

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]



**PARALLEL GENETIC SCHEDULING FOR
PARALLEL APPLICATIONS**

BY

AHMED OMER SALEH BIN MAHFOOD

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In
COMPUTER SCIENCE

MAY 2001

UMI Number: 1409812

UMI[®]

UMI Microform 1409812

**Copyright 2002 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.**

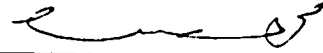
**ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346**

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **AHMED OMER SALEH BIN MAHFOOD** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee



Dr. Muhammad Al-Suwaiyel (Chairman)



Dr. Sadiq Sait (Co-Chairman)



Dr. Muhammad Sarfraz (Member)

Kanan Faisal 3/10/2001
Department Chairman


Dean of Graduate Studies

١٤٢٢/٤/١٦
Date



Dedicated to

My Parents

and

My Grand Mother

Acknowledgments

All praises be to Allah for his limitless help and guidance that enable me to complete this work. Peace and blessings of Allah be upon his prophet Muhammad.

I acknowledge the generous help and support provided by the King Fahd University of Petroleum and Minerals, and its Information and Computer Science Department for this research.

I would like to thank my thesis advisor, Dr. Muhammad Al-Suwaiyel, for his guidance and support. I would also like to thank Dr. Sadiq Sait and Dr. Muhammad Sarfraz for their consistent support and cooperation.

I also wish to express my profound gratitude and appreciation to Dr. Muslim Bozyigit with whom I started this thesis. But due to extraordinary circumstances I could not complete it with him. My interest in the field of Parallel Processing is because of the course I took with him.

Finally, and most importantly thanks to my parents and other family members for their love, support, and encouragement through the years.

Contents

List of Tables	viii
List of Figures	x
Abstract (English)	xi
Abstract (Arabic)	xii
1 Introduction	1
1.1 Parallel Processing Architectures.....	3
1.2 Genetic Algorithms	8
1.2.1 Genetic Algorithm Components.....	10
1.2.2 Parallel Genetic Algorithms	13
1.3 Thesis Outline	19
2 Problem Definition	21
2.1 Parallel Computation Models.....	21
2.2 Parallel Systems	22
2.3 The Scheduling Problem.....	24

2.4	Scheduling Taxonomy	25
2.5	Thesis Objective	28
3	Literature Review	30
3.1	Static Scheduling Approaches.....	30
3.2	Dynamic Scheduling Approaches	41
3.3	Parallel Scheduling Approaches.....	44
4	Sequential Genetic Scheduling Algorithm	48
4.1	Test Problems Generation.....	48
4.2	Design of the LGA	52
4.2.1	String Representation.....	52
4.2.2	Fitness Function.....	53
4.2.3	Initial Population	55
4.2.4	Reproduction	56
4.2.5	Crossover.....	56
4.2.6	Mutation	57
4.2.7	Control Parameters.....	57
4.3	Experimental Results.....	59
5	Parallel Genetic Scheduling Algorithm	68
5.1	Design of the PLGA	68
5.2	Computational Environment.....	74
5.2.1	PVM Overview.....	76

5.2.2	Implementation Issues.....	77
5.3	Experimental Results.....	78
6	Conclusion	87
	References	90

List of Tables

4.1	Task graphs generated by Random Parallel Computation Generator (RPCG).....	50
4.2	Task graphs generated by Optimal Parallel Computation Generator (OPCG).....	51
4.3	Comparison of HGA and LGA for task graphs generated by RPCG on 8-processor fully connected topology.....	62
4.4	Comparison of HGA and LGA for task graphs generated by RPCG on 8-processor mesh topology	63
4.5	Relative comparison of HGA and LGA for task graphs generated by RPCG on fully connected topology of different number of processors (A_p).....	65
4.6	Comparison of HGA and LGA for task graphs generated by OPCG.....	66
5.1	Comparison of LGA and PLGA for task graphs generated by RPCG on 8-processor fully connected topology ($SP_s = P_s/W$).....	72
5.2	Comparison of LGA and PLGA for task graphs generated by RPCG on 8-processor fully connected topology ($SP_s = P_s/W + 0.4P_s$).....	73

5.3	Comparison of migration frequencies (<i>MF</i>) for task graphs generated by RPCG on 8-processor fully connected topology.....	75
5.4	Ratios of schedule lengths generated by PLGA to those of LGA for task graphs generated by RPCG on 8-processor fully connected topology.....	81
5.5	Ratios of schedule lengths generated by PLGA to those of LGA for task graphs generated by RPCG on 8-processor mesh topology.....	82
5.6	The speedup in the running times of PLGA over LGA for 8-processor fully connected topology	83
5.7	The speedup in the running times of PLGA over LGA for 8-processor mesh topology	84
5.8	Comparison of LGA and PLGA in terms of the quality of the schedules for task graphs generated by OPCG	86

List of Figures

1.1	An array processor system	4
1.2	A multiprocessor system.....	6
1.3	A multicomputer system.....	7
2.1	Example of (a) A task graph with communication costs (b) A parallel system.....	23
2.2	Scheduling taxonomy	27
4.1	String representation.....	54
4.2	An example of crossover operation.....	58
4.3	Structure of the level-based genetic algorithm (LGA)	60
4.4	Example where the optimal schedule does not satisfy the height constraints	67
5.1	Structure of the parallel level-based genetic algorithm (PLGA).....	70
5.2	Implementation of the parallel algorithm	79

Thesis Abstract

Name: AHMED OMER SALEH BIN MAHFOOD

Title: PARALLEL GENETIC SCHEDULING FOR PARALLEL APPLICATIONS

Degree: MASTER OF SCIENCE

Major Field: COMPUTER SCIENCE

Date of Degree: MAY 2001

In this thesis, a parallel genetic algorithm for scheduling parallel tasks on a multiprocessor system is proposed. We first designed a fast and efficient sequential genetic algorithm upon which the parallel genetic algorithm developed. The parallel genetic algorithm is based on the island model of GA where a population is divided into a number of independent subpopulations. Each subpopulation is evolved by an independent GA and periodically fit strings migrate between the subpopulations. The parallel genetic algorithm was implemented on a network of SUN workstations. It has been observed through the experiments that generally the schedules generated by the parallel genetic algorithm are better than the schedules generated by the sequential genetic algorithm. Furthermore, the parallel genetic algorithm achieved some speedup over the sequential genetic algorithm. The speedup increases as the number of workstations in the parallel machine increases.

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

May 2001

ملخص الرسالة

اسم الطالب : احمد عمر صالح بن محفوظ

عنوان الرسالة : جدولة جينية متوازية للتطبيقات المتوازية

التخصص : علوم الحاسب الآلي

تاريخ التخرج : مايو ٢٠٠١م

في هذه الرسالة، تم تصميم خوارزمية جينية متوازية للقيام بجدولة التطبيقات المتوازية على نظام متعدد المعالجات. لقد قمنا أولاً بتصميم خوارزمية جينية متتابعة سريعة وفعالة ثم بنينا عليها الخوارزمية الجينية المتوازية. الخوارزمية الجينية المتوازية تعتمد على نوعية من الخوارزميات الجينية تسمى بالجزر، حيث إن مجموعة الحلول تنقسم إلى عدة مجموعات. كل مجموعة تتطور بواسطة خوارزمية جينية منفصلة ومن حين لآخر تتبادل المجموعات الحلول الجيدة. الخوارزمية الجينية المتوازية جربت على شبكة من أجهزة (SUN). ولقد لوحظ من خلال النتائج إن الجداول التي تنتجها الخوارزمية الجينية المتوازية أفضل بشكل عام من الجداول التي تنتجها الخوارزمية الجينية المتتابعة. علاوة على ذلك إن الخوارزمية الجينية المتوازية تحقق بعض التسارع على الخوارزمية الجينية المتتابعة وان التسارع يزيد بزيادة عدد الأجهزة في نظام المعالجة المتوازية.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران، المملكة العربية السعودية

مايو ٢٠٠١م

Chapter 1

Introduction

During the past years, dramatic increases in computing speed were achieved. Most of these achievements in computing speed were due to the use of inherently faster electronic components. However, the physical limitation imposed by fundamental electrical properties makes it impossible to achieve further dramatic improvements in the speed of such uniprocessor computers. Thus, the scientists started to examine a new data-processing approach, which is generally known as *parallel processing*. The idea here is that if several operations that form a computation can be performed simultaneously then the time taken by the computation can be significantly reduced. To implement the parallel processing approach we should use the combined power of more than one processor.

There are two reasons for the popularity and the attractiveness of parallel processing. Firstly, the declining cost of computer hardware has made it possible to assemble parallel machines with thousands of processors. Secondly, the availability of applications which are beyond the capability of conventional computers. Ocean resource

exploration, numerical weather forecasting, airplanes and space vehicles modeling, genetic engineering, interactive graphics, and managing large databases are few examples of these applications. Without using parallel processing, many of these applications to advance human civilization could hardly be realized.

The performance of parallel processing systems depends mainly on the operating system. This is because the operating system performs, in addition to its usual issues, the task of scheduling a given set of computation tasks on the set of available processors in order to maximize the throughput of the parallel system. Thus, the efficiency of the whole system largely depends on the efficiency of the scheduling policy used by the operating system.

The problem of scheduling parallel tasks is one of the most challenging problems in parallel processing systems whose optimal solution is known to be NP-complete. Therefore, various methods have been proposed in the literature to find an acceptable schedule. These methods can be classified as *heuristic-based* or *search-based*. The heuristic-based methods are inexpensive in terms of their execution times, but they may not provide good quality solutions. The *Earliest Task First* (ETF) and *Dynamic Critical Path* (DCP) are two examples of such methods. The search-based methods produce good quality solutions but they are computationally very expensive. Some such methods are Genetic Algorithms, Simulated Annealing, and Tabu Search.

It is desirable to execute the scheduling policy as fast as possible. Therefore, the execution time of the scheduling policy is a major overhead that should be minimized especially when the scheduling policy belongs to the search-based methods. Thus, we set the objective of this thesis to minimize this overhead by developing a practical parallel

algorithm for scheduling parallel applications.

The rest of this chapter is organized as follows. In Section 1.1, parallel processing architectures are briefly discussed. The components of genetic algorithms as well as different parallel models of these algorithms are described in Section 1.2. Finally, thesis outline is given in Section 1.3.

1.1 Parallel Processing Architectures

Parallel computers can be divided into three architectural categories as discussed in [22]. The first category is called *array processor system*. Such architecture employs a central control unit, multiple arithmetic logic units called processing elements (PE), and an interconnection network among the PEs, as shown in Figure 1.1 [34]. The control unit broadcasts array instructions to all processing elements and all active processing elements execute the same instruction at the same time but on different data. These data are fetched from local memories of the processing elements. This is why this architecture is also called *single-instruction multiple-data* (SIMD) architecture. The interconnection network facilitates data communication among processing elements. Array processor systems are mainly developed to perform parallel computations on vector or matrix types of data such as matrix multiplication, summation of vector elements, and parallel sorting.

