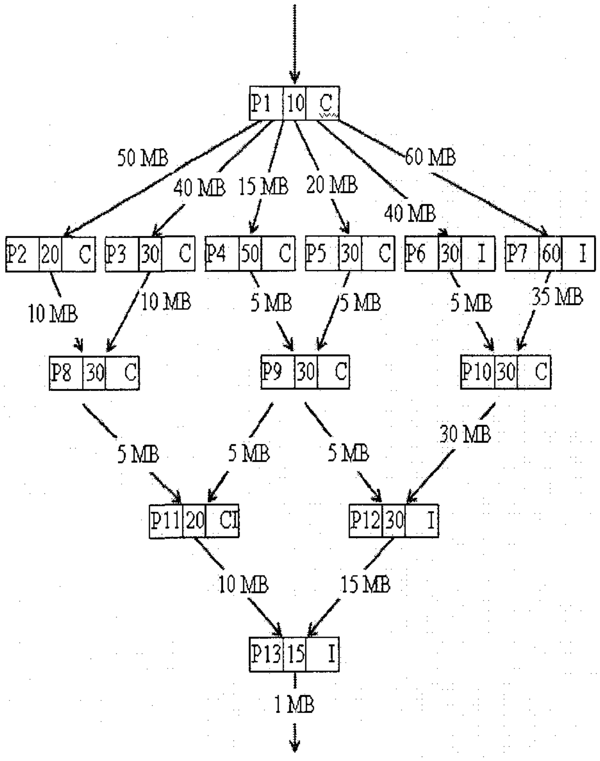
When the required resource is not found in the preferred class, it will apply Process Completion Predictor algorithm to allocate the current process to the next preferred class available.

2.1.8 Application Completion Predictor Algorithm

If the required resource is still not found after implementing all the algorithms above, the Application Completion Predictor Algorithm will be invoked to try to release the required resource via delaying some allocated processes which are in the critical branches of certain applications. Since it will introduce delay to other applications, one must consider the overall benefit of applying the system, the advantages achieved must be seriously considered

2.1.9 Priority Management Algorithm

The system manages the priority at three levels: normal, mid and top. All system level applications have the top priority. Those applications with mid level priority will be allocated without further delay unless any top level application interrupts them. All the applications are assigned with normal level priority when they arrive. They will be upgraded to mid level once an application is delayed beyond a certain allowable delay.



- P1, P2..etc are process names
- Middle value indicates the required time
- C stands for compute bound process
- I stands for IO bound process
- CI for the hybrid process
- Arrow is used for process interdependency. Means a dependent process cannot start until parent process is not successfully completed.
- The volume of data transfer between two processes is attached to the arrows

Figure 1: Interdependency graph of Application, [11-Page9]

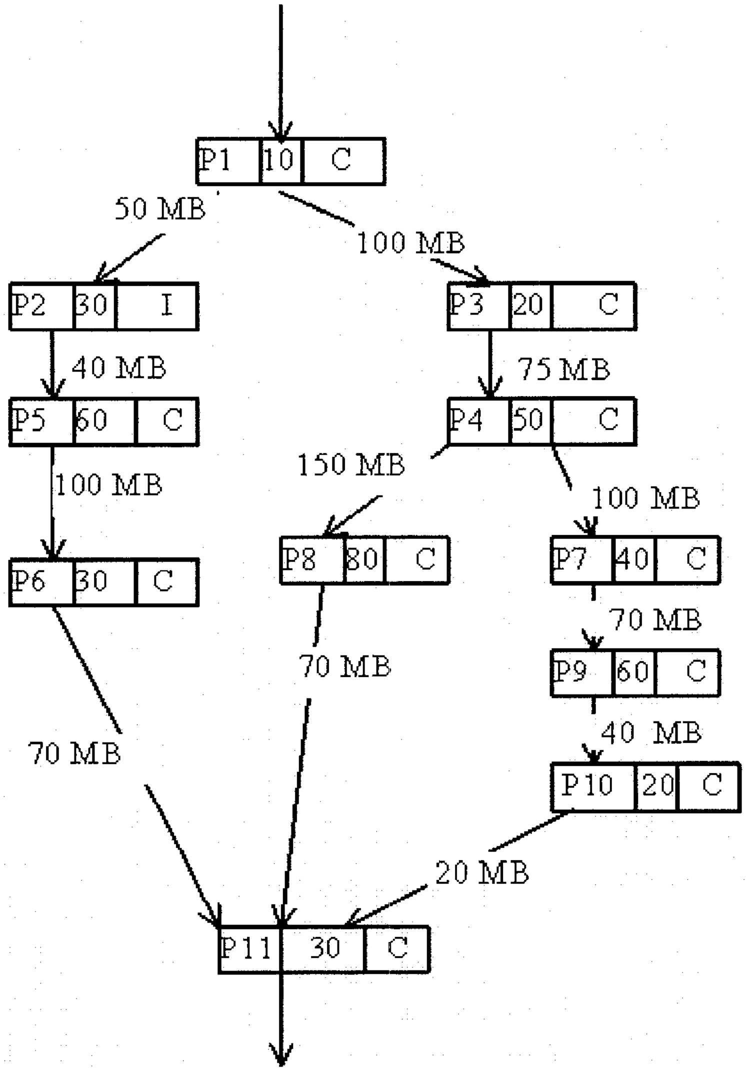


Figure 2: Interdependency graph of Application 2, [11-Page9]

App No	Number of branches	Tc	Ts	Tapd
Appl	8	145	Ts(9) = Max (60,60,40,40) = 60	Tapd(9)=Min(40,50,40,50)=40
App2	3	230	Ts(4)=Max(30)=30	Tapd(4)=Min(90,50)=50

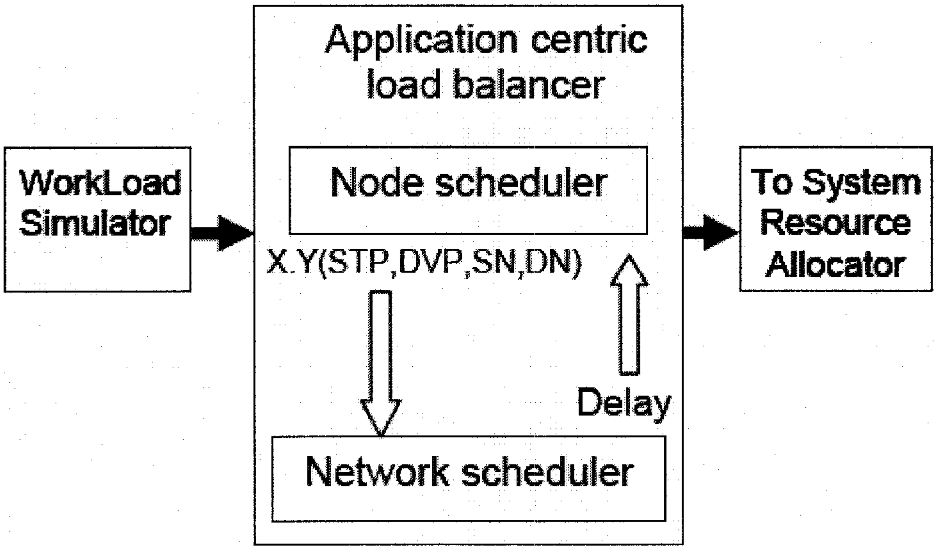
Table 1: Sample output of Critical branch heuristic, [11-Page9]

2.2 *Network load balancing*

Researchers in the area of application centric load balancing did not consider the network delays, since the work was for optimum utilization of clusters of computers. When the network delay was considered by a few researchers, they made simplifying assumptions such as "a constant latency, based on past network data traffic history and high data rate availability" [7-pagel]. In [7], a network load balancing model is introduced into the application centric load balancer [11] working on a workload with dependencies. This network model addresses the allocation problem of the network resources, and the target of the algorithm is to increase the session throughput.

Actually, it is very important to consider the network delay in load balancing for DAG kind of workload with inter-dependencies. First, the network links, which are required to receive the data from parents, may be busy in carrying other data, when all the parent processes are finished. This will directly introduce extra delay on the start time of a process. Such delays, due to the non-availability of a network link, have not been considered by most of the researchers. This may influence the utilization of the processing resources, and will eventually affect the session completion time. These delays may be introduced by the amount of data, required to be transferred from the parent processes to the child process and the different data rates of the links

The structure of the load balancing system is shown in Figure 3 [7-page2] as below.



X.Y(ST,DV,SN,DN)
X = Application Number, Y = Process Number
STP = Tentative start time (i.e. End time of a parent)
DVP = Data Volume to be transferred from a parent,
SN = Source Node of a parent,
DN = Destination Node of a parent,

Figure 3: Load Balancing Model, [7-Page2]

Basically, the system is composed of two parts, the node scheduler and the network scheduler. The node scheduler keeps a node allocation graph as the sample shown in

Figure 4 [7-page7]. When a process arrives, the node scheduler will pass the information of the process to the network scheduler, and receive delay estimations from the network scheduler. This information includes "the End Time of Parent processes (ETP) and the size of Data Volume to be transferred from a parent process (DVP) to the child process" [7-page3].

The network scheduler keeps the link allocation graph. Based on ETP and DVP passed from the node scheduler, it allocates the data transportation on the link allocation graph, acquires the delay estimations and returns it to the node scheduler. Finally, the node scheduler allocates the process on the node allocation graph, taking into account the delay estimations. With this two-step mapping, the load balancer will make the final allocation decision. This procedure will work on all of the processes of all applications that need to be allocated. Once it is done, the load balancer will forward the allocation result to the System Resource Allocator. The System Resource Allocator will do the real job to assign applications on the resources. A typical Network allocation graph is given in Figure 5 [7-page4].

1.1 = Application number. Process number
(Pr, xMB) = (Parent number, Data volume to be transferred to a child in MB)
ET = End Time of Data transfer

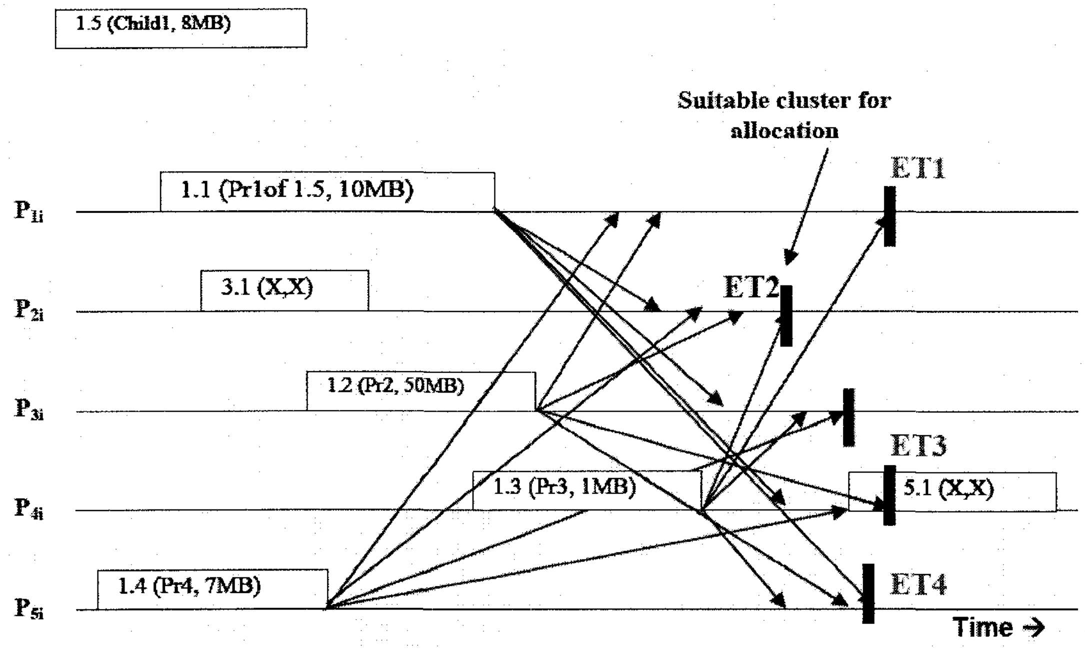


Figure 4: Node allocation graph, [7-Page7]

Communication Net Scheduling Algorithm

The pseudo code for communication net scheduling algorithm is cited from [7-page3] as below:

```
1. Get parameters (ET, DV, SN, DN)
2. Set STL = ET, DVtba = DV,
3. Consider each Link Vector i between Source Node and Destination Node
4. Find STL (i)
5. Find ETL (i)
6. Find the Data Communication Rate for Links: DRL (i) = DRmax – DRused
7. Repeat 3 to 6 till i = N; for N = the number of links involved
8. Calculate STL = max {STL (i)}; for i = 1 to N
```

```
9. Calculate $ETL = min \{ET (i)\}; for\ i = 1\ to\ N$
10. Calculate $DRL = Min \{DRL (i)\}; for\ i = 1\ to\ N$
11. Set $DVtba = [DVtba - \{DRL * (ETL - STL)\}]$
12. Set $STL = ETL$
13. Repeat step 3 to 12 till $DVtba = 0;$
14. Return $Delay = (ET - ETL)$
```

The objective of this algorithm is to find out the time delay for transferring the result data from parents to the child process. In order to do so, we need the data volume transferring from parents (DVP), and the "data communication rate for the links (DRL)" [7-page3] involved. Since DVPs are parameters passed from node scheduler, the only thing we need to calculate is the DRL for each network link involved in the communication. The algorithm picks the minimum DRL from all the links as the common communication rate for that certain amount of time, and calculates how much data can be transferred in that duration. This cycle will keep working until all the data from parents has been transferred to the child process.