The second category is centered by so-called *multiprocessor system*. This system consists of two or more comparable processors that can execute independent instruction streams using local data. Therefore, this system belongs to *multiple-instruction multiple-*

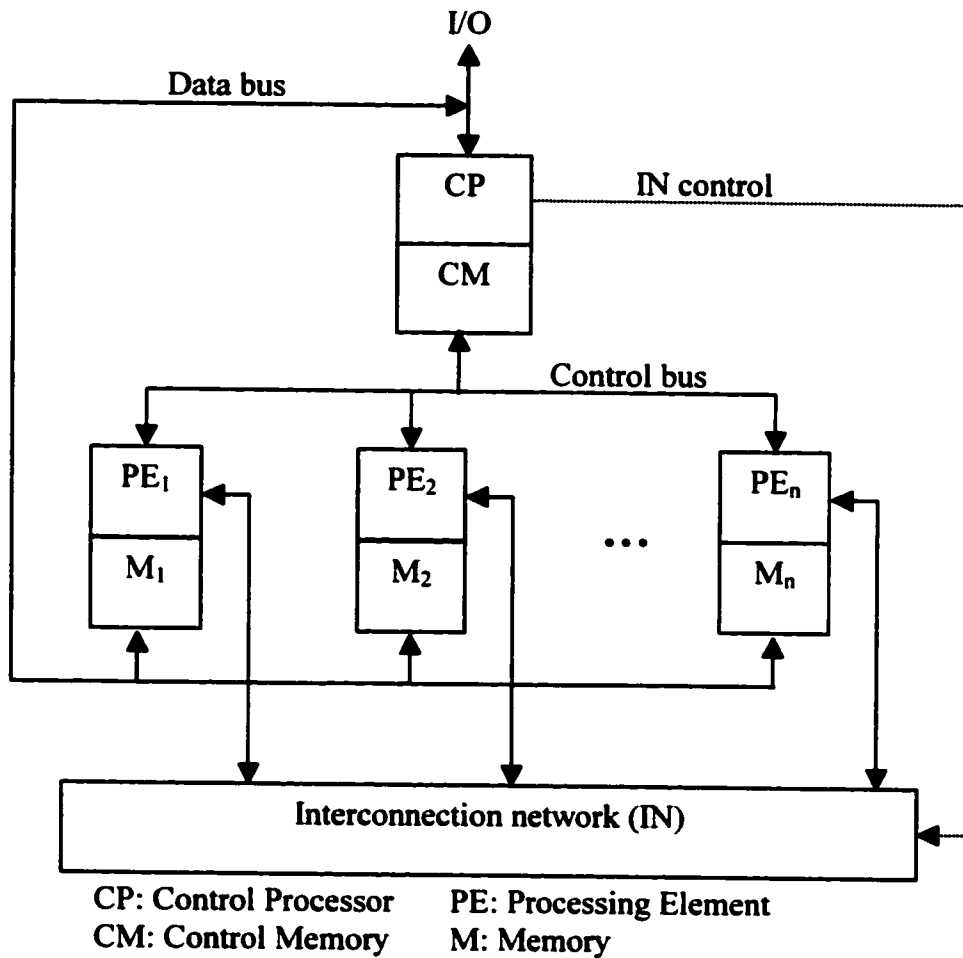


Figure 1.1: An array processor system

data (MIMD) class of computers. In this system, there is a shared memory that can be accessed by all processors in the system. This shared memory is usually partitioned into several modules each connected to the interconnection network, as shown in Figure 1.2 [34]. Processors communicate by read and write accesses to the shared memory. Different interconnection networks have been used to connect the processors and memories of these architectures. Crossbar switches and common buses are two examples of such interconnection networks.

The third architecture is called *multicomputer system*, which also belongs to MIMD class of computers. In a multicomputer system, each processor has its own memory called local or private memory, which is accessible only to that processor. The communication between processors is achieved via messages exchanged directly between them through an interconnection network. Multicomputer systems typically use interconnection networks with direct, point-to-point connections between processors. These interconnection networks include ring, tree, mesh, and hypercube. A multicomputer system is shown in Figure 1.3 [34]. The processors of a multiprocessor system access the memory modules that constitute the shared memory much more frequently compared to the communication among processors of a multicomputer system. Therefore, the terms *tightly coupled* and *loosely coupled* have been associated with multiprocessor and multicomputer systems, respectively.

With advances in networking and communication technologies, new line of the multi-computer systems is emerging. These systems are based on a network of workstations. Although each workstation is normally used as a separate computer, many workstations interconnected by a local area network (LAN) can be viewed as a multicom-

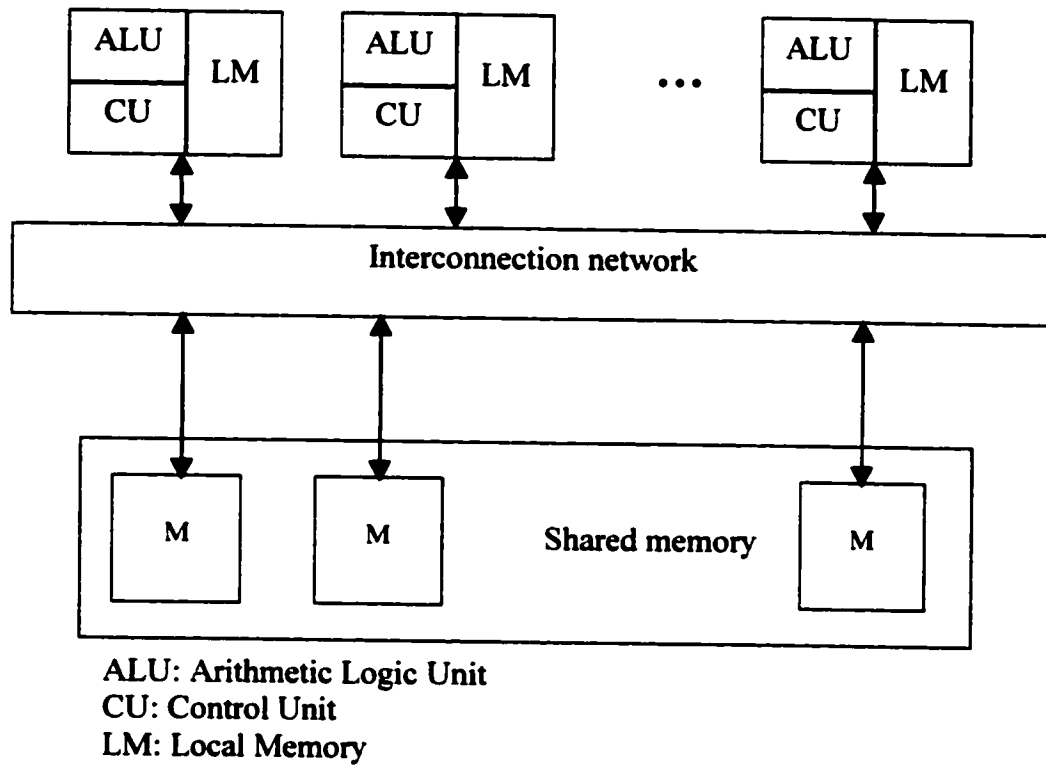
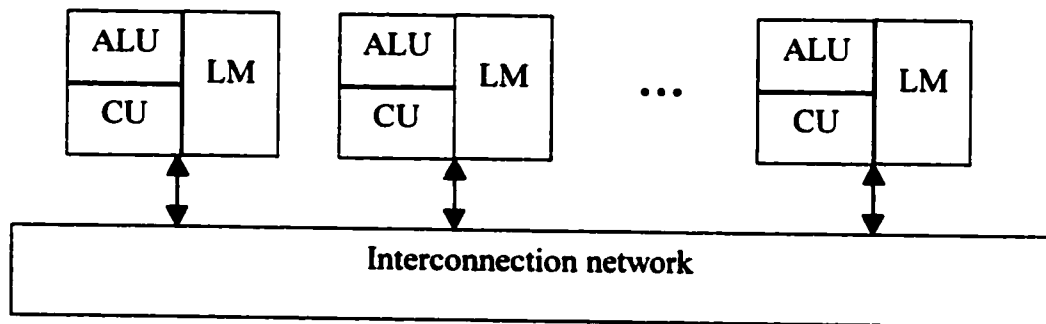


Figure 1.2: A multiprocessor system



ALU: Arithmetic Logic Unit
CU: Control Unit
LM: Local Memory

Figure 1.3: A multicomputer system

puter system. Systems of this type are often called *distributed computer systems*. The great advantage of such systems is that they are usually available in most of today's commercial, educational, and government organizations. Thus, we will use them as our experimentation test-bed.

Several software packages were developed to assist programmers in using distributed computer systems for parallel processing. Among the most well known packages are P4 developed at Argonne National Laboratory, Express from ParaSoft Corporation, and PVM that is a collaborative venture between Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University [20]. We will use PVM for implementing parallel scheduling algorithm, therefore, description of it will be found in Section 5.2.

1.2 Genetic Algorithms

In the early 1970s, John Holland invented a technique that mimics the process of natural biological evolution for solving difficult optimization problems [25]. Later, this technique was called *genetic algorithm* (GA). Since its invention, many variations of the basic technique have been introduced. Therefore, we will refer to this set of techniques as *genetic algorithms* (GAs).

Genetic algorithms are stochastic, robust, problem-independent search techniques. Two mechanisms link GAs with the optimization problem being solved. One is the way of *coding* the solutions of the problem. The coded form of a solution is called a *string*. The other is the *fitness function* that measures the merit of a string in the context of the

problem. Genetic algorithms have proved to be robust because they always produce high quality solutions. The ability of GAs to produce high quality solutions lies in the fact that they combine the exploitation of past good solutions with the exploration of new areas of the search space.

Genetic algorithms have been successfully applied to solve various optimization problems. These problems include pattern matching [5], bin packing [17], traveling salesman problem [30, 31], standard cell placement [33], graph partitioning [37], job shop scheduling [43], and multiprocessor scheduling [14, 26, 48]. Detailed discussion of the application of GAs to several problems from various fields of science and engineering can be found in [15, 21].

The success of GAs to solve various optimization problems can be attributed to the following characteristics:

- 1) GAs work with a coding of the parameter set rather than the parameters themselves.
- 2) GAs search from a population of search nodes.
- 3) GAs use probabilistic transition rules.

To design a genetic algorithm for solving any optimization problem, we have to design its components. These components are as follows: (1) a string representation of a solution, (2) a fitness function, (3) a method to create initial population, (4) genetic operators (reproduction, crossover, mutation), and (5) values of the control parameters. These parameters are population size, crossover probability, and mutation probability.

Typically, a genetic algorithm consists of the following steps:

- 1) Generation of the initial population of strings randomly.
- 2) Calculation of the fitness values of strings according to the fitness function.

- 3) Generation of a new population of strings by applying genetic operators to the old population of strings.
- 4) Step (2) and (3) are repeated for a predefined number of generations or until the population converges.

1.2.1 Genetic Algorithm Components

The construction of an efficient GA for any problem depends on how carefully its components were designed. Therefore, the remainder of this section provides an overview of the major components of GAs.

1.2.1.1 String Representation. The primary step in constructing GAs to solve an optimization problem is to find an efficient *string representation* for the solutions. Inefficiency in string representation may lead to high computational requirements for calculating fitness values and applying genetic operators. The most commonly used string representation in GAs is the binary alphabet $\{0,1\}$ arranged in one dimension. Other string representations found in the literature are binary, integer, or real-valued alphabet arranged in one or two dimensions.

1.2.1.2 Fitness Function. *Fitness function* plays the same role in GAs that the environment plays in natural evolution. The interaction of an individual with its environment provides a measure of its fitness, and the interaction of a string with a fitness function provides a measure of its fitness in the problem domain. The fitnesses of

the strings are used by GAs for carrying out reproduction [15].

In genetic algorithms, it is required that the fitness values must be nonnegative and that the fitter a string, the larger its fitness value. Therefore, it is necessary to map the objective function to a fitness function. When the objective function is a cost function, the following transformation is used [21]:

$$f(x) = C_{\max} - g(x) \quad \text{when } g(x) < C_{\max}$$

$$= 0 \quad \text{otherwise}$$

where $g(x)$ is the value of each individual solution. C_{\max} may be taken as the largest g value in the current population, as the largest g value observed thus far, or as the largest g value observed in the last k generations.

When the objective function is a profit or utility function, the following transformation is used [21]:

$$f(x) = u(x) + C_{\min} \quad \text{when } u(x) + C_{\min} > 0$$

$$= 0 \quad \text{otherwise}$$

where $u(x)$ is the value of each individual solution. C_{\min} may be the absolute value of the worst u value in the current population or the absolute value of the worst u value observed in the last k generations.

1.2.1.3 Initial Population. The way of constructing initial population influences the quality of final solution. This is because GAs work by adopting good characteristics from strings of the initial population to generate strings in subsequent populations. Generally, the initial population is constructed randomly. It is also reported that certain strings of the initial population may be solutions generated by some

constructive heuristics. The method of including solutions of other heuristics is called *seeding*. Another factor that affects the quality of final solution is the population size. A small population size, due to the lack of genetic diversity in the population, results in insufficient investigation of the search space and hence only suboptimal solutions are found. On the other hand, a large population size maintains genetic diversity in the population and hence prevents premature convergence to suboptimal solutions, but it may require unacceptable large computation time. There is no fixed rule to select the population size. It is usually determined empirically. For most GA applications, typical values of the population size range between 30 and 100.

1.2.1.4 Reproduction. *Reproduction* selects strings from a current population to form an intermediate population. The intermediate population is the mating pool from which offspring strings are generated by crossover and mutation. The selection of strings must mimic the process of natural selection where the fittest strings have higher chance to reproduce. Therefore, the selection is based on the fitness values in such a way that strings with higher fitness values should have higher probability of contributing one or more offspring strings in the next generation.

Various selection methods have been proposed in the literature. The simplest method is called *roulette wheel method*. This method starts by determining a real value (*sum*) as the summation of fitness values of all strings in a population. Strings are then mapped into contiguous interval $[0, \textit{sum}]$. The size of the interval assigned to a string is proportional to the fitness value of the string. A string is selected if the segment corresponding to this string spans the randomly generated number in the interval $[0, \textit{sum}]$.

This process is repeated until a mating pool of desired size is constructed [10].

1.2.1.5 Crossover. *Crossover* is the main operator for producing new strings. It operates on two parent strings to generate one or two offspring strings. Crossover operator imitates its counterpart in nature such that a new string generated by it has some characteristics of both parent strings. In the literature, several types of crossover operator are proposed. Single-point crossover, Order crossover (OX), Partially Mapped Crossover (PMX), and Cycle Crossover (CX) are few examples of these types. The simplest type of the crossover operator is *single-point crossover*. It starts by choosing a random point over the length of both parent strings. Then, right parts of the two parents are swapped to generate two new strings.

1.2.1.6 Mutation. *Mutation* operator infrequently introduces new characteristics not present in any string of a population, and thus it prevents premature convergence of the population. The mutation operator guarantees that the probability of searching any string will never be zero. In the simple mutation, a position in a string is chosen randomly and its value is replaced again by randomly chosen one from a certain set of values. Typically, mutation is applied with a very low probability.

1.2.2 Parallel Genetic Algorithms

There are three reasons for exploring parallel genetic algorithms. Firstly, to reduce the execution time of GA by computing the calculations associated with it in parallel.

Secondly, to maintain diversity and reduce the probability of the premature convergence. This can be done by employing a number of independent subpopulations and by using local selection rules, which allow a string to mate with strings in its local neighborhood. Thirdly, to mimic the natural evolution more closely. In nature, a population is typically many independent subpopulations that occasionally interact.

There are several ways to parallelize GAs. A simple way is to parallelize the loop (evaluation of fitness values, reproduction, crossover, and mutation) that creates the next generation from the previous one. Calculating fitness values and performing crossover and mutation can be trivially parallelized. In the reproduction step, to select parent strings all strings in the population have to be accessed by all processors, this can be a parallel bottleneck. Therefore, this way to parallelize GAs is only suitable for tightly coupled machines. For loosely coupled machines, some data must be passed to all processors. This data transferring may take an enormous amount of time, especially if there is no direct connection between two arbitrary processors [24].

Parallel genetic algorithms found in the literature can be classified according to the population structure and the method of selecting strings for breeding into three categories: *global*, *island*, and *cellular* [10, 46]. The following paragraphs describe these categories one by one.

1.2.2.1 Global Genetic Algorithms. In a global GA, the entire population is treated as a single breeding unit. This type of GA is realized as the master-slave model. In this model, slaves execute the task which is significantly more computationally expensive and the master runs the rest of the GA program.

In [18], a global GA is proposed to solve a real-time control problem on transputer based parallel processing system. For this problem, calculating the fitness value of a string takes a relatively long time compared with reproduction, crossover, and mutation. Thus, the main GA program is run on a host transputer and strings are distributed for evaluation of fitness functions over a number of transputers. The experiments used a population size of 250 strings and a number of transputers ranged from 4 to 72. The results showed an increase in speedup with more transputers in the parallel processing system. However, the relative increase in speedup dropped as the number of transputers grew. This means that the graph of speedup versus the number of transputers is not linear.

1.2.2.2 Island Genetic Algorithms. In an island GA, also called *migration GA*, a population is divided into a number of subpopulations, each of which is a separate breeding unit evolved by an independent GA. From time to time, fit strings are migrated between subpopulations.

The migration of strings between subpopulations is the most important feature of the island GA for two reasons. Firstly, this feature maintains genetic diversity within each subpopulation, since strings are migrated between subpopulations that have evolved independently under the control of different GAs. Secondly, this feature increases the selective pressure within each subpopulation for reproduction, since only fit strings are migrated between subpopulations and hence only good genetic materials are exchanged between subpopulations.

The performance of the island GA is based on several parameters. These parameters

are the number of generations between two consecutive migrations, the number of strings migrated, the migration paths between subpopulations, and the strategy used to select strings for migration. If a large number of strings migrate too frequently, then the local differences between subpopulations will be driven out and hence the parallel algorithm will be inefficient. Conversely, If migration is too infrequent, subpopulations may be prematurely converged.

In [31], an island GA is proposed to solve travelling salesman problem (TSP). The parallel algorithm is implemented on a network of workstations. The basic idea in the parallel algorithm is based on the using of several cooperating copies of the sequential GA, each of which is executed on a different workstation. Each copy of the sequential GA uses different values for crossover probability and mutation probability. After each generation, every workstation multicasts a string, which is selected using the roulette wheel method, to all other workstations. When a string is received, it replaces the worst string in the population. Experimental results showed that the parallel algorithm yields high quality and consistent solutions.

The paper [33] described a parallel placement algorithm based on the island GA model for standard cells. The parallel algorithm is organized as one master program and several slave programs. Each slave program runs on a different workstation, therefore, the number of slave programs depends on the number of workstations available in the distributed system. The master program creates slave programs and determines the communication paths between them. Each slave program is a sequential GA evolving its own subpopulation. From time to time, copies of the selected fit strings are sent along one communication path. Before termination, each slave program sends the best string

found in its subpopulation to the master program. Then, the master program identifies the best overall solution. It was observed that the parallel algorithm produces solutions of the same quality as the sequential algorithm while providing linear speedup. The effect of running the parallel algorithm on a heterogeneous computing environment was studied. The results found that the parallel algorithm is robust, since it maintains the solution quality and the speedup.

In [44], an island GA is applied to solve a function maximization problem on a hypercube multiprocessor system. Each processor runs GA on its own subpopulation, periodically selects good strings from its subpopulation and sends copies of them to one of its neighbor processors. It also receives copies of good strings from this neighbor. Then, bad strings in the subpopulation are replaced with the received ones. The migration paths varied over time, taking place over a different dimension of the hypercube each time. Good strings are chosen probabilistically from strings whose fitness values are greater than or equal to the average fitness of the subpopulation. Similarly, bad strings are chosen probabilistically from strings whose fitness values are less than or equal to the average fitness of the subpopulation. Near-linear speedup was reported when the parallel algorithm is compared with the sequential GA such that the sum of the subpopulations in the parallel algorithm is equal to the size of the population in the sequential GA. The effect of varying crossover and mutation rates among subpopulations was also investigated. The results found that the parallel algorithm performs well even without knowing the best parameter settings.

The paper [30] proposed an island GA based on a client-server model to solve travelling salesman problem (TSP). Each client executes the sequential GA on its own

subpopulation, and sends a copy of the best string in its subpopulation to the server whenever the best string is updated. The server retains a copy of the best string of each subpopulation and it sends copies of these strings to a client that cannot update its best string for a specific number of generations. When the client receives these strings, it replaces bad strings in its subpopulation with the received ones. The parallel algorithm is implemented on a network of workstations. Experimental results showed that the parallel algorithm yields solutions of the same quality as the sequential algorithm. The running time of the parallel algorithm decreases as the number of workstations increases in the parallel machine.

1.2.2.3 Cellular Genetic Algorithms. In a cellular GA, also known as *diffusion* or *neighborhood* GA, each string is assigned a geographic location on the population surface and allowed to breed with strings in its local neighborhood. The neighborhood of a string is usually the immediate adjacent strings on the population surface. Typically, only one string is assigned to each processor of the parallel computer. Therefore, the neighborhood of the strings is determined by processor connection topology. However, if the population size is greater than the number of processors, the population is divided into a number of blocks equal to the number of processors and strings in one block are assigned to the same processor. The parameters that affect the performance of the cellular GA are neighborhood size, processor connection topology, and string selection strategy for mating.

The paper [43] proposed a cellular GA to solve the jobshop-scheduling problem. In the proposed algorithm, strings are arranged on a torus, and the neighborhood of a string

x is defined as the set of strings within the *hamming distance* 1 from x . The cellular GA is executed on a single processor and compared with the sequential GA. The results showed that the cellular GA is better than the sequential GA in terms of the quality of solutions. It was also observed by implementing the cellular GA on transputer based parallel processing system that the cellular GA is quite effective in reducing the computational time.

1.3 Thesis Outline

The rest of the thesis is organized as follows:

Chapter 2 formulates the scheduling problem addressed in this thesis. Section 2.1 and Section 2.2 define the models used to represent parallel computations and parallel systems, respectively. In Section 2.3, the scheduling problem is overviewed. Then, the classification of the scheduling algorithms is described in Section 2.4. Finally, Section 2.5 states the objective of this thesis.

Chapter 3 briefly describes various scheduling algorithms found in the literature. Static scheduling algorithms and dynamic scheduling algorithms are reviewed in Section 3.1 and Section 3.2, respectively. Section 3.3 is devoted to parallel approaches to solve the scheduling problem.

Chapter 4 presents the sequential genetic scheduling algorithm used as the basis for the parallel genetic scheduling algorithm. Section 4.1 discusses workload generation methods and presents test problems used in the thesis. The design of the sequential genetic scheduling algorithm is presented in Section 4.2. Experimental results are shown

in Section 4.3.

Chapter 5 starts by presenting the proposed parallel genetic scheduling algorithm in Section 5.1. In Section 5.2, the environment in which the experiments were performed is described as well as implementation issues are discussed. Finally, the results of our experiments are shown in Section 5.3.

Chapter 6 presents our conclusions and suggestions for future research.

Chapter 2

Problem Definition

In this chapter, the models that are used in the literature to represent parallel computations and parallel systems are defined. Next, the formulation of the scheduling problem and the scheduling taxonomy are presented. Finally, the objective addressed in this thesis is stated.

2.1 Parallel Computation Models

A parallel computation is represented by a set of tasks (computational units), which are interconnected into a graph or a digraph, called a *task graph*, depending on the nature of their interrelationship. There are two models to represent parallel computations. The first model assumes no data transfer between tasks of the task graph and thus no communication between processors. This kind of a task graph is called a task graph with no communication overhead between tasks. The scheduling of such task graphs has been studied extensively and several solutions have been proposed [1, 29, 42]. However, this

model is no longer valid for loosely coupled systems, since inter-processor communication overhead is an important aspect in such systems and is not negligible. The second model, which is addressed in this thesis, considers inter-task communication costs.

In this study, a parallel computation can be viewed as a finite set $\Gamma = \{T_1, T_2, T_3 \dots T_n\}$ of cooperating and communicating tasks. The behavior of this set of tasks is modeled by a directed acyclic graph (DAG). Each node in the task graph represents a task $T_i \in \Gamma$ and each edge represents a precedence constraint between a task and its immediate successor. The set of edges in the task graph is denoted by \rightarrow . Edges are associated with non-negative integers $\eta(T_i, T_j)$, which expresses the number of data units to be sent from task T_i to its immediate successor T_j upon the completion of T_i . Each node T_i of the task graph is associated with a positive number $\mu(T_i)$, which expresses the task computation time. Therefore, a general precedence constrained computation with communication costs can be denoted by a quadruple $G(\Gamma, \rightarrow, \mu, \eta)$. Figure 2.1(a) shows an example of a task graph with communication costs.