In the step of finding the DRL for each link involved, it needs to search all the links one by one. It is worthless to continue calculating the DRLs in their entirety if the DRL of a particular link is zero. The futility increases when the DRL turns out a zero in the last few links. In order to accommodate this problem, the algorithm implements the binary searching method to find the DRLs. This will improve the search efficiency and save a lot of time in cases when there is a link with zero DRL in the last few links. An example of Network allocation graph is shown in Figure 5:

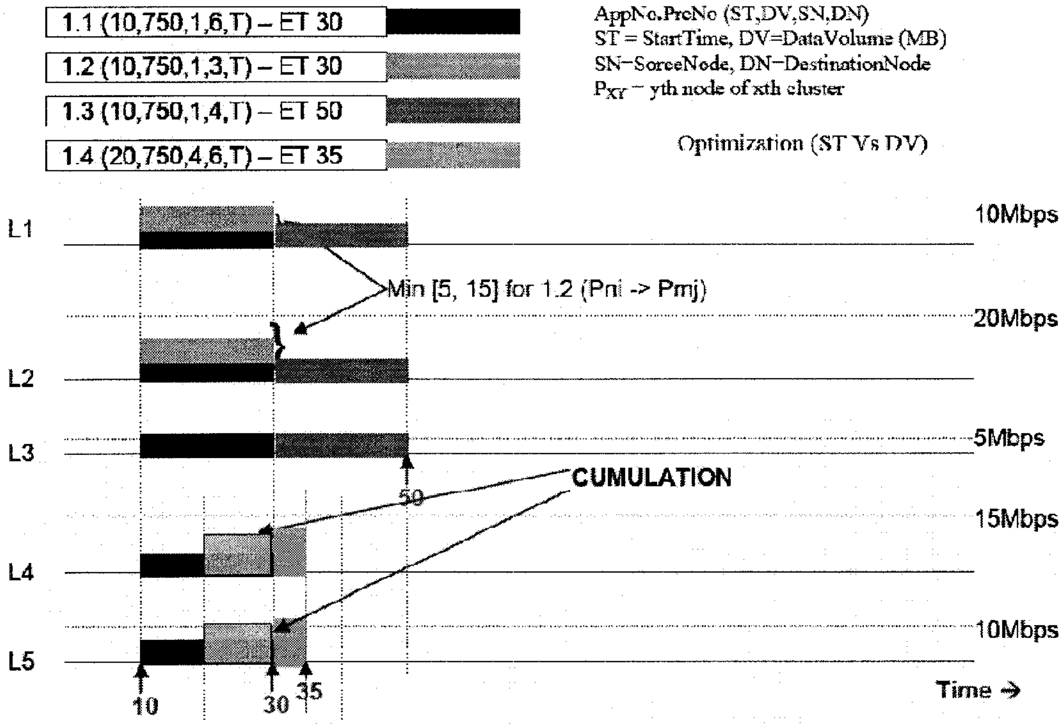

1.1 (10,750,1,6,T) – ET 30
1.2 (10,750,1,3,T) – ET 30
1.3 (10,750,1,4,T) – ET 50
1.4 (20,750,4,6,T) – ET 35
AppNo.PreNo (ST,DV,SN,DN)
ST = StartTime, DV=DataVolume (MB)
SN–SorceNode, DN–DestinationNode
P_{XY} – yth node of xth cluster
Optimization (ST Vs DV)
L1
10Mbps
Min [5, 15] for 1.2 (Pni -> Pmj)
20Mbps
L2
L3
5Mbps
CUMULATION
50
15Mbps
L4
10Mbps
L5
10
30
35
Time →

Figure 5: Network Allocation Graph, [7-Page4]

The network allocation graph of Figure 5 refers to a network of five links connecting 6 compute-nodes. Link L1 connects Node 1 and Node2, L2 connects Node2 and Node3, and so on. The x axis represents the time, and the y axis represents the communication rate for each link. The meta data of four processes, 1.1, 1.2, 1.3 and 1.4, which need to be assigned on the network, is shown in the Figure. According to the information above, the result data of process 1.1 will be transferred from Node 1 to Node 6. It starts from time t=10, with a data volume of 750 MB. Since the data passes all the five links, it can only transfer under the minimum common rate, which is 5Mbps of L3. It will take a total of 20 minutes to transfer the data. Similarly process 1.4, which will pass 750 MB data through L4 and L5, can begin transfer of data at time t=20. Since both L4 and L5 are being used by process 1.1 for transfer of data at a data rate of 5 Mbps from t=10 to t=30, L4 and L5 can make available to process 1.4 a data rate of 10 and 5 Mbps respectively. Therefore, data transfer for process 4 can only work at 5 Mbps from t=20 to t=30. After t=30, data can be transferred by process 1.4 through L4 and L5 at 10 Mbps. In total, it takes 15 minutes to finish the transportation.

The advantage of this algorithm is that once a process is ready for allocation; it does not need to care about any other processes allocated on the links involved. The only thing that needs to be considered is the available data transfer rate, after the cumulative effect of all the previous allocations is taken into account. This localization makes the calculation very simple and fast.

Optimization Algorithm

In order to improve the performance, four possible optimizations are introduced into the original network scheduling algorithm. The basic idea is to calculate the network delays for different data transfer sequence schedules from all parents, and pick the one with minimum duration as the final network allocation recommendation. There are two main parameters for a process to make the transfer decision, a list of "End Time of all the parents (ETPs)" and a list of "Data Volume to be transferred from all parents (DVPs)" [7-page3]. The first and second optimizations schedule the transfer sequence in ascending and descending order of ETPs respectively, and calculate the delay expectation. Whereas, the third and fourth optimizations arrange the data transfer order in ascending and descending order of DVPs respectively. Finally, the network allocation with minimum delay duration will be returned to the node scheduler as the network delay recommendation.

The experimental results are shown in Figure 6 [7-page5]. The x axis represents the number of compute nodes in each case; and the y axis shows the normalized network utilization. The blue curve and pink cure represents the network utilization with and without optimization respectively. From the chart, we can see that the optimization algorithm leads to an improvement in network utilization for every case of compute-nodes varying from 2 to 45.

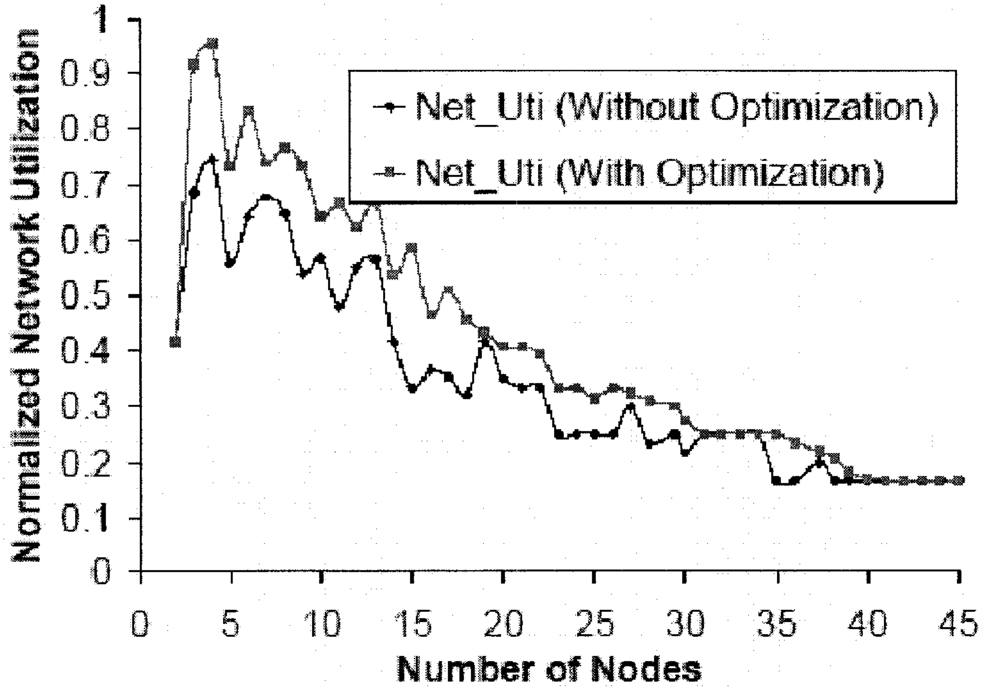


Figure 6: Normalized network utilization nodes, [7-Page5]

3. Motivation and Model Formalization

3.1 Motivation

In the application centric load balancer [11] for distributed system, an application was put in the waiting queue when it arrived. At every sync point, the program will call Queue Shuffling Algorithm, and put all qualified applications into ready-to-run queue, and then allocate the applications in the ready-to-run queue according to the Class Allocation algorithm.

In this process, the applications are allocated on the resources simply based on its arrival sequence. Similarly in [1], the jobs in the global job-waiting queue are processed in "First-Come-First-Serve" order. Since nothing else is considered, this sequence may not give optimum utilization of the resources, especially in the heavy load scenario. If we can implement some scheduling algorithm to rearrange the schedule of the application allocation, it may produce more effective utilization of the resources. Therefore, we started studying the case of Genetic Algorithm to schedule the sequence of the allocation. The objective is to find the shortest session completion time.

3.2 Model Formalization

In our study, we focused on scheduling the application allocation on heterogeneous compute nodes connected by a distributed network.

3.2.1 Compute Nodes

In this thesis, we remove the homogeneous simplification of the compute nodes, and we considering that every node may have a different compute capacity. As mentioned before, we implement the normalization process [11] for each node, and finally build up a Normalized Lookup Table. It is used to calculate the completion time on any specific node. Generally, if a node has twice the compute capacity as the other one in NLT, it will take half the time to finish the same process on the first one than the second.

3.2.2 Distributed Networks: Topology

A network with bus topology has been assumed in this study. The network links may have its own data transfer rate, and it introduces network delay, depending upon the amount of data to be transferred from a parent process to the child process.

3.2.3 Work Load

In this thesis, we focus our research on scheduling generalized workloads composed of sub jobs with internal dependencies. For our experiments, we use the workload simulator, [7], to generate the work load for our proposed algorithm. This simulator has a detailed parameter system, which allows generation of the workload for testing the scheduler.

4. Proposed Algorithm

The solution space [8-page2] for allocating applications to a distributed system is very large. The grid scheduling problem has been proven to be a NP-complete problem. However, based on the objectives supplied, a Genetic Algorithm (GA) based scheduler can find a solution near to the optimum value within an affordable amount of computing effort. This algorithm works well even if the solution space is very large.

Instead of going through the whole solution space to find the optimum value, the first step in a GA-based scheduler is to first select a very small portion from the solution space, which is called a generation. Secondly for each schedule, in the generation, a quality benchmark, usually called the fitness value is calculated. As a third step, for all the schedules in the generation the standard deviation in the fitness values is calculated.. As a fourth step, another portion of the solution space, called the second generation, is calculated from the first generation in such a way that the schedules in the second generation may be closer to the optimum value. Then the second, the third and the fourth steps are repeated on the second generation to generate the third generation. The GA-based scheduler continues to generate new generations until the value of standard deviation reaches a specified low range. This is when the schedule with the best fitness value in the last generation provides a near-optimum scheduling solution.

Figure [8] is typical implementation of GA. First, it generates an initial generation Figure [8] is a typical implementation of the Genetic Algorithm. First, it generates an initial generation randomly. Then it calculates the fitness value for each schedule, and also gets the mean fitness value and standard deviation from the mean for the whole generation.

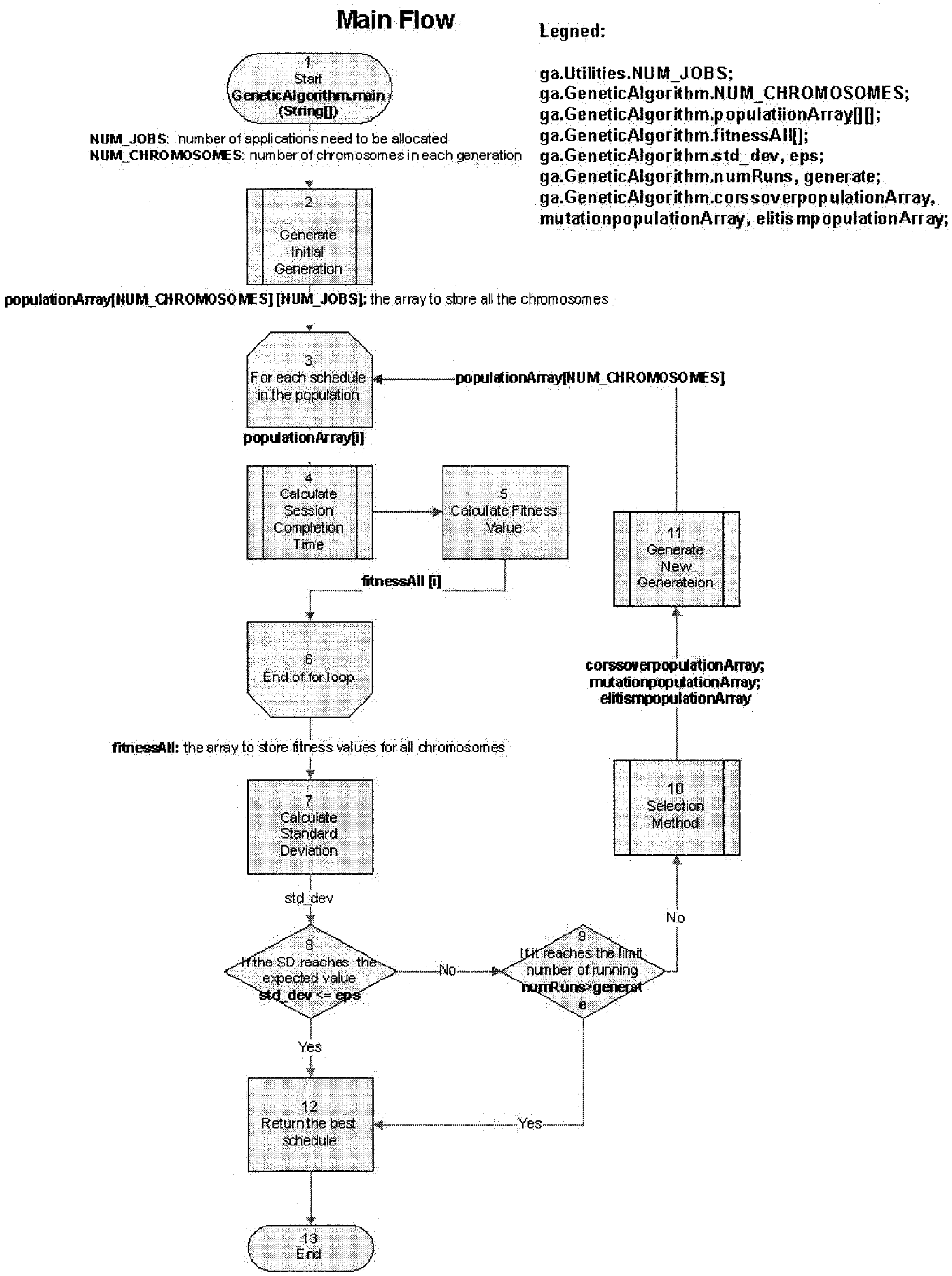
After that, it implements roulette wheel selection method to pick up schedules from existing generation as the parents, and applies three methods, crossover method, mutation method, and elitism, to generate a new population. This loop will keep running until the SD reaches the expected threshold or the number of generations reaches the limit.

In GA terminology, a Chromosome denotes a schedule of independent applications which need to be allocated to the distributed system of processors. In general, each application is a DAG type of job, which has many processes with multi-level inter dependencies. A new chromosome is created by modifying the sequence schedule of an existing chromosome [8].