2.2 Parallel Systems

A parallel system is denoted as $S(P, R)$, where P is a set of m identical processors (identical in both functional capability and speed) and R is a set of routing times between them. The time to transfer a unit of data between two processors P_u and P_v is represented by $r(P_u, P_v)$. Therefore, if tasks T_i and T_j are mapped to processors P_u and P_v , respectively,

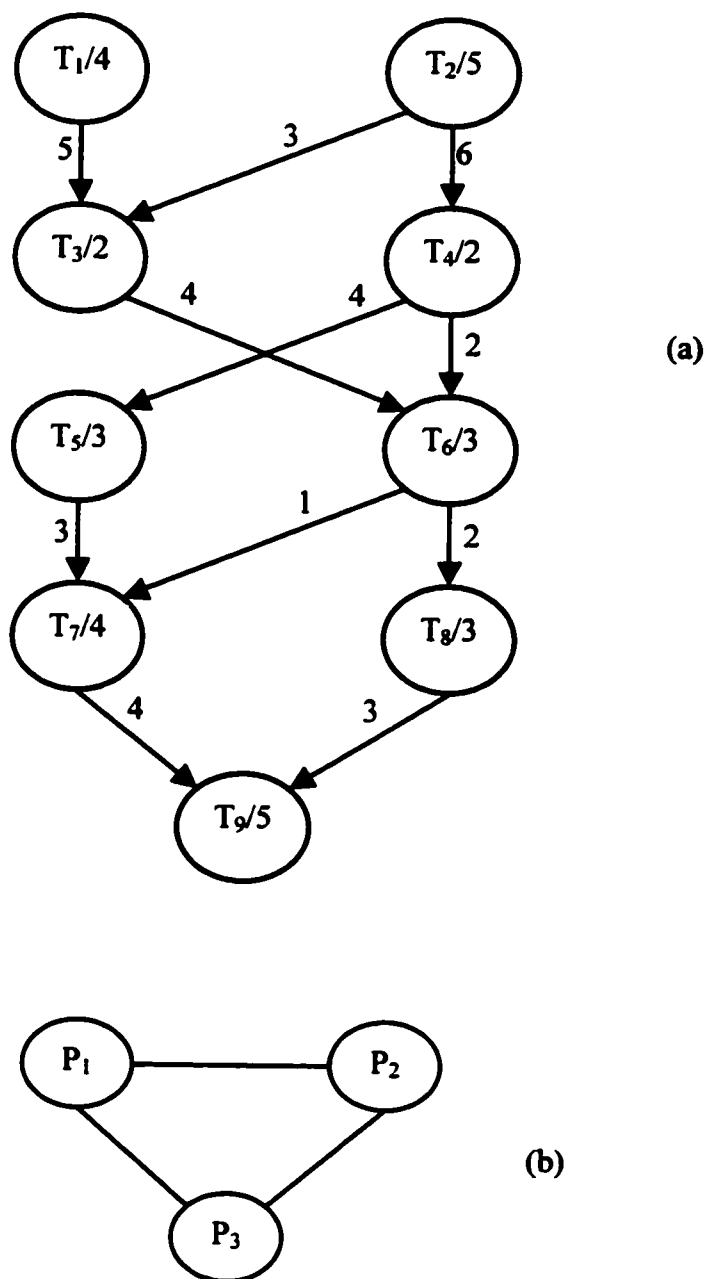


Figure 2.1: Example of (a) A task graph with communication costs
(b) A parallel system

then the time to transfer $\eta(T_i, T_j)$ data units from P_u to P_v is given by $r(P_u, P_v) \times \eta(T_i, T_j)$. Note that if tasks T_i and T_j are assigned to the same processor, then the time to transfer data between them is zero. An example of a parallel system, where every processor is connected to every other processor, is shown in Figure 2.1(b). However, in reality different interconnection topologies are possible such as ring, mesh, hypercube, irregular (arbitrary), etc.

2.3 The Scheduling Problem

The scheduling problem can be defined as mapping a given set of partially ordered computational tasks that constitute an application to an available set of processors in order to minimize the finishing time of the application. The scheduling problem in its most general form is known to be NP-complete [13, 40]. The complexity of the scheduling problem depends on the topology of the task graph, the uniformity of tasks' execution times, task preemption, the topology of the parallel system, the number of parallel processors, and the uniformity of processors. To tackle the problem, restrictions are placed on the task graph and the parallel system models. Even restricted forms of the problem are found to be NP-complete such as scheduling one-time-unit tasks on an arbitrary number of processors, or scheduling one- or two-time-unit tasks on two processors [13]. However, there are two cases where optimal schedules could be found in polynomial time when inter-task communication is not considered. These cases are scheduling tree-structured task graphs with equal execution times of the tasks on arbitrary number of processors [27], and scheduling arbitrary-structured task graphs with

equal execution times of the tasks on two processors [40, 42]. If the inter-task communication introduced in any one of the above cases then the problem becomes NP-complete.

2.4 Scheduling Taxonomy

Different scheduling algorithms can be classified according to the following characteristics:

a) Local Versus Global: Local scheduling deals with the assignment of tasks to the time-slices of a single processor. Global scheduling deals with the assignment of tasks to the processors in a parallel system.

b) Static Versus Dynamic: In static scheduling, all information regarding the task graph must be entirely known before execution time. Therefore, each task has a static assignment to a particular processor, and each time that task is submitted for execution, it is assigned to that processor. The disadvantage of static scheduling is its inadequacy in handling non-determinism in programs (loops and conditional branches). For example, the direction of a conditional branch or the size of a loop could be unknown before a program starts execution.

Under the static scheduling, we distinguish between optimal solutions and sub-optimal solutions. A sub-optimal solution may be reached when an optimal solution is computationally infeasible. There are two ways to obtain sub-optimal solutions, namely, approximate and heuristic. In approximate, a sub-optimal solution can be reached by

using the same computational model representing the algorithm but instead of searching the entire solution space for an optimal solution, we are satisfied when we find a good one. In heuristic, we use our intuition to come up with a solution. For example, clustering tasks that heavily communicate with each other on the same processor reduces communication overhead between processors.

In dynamic scheduling, information regarding the task graph is not known before the program is in execution due to conditional branches and loops. Therefore, the scheduling decisions are made on the fly. The disadvantage of dynamic scheduling is its inadequacy in finding global optimums and the time delay incurred by the on-line scheduling.

In dynamic scheduling, if the work involved in making scheduling decisions is distributed among different processors, then the scheduler is physically distributed. Otherwise, if the scheduling decisions are assigned to only one processor, then the scheduler is physically non-distributed. Under the physically distributed scheduling, we may distinguish between cooperative and non-cooperative systems. In cooperative systems, the local schedulers at each processor cooperate to come up with a global schedule that is based on the status of the whole system. In non-cooperative systems, individual processors work independently and arrive at scheduling decisions regarding their own resources, which will affect local performance only. The cooperative dynamic scheduling branch is further classified into optimal and suboptimal solution cases. Same discussion as was presented for the static scheduling applies here as well. The structure of this portion of the taxonomy is shown in Figure 2.2 [9].

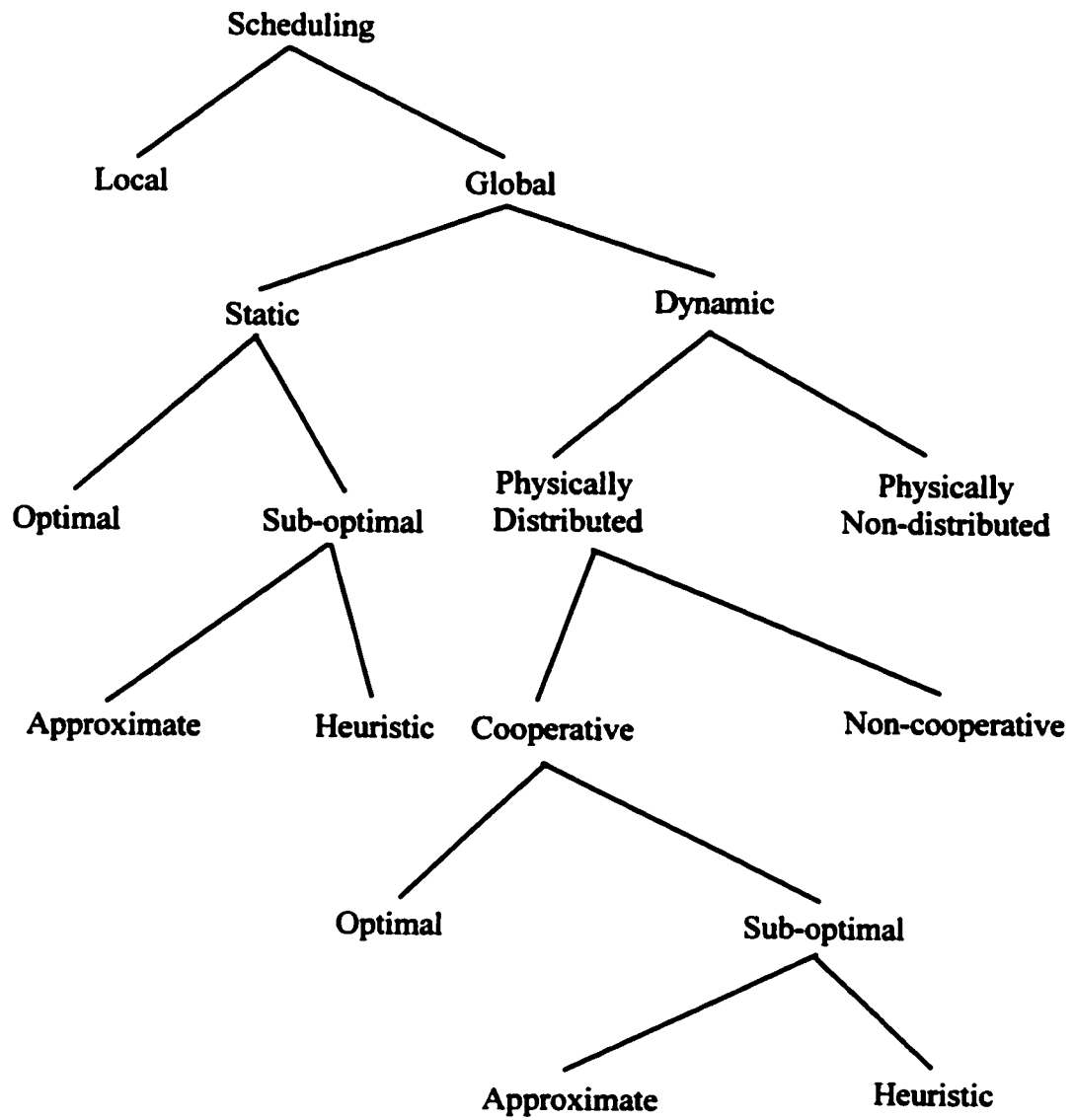


Figure 2.2: Scheduling taxonomy

c) Adaptive Versus Non-adaptive: An adaptive scheduler changes its scheduling decisions in response to the previous behavior of the system. Adaptive schedulers are usually dynamic because they may collect information about the system and make scheduling decisions on the fly. A non-adaptive scheduler does not change its scheduling decisions according to the previous behavior of the system.

d) Preemptive Versus Non-preemptive: Preemptive scheduling permits a task to be interrupted and removed from the processor under the assumption that it will start later from the point where it was interrupted. Non-preemptive scheduling permits a task to run to completion without interruption once it has begun execution.

2.5 Thesis Objective

This thesis deals with genetic based static scheduling of parallel computations with communications on multiprocessor systems. The parallel computation is represented by a directed acyclic graph (DAG) in which nodes represent tasks and edges represent precedence constraints between tasks. The multiprocessor system is composed of a number of identical processors. Two interconnection topologies, namely, fully connected and mesh are considered. Contention free communication network will be assumed. Contention occurs when two or more parallel tasks running on different processors send messages through one or more common channels. Contention delays the arrival of messages to their destinations. In a contention free system, the communication channels have enough capacity to accommodate all transmissions without any conflict. Thus,

there is no delay and the time required for sending a unit of data from one processor to another is fixed. Under the above conditions, the objective is to minimize the finishing time of a given parallel computation on a given multiprocessor system.

The objective of this thesis is to design a fast and efficient parallel genetic scheduling algorithm. To achieve this objective, an efficient sequential genetic scheduling algorithm will be designed. Then, various approaches to parallelize this algorithm will be studied and the most suitable approach that can be realized easily on a network of workstations will be selected. Network of workstations has been selected as a parallel machine because it is readily available in most commercial, educational, and government organizations. To implement the parallel genetic scheduling algorithm, PVM (*Parallel Virtual Machine*) will be used. PVM is a software package that assists programmers to use distributed systems for parallel processing.

Chapter 3

Literature Review

Various scheduling strategies found in the literature are presented in this chapter with special attention given to scheduling techniques based on genetic algorithms. The chapter is divided into three sections. The first section presents static scheduling approaches. In the second section, some dynamic scheduling algorithms are reviewed. In the last section, parallel approaches to solve the scheduling problem are discussed.

3.1 Static Scheduling Approaches

In [28], a heuristic called *Earliest Task First* (ETF) is proposed. ETF maintains a ready set. This set contains tasks whose predecessors have been scheduled. At each scheduling step, the earliest starting time of each task in the ready set is computed. This is done by computing the starting times of each task on all processors. Then, the task with the earliest starting time is scheduled on the processor where this time obtained. The time complexity of ETF is $O(nm^2)$, where n and m are number of processors and number of

tasks, respectively.

The reference [32] presents a static scheduling algorithm for allocating task graphs with arbitrary computation and communication costs to a multiprocessor system with unlimited number of fully connected identical processors. The proposed algorithm is called *Dynamic Critical-Path* (DCP) scheduling algorithm. The idea used in DCP is that during the scheduling process the critical path (CP) of partially scheduled task graph is not constant. This is because the communication cost between two tasks is considered zero if the tasks are scheduled to the same processor. Therefore, DCP assigns dynamic priorities to the tasks at each scheduling step and selects a task according to the modified priorities. When scheduling a task on a processor, DCP does not simply minimize the starting time of this task but also estimate the scheduling effect on the task's successors starting times. The complexity of the algorithm is $O(v^3)$, where v is the number of tasks in a task graph.

In [39], a scheduling heuristic called *Mapping Heuristic* (MH) is described. MH is a modified list scheduling technique. MH considers real-world constraints such as interconnection topology of the target machine, processor speed, link transfer rate, and contention delays. MH uses the level of each node in the task graph as its priority. It breaks ties by selecting the task with the largest number of immediate successors. If this does not break the tie, it selects one at random.

When a task is ready, a processor is selected to run that task in such a way that the task cannot finish earlier on any other processor. The finishing time of a task is determined by considering: 1) processor speed, 2) link transfer rate, 3) message passing

route, 4) number of hops, and 5) delay due to contention. When a task finishes execution, the number of conditions that prevent any of its immediate successors from being run is decreased by one. When the number of conditions associated with a particular successor becomes zero, then that successor is ready and can be scheduled.

The paper [7] deals with a scheduling problem known as the flow shop problem. Flow shop is a system where all tasks are decomposed into chains of subtasks. The subtasks are executed on different processors in turn in the same order. The flow shop scheduling problem is NP-hard, except for a few special cases. This paper presents a heuristic to schedule tasks with arbitrary parameters (release times, deadlines and processing times). The complexity of this heuristic is $O(n \log(n) + nm)$, where n and m are number of tasks and number of processors, respectively.

The heuristic determines the effective release times and effective deadlines of all subtasks. The effective release time of a subtask is the earliest point in time at which the subtask can be scheduled. The effective deadline is the point in time by which the execution of a subtask must be completed to allow later subtasks to complete by their deadlines.

The heuristic starts by determining the subtask with the largest processing time on each processor. Then, on each processor inflate all the subtasks by making their processing times equal to the processing time of the longest subtask. Determine the bottleneck processor P_b . The bottleneck processor is the one on which subtasks have longest processing time. Schedule the subtasks on P_b according to the *earliest-effective-deadline-first* (EEDF) algorithm. If the resultant schedule is feasible, the subtasks meet

their deadlines, propagate the schedule onto the remaining processors.

Inflating processing times of the subtasks increases the workload that is to be scheduled on the processors. This increases the number of release time and deadline constraints that are not met. Therefore, a compaction step is added to the heuristic that reduces the idle time.

In [35], a task clustering heuristic with duplication is proposed for the scheduling problem. The algorithm takes into account inter-task communications and assumes that there is no limit on the number of processors. The algorithm produces a schedule whose makespan is at most twice the optimal. The algorithm runs in $O(n(n \log(n) + e))$ time, where n is the number of tasks and e is the number of edges.

The algorithm computes a lower bound $e(v)$ on the earliest possible start time for each task v . This is accomplished by finding a cluster, $C(v)$, containing v that allows v to be started as early as possible. All the nodes in the cluster, $C(v)$, are executed on the same processor and all other nodes are executed on other processors. Once the clusters are determined, they are mapped to different processors. The nodes mapped to the same processor are executed in non-decreasing order of their e values.

The reference [4] presents a scheduling algorithm called *Optimal Assignment with Sequential Search* (OASS). OASS constructs the problem as a search tree. It then searches the nodes of the tree starting from the root. Intermediate nodes represent partial solutions while the complete solutions (goals) are represented by leaf nodes. Associated with each node is a cost f . The value of f for a node v is computed as $f(v) = g(v) + h(v)$. In the above equation, $g(v)$ is the cost of the search path from the root to the node v , and

$h(v)$ is a lower bound estimate of the path cost from the node v to the goal node. The algorithm always selects a node with the best cost (node with the smallest f) for expansion. Therefore, the algorithm guarantees an optimal solution. The order in which tasks are considered for allocation is determined by a heuristic called *minimax sequencing*. In this heuristic, we choose a task that would have a maximum bearing on f as the first task to be assigned.

The reference [3] introduces an approach based on the problem-space genetic algorithm (PSGA) for scheduling task graphs on multiprocessor systems in order to reduce the task turnaround time and to increase the throughput of the system. The proposed PSGA based approach combines the power of the genetic algorithm with a list scheduling heuristic to search a large solution space efficiently and effectively.

The algorithm reads the task graph, population size N_p , and number of generations N_g . Then, it generates an initial population of strings. The string is an array of integer numbers representing the priorities of the tasks. After that, the algorithm repeats the following steps N_g times.

- 1) Apply the list scheduling heuristic to generate a solution for each string in a current population.
- 2) Calculate the fitness value of each string.
- 3) Select strings based on their fitness values from the current population.
- 4) Apply crossover and mutation to generate a new population.
- 5) Replace the current population with the new population.

Experimental results showed that PSGA has a fast convergence rate as compared to

the standard genetic algorithm. This is due to the heuristic guided search in PSGA instead of a blind search in standard GA.

The paper [2] deals with a task assignment problem. The task assignment problem is different from the scheduling problem. In the task assignment problem, precedence constraints are not taken into consideration, while in the scheduling problem precedence constraints are preserved in assigning tasks to processors. The task assignment problem is NP-complete, except for a few special cases.

The paper proposes a problem-space genetic algorithm (PSGA) for the task assignment problem to reduce the task turnaround time and to increase the throughput of the system. The PSGA approach combines the genetic algorithm with a simple and fast problem-specific heuristic. The proposed algorithm is extended for task assignment in heterogeneous distributed computing systems.

In [6], a genetic algorithm to find a minimal schedule assignment of tasks to processors in a multiprocessor system is proposed. The string representation is integer alphabet arranged in one dimension. The ordinal value of an integer in the string represents a task while its cardinal value represents a processor.

To construct a string for initial population, a random permutation of numbers from one to n is generated, where n is the number of tasks in a task graph. Then equal number of tasks is assigned to each processor. This is referred to in the literature as *pre-scheduling*. A simple one-point crossover operator is used. It operates on two parent strings and generates one child string. The mutation operator applied is the pair wise swap of assignment of tasks to processors.

The reference [11] describes a simple randomized heuristic based on the principle of genetic algorithms for the mapping problem. The mapping problem is the problem of assigning parallel tasks onto multiprocessor architecture to minimize the overall finishing time. An integer string representation has been chosen for this problem. The ordinal value of an integer in the string represents a task while its cardinal value represents a processor to which that task is mapped. The heuristic begins with an initial generation of a uniformly random population of strings.

The heuristic applies two genetic operators on strings, namely, crossover and mutation. The application of crossover operator involves a selection of strings. The selection of strings is based on their fitness values. Two random crossover points are chosen over the selected parent strings. The sub-strings demarcated by the crossover points are exchanged between the parent strings to create two new strings.

The point wise mutation is applied after the crossover. The number of positions that are to be changed in a string, are chosen from a uniform distribution of values between zero and one third of the string length. The actual positions to be changed are then chosen over the string. A new value for a chosen position in the string is selected from a uniform distribution of values between one and the maximum number of processors. Then, the heuristic replaces the worst string in the current generation with the new string if the new string is better than the former.

In paper [12], the problem of mapping is decomposed into sub-problems of task clustering and clusters allocation. Such decomposition reduces the size of the allocation problem resulting in a relatively less expensive objective function evaluation.

The proposed hybrid genetic mapping algorithm is a combination of the recursive clustering algorithm for clustering tasks and the genetic algorithm for allocating clusters to processors. Inter-cluster communication is minimized during the clustering phase. Inter-processor communication is minimized during the allocating phase.

The major advantage of this combination is that the clustering phase reduces the number of allocable units to the number of processors. Thus, irrespective of the task graph size, the total number of processors bound the size of the allocation problem in the system.

The reference [14] proposes an approach where a genetic algorithm is improved with the introduction of some knowledge about the scheduling problem. Using a list heuristic in the crossover and mutation operations does the introducing of knowledge.

The string representation is based on several lists of tasks, where each list represents the tasks scheduled to some specific processor. To verify the feasibility of a string, the combined precedence relations implied by the task graph and by the tasks scheduled to the same processor in the string are used.

The algorithm uses an iterative method to generate strings of an initial population. This method guarantees uniform distribution of tasks among processors. At each step, the set of tasks that can be scheduled is determined. A task can be scheduled if all its predecessors are already scheduled. Then, a task from this set and a processor where this task will be scheduled are chosen randomly. The disadvantage of this algorithm is its running time, which is one order of magnitude larger than the pure genetic algorithm.

The paper [26] proposes an algorithm based on genetic algorithms to solve the problem

of multiprocessor scheduling. In this algorithm, the string representation is based on several lists of tasks. Each list represents tasks executed on a processor and the appearance of tasks in a list depends on their order of execution. The tasks in each list are ordered in ascending order of their heights. The algorithm generates an initial population of strings randomly. A legal string is one that satisfies the following: The precedence constraints among the tasks are satisfied and every task appears only once in the string.

In each generation, the algorithm computes fitness values of the strings in the current population. Then, three genetic operators are applied on the population, namely, reproduction, crossover, and mutation. The reproduction operator selects strings based on their fitness values from the current population to form a mating pool. The crossover operator picks any two strings from the mating pool and exchanges portions of them to create new strings. The mutation operator randomly exchanges two tasks with the same height in each string. The algorithm is applied repeatedly for a specific number of generations.

In [38], a genetic algorithm for mapping tasks to processors in a reconfigurable parallel architecture is proposed. A reconfigurable parallel architecture is a multiprocessor architecture whose interconnection network topology can be modified using programmable switches, e.g. transputer-based parallel processors. The problem of mapping tasks to processors in a reconfigurable parallel architecture is consists of the following sub-problems.

- 1) Assigning tasks to processors (mapping problem).

- 2) Configuring the processor graph (link allocation problem).
- 3) Ordering the execution of the tasks allocated to a processor (scheduling problem).

The string representation consists of the following components: mapping information, link allocation information, and scheduling information. The algorithm uses a composite fitness function, which measures the total execution time of the tasks under a given mapping, a given configuration of processors, and a specified schedule. In addition, it uses three different crossover operators, one for each component of the string.

The paper presents an extension of the algorithm that can take into account heterogeneous platforms, where each task can be executed on a particular class of processors.

The paper [45] describes a genetic algorithm to schedule tasks on processors in order to minimize the maximum task finishing time. In this algorithm, the string representation is based on two-dimensional matrix called *allocation matrix*. The allocation matrix, A , is an $n \times n$ matrix, where n is the number of tasks in a task graph. For a string, $A_{ij} = 1$ if task i precedes task j on the same processor, otherwise $A_{ij} = 0$.

The crossover operator mates two selected strings to form two new strings. It selects a column from one to $(n-1)$ and cuts the allocation matrix vertically in two parts at the selected column. Then, it exchanges the right parts of the two allocation matrices. The mutation operator for this algorithm selects randomly a number of positions in an allocation matrix and complements their values.

The disadvantage of this algorithm is that the crossover and the mutation operators

may generate invalid strings. Thus, a repair operation is applied after each crossover and mutation.

The paper [48] presents a genetic algorithm to solve the problem of task scheduling in distributed computing system. The string representation is based on several lists. Each list is associated with a processor. To verify the feasibility of a string, tasks are arranged in ascending order of their heights in each list.

The crossover operator selects a task randomly and assigns the selected task and all its immediate successors to a set (E). Then, for each processor in both parent strings the following steps are performed. First, extract tasks from the list of tasks associated with this processor such that the extracted tasks are in the set (E). Then swap the extracted tasks between the two parent strings.