4.1 Main Strategy

Based on this structure, we propose an adaptive GA for task scheduling in distributed system by using the ideas of adaptation of GA parameters proposed in [5]. The main flow of the proposed algorithm is given in Figure 7:

The model, given in section 3.2 is assumed. It is assumed that a number of applications, which are to be scheduled have been received. For testing the algorithm, the applications would be generated by using the fuzzy generator [7]. In a real world scenario, the applications would be submitted by various users of the distributed system to a broker. The broker would use the proposed GA-based scheduler to allocate the applications such that the compute-nodes and the network resources are optimally utilized and a near-optimum value of the session completion time is obtained.



Legned:

ga.Utilities.NUM_JOBS;
ga.GeneticAlgorithm.NUM_CHROMOSOMES;
ga.GeneticAlgorithm.populatiionArray[][];
ga.GeneticAlgorithm.fitnessAll[];
ga.GeneticAlgorithm.std_dev, eps;
ga.GeneticAlgorithm.numRuns, generate;
ga.GeneticAlgorithm.corssoverpopulationArray,
mutationpopulationArray, elitismpopulationArray;

Figure 7: Main Flow of Proposed Algorithm

The flow chart in Figure 7, illustrates an overview of how the algorithm works.

Basically, the system requires that two main parameters be set by the user to begin the evolution process: the number of applications, NUM_JOBS, tells the system how many applications need to be allocated in each chromosome; and the number of chromosomes, NUM_CHROMOSOMES, decides the number of chromosomes in each generation. The steps in the algorithm are as follows:

1. Once the system gets these two parameters, it runs the 'Generate initial population' model to get the initial population of NUM_CHROMOSOMES chromosomes with NUM_JOBS applications in each of them..
2. Then the program goes through the initial population, runs the 'Calculate Session Completion Time' to calculate the session, and then calculates the fitness value for each of the chromosomes and finally calculates the Standard Deviation for this generation.
3. The system generates crossover rate, mutation rate and elitism rate.
4. According to the rates in step 3, it calls the 'Selection Method' model to pick up a certain amount of chromosomes from the current generation as the parents.
5. It calls the 'Generate New Generation' model to initiate a new generation from the parents above.
6. It increments the number of runs, numRuns++.

The algorithm repeats step 2 to step 6 until the SD reaches an expected low threshold, eps, or the times counter, numRuns, reaches the expected high threshold. Then it picks up the chromosome with the best fitness value as the final near-optimum solution.

4.2 Generate initial population

Since all the calculations are based on the initial population, it plays a very important role in GA. In [8], the initial population is simply generated via randomly picking up a small portion of chromosomes from the solution space, without any consideration of the quality of the chromosome. This algorithm has severe drawbacks. First, because of its inherent uncertainty, it is not likely to generate proper chromosomes to start with. Moreover, in some circumstances, certain parts of the solution space will not be reachable with this initial population.

For generating the initial population, the proposed algorithm introduces a strategy so that the search covers as large a portion of the solution space as possible. As an example, assume that there are 10 applications that need to be allocated, and the number of chromosomes in the initial generation is 100. Instead of generating 100 chromosomes randomly, it will go through the 10 applications one by one, and generate 10=100/10 chromosomes, starting with each application respectively. This will ensure the chromosomes contained in the initial generation, come from all directions of the solution space. Implementing with recursive calculation, this algorithm will yield a superior result. Figure 8 shows this process of generation of the initial population.

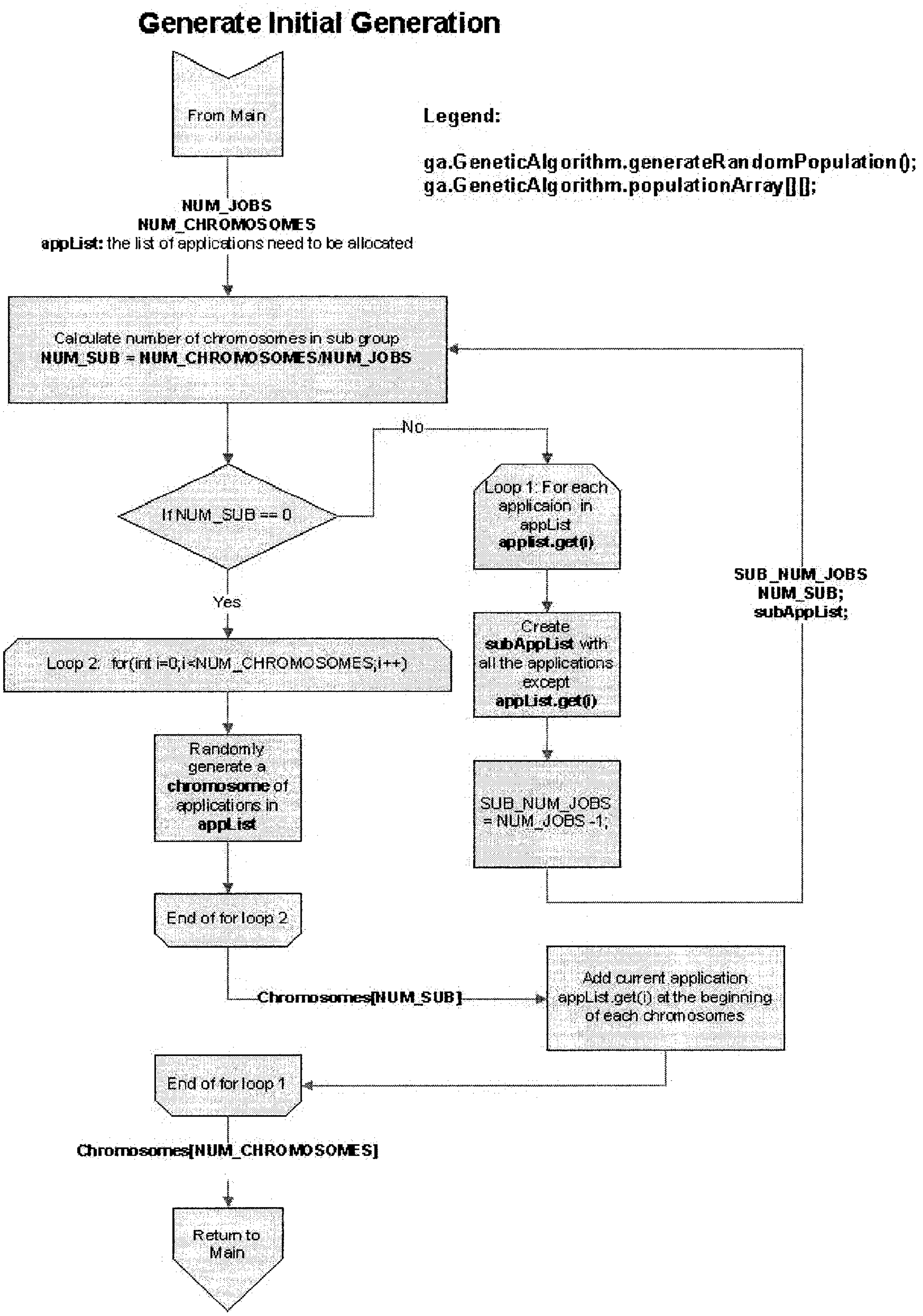

Generate Initial Generation
From Main
Legend:
ga.GeneticAlgorithm.generateRandomPopulation();
ga.GeneticAlgorithm.populationArray[][];
NUM_JOBS
NUM_CHROMOSOMES
appList: the list of applications need to be allocated
Calculate number of chromosomes in sub group
NUM_SUB = NUM_CHROMOSOMES/NUM_JOBS
No
If NUM_SUB == 0
Loop 1: For each applicaion in appList
applist.get(i)
Yes
SUB_NUM_JOBS
NUM_SUB;
subAppList;
Loop 2: for(int i=0;i<NUM_CHROMOSOMES;i++)
Create subAppList with all the applications except appList.get(i)
Randomly generate a chromosome of applications in appList
SUB_NUM_JOBS = NUM_JOBS -1;
End of for loop 2
Chromosomes[NUM_SUB]
Add current application appList.get(i) at the beginning of each chromosomes
End of for loop 1
Chromosomes[NUM_CHROMOSOMES]
Return to Main

Figure 8: Sub Flow – Generate Initial Population

4.3 Calculation of the Session Completion Time

Each chromosome of the generation provides a sequence of all the applications, which need to be scheduled. The objective of this module of the algorithm is to allocate all the applications and calculate the total session completion time, Ts, and find the maximum Critical Branch time, Tc, from all the applications.

The strategy is quite simple. It goes through each of the applications one by one in the scheduled order, calls on the 'Allocate Application' model to allocate it on the environment and finally get its Critical Branch time, Tci, and the session completion time Tsi. Then the system compares Tci, Tsi with Tc and Ts, and set Tc = Tci if Tc<Tci. It does the same to Ts. Once the whole process ends, it will get the right values for Tc and Ts, which are going to be used in the fitness value calculation. The flow chart in Figure 9 shows the detailed procedure, step-by-step.

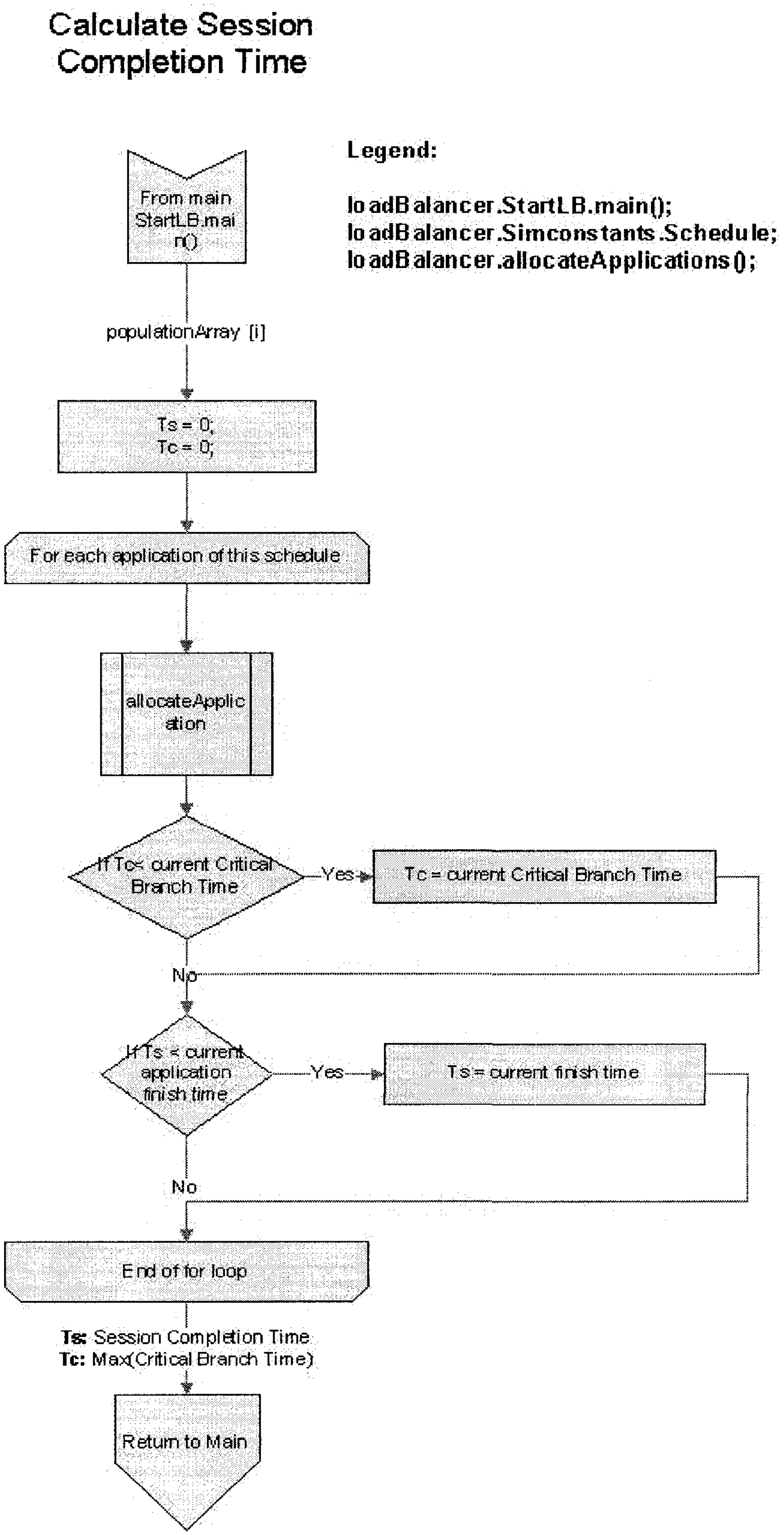

Calculate Session Completion Time
Legend:
loadBalancer.StartLB.main();
loadBalancer.Simconstants.Schedule;
loadBalancer.allocateApplications();
From main StartLB.main()
populationArray [i]
Ts = 0;
Tc = 0;
For each application of this schedule
allocateApplication
If Tc< current Critical Branch Time
Yes
Tc = current Critical Branch Time
No
If Ts < current application finish time
Yes
Ts = current finish time
No
End of for loop
Ts: Session Completion Time
Tc: Max(Critical Branch Time)
Return to Main

Figure 9 Sub Flow – Calculate Session Completion Time

4.3.1 Allocation of Applications

The algorithm maintains a process pool for all the processes which are ready to run. It goes through the process pool, and allocates one process per time until all the processes in the pool are allocated. There are two critical parts in this process. First is how the algorithm maintains the process pool; second is how to pick up the process from the pool in each allocation.

Basically, the pool maintenance is based on the internal process dependency structure of the work load. Once this model gets an application composed of a couple of processes with internal dependencies, it puts the initial process into the pool and then allocates it. After the first process has been allocated, it checks the dependency and releases all its dependent processes into the pool for allocation. This process is repeated until all the processes of this application are allocated. Every time, when a process has been allocated, the system checks all its dependents and allocates those children, whose parents have been already allocated.

If there is more than one process in the pool for allocation, the algorithm calculates Tapd – Tpd for each process, and picks the process with the minimum value. Tapd is the allowable process duration and Tpd is process duration. This process is then allocated. Figure 10 shows the detailed flow of this process.