In mutation, a processor with the latest finishing time is selected. On this processor, the longest idle period is determined. The task right after this idle period is extracted from the list of tasks associated with this processor and inserted into a list of tasks associated with a processor where *data dominant parent* (DDP) of this task is assigned. Data dominant parent is a task that transmits the largest volume of data to this task.

In [47], a genetic algorithm is presented to solve the problem of multiprocessor scheduling in heterogeneous environment. In heterogeneous environment, the characteristics of tasks and the properties of processors should be considered. The string representation is an array of integer numbers, where the ordinal value of an integer represents a task while its cardinal value represents a processor.

The algorithm introduced a new crossover operator, called *x-pocket* crossover. When

string X and string Y exchanging their i^{th} genes to each other, both i^{th} genes are inserted into a pocket. All processors with the most suitable types for the i^{th} task also inserted into the pocket. Then, the crossover picks two processors from the pocket and assigns one of them to the i^{th} gene of X and the other to that of Y .

The paper [49] presents a genetic algorithm to solve a scheduling problem known as the hybrid flow shop. The hybrid flow shop is a generalization of the flow shop problem, and characterized as the scheduling of jobs in a flow shop environment where there may exist multiple machines in one or more stages. For stages with multiple machines, jobs may be processed on any machine.

The proposed algorithm is based on the list scheduling principle. It develops job sequences for the first stage, and then jobs are queued in a FIFO manner for the remaining stages. Job sequence, denoted as integers, is used in the proposed algorithm as string representation.

Two genetic operators, namely, crossover and mutation are applied on the strings. Partially mapped crossover (PMX) is used. The crossover operator may generate invalid strings; therefore, each crossover must be succeeded by a repair operation. In mutation, two jobs are selected randomly and their positions are exchanged.

3.2 Dynamic Scheduling Approaches

The paper [23] describes a dynamic scheduling algorithm, called *self-adjusting scheduling algorithm* (SASH), for a heterogeneous computing system interconnected

with an arbitrary communication network. A completely heterogeneous system is one in which the degree of uniformity in both the processing elements and the communication channels is low.

The algorithm uses a maximally overlapped scheduling and execution paradigm to schedule a set of independent tasks. Overlapped scheduling and execution in SASH is accomplished by dedicating a processor to execute the scheduling algorithm. SASH performs repeated scheduling phases in which it generates partial schedules. At the end of each scheduling phase, the scheduling processor places the tasks scheduled in that phase onto the local queues of the working processors. The scheduling processor then schedules the next subset of tasks while the working processors execute the previously scheduled tasks. During a single scheduling phase, the algorithm schedules tasks until the least-loaded working processor has completed executing all the tasks on its local queue.

A unique aspect of the algorithm is that it easily adapted to different task granularities, to dynamically varying processor and system loads, and to system with varying degrees of heterogeneity. The experiments showed that the performance resulting from SASH, compared to the performance resulting from existing scheduling heuristics, could outweigh the loss in performance caused by dedicating a processor for scheduling, even in systems with a small number of processors.

The paper [41] presents dynamic uniprocessor scheduling algorithm for hard real-time tasks. The algorithm accepts or rejects a task by performing feasibility analysis, i.e. decides whether a new task can be scheduled to meet its timing constraints.

When a new task T arrives, T is placed in order into *earliest deadline first list* (EL). EL records all previously scheduled tasks. A subset of tasks in EL, whose scheduling interval conflict with the scheduling interval of T , is rescheduled. A task T is considered schedulable if all of the tasks in the subset are schedulable. Otherwise, T is considered unschedulable, and it is removed from EL.

The complexity of this algorithm is $O(n \log(n))$ in the worst case, where n is number of tasks in EL. The paper also presents a simple extension of the algorithm that can take into account precedence constraints among tasks.

In [36], a hard real-time genetic algorithm called *MicroGA* is proposed. The objective of MicroGA is to dynamically schedule as many tasks as possible such that each task meets its execution deadline, while minimizing the total delay time of all tasks.

The algorithm represents a schedule of tasks in a string as a permutation of the tasks known to the system at a time. Each gene represents a task and the corresponding order of the tasks determines the schedule. Each task has an associated run-time and deadline. Since the processing must be done in real-time, we use a small population size and run MicroGA for a small fixed number of generations (N_g). After each N_g generations, the first task in the best string is scheduled for execution. Then, in each string the scheduled task is replaced by a new task. The new task is obtained from the external task queue.

The paper [19] presents four variations of a new scheduler called *least genetic* (LG) to schedule dynamically evolving parallel programs in distributed multiprocessor systems. The least-genetic scheduler is consists of two schedulers, namely, lest loaded and genetic. The least loaded scheduler schedules a single task at a time by assigning it

to the least loaded processor. On the other hand, genetic scheduler schedules a set of tasks in a single activation by the application of genetic algorithm. Tasks are accumulated to fill the string before genetic scheduler is activated. This accumulation will only be in operation when the host machine perceives that every machine in the system has at least a minimum number of tasks in its ready queue. The proposed scheduler considers both load balancing and communication minimization.

3.3 Parallel Scheduling Approaches

The reference [4] also describes a parallel version of OASS (refer to Section 3.1) called *Optimal Assignment with Parallel Search* (OAPS). OAPS generates optimal solution for assigning an arbitrary task graph on an arbitrary network of homogeneous or heterogeneous processors.

Initially the search space is divided according to the number of scheduling processors P and the maximum number of successor S of the initial node. There are three cases:

- 1) If $P < S$, each scheduling processor will get one node and get additional nodes in Round Robin fashion.
- 2) If $P = S$, each scheduling processor will get one node.
- 3) If $P > S$, each scheduling processor will keep expanding nodes until the number of nodes are greater than or equal to the number of scheduling processors. Then, the resulted nodes are sorted in increasing order of cost values of the nodes. First node will go to the first scheduling processor, second node will go to the second

scheduling processor and so on. Extra nodes will be distributed in Round Robin fashion.

Given an initial partition, every scheduling processor first sets up its neighborhood to find out which scheduling processors are in its neighbor. Then, every scheduling processor will run the Optimal Assignment with Sequential Search (OASS).

If there is no communication between scheduling processors after the initial partitioning, some of the scheduling processors may work on good part of the search space, while others may expand unnecessary nodes. Thus, scheduling processors need to communicate to share the best part of the search space. In OAPS, scheduling processors achieve this by periodically selecting a neighbor and then sending its best node to that neighbor.

The paper [8] proposes a parallel scheduling algorithm (PSA) for scheduling a set of n partially ordered tasks on m -processor parallel computing system. The proposed algorithm is based on Earliest Task First (ETF) approach. Therefore, PSA schedules a task T on a processor P if the earliest starting time of T on P is the smallest among all the ready tasks and all the ready processors. A task is ready if all its predecessors are already scheduled. All scheduling processors share the scheduling load. Thus, each scheduling processor executes PSA algorithm for a different subset of tasks and processors. The scheduling processors cooperate in terms of task ready time, processor free time, and task completion time. Using $O(mn)$ scheduling processors, where n is the number of tasks and m is the number of application processors, the time complexity of the algorithm is $O(n (\log(m) + \log(n)))$.

The paper [16] presents parallel scheduling algorithm developed by using binary trees. The algorithm considers tasks with release times and deadlines. The algorithm consists of three main steps, namely, preprocessing, a bottom-up pass, and a top-down pass.

The preprocessing step in the proposed algorithm sorts tasks by release times into non-decreasing order. Tasks with the same release time are sorted into non-decreasing order of deadlines. Then, a binary computation tree with k leaves is associated with the problem, where k is the number of distinct release times of the tasks. With each node in the tree, a time interval (t_l, t_r) is associated.

In the bottom-up pass, the used and the transferred sets of tasks for each node in the computation tree are determined by using the available sets of tasks. The available set consists of exactly of those tasks that have a release time greater than or equal to t_l and less than t_r . The used set consists of exactly of those available tasks that will be scheduled between t_l and t_r . The remaining tasks make up the transferred set.

In the top-down pass, the used sets are updated so that the used set for a node representing the interval (t_l, t_r) is precisely the subset of the tasks that is scheduled in this interval for the entire task set.

The paper [36] also presents a parallel model of *MicroGA* algorithm (refer to Section 3.2). The parallel MicroGA uses one host and several processors. The general scheduler runs on the host and its only function is to maintain a queue of tasks and distribute these tasks to the processors, as they become available. The processors take the tasks from the host and process them in the same way as sequential MicroGA do.

The paper presents two models of parallel MicroGA, one allowing communications between scheduling processors and the other not allowing communications. In the first model also called migration model, certain number of the largest delay tasks appearing in the best string of each processor is passed to one of the neighbor processors. This step is done once every execution of MicroGA for a fixed number of generations. The paper shows that the migration model of parallel MicroGA is better than no migration.

Chapter 4

Sequential Genetic Scheduling Algorithm

In this chapter, a new sequential genetic scheduling algorithm called *level-based genetic algorithm* (LGA) is designed. This algorithm will then be used as a building block upon which to develop the parallel genetic scheduling algorithm.

In Section 4.1, test problems generation techniques are described. Then, detailed discussion of the sequential genetic scheduling algorithm developed is presented in Section 4.2. Finally, Section 4.3 presents the experimental results.

4.1 Test Problems Generation

To evaluate the performance of LGA, a *random parallel computation generator* (RPCG) is implemented and used to generate task graphs of various sizes. The computation times of the tasks as well as the communication costs between tasks are selected randomly from the integer range of 1 to 20. The number of tasks at each height level is a random

number between 8 and 24. The height of a task is based on the number of predecessors it has. If a task has no predecessor, its height is zero; otherwise, the task's height is the maximum of its predecessors' heights plus one. The number of successors for a task is a random number between 2 and 10. A task selects 70% of its successors from tasks at the next height level and the remaining 30% from tasks at other height levels. RPCG also generates system graphs with various sizes and topologies. Two system topologies are considered, namely, fully connected and mesh. Task graphs generated by RPCG and used in this thesis are given in Table 4.1. The columns in this table are the task graph name, the number of tasks in the task graph, and the schedule length of the task graph in time units on a single processor.

The sequential genetic scheduling algorithm is also tested on a number of task graphs for which optimal schedules are known in advance. The technique used to generate such task graphs is presented in [6]. We called it *optimal parallel computation generator* (OPCG). OPCG generates a random Gantt chart such that each slice in the Gantt chart corresponds to a task and its width represents the computation time of that task on the given processor. Then, precedence relations between tasks are defined as follows. Several pairs of tasks (T_i and T_j) are chosen randomly such that the finishing time of T_i is before the starting time of T_j . If T_i and T_j are on different processors, the communication cost is equal to or less than the separation between the finishing time of T_i and starting time of T_j ; otherwise, the communication cost is any arbitrary number. Task graphs generated by OPCG and used in this thesis are presented in Table 4.2. The columns in this table are the task graph name, number of tasks in the task graph, number of processors in the system graph, the length of the optimal schedule, and the schedule

Task Graph	No. Tasks	Schedule Length on One Processor
m100	100	991
m150	150	1481
m200	200	1904
m250	250	2528
m300	300	3061
m350	350	3493
m400	400	4073
m450	450	4397
m500	500	4874

Table 4.1: Task graphs generated by Random Parallel Computation Generator (RPCG)

Task Graph	No. Tasks	No. Processors	Optimal Schedule Length	Schedule Length on One Processor
on140m5	140	5	310	1550
on165m7	165	7	287	2009
on191m6	191	6	375	2250
on219m4	219	4	602	2408
on293m7	293	7	503	3521
on329m5	329	5	735	3675
on350m4	350	4	958	3832
on373m3	373	3	1333	3999

Table 4.2: Task graphs generated by Optimal Parallel Computation Generator (OPCG)

length of the task graph on a single processor. In our implementation, the computation time of a task is selected randomly in the range 1–20, and the number of successors for each task is a random number between 2 and 10. Fully connected parallel systems are considered to generate such task graphs. Therefore, the lengths of the optimal schedules given in Table 4.2 are only true if these task graphs are scheduled on fully connected parallel systems with the number of processors specified in this table.

4.2 Design of the LGA

In this section, basic components of LGA are discussed one by one in detail. These components are string representation of schedules, fitness function, method to create initial population, genetic operators (reproduction, crossover, and mutation), and control parameters.

4.2.1 String Representation

String representation used in LGA is based on several lists of computational tasks. Each list corresponds to tasks scheduled on some specific processor, therefore, the number of lists are equal to the number of processors in a parallel system. The order of the tasks in a list indicates the order of execution. To ensure that this representation presents *feasible schedules*, tasks are arranged in descending order of their levels in each list. A schedule is called feasible if the precedence constraints between tasks are satisfied and every task appears exactly once in the schedule.

The level of a task (T_i) in a task graph is denoted by $l(T_i)$ and defined as follows:

$$l(T_i) = \mu(T_i) + \begin{cases} 0 & \text{if } succ(T_i) = \emptyset \\ \text{Max} \{ l(T_j) + \eta(T_i, T_j) : T_j \in succ(T_i) \} & \text{otherwise.} \end{cases} \quad (4.1)$$

Where,

$succ(T_i)$ is the set of successors of T_i .

$\mu(T_i)$ is the execution time of T_i .

$\eta(T_i, T_j)$ is the communication cost between T_i and T_j .

This ordering maintains the precedence constraints between tasks assigned to the same processor and ignores the precedence constraints between tasks assigned to different processors. This is because the precedence constraints between tasks assigned to different processors do not come into play until the moment of calculating the finishing time of the schedule. Figure 4.1 shows an example of a string representation for the task graph and the system graph presented in Figure 2.1.

4.2.2 Fitness Function

A suitable *objective function* is required to formulate the scheduling problem as an optimization problem. The objective function used in LGA is based on the finishing time of the schedule. The finishing time (FT) of a schedule (S) is defined as follows:

$$FT(S) = \text{Max}_{1 \leq i \leq m} \{ \text{finishing time } (P_i) \} \quad (4.2)$$

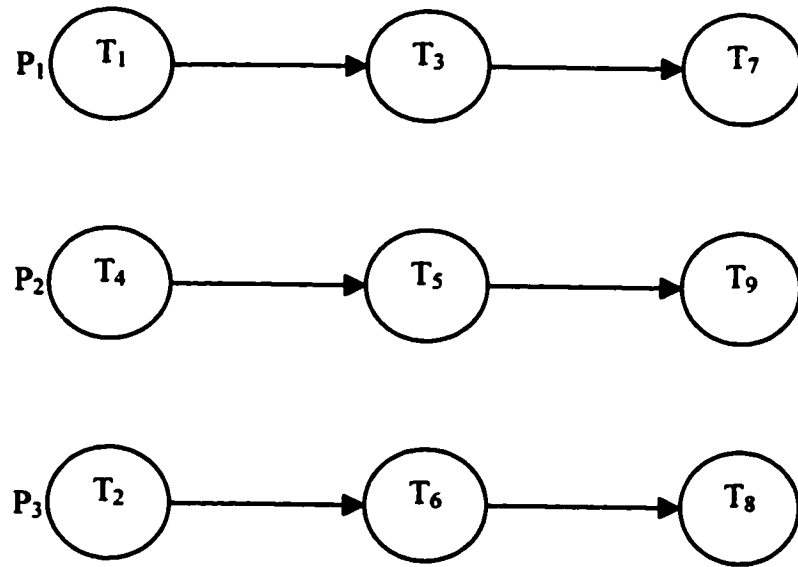


Figure 4.1: String representation

Where,

finishing time (P_i) is the finishing time of the last task on processor P_i .

m is the number of available processors.

Genetic algorithms work naturally on the maximization problem, while the mentioned objective function (finishing time of a schedule) has to be minimized. Therefore, it is necessary to convert the objective function into maximization form called *fitness function*. The fitness function can be defined as follows:

$$f(S) = FT_{max} - FT(S) \quad (4.3)$$

Where,

$f(S)$ is the fitness value of schedule S .

FT_{max} is the maximum finishing time observed in the current population.

4.2.3 Initial Population

In LGA, initial population is constructed randomly. The technique to generate initial population is divided into two steps, namely, preprocessing and assigning. In the preprocessing step, tasks are separated into sets according to their levels and the sets are arranged in descending order of levels. The preprocessing step is done only once. In the assigning step, a task is picked randomly from a set and assigned randomly to a processor until all tasks of this set are assigned. This process is performed for all sets one after the other as they are arranged. In this way, a string of the initial population is constructed. The assigning step is repeated until all strings of the initial population are

generated. This technique guarantees that the distribution of tasks among processors is uniform and tasks assigned to a processor are arranged in descending order of their levels.

4.2.4 Reproduction

The level-based genetic algorithm (LGA) uses *roulette wheel* technique for reproduction. Refer to Section 1.2 for detailed discussion of this technique. The size of the mating pool generated by reproduction equals to the population size.

4.2.5 Crossover

Crossover is the main operator in LGA. It takes a pair of parent strings (schedules) and swaps their substrings with each other to produce two new strings. In the scheduling problem, the swapping of substrings changes the assignment of tasks to processors. The crossover operator, adopted from [48], is described in the following paragraph.

First, a task T_i is selected randomly from the task graph. Then, immediate successors of task T_i are selected. The set containing task T_i and all its immediate successors is called *swapped_tasks*. Let the first parent string denoted by X and the second parent string denoted by Y . Next for each processor, P_j , in both strings X and Y do the following steps. Extract the tasks in *swapped_tasks* from the list of tasks assigned to processor P_j . The set of extracted tasks from string X and the set of extracted tasks from string Y are called *extracted_X* and *extracted_Y*, respectively. Then, insert the tasks in *extracted_Y*

into the list of tasks assigned to processor P_j in string X . Similarly, insert the tasks in *extracted_X* into the list of tasks assigned to processor P_j in string Y . The insertion is done in such a way that the tasks arranged in descending order of their levels. Thus, the crossover operator generates feasible strings. A working example of the crossover operation is shown in Figure 4.2 for the task graph shown in Figure 2.1. The tasks in *swapped_tasks* are T_6 , T_7 , and T_8 for this example.

4.2.6 Mutation

In LGA, mutation randomly selects a task T_i in a schedule S and removes it from the processor P_j to which it is assigned. Then, it selects another processor P_l randomly and inserts the selected task T_i into the list of tasks assigned to this processor. The insertion is done in such a way that the descending order of the tasks according to their levels is not violated.

4.2.7 Control Parameters

Crossover probability, *mutation probability*, and *population size* are control parameters on which the performance of genetic algorithms depends. The setting of these parameters is very difficult and requires extensive experimentation. We relayed on the empirical results that exist in the literature to set these parameters for LGA. The control parameters used in LGA are as follows:

- Crossover probability : 0.9

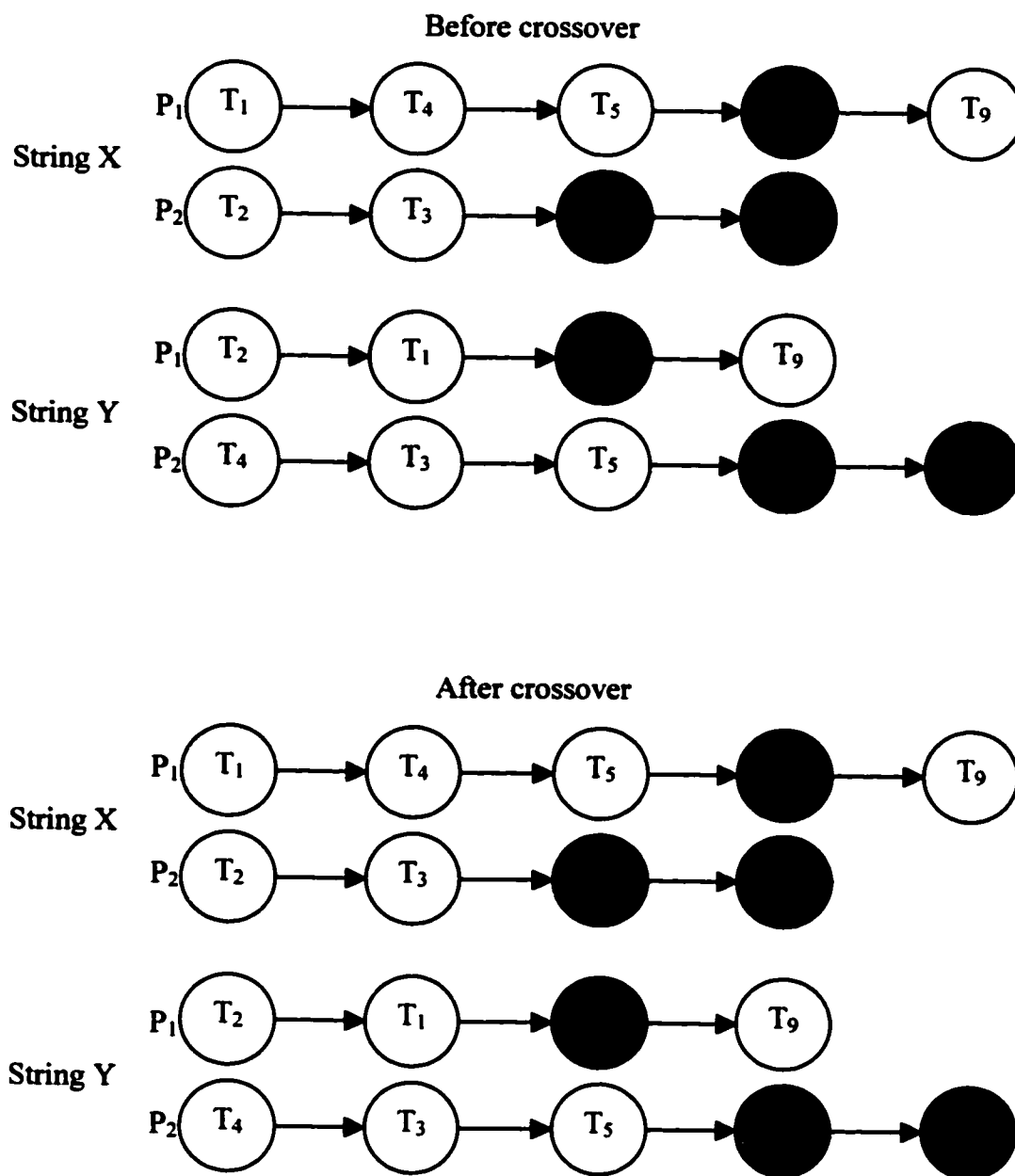


Figure 4.2: An example of crossover operation

- Mutation probability : 0.05
- Population size : 50

Now, we can combine all the components discussed above to form the level-based genetic scheduling algorithm (LGA). The structure of LGA is shown in Figure 4.3.

4.3 Experimental Results

The proposed level-based genetic scheduling algorithm (LGA) has been implemented as a set of C procedures on a Sun Ultra Sparc 10 workstation and tested on task graphs presented in Table 4.1 and Table 4.2. To evaluate the effectiveness of the level based ordering technique used in LGA, we modified LGA such that the modified algorithm uses the ordering technique presented in [26, 48]. We called it *height-based genetic algorithm* and denoted it by HGA for short. HGA arranged tasks assigned to the same processor in ascending order of their heights. The height of a task (T_i) in a task graph is defined as follows:

$$height(T_i) = \begin{cases} 0 & \text{if } pred(T_i) = \emptyset \\ 1 + \underset{T_j \in pred(T_i)}{Max} height(T_j) & \text{otherwise.} \end{cases} \quad (4.4)$$

Where,

$pred(T_i)$ is the set of predecessors of T_i .

In order to evaluate the impact of various parameters on the performance of the algorithms, three experiments were conducted. These parameters are task graph size, system graph size, and system topology. the schedule length is the performance measure

P_s = Population size.

N_g = Number of generations.

P_c = Crossover probability.

P_m = Mutation probability.

Create initial population P_{current} of size P_s .

Compute the fitness value of each string in P_{current} .

For $i = 1$ to N_g do

 Begin

 Perform *reproduction* to construct mating pool of size P_s .

 For $j = 1$ to P_s do

 Begin

 Pick two strings from the mating pool.

 Generate a random number p .

 If $p < P_c$ then perform *crossover* and put the new strings in P_{new} .

 Otherwise, put the two strings picked in P_{new} .

 End.