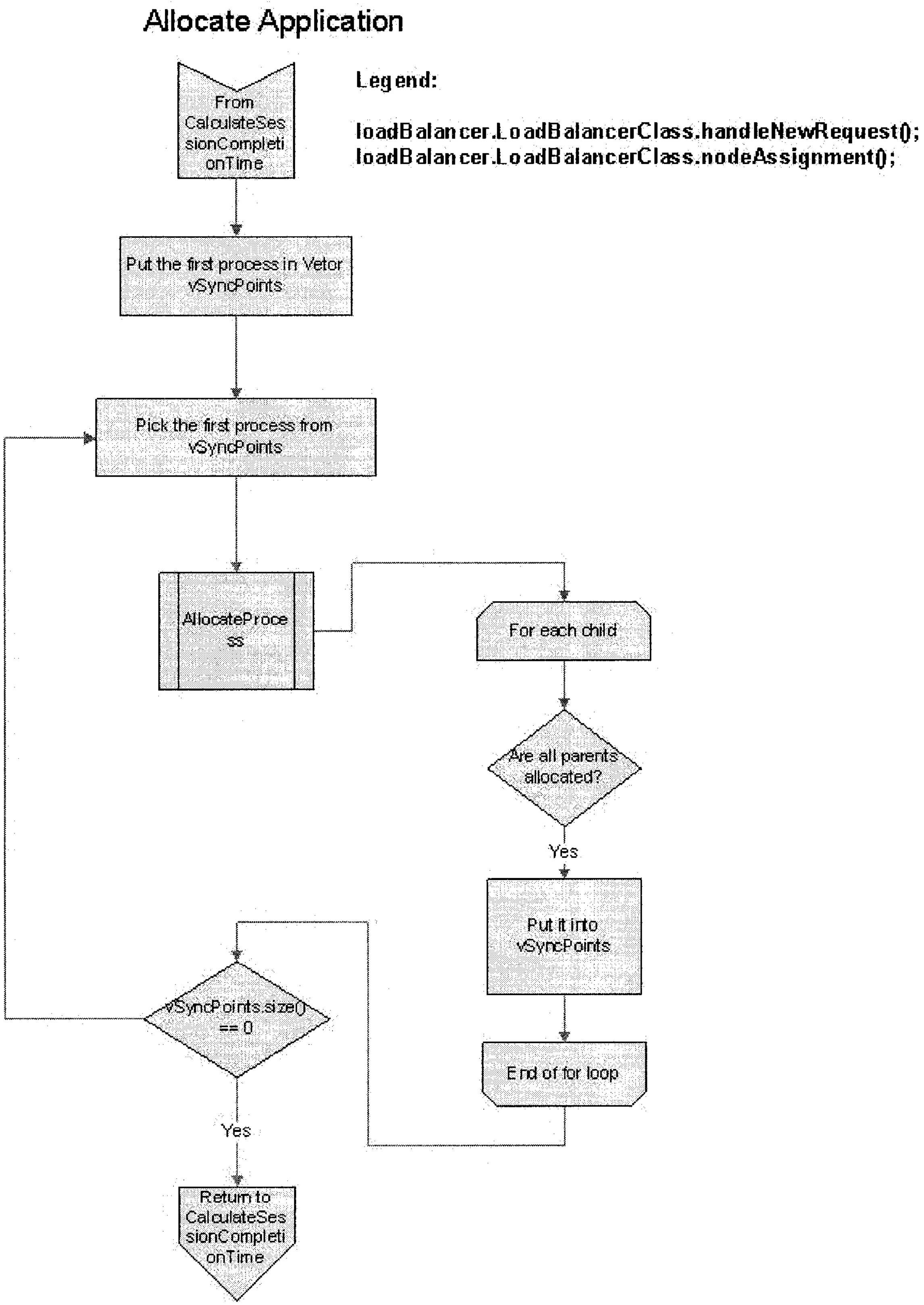

Allocate Application
Legend:
loadBalancer.LoadBalancerClass.handleNewRequest();
loadBalancer.LoadBalancerClass.nodeAssignment();
From CalculateSessionCompletionTime
Put the first process in Vetor vSyncPoints
Pick the first process from vSyncPoints
AllocateProcess
For each child
Are all parents allocated?
Yes
Put it into vSyncPoints
End of for loop
vSyncPoints.size() == 0
Yes
Return to CalculateSessionCompletionTime

Figure 10: Sub Flow - Allocate Application

4.3.2 Allocation of a Process

The basic strategy here is to check all the compute nodes in the network for the first free time with enough duration to allocate the current process and choose the one with minimum start time as the target to allocate the process.

In the above process, once a node is picked, we first need to calculate the potential start time for the process. It is the sum of the last parent finish time plus the network delay introduced by transferring all the parents' data to the current node. This delay is calculated by the 'calculate Network Delay' mode. Then the algorithm starts searching for the first "enough time duration" on the node after the potential start time.

After the algorithm checked all the nodes, it picks the node with the earliest start time of the time durations, allocates the process on this node and finally updates the node allocation graph. Figure 11 shows the details of this flow.

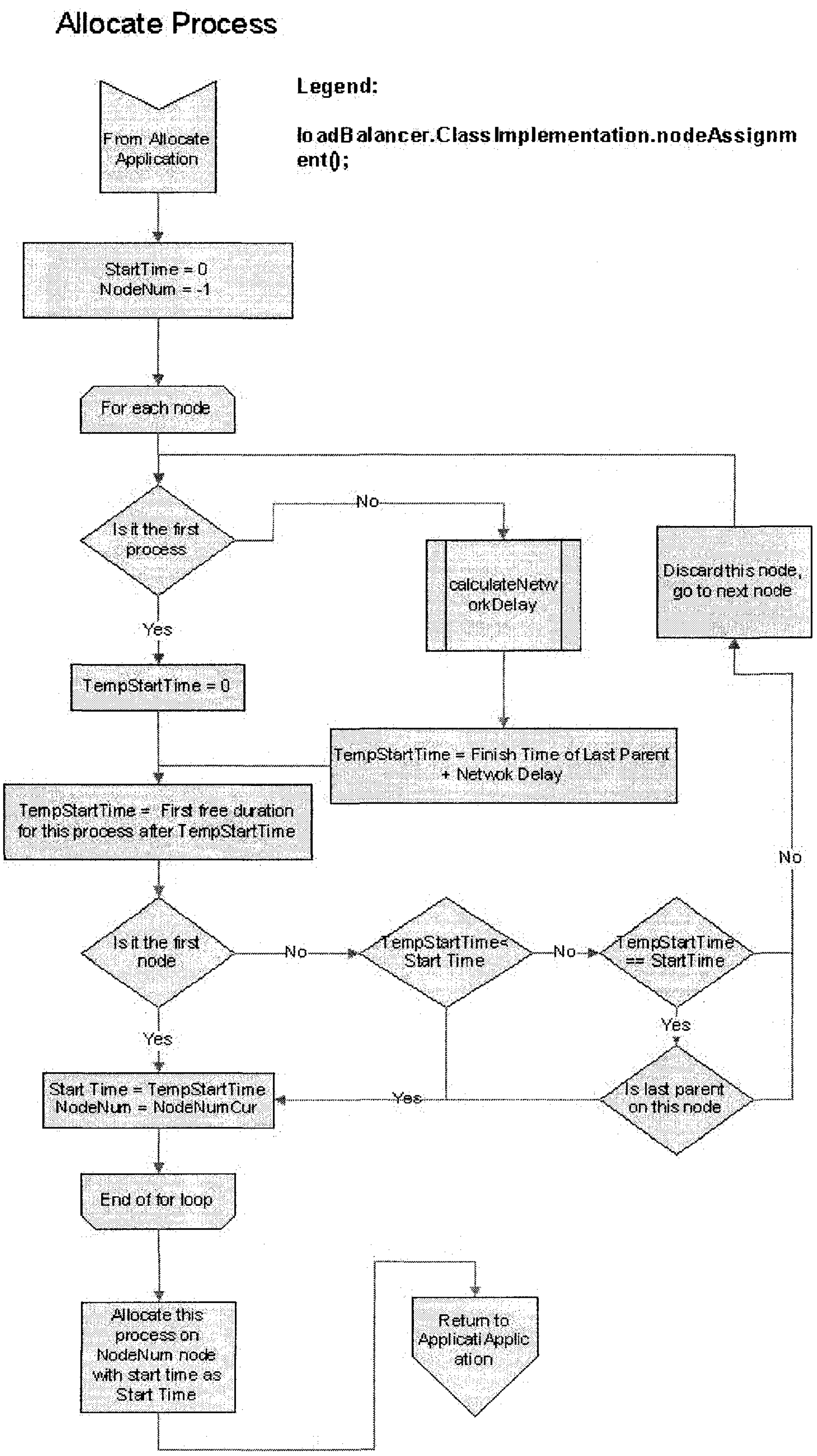

Allocate Process
Legend:
lo adB alancer.Class Implementation.nodeAssignm ent();
From Allocate Application
StartTime = 0
NodeNum = -1
For each node
Is it the first process
No
Yes
calculateNetworkDelay
Discard this node, go to next node
TempStartTime = 0
TempStartTime = Finish Time of Last Parent + Netwok Delay
TempStartTime = First free duration for this process after TempStartTime
No
Is it the first node
No
TempStartTime< Start Time
No
TempStartTime == StartTime
Yes
Yes
Yes
Start Time = TempStartTime
NodeNum = NodeNumCur
Is last parent on this node
End of for loop
Allocate this process on NodeNum node with start time as Start Time
Return to Applicati Applic ation

Figure 11: Sub Flow – Allocate Process

4.3.3 Calculation of Network Delay

At this point, we have selected the node to allocate the process. The algorithm goes through all the parents one by one. For each of them, it first finds the node where this parent is located, and builds the route to transfer the data from the parent node to the child node. Then it checks all the links in the route and finds the common time duration and common minimum transfer rate. Using the common minimum transfer rate, it calculates the amount of data, which can be transferred in this time duration. This process is repeated until the entire data from the parent's node can be transferred to the node where the child is located. This finally gives the time delay for this transfer of data.

Once the above process has been completed for all the parents, the algorithm will get the final time delay, which is the difference between the earliest start time and the latest end time of all the transfers. Figure 12 shows the detailed flow for this calculation.

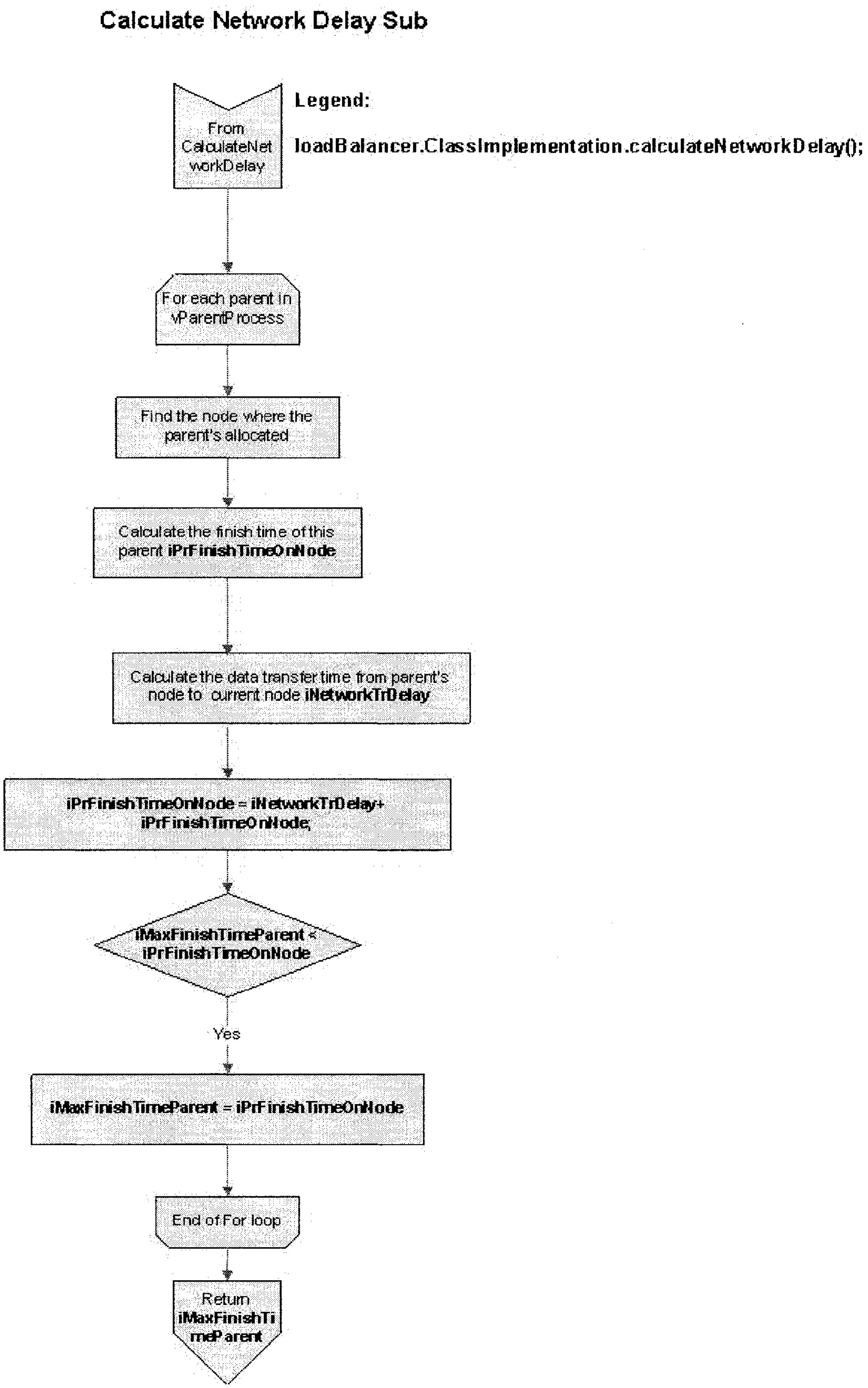

Calculate Network Delay Sub
Legend:
loadBalancer.ClassImplementation.calculateNetworkDelay();
From CalculateNetworkDelay
For each parent in vParentProcess
Find the node where the parent's allocated
Calculate the finish time of this parent iPrFinishTimeOnNode
Calculate the data transfer time from parent's node to current node iNetworkTrDelay
iPrFinishTimeOnNode = iNetworkTrDelay+ iPrFinishTimeOnNode;
iMaxFinishTimeParent < iPrFinishTimeOnNode
Yes
iMaxFinishTimeParent = iPrFinishTimeOnNode
End of For loop
Return iMaxFinishTimeParent

Figure 12: Sub Flow – Calculate Network Delay Sub

4.3.4 Optimization of Network Delays

Since all the data transfers share the same network, there would be some overlaps on certain links. These overlaps will affect the data transfer rate of those links in certain time durations and finally affect the total network delay. This means the transfer sequence does affect the total network delay. An optimum strategy is introduced to reduce the network delay by rearranging the transfer sequence of the processes.

There are two main attributes for each parent: the end time of the parent process, and the data volume, that needs to be transferred from the parent node to the child node. The algorithm gets the list of end times, ETP [7], and the list of data volumes, DVP [7], of all the parents. Then it arranges the transfer in four different sequences, and finally picks the one with the shortest value as the network delay recommendation. The list of the four sequences is as follows:

1. Arranging the transfer in the ascending order of ETP[];
2. Arranging the transfer in the descending order of ETP[];
3. Arranging the transfer in the ascending order of DVP[];
4. Arranging the transfer in the descending order of DVP[];

Figure 13 shows the process in greater details.

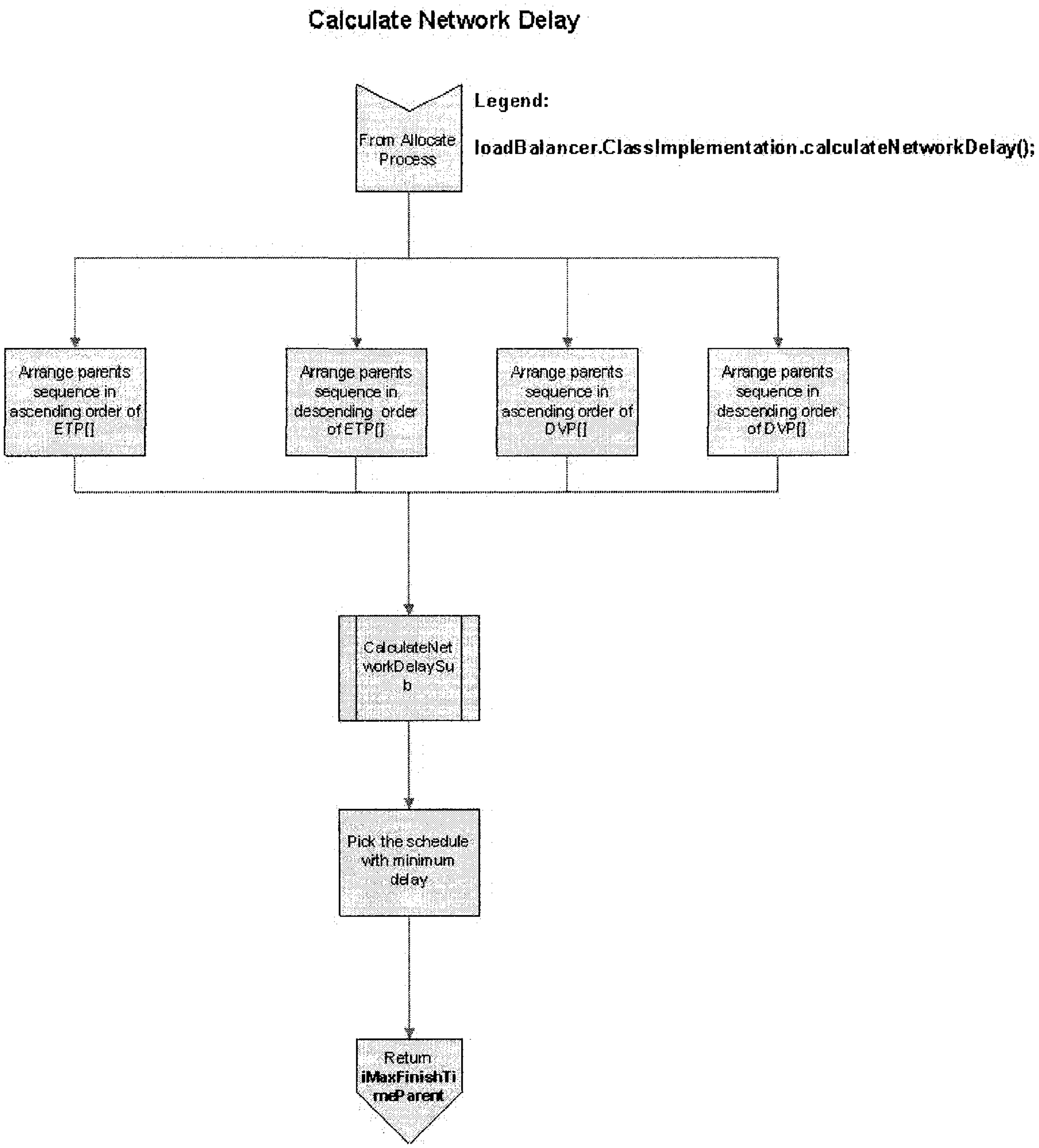

Calculate Network Delay
Legend:
loadBalancer.ClassImplementation.calculateNetworkDelay();
From Allocate Process
Arrange parents sequence in ascending order of ETP[]
Arrange parents sequence in descending order of ETP[]
Arrange parents sequence in ascending order of DVP[]
Arrange parents sequence in descending order of DVP[]
CalculateNetworkDelaySub
Pick the schedule with minimum delay
Return iMaxFinishTimeParent

Figure 13: Calculate Network Delay

4.4 Fitness Function and Standard Deviation

First, we need a benchmark to measure the quality of a schedule. Based on the optimization objectives, fitness function is introduced as a measure of the quality of the schedule. Generally, a fitness function is created with several objectives. The system user may want to reduce turn-around time of the application submitted by the user; while a resource provider may want to shorten the session completion time, and enhance the utilization of the compute resources. Therefore, weight coefficient is applied to each part of the function representing different objectives. This allows the fitness function to accommodate its focus on certain objectives dynamically [8].

In this thesis, the only goal of our schedule is minimizing the session completion time. Therefore the weight for it is 1. Let t_c (i) be the critical time of the i-th job, Tc is the Max (t_c (i)) over all the jobs being scheduled and t_{st} is the end time of the last task for a given schedule. Then, denoting ***w*** as the makespan for the session,

$$\boldsymbol{w = 1 - (T_c / t_{st})} \qquad \textbf{(1)}$$

The fitness function F:

$$\boldsymbol{F = 1 - w} \qquad \textbf{(2)}$$

After the fitness function for each of the chromosomes of a generation, which is the small portion picked from the whole solution space, has been calculated, the average value of the fitness functions and the standard deviation are computed. Standard Deviation (SD), which is a very popular benchmark used to measure the spread of values in a population, is defined as the square root of the average of the square of the difference between fitness values and the mean. Equation (3) gives the standard deviation:

$$\partial = \sqrt{\frac{1}{N}(\sum_{i=1}^{N}(fi - \overline{f})^{2}} \qquad \textbf{(3)}$$

Where N is the number of chromosomes in each generation; f_i is the fitness value of each chromosome; and $\overline{f}$ is the mean of all fitness values in existing population.

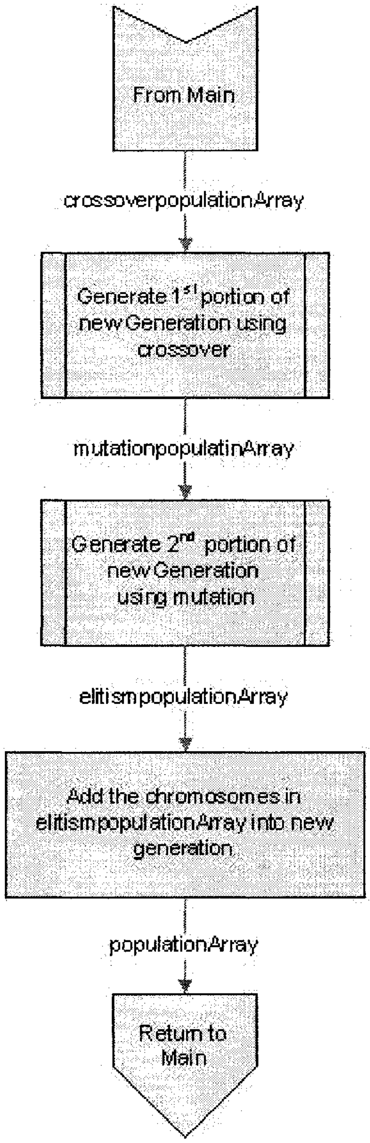
$$\overline{f} = \frac{1}{N}\sum_{i=1}^{N} fi \qquad \textbf{(4)}$$

The genetic algorithm will converge once the SD reaches an expected low value. The best solution achieved so far would be the near optimum result we are looking for.

4.5 Generation of New Population

There are two main steps to generate the new population. First, we need to use some selection method to pick up a proportion of existing generation as the parents to generate a new generation. Then certain kinds of strategy are used to generate the new generation. These methods include crossover, mutation and elitism. Figure 14 shows the main flow of this process.

Generate New Population


From Main
crossoverpopulationArray
Generate 1st portion of new Generation using crossover
mutationpopulatinArray
Generate 2nd portion of new Generation using mutation
elitismpopulationArray
Add the chromosomes in elitismpopulationArray into new generation
populationArray
Return to Main

Legend:

ga.GeneticAlgorithm.generateNewPopulation();
ga.GeneticAlgorithm.crossover1();
ga.geneticAlgorithm.crossover2();
ga.geneticAlgorithm.Mutation();

Figure 14 Sub Flow – Generate New Population

4.5.1 Selection method

In the selection method, proper parents are selected based on their fitness values. Typically, it ensures that the candidates with higher fitness values will be more likely to be picked. Some selection methods select solutions with best fitness values. Other methods prefer to select the best solution from a randomly picked portion of the existing generation each time. This process dramatically reduces the computation time over the previous one. In [8], a roulette wheel selection method is used to pick up the candidates to generate a new generation. The pseudo code and its complexity calculation are as follows:

```
generates a sorted array for existing generation according to their probabilities

for (int i=0; i<c1*n; i++ {                                              c1*n
        generate random probability radom1;
c1*n

        for (int j=0;j<n;j++){                                            $\sum_{j=1}^{n} j$

                if (probability[j]<=random1<probability[j]){
                        select this parent;
                        break;
                }
        }
}
$O(n) = c1*n+c2*n+c3*\sum_{j=1}^{n} j = c1*n+c2*n+c3*n(n)/2 = n^2;$
```

Instead of using the roulette wheel selection method, in our proposed algorithm, we chose to use the tournament selection method, which is another popular method of selection. It is simpler to implement and will provide faster comutation. The pseudo code and its complexity are as below:

```
For (int i=0; i<c1*n; i++){                                   c1*n
        Randomly pick up two parents a, b;                    c2*n
        Choose the one with higher probability                        c3*n
}

$O(n) = c1*n+c2*n+c3*n = n;$
```

Figure 15 shows the main flow of selection method.

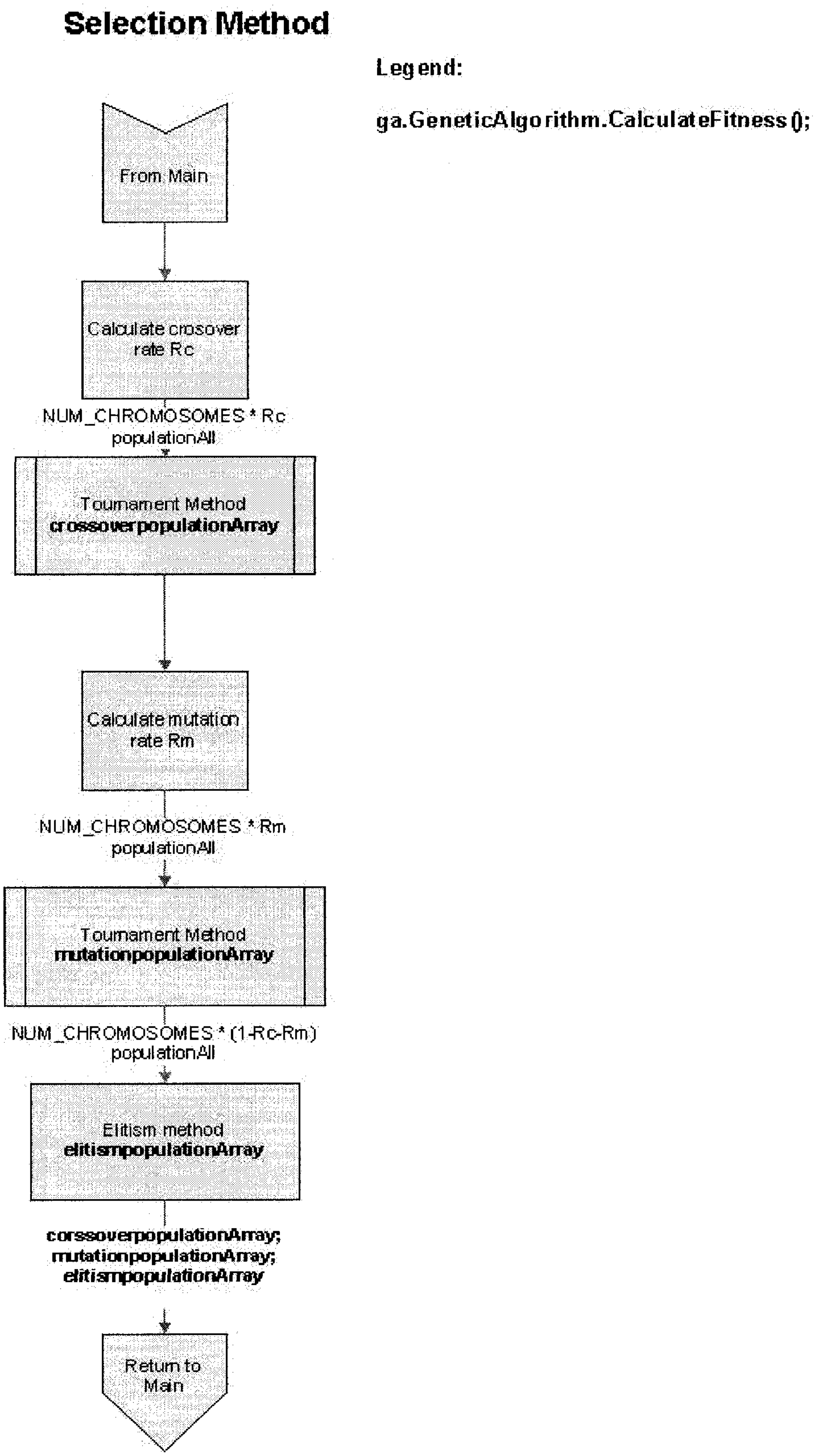


Figure 15: Sub Flow – Selection Method

The crossover rate r_c and mutation rate r_m are the proportions of how many chromosomes are picked from existing generation as the parents to generate the new generation. These values are very important in GA, since these values play critical roles in its convergence. Like most of the GAs in the literature, [8] chooses to use values, obtained through experiments. Picking a value of these parameters improperly will make the whole GA calculation end up with premature convergence or convergence stagnation. Instead of using constants, we introduce the idea of SAGA [5], and choose to change these values on the fly while generating the new generation. The change is based on the quality of the generation. The key idea of this algorithm is to dynamically balance the local and global searching in the process and thereby to improve the optimization. Below are the equations in our algorithm to calculate r_c and r_m:

$$\mathbf{r_c = 1-} \frac{1}{N}\sum_{i=1}^{N}\frac{Tc}{Ti} \qquad \mathbf{(5)}$$

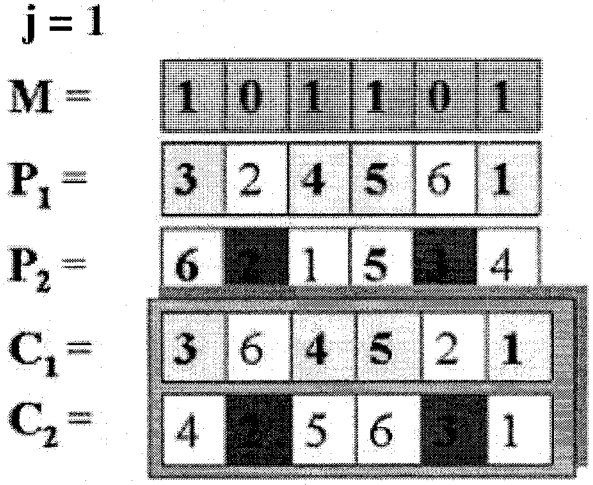
$$\mathbf{r_m = (}\frac{1}{N}\sum_{i=1}^{N}\frac{Tc}{Ti}\mathbf{)/10} \qquad \mathbf{(6)}$$

where Tc is the maximum Critical Branch Time and Ti is the session end time for the ith chromosome.

Crossover is the process to bring the results approaching to local optimum, while mutation tries to break this phase to prevent the premature convergence. In our algorithm, the r_c and r_m are selected such that r_c will go down and r_m will rise through the evolution. This strategy ensures that the GA process will not reach premature convergence by bringing more diversity through high mutation rate.

4.5.2 Crossover

The idea of crossover method is to generate a child chromosome inheriting the most important sequence characters from the parent. Generally, a chromosome with duplicate application can be generated in the crossover process. To solve this problem, as in [8], we use the uniform permutation crossover [25] to generate two child chromosomes from two parents. First, we randomly generate a binary number of the same length as each chromosome. This number, which is composed of only 0s and 1s, is used as a mask, M, in the crossover process. Then we pick up two chromosomes from the candidates pool for crossover, as P1 and P2. As shown in Figure 7 [8-page4], there are two steps in the crossover process. First, we will copy the values in P1 to C1 at the ith positions where the M[i] = 1; then we'll try to fill out the rest of the parts of C1 by picking values from P2, which are not in C1 yet. This process will go from left to right in Both P2 and C1 to keep the same sequence in P2. The same process will be applied for C2, P1 and P2 except we will copy the values from P2 at the positions where the values equal to 0 in the mask M. [15]. Figure 16 shows the process diagrammatically.



j = 1

M =	1	0	1	1	0	1
P_1 =	3	2	4	5	6	1
P_2 =	6	2	1	5	3	4
C_1 =	3	6	4	5	2	1
C_2 =	4	2	5	6	3	1

Figure 16: Crossover Process, [8-Page4]

The flow chart in Figure 17 describes the implementation of this process.

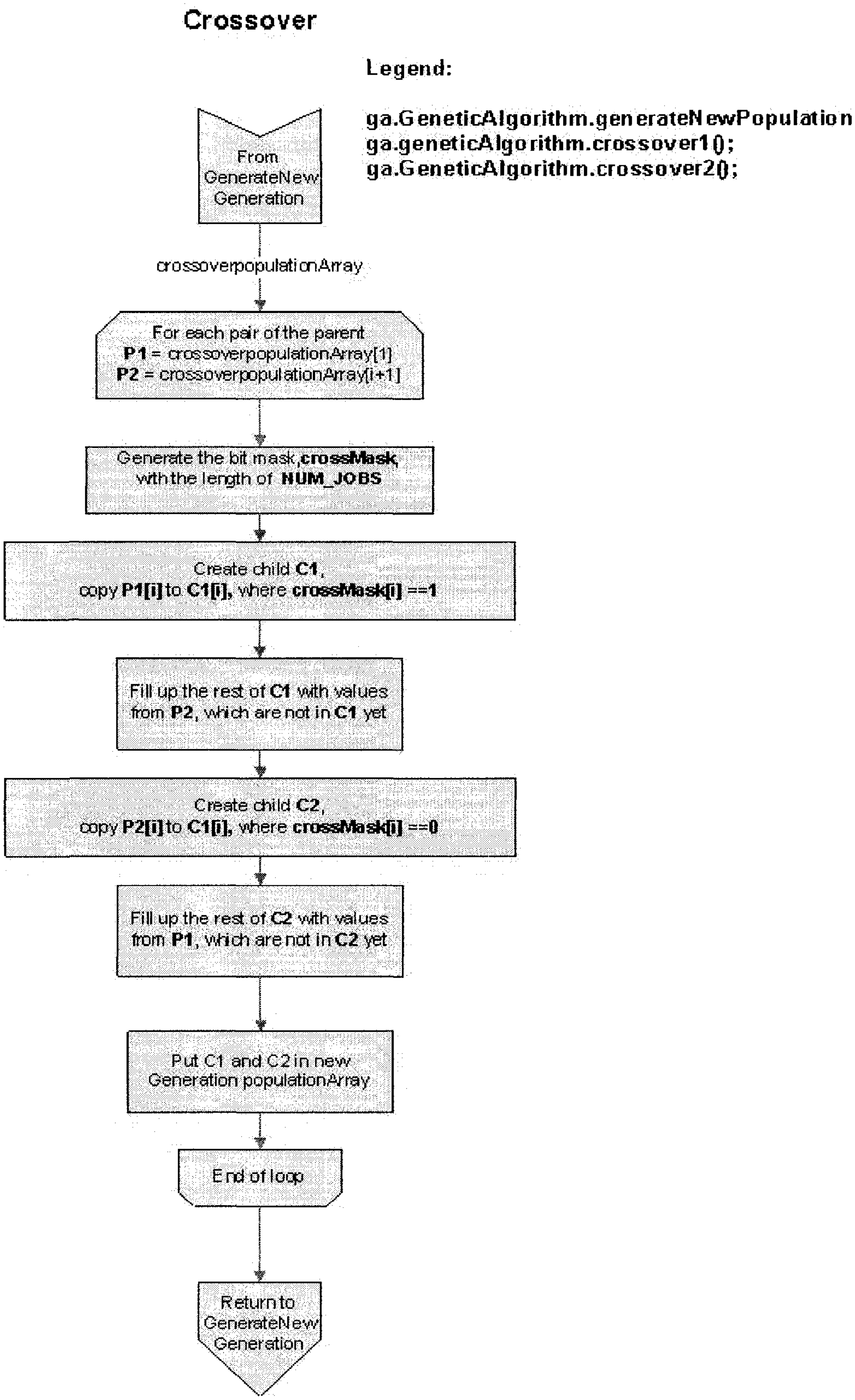

Crossover
Legend:
ga.GeneticAlgorithm.generateNewPopulation
ga.geneticAlgorithm.crossover1();
ga.GeneticAlgorithm.crossover2();
From GenerateNew Generation
crossoverpopulationArray
For each pair of the parent
P1 = crossoverpopulationArray[1]
P2 = crossoverpopulationArray[i+1]
Generate the bit mask,crossMask, with the length of NUM_JOBS
Create child C1,
copy P1[i] to C1[i], where crossMask[i] ==1
Fill up the rest of C1 with values from P2, which are not in C1 yet
Create child C2,
copy P2[i] to C1[i], where crossMask[i] ==0
Fill up the rest of C2 with values from P1, which are not in C2 yet
Put C1 and C2 in new Generation populationArray
End of loop
Return to GenerateNew Generation

Figure 17: Sub Flow - Crossover

4.5.3 Mutation

The goal of mutation is to bring diversity into the new generation to prevent the genetic algorithm from reaching local optimum too early. Here, we are using the same algorithm as in [8]. The process of mutation generates a random integer j for each position i in a parent, and switches the values between ith and jth positions to create a new child. By mutating the schedule of a parent, it generates a child with more differences. This ensures that the solutions generated by the GA process do not converge to a local optimum value. Since a high mutation rate may also prevent the algorithm from final convergence, a mutation rate r_m is picked very carefully and usually much less than the crossover rate r_c. Figure 18 gives the flow chart of the implementation of this process.

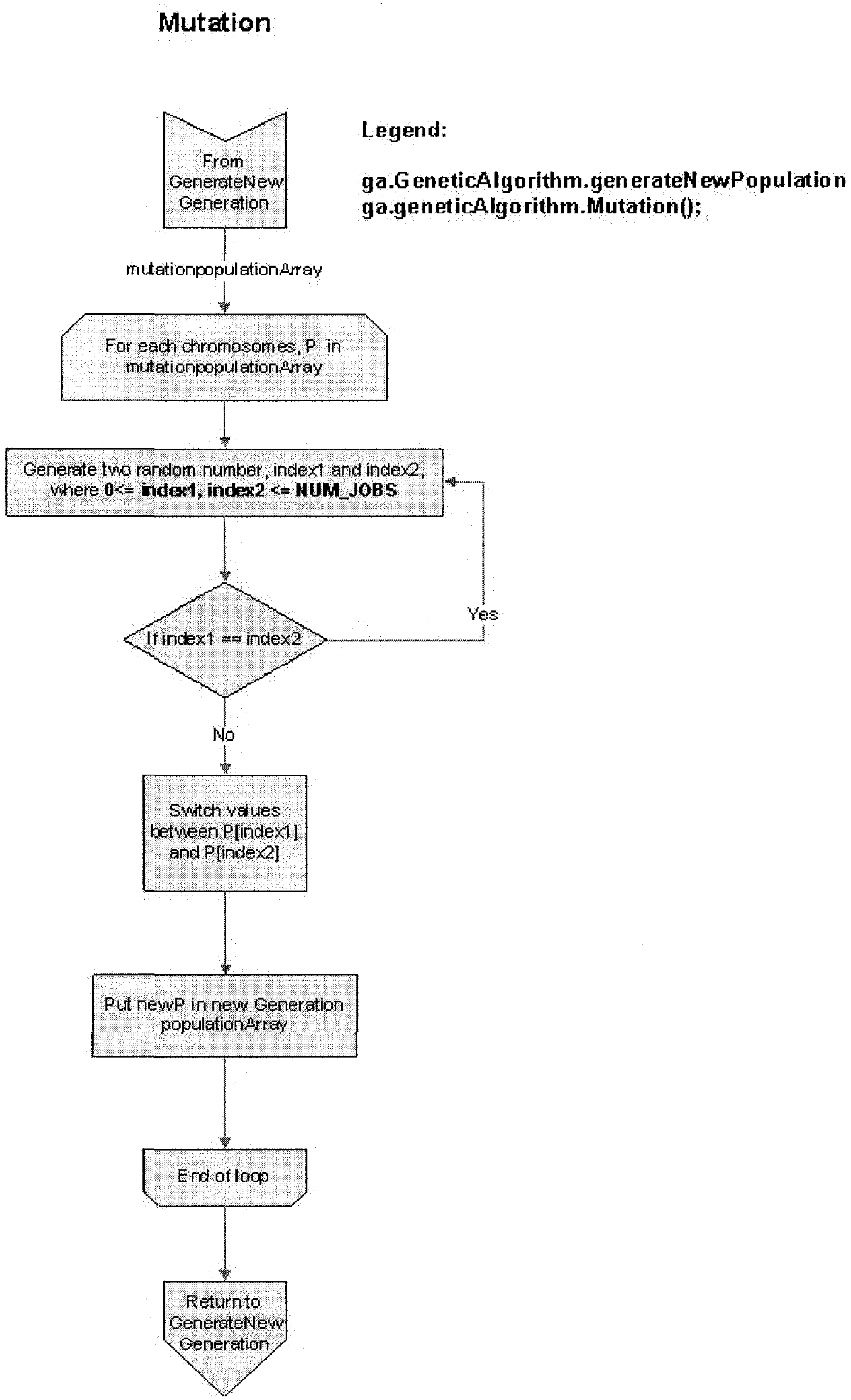

Mutation
Legend:
ga.GeneticAlgorithm.generateNewPopulation
ga.geneticAlgorithm.Mutation();
From GenerateNew Generation
mutationpopulationArray
For each chromosomes, P in mutationpopulationArray
Generate two random number, index1 and index2, where 0<= index1, index2 <= NUM_JOBS
If index1 == index2
Yes
No
Switch values between P[index1] and P[index2]
Put newP in new Generation populationArray
End of loop
Return to GenerateNew Generation

Figure 18: Sub Flow - Mutation

4.5.4 Elitism

The crossover process and mutation process will generate the same number of children as the parents. Since the process of obtaining a new generation implements a selection method to pick up chromosomes with better fitness value to generate children, the number of new generated chromosomes is usually less than the existing generation. In order to have a new generation with the same size, we implement the elitism method to make up the difference. It will pick up the remaining required chromosomes from the best existing ones, and put them into the new generation without any changes. This strategy makes sure the best chromosomes will be carried over in the whole process.

4.6 Further improvement

Tahera et. Al. [6-page2] have incorporated strategies, such as gender assignment; age assignment; and fertility rate in GA, while generating a new generation and got promising results in a mechanical design problem. Since GA derives its processes from natural selection and genetic production, which have been proved in human generation, there is a good reason to believe that introducing more features from the natural selection process will improve the proposed algorithm. Therefore we implement similar strategies of gender assignment; age assignment; and fertility rate based on [6].

4.6.1 Gender assignment

Unlike in general genetic algorithms, once a chromosome is created, it is endued with a gender of either male or female. As a result, this divides each generation into two groups. Just like in biological world, the mating, to reproduce children, can only occur between two chromosomes with different gender. The number of children generated by each parents is determined by the fertility rate, which is discussed in section 4.6.3.

The proposed algorithm uses genderProbabilityFactor to manage the proportion of males and females in each generation. Its values fall in the range between 0 and 1. When its value is 0, all chromosomes in this generation are male. Vice versa, each chromosome is endued with a female gender when its value is 1. In both conditions, our algorithm works just like a general genetic algorithm without consideration of gender.

In the proposed algorithm, the value of genderProbabilityFactor is set to 0.5. This means the proportion of males and females in each generation is exactly 50%.

4.6.2 Age assignment

Besides gender, a chromosome is assigned another attribute of age. In most of the previous works on GA, parents are discarded after they generate their children, and may not be carried into the new generation.

On the other hand, with continuous aged idea, parents may be carried into the new generation if they satisfy the fitness value condition, which means both the parents and children can exist in the same generation. In the proposed algorithm, each chromosome is initially assigned an age of 0 when it is born. Its age increases by 1, every time when it is carried into a new generation. Finally, once it reaches its death age, the chromosome dies, and will be discarded.

In the proposed algorithm, we define the death age = 3. Once a chromosome reaches 3, it will be discarded.

4.6.3 Fertility rate

We also introduce the fertility rate feature into each chromosome. It represents the ability of parents to generate children. Usually, the fertility rate is affected by the age of the chromosome. Like in the biological world, when parents are young, the fertility rate is high; parents are likely to generate more children. The fecundity of parents decreases as while the parents' age increases. Once it reaches a certain low threshold, fewer children are produced.

In the proposed algorithm, we define the fertility rate of a chromosome as a function of its age. When the age sum of parents is less than 4, they can generate 2 children as usual. Once the age sum reaches 4, the parents can only generate 1 child. Moreover, the generated children will inherit more features from the parent whose age is less than the others.

According to the experiment's results, it looks like these strategies do help our algorithm to change the genetic operators dynamically, and improve the performance on finding the near-optimum solution in an adaptive environment.

4.6.4 The updated algorithm

The pseudo code for the updated algorithm is as follows:

1. Generate initial generation of N chromosomes; endue a gender attribute to each chromosome according to genderProbabilityFactor; set its initial age of 0;

2. Calculate fitness value for each chromosome; calculate SD for this generation;

3. Generate first portion of Children by crossover method;

4. Generate second portion of Children by mutation method;

5. Generate third portion of Children by elitism method;

6. Calculate fitness value for each child;

7. Sort children and parents by their fitness values, increase a chromosome's age by 1 if it is parent;

8. Pick up the first N chromosomes whose age less than death age as new generation;

9. Repeat step 2 to 8 till SD reaches the predefined low threshold or the number of times reaches the limit.

10. Return the best chromosome as the near-optimum result.

5. Experiment and Expected Result

In order to achieve a good result, there are several parameters that must be chosen very carefully. In the experiment, we use the rules in [8], which did a lot of experiments to find the right parameters. Suppose there are n applications to be allocated, if n*(n-1) < 100, then the number of schedule in each generation is n*(n-1), otherwise it has been set to a constant value of 100.

5.1 Same Load, various environments

Basically, there are two main parameters that can be changed, the number of nodes; and the number of applications that need to be allocated. First, we test for the same number of applications, allocated on a distributed environment with various number of computes nodes. We have tested 10 applications on environment with nodes number 5, 10, 15, and 20. In order to ensure the data accuracy, we performed the same experiment three times. The experiment results are shown in Table 2:

Table 2: Experiment result of 10 applications on various numbers of nodes

Structure	Algorithm	Time of Experiments	EndTime	Improvement (%)
10Applications				
5Node	LB [7]		**2928**	0.00
	GA [8]	1	2808	4.10
		2	2807	4.13
		3	2810	4.03
	PA	1	2800	4.37
		2	2800	4.37
		3	2800	4.37
10Node	LB [7]		**1649**	0.00
	GA [8]	1	1465	11.16
		2	1468	10.98
		3	1464	11.22
	PA	1	1452	11.95
		2	1447	12.25
		3	1447	12.25
15Node	LB [7]		**1193**	0.00
	GA [8]	1	1007	15.59
		2	1009	15.42
		3	1009	15.42
	PA	1	1005	15.76
		2	1005	15.76
		3	1005	15.76
20Node	LB [7]		**959**	0.00
	GA [8]	1	797	16.89
		2	791	17.52
		3	796	17.00
	PA	1	791	17.52
		2	791	17.52
		3	791	17.52

In the experiments, we have tested three algorithms: LB [7] represents the original LB algorithm without any scheduling strategy in [7]; GA [8] represents the standard genetic algorithm used in [8]; and PA is our proposed algorithm with adaptive features. Column 'EndTime' records the session completion time of each experiment. And the final 'Improvement' column records the percentage improvements achieved by both GA [8] and our proposed algorithm to the original algorithm without scheduling. The comparison results are shown in Table 3.

Table 3: Comparison Result of 10 Applications on various number of Nodes

Num of Nodes	Algorithms	Improvement1 (%)	Improvement2 (%)
5Node	GA [8]	4.03	0.34
	PA	4.37	
10Node	GA [8]	10.98	1.27
	PA	12.25	
15Node	GA [8]	15.42	0.34
	PA	15.76	
20Node	GA [8]	16.89	0.63
	PA	17.52	

In Table 3, there are two improvement columns: Improvement1 represents the improvement of both standard GA in [8] and our proposed algorithm from the original algorithm; and Improvement 2 records the improvement of our proposed algorithm from the standard GA. Both improvements are percentage perspective. From the data above, we can reach three conclusions:

1. In each scenario, both our proposed algorithm and the original GA algorithm in [8] scheduling did reduce the session completion time, and improves the resource utilization.
2. As the available resources rise, the scheduling algorithms will give better results. A range from 4.03% to 17.52% percentage improvement has been achieved.
3. The proposed algorithm consistently performs better than the standard GA. Figure 19 shows the results:

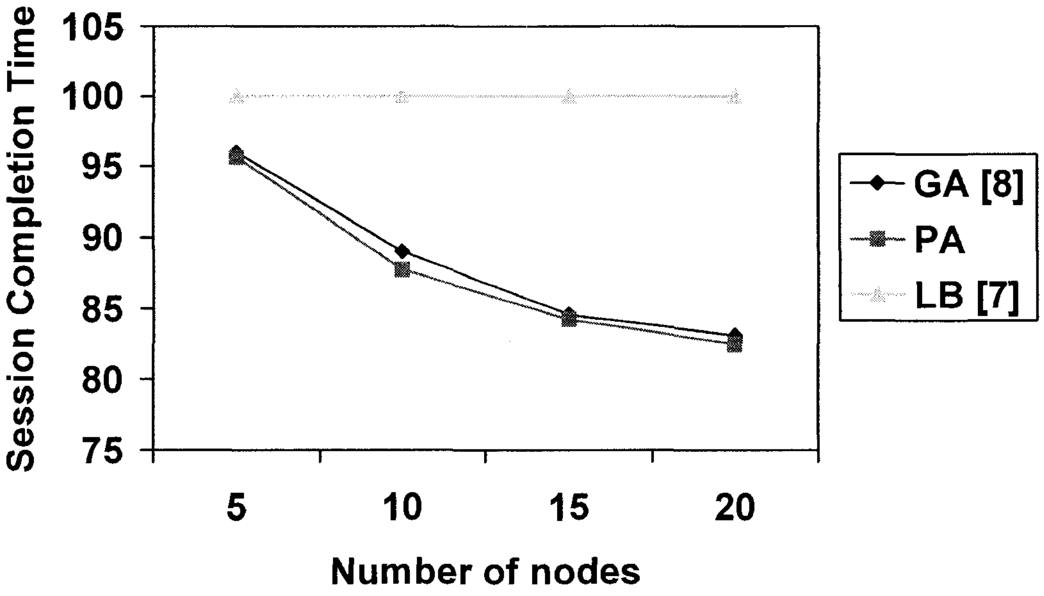


Figure 19: Generalized SCT of 10 Applications based on LB [7]

5.2 Same environment, various loads

Secondly, we tested our algorithm on the same distributed environment with different number of applications. In the experiments, we chose 15 as the number of nodes, and the numbers of applications are 10, 15, 20, and 25. Table 4 shows the results of the experiments:

Table 4: Experiment result of various num applications on same numbers of nodes

Structure	Algorithm	Time of Experiments	EndTime	Improvement1 (%)
10App				
15Node	LB [7]		**1193**	0.00
	GA [8]	1	1007	15.59
		2	1009	15.42
		3	1009	15.42
	PA	1	1005	15.76
		2	1005	15.76
		3	1005	15.76
15App				
15Node	LB [7]		**1517**	0.00
	GA [8]	1	1453	4.22
		2	1455	4.09
		3	1454	4.15
	PA	1	1427	5.93
		2	1428	5.87
		3	1427	5.93
20App				
15Node	LB [7]		**2210**	0.00
	GA [8]	1	1947	11.90
		2	1946	11.95
		3	1949	11.81
	PA	1	1908	13.67
		2	1900	14.03
		3	1890	14.48
25App				
15Node	LB [7]		**2574**	0.00
	GA [8]	1	2453	4.70
		2	2452	4.74

		3	2450	4.82
	PA	1	2331	9.44
		2	2331	9.44
		3	2332	9.40
35App				
15Node	LB [7]		3478	0.00
	GA [8]		3376	2.93
	PA		3186	8.40
45App				
15Node	LB [7]		4332	0.00
	GA [8]		4273	1.36
	PA		4022	7.16

Table 5 shows the relative improvements obtained from GA[8] and the proposed algorithm in all these experiments.:

Table 5: Comparison Result of various num of apps on constant num of Nodes

Num of Apps	Algorithms	Improvement1 (%)	Improvement2 (%)
10	GA [8]	15.42	
	PA	15.76	0.34
15	GA [8]	4.09	
	PA	5.93	1.85
20	GA [8]	11.81	
	PA	14.48	2.53
25	GA [8]	4.70	
	PA	9.44	4.74
35	GA [8]	2.93	
	PA	8.40	5.46

45	GA [8]	1.36	
	PA	7.16	5.79

As in Table 3, the first improvement column in Table 5 shows the percentage improvement of both standard GA in [8] and the proposed algorithm wrt the LB algorithm without any scheduling. The second improvement column records the improvement of the proposed algorithm wrt the standard GA. First, the proposed algorithm consistently performs better than the standard GA. In a distributed environment, as the workload increases, this improvement becomes greater.. There is no improvement in the scenario when the load is fixed and the available resources increase. Figure 20 shows the trend lines:

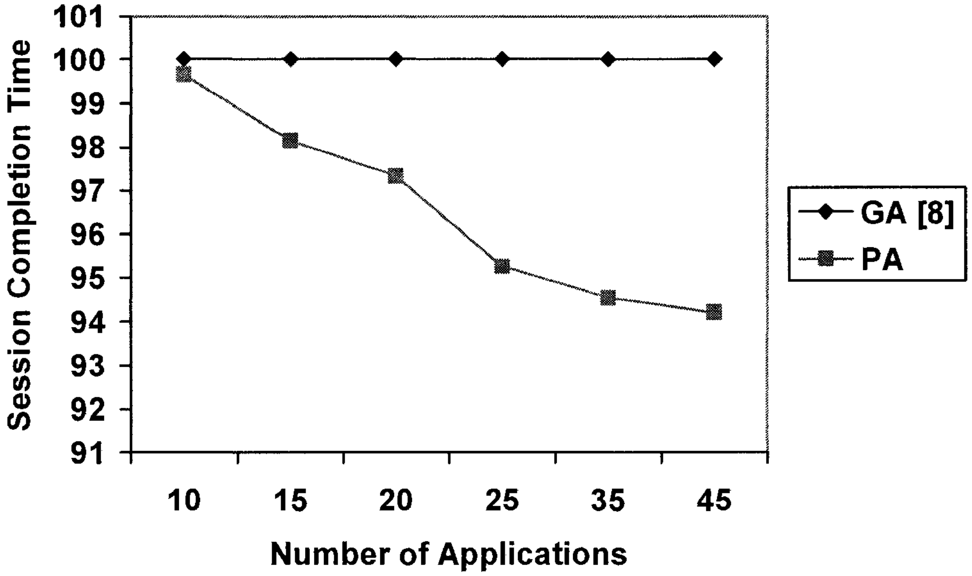


Figure 20: Generalized SCT on 15 Nodes based on GA [8]

5.3 Implementing Gender and Age assignments

Since the proposed algorithm is able to take care of distributed systems, with a heavy computational load, a new generated work load pool of 200 applications was created for testing. We test our algorithm on a fixed distributed environment of 25 nodes. In each experiment, we vary the numbers of applications from 10, 15, 20 to 25 30 35 and 40. Please see the data of the experiment results in Table 6:

Table 6: Experiment result of various num applications on 25 nodes

25Nodes						
Num Of Apps	Algorithms	Num of times	Session completion time	Impv1 (%)	Impv2 (%)	Impv2 (%)
10Apps	LB [7]		**922**	0.00		
	GA [8]	1	675	26.79		
		2	680	26.25		
		3	675	26.79		
	PA	1	671	27.22		
		2	670	27.33		
		3	671	27.22	1.47	
	PA with G		669	27.44	1.62	0.15
15Apps	LB [7]		**1059**	0.00		
	GA [8]	1	911	13.98		
		2	910	14.07		
		3	911	13.98		
	PA	1	885	16.43		
		2	883	16.62		
		3	883	16.62	3.07	
	PA with G		880	16.90	3.40	0.33
20Apps	LB [7]		**1386**	0.00		
	GA [8]	1	1231	11.18		
		2	1231	11.18		
		3	1230	11.26		
	PA	1	1177	15.08		
		2	1177	15.08		
		3	1179	14.94	4.39	
	PA with G		1170	15.58	4.96	0.57
30Apps	LB [7]		**1864**	0.00		
	GA [8]	1	1727	7.35		
		2	1717	7.89		
		3	1715	7.99		

	PA	1	1697	8.96		
		2	1686	9.55		
		3	1693	9.17	1.74	
	PA with G		1686	9.55	2.37	0.64
40Apps	LB [7]		**2367**	0.00		
	GA [8]	1	2252	4.86		
		2	2243	5.24		
		3	2236	5.53		
	PA	1	2225	6.00		
		2	2223	6.08		
		3	2223	6.08	1.20	
	PA with G		2204	6.89	2.13	0.93

The format of the result data is almost the same as in the previous experiments. We tested different algorithms on the same scenario, and compared the session completion time. The extra row with algorithms PA with G is used to record the experiment result of the proposed algorithm with the implementation of Gender and Age assignment. The last column Impv3 records the percentage improvement of PA with G over the previous proposed algorithm PA. The trend lines of the improvement are shown in Figure 21:

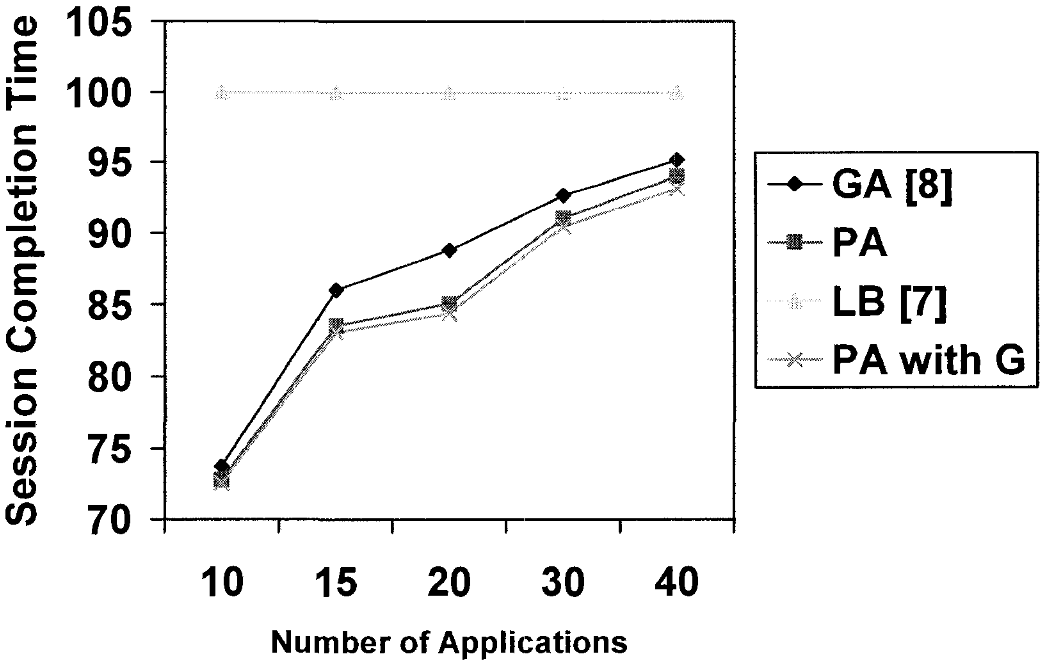


Figure 21: Generalized SCT on 25 Nodes based on LB [7]

As mentioned before , Table 6 introduced an extra row in each scenario storing the experiment result data produced by the proposed algorithm implemented with Gender and Age assignments. Table 7 shows the analysis to understand the effect of introduction of the gender and age assignments:

Table 7: Comparison Result between PA with Gender and PA without Gender

25Nodes			
Num Of Apps	Algorithms	Impv2 (%)	Impv2 (%)
10Apps	PA	1.47	
	PA with G	1.62	0.15
15Apps	PA	3.07	
	PA with G	3.40	0.33
20Apps	PA	4.39	

	PA with G	4.96	0.57
30Apps	PA	1.74	
	PA with G	2.37	0.64
40Apps	PA	1.20	
	PA with G	2.13	0.93

In Table 7, the column Impv2 is the percentage improvement of the proposed algorithm with gender and age assignments over the previous version. The implementation of the gender and age assignment leads to a consistent improvement in performance. Figure 22 shows the trend lines of the improvement.

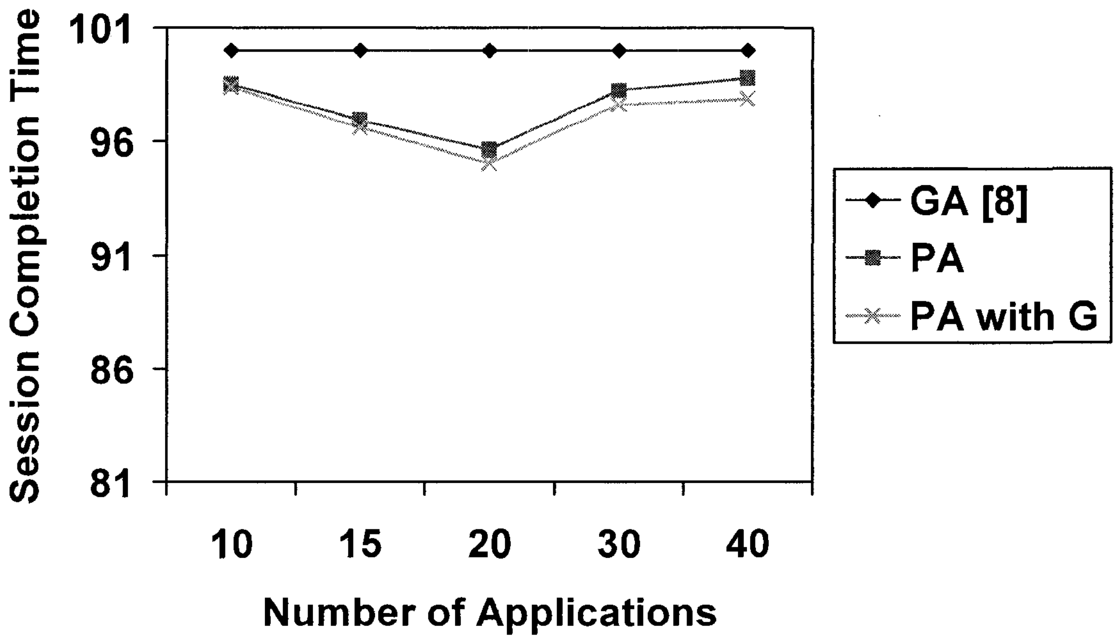


Figure 22: Generalized SCT on 25 nodes based on GA [8]

In order to find the improvement range, we tested the two algorithms with more workloads. Table 8 shows the results of the experiments:

Table 8: Percent Improvement after implementing Gender and Age assignments

Num Of Apps	Session completion time		
	PA	PA with G	Improvement (%)
10	670	669	0.15
20	1177	1170	0.57
30	1697	1686	0.64
40	2225	2204	0.93
50	3643	3582	1.67
60	4348	4260	2.02
70	5161	5052	2.11
80	5960	5796	2.75
90	6750	6533	3.21
100	7501	7243	3.44
110	8276	7981	3.56
120	8977	8654	3.60

The trend lines are shown in Figure 23 as below:

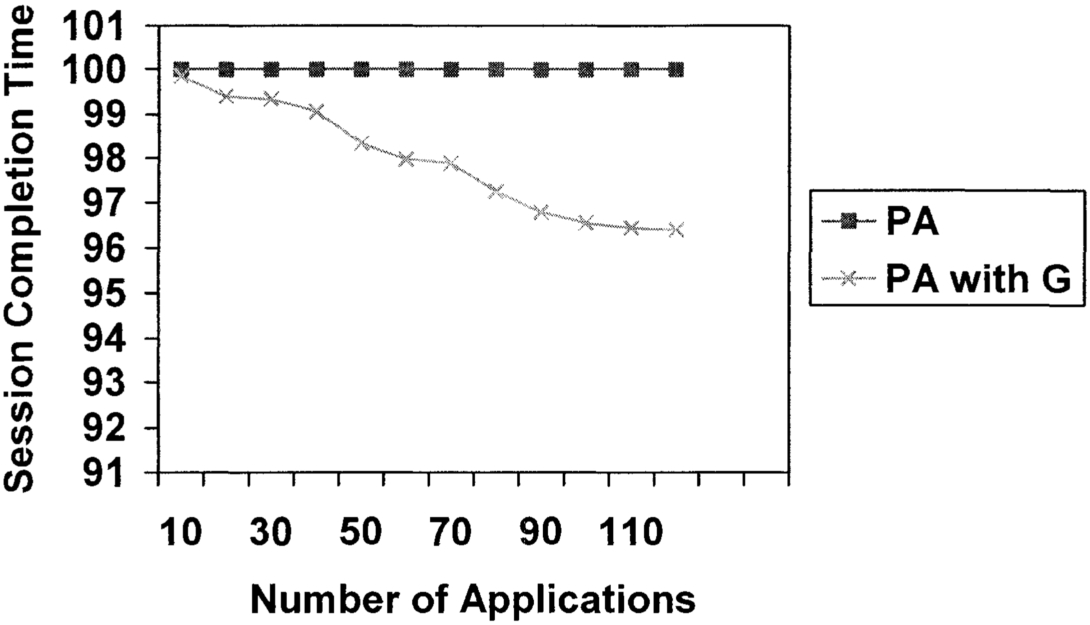


Figure 23: Generalized SCT on 25 nodes based on PA without Gender and Age

From the figure23, we can derive the following conclusions:

1. Implementing Gender assignment improves the performance of the proposed algorithm;
2. This improvement keeps rising as the work load increases and the improving trend reduces towards zero, once the work load reaches a threshold.

6. Conclusions and future work

In this thesis, we have proposed a genetic algorithm based scheduling system for the application centric loading balancing in a distributed heterogeneous environment.

In our research, we have considered the generalized distributed system with heterogeneous compute-nodes. We also take the network delays into consideration, and deal with the allocation problem on a distributed heterogeneous network. From the work load perspective, the proposed algorithm deals with the allocation of direct acyclic graph (DAG) kind of generalized workload, which is composed of sub processes with internal dependencies. All of these ensure that the proposed algorithm attempts to solve problems of the real world.

In the genetic algorithm, we have introduced certain adaptive features. By dynamically changing the genetic parameters through the whole evolution process, it balances the local and global searching in the process on the run. By doing this, the algorithm prevents premature convergence or convergence stagnation, and directs the process towards optimal convergence more efficiently.

Since GA is originally developed from natural selection and genetic production, we have introduced more features from the biological world into the proposed algorithm. Using the ideas used for a mechanical design problem [6], we implemented gender and aging strategies in the proposed algorithm, and managed the child generation according to the parents' fertility rate.

We have designed the system, based on object oriented concept. So with moderate efforts, experiments with more features and new strategies for optimization can be performed.

Due to its inherited nature, genetic algorithm is more time consuming than other heuristic algorithms. Hence if the proposed algorithm is implemented into a distributed system of compute-nodes, the scheduling process can become faster.

Bibliography

[1]. Kai Lu and Albert Y. Zomaya, A Hybrid Policy for Job Scheduling and Load Balancing in Heterogeneous Computational Grids, *Sixth International Symposium on Parallel and Distributed Computing, IEEE* 2007

[2]. Grzegorz Malewicz, Ian Foster, Arnold L. Rosenberg, Michael Wilde, A Tool for Prioritizing DAGMan Jobs and Its Evaluation, 2006

[3]. Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud legrand, Loris Marchal and Yves Robert, Centralized versus distributed schedulers for multiple bag-of-task applications, *IEE proceedings-E*, vol. 138, no. 5, 2006, Page(s): 313-318

[4]. Lu Tian and K. Mani Chandy, Resource Allocation in Streaming Environmnets, *Grid Computing Conference,* 2006

[5]. Lan Zhou, Sun Shi-Xin,A Self-Adaptive Genetic Algorithm for Tasks Scheduling in Multiprocessor System, *IEEE proceedings*, 2006

[6]. Khadiza Tahera, Raafat N. Ibrahim, Paul B. Lochert Adopting dynamic operators in a Genetic Algorithm , *Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM* 2007, Page(s): 1533-1540

[7]. A. K. Aggarwal, Brajendra K. Singh and Kemal Tepe, Communication net Load Balancing for Distributed Bus networks, 2005

[8]. Mona Aggarwal, Robert D. Kent and Alioune Ngom, Genetic Algorithm Based Scheduler for Computational Grids, *Proceedings of the 19th Conference,* 2005

[9]. Foster, I. and Kesselman, C. (eds.), The Grid: Blueprint for a New Computing Infrastructure, 2003.

[10]. Ian Foster, Carl Kesselman, Steven Tuecke, The Anatomy of The grid: Enabling Scalable Virtual Organizations, *International Journal of Supercomputer Applications*, 2001.

[11]. H. S. Bhatt, B. K. Singh, A. K. Aggarwal, "Application centric load balancing strategy based on hybrid model", *IXth International Conference on Advanced Computing & Communication, ADCOM-2001, organised by IEEE and ACS India*, 2001, Page(s): 231-238.

[12]. Shaout A., McAuliffe P., Job scheduling using fuzzy load balancing in distributed system, *Electronics Letters,* vol. 34, 1998, Page(s): 1983-1985

[13]. Chulhye Park, Kuhl J.G., A fuzzy-based distributed load balancing algorithm for large distributed systems, Autonomous Decentralized Systems, *Proceedings. ISADS 95, Second International Symposium,* 1995, Page(s): 266-273

[14]. Baumgartner K.M., Wah B.W., A global load balancing strategy for a distributed computer system, *Distributed Computing Systems in the 1990s, 1988. Proceedings., Workshop on the Future Trends of*, 1988, Page(s): 93-102

[15]. A. Ngom, 1997, Genetic algorithms for the jump number scheduling problem, Order - *A Journal on the Theory of Ordered Sets and its Applications*, 1998, Page(s): 59-73.

[16]. Gopal, S., Vajapeyam, U., Load balancing in a heterogeneous computing environment, System Sciences, *Proceedings of the Thirty-First Hawaii International Conference on,* vol. 7, 1998, Page(s): 796 -804

[17]. Andersen P.H., Antonio J.K., Implementation and utilization of a heterogeneous multicomputer cluster for the study of load balancing strategies, High Performance Distributed Computing, 1998. Proceedings. *The Seventh International Symposium on* , 1998, Page(s): 362-363

[18]. Zhang Y., Kameda H., Hung S.L., Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems, *Computers and Digital Techniques, IEE Proceedings*, vol. 144, 1997, Page(s): 100-106

[19]. Kun-Ming Yu, Wu, S.J.-W., Tzung-Pei Hong, A load balancing algorithm using prediction, Parallel Algorithms/ Architecture Synthesis, *Proceedings, Second Aizu International Symposium*, 1997, Page(s): 159-165

[20]. M. Coli, P. Palazzari, "Load Balancing with Internode Precedence Relations: A New Method for Static Allocation of DAGs into Parallel Systems", *Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing*, PDP 1996, pp 252-257 16. Malloy

[21]. Borzemski L., Load balancing in parallel and distributed processing of tree-based multiple-task jobs, Parallel and Distributed Processing, *Proceedings. Euromicro Workshop on*, 1995, Page(s): 98-105

[22]. Haresh Bhatt, Mritunjay, M.K. Shah, N.P. Darji, and CVS Prakash, Optimization of image processing software on VAX/VMS system configuration, *Proceedings of Seminar on Supercomputing for Scientific Visualization*, 1994

[23]. Yongbing Zhang, Kameda H., Shimizu K., A Comparison Of Adaptive And Static Load Balancing Strategies By Using Simulation Methods, intelligent Control and Instrumentation, *Proceedings., Singapore International, Conference on* , vol. 2, 1992, Page(s): 1162 -1167

[24]. Sarje A.K., Sagar G., Heuristic model for task allocation in distributed computer systems, *IEEE proceedings-E,* vol. 138, no. 5, 1991, Page(s): 313-318

[25]. Goldberg D.E, Genetic Algorithm in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, 1989.

Vita Auctoris

Name:	Sheng Bai
PLACE OF BIRTH:	Beijing, China
YEAR OF BIRTH:	1974
EDUCTION:	Beijing University of Technology, Beijing, China 1993-1998 BEng University of Windsor, Windsor, Ontario, Canada 2003-2008 M.Sc.