 For $j = 1$ to P_s do

 Begin

 Perform *mutation* with probability P_m on strings of P_{new} .

 End.

 Compute the fitness value of each string in P_{new} .

$P_{\text{current}} \leftarrow P_{\text{new}}$.

 End.

Return string with highest fitness value in the last population.

Figure 4.3: Structure of the level-based genetic algorithm (LGA)

considered in the evaluation. Both algorithms are set to run for a fixed number of generations, which is set to be 10 times the number of tasks in a task graph. Henceforth, this linear relationship between the number of generations and the number of tasks in a task graph was considered in our experimentation. For LGA as well as HGA, the population size was fixed at 50 and the population was randomly seeded by using the *time* function of *C*.

The first experiment measures the effect of changing the task graph size and the system graph topology on the performance of the algorithms. In this experiment, the schedules generated by HGA and LGA for task graphs shown in Table 4.1 are compared. Two system topologies, namely, fully connected and mesh are considered, and the number of processors is fixed to eight. The results are shown in Table 4.3 and Table 4.4 for fully connected and mesh systems, respectively. In each table, the first column gives the name of the task graph. The length of the schedule generated by HGA and that generated by LGA are shown in the second and the third columns, respectively. The last column shows the percentage of improvement obtained in schedule quality by LGA over HGA. It is clear from the tables that the performance of LGA is consistently better than the performance of HGA for all cases. The improvement gained in schedule quality by LGA over the HGA is better for the mesh system compared to the fully connected system such that the average improvement is 6.0% for the mesh system and 5.89% for the fully connected system.

The second experiment measures the impact of changing number of application processors (A_p) on the performance of the algorithms. In this experiment, fully connected systems of sizes 4,8,16 and 32 of similar processors are considered. The com-

Task Graph	Schedule Length by HGA	Schedule Length by LGA	Improvement in Schedule Quality %
m100	224	223	0.45
m150	326	310	4.91
m200	441	412	6.57
m250	558	536	3.94
m300	704	660	6.25
m350	833	774	7.08
m400	886	826	6.77
m450	974	898	7.80
m500	1149	1042	9.31

Table 4.3: Comparison of HGA and LGA for task graphs generated by RPCG on 8-processor fully connected topology

Task Graph	Schedule Length by HGA	Schedule Length by LGA	Improvement in Schedule Quality %
m100	313	290	7.35
m150	438	408	6.85
m200	583	574	1.54
m250	778	745	4.24
m300	943	920	2.44
m350	1113	1049	5.75
m400	1208	1113	7.86
m450	1334	1198	10.19
m500	1538	1418	7.80

Table 4.4: Comparison of HGA and LGA for task graphs generated by RPCG on 8-processor mesh topology

parison results are provided in Table 4.5. In this table, a ratio greater than 1.0 indicates a case where LGA performed worse than HGA. As shown in the table, the schedules generated by LGA are better than or equal to those generated by HGA except for one case in which LGA performed worse than HGA.

The last experiment designed to measure how good are the schedules generated by the algorithms. Therefore, task graphs shown in Table 4.2 are scheduled by both LGA and HGA, and the results are shown in Table 4.6. The first column in the table gives the name of the task graph. The length of the optimal schedule is shown in the second column of the table. The length of the schedule generated by HGA and its quality compared to the optimal schedule are shown in the third and the fourth column, respectively. The two last columns in the table show the length of the schedule generated by LGA and its quality compared to the optimal schedule. It is evident from the table that there is no single case in which LGA technique performed worse than HGA technique. For LGA, the schedule quality in all cases is within 80% of the optimal and the average schedule quality is within 87% of the optimal, while for HGA the average schedule quality is within 69% of the optimal.

As noticed from the above experiments, LGA outperformed HGA under a wide range of parameters. This is because some feasible schedules cannot be generated by HGA for some scheduling problems. In fact, the search space of HGA may not contain any optimal schedule. A simple example of such problems is the scheduling of the task graph shown in Figure 4.4 [14] on two processors. The length of the optimal schedule for this problem is 13. LGA can generate the optimal schedule, while HGA can never reach it due to the height ordering restriction.

Task Graph	LGA/HGA			
	$A_p=4$	$A_p=8$	$A_p=16$	$A_p=32$
m100	0.98	0.99	1.00	0.96
m150	0.98	0.95	0.99	0.99
m200	0.94	0.93	0.98	1.00
m250	0.95	0.96	0.95	1.01
m300	0.92	0.94	0.97	0.99
m350	0.93	0.93	0.96	0.99
m400	0.92	0.93	0.98	0.99
m450	0.94	0.92	0.95	0.99
m500	0.90	0.91	0.97	0.97

Table 4.5: Relative comparison of HGA and LGA for task graphs generated by RPCG on fully connected topology of different number of processors (A_p)

Task Graph	Optimal Schedule Length	HGA		LGA	
		Schedule Length	Sol. Quality %	Schedule Length	Sol. Quality %
on140m5	310	416	74.52	362	85.63
on165m7	287	403	71.21	358	80.17
on191m6	375	523	71.70	442	84.84
on219m4	602	894	67.34	678	88.79
on293m7	503	742	67.79	579	86.87
on329m5	735	1097	67.00	844	87.08
on350m4	958	1451	66.02	1044	91.76
on373m3	1333	2059	64.74	1496	89.10

Table 4.6: Comparison of HGA and LGA for task graphs generated by OPCG

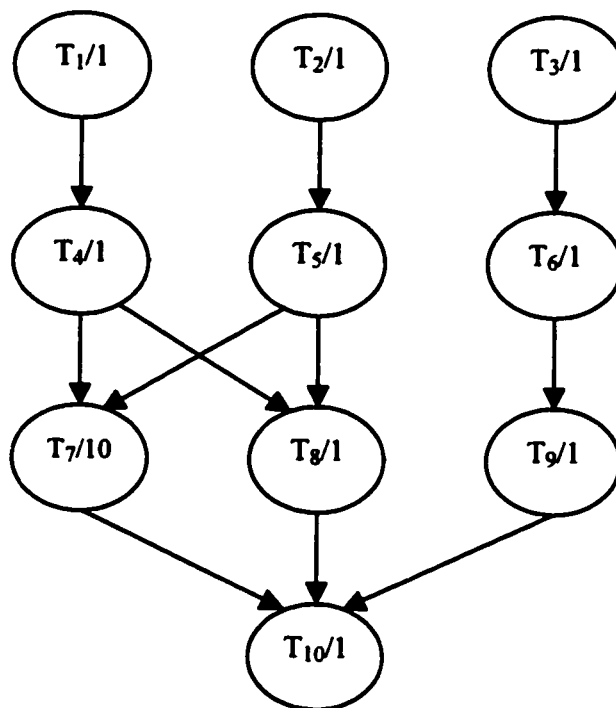


Figure 4.4: Example where the optimal schedule does not satisfy the height constraints

Chapter 5

Parallel Genetic Scheduling Algorithm

This chapter presents the parallel version of the level-based genetic scheduling algorithm. In Section 5.1, detailed discussion of the parallel level-based genetic algorithm (PLGA) is provided. The environment in which the experiments were performed and the implementation issues are presented in Section 5.2. Finally, the empirical results are shown in Section 5.3.

5.1 Design of the PLGA

The parallel level-based genetic algorithm (PLGA) for the scheduling problem is based on the island model of GA. The island model of GA was selected for two reasons. Firstly, the island GA can be implemented easily on a network of workstations. Secondly, the island model of GA requires less communication overhead compared with other models.

The basic idea of PLGA is to divide the population into several subpopulations; each of these subpopulations is randomly initialized and placed on a different scheduling processor (workstation). Each workstation runs the level-based genetic algorithm (LGA) on its own subpopulation, periodically selects the best string from its subpopulation and broadcasts copies of it to all other workstations. It will also receive copies of the best strings broadcasted from every other workstation. When a workstation receives a string, it replaces the string with the lowest fitness value in its subpopulation by the received string if the fitness value of the first is lower than the fitness value of the second.

It is possible that after a certain number of migration operations, each subpopulation contains a copy of the globally fittest string and only copies of this string migrate between subpopulations. Inserting copies of this string into subpopulations may lead to lose the diversity between subpopulations and thus drift the whole population to be prematurely converged. Therefore, each workstation broadcasts the best string in its subpopulation only if it is updated since last migration operation.

The structure of PLGA that runs on each workstation is shown in Figure 5.1. The genetic operators (reproduction, crossover, and mutation) as well as the fitness function and the method to construct initial population are the same as those used in the level-based genetic algorithm (LGA).

Subpopulation size assigned to each workstation affects both the quality of the schedule generated and the running time of PLGA. Therefore, to show the usefulness of PLGA for the scheduling problem, it is important to compromise between the schedule quality and PLGA running time by selecting a suitable subpopulation size.

In the first experiment, the affect of assigning a subpopulation of size (P_s/W) to each

SP_s = Subpopulation size.
 N_g = Number of generations.
 P_c = Crossover probability.
 P_m = Mutation probability.
 W = Number of scheduling processors (workstations).

Create initial subpopulation P_{current} of size SP_s .
 Compute the fitness value of each string in P_{current} .
 For $i = 1$ to N_g do
 Begin
 Best-string \leftarrow string with highest fitness value in the current subpopulation P_{current} .
 Perform *reproduction* to construct mating pool of size SP_s .
 For $j = 1$ to SP_s do
 Begin
 Pick two strings from the mating pool.
 Generate a random number p .
 If $p < P_c$ then perform *crossover* and put the new strings in P_{new} .
 Otherwise, put the two strings picked in P_{new} .
 End.
 For $j = 1$ to SP_s do
 Begin
 Perform *mutation* with probability P_m on strings of P_{new} .
 End.
 Compute the fitness value of each string in P_{new} .
 $P_{\text{current}} \leftarrow P_{\text{new}}$
 If (migration generation)
 Begin
 If *Best-string* is updated, send copies of it to all workstations.
 For $j = 1$ to $(W - 1)$ do
 Begin
 Receive copy of a broadcasted string.
 Replace string with lowest fitness value in P_{current} by the received
 string.
 End.
 End.
 End.
 End.

Figure 5.1: Structure of the parallel level-based genetic algorithm (PLGA)

workstation is studied, where $P_s = 50$ (population size used by LGA) and W is the number of workstations in a parallel machine. The parallel machine used had four workstations and the migration occurred after every generation (migration frequency = 1). PLGA selects subpopulation sizes in such a way that the sum of them equals the population size in LGA. Therefore, for the case of four workstations, one of the workstations has a subpopulation of size 14 and each of the remaining workstations has a subpopulation of size 12. We compared PLGA with LGA in terms of the schedule length and the running time. The comparison results are shown in Table 5.1. As shown in the table, PLGA performed better than LGA except for one case in which the schedule generated by PLGA is 2% poorer than the schedule generated by LGA. However, PLGA required at most 50% of the time required by LGA.

To improve the schedule quality, we conducted several experiments and tried different subpopulation sizes. In these experiments, a parallel machine consists of four workstations was used and the migration frequency was set to one. We found that the most suitable subpopulation size is $(P_s/W + 0.4P_s)$. By using this subpopulation size, the total number of strings in the whole population increases as the number of workstations increases in the parallel machine. We compared PLGA with LGA and the comparison results are presented in Table 5.2. It is clear from the table that the schedules generated by PLGA are better than the schedules generated by LGA. Furthermore, PLGA required at maximum 85% of the time required by LGA. Note that the running time of PLGA also depends on the migration frequency and can be reduced by using larger migration frequency. Henceforth, the subpopulation size is set to $(P_s/W + 0.4P_s)$ in our implementations.

Task Graph	LGA		PLGA	
	Schedule Length	R. Time (seconds)	Schedule Length	R. Time (seconds)
m100	223	22	212	11
m150	310	49	316	20
m200	412	93	403	33
m250	536	138	517	48
m300	660	206	649	68
m350	774	280	745	93
m400	826	371	790	124
m450	898	462	862	149
m500	1042	589	1013	187

Table 5.1: Comparison of LGA and PLGA for task graphs generated by RPCG on 8-processor fully connected topology ($SP_s = P_s/W$)

Task Graph	LGA		PLGA	
	Schedule Length	R. Time (seconds)	Schedule Length	R. Time (seconds)
m100	223	22	207	19
m150	310	49	306	39
m200	412	93	394	73
m250	536	138	514	106
m300	660	206	635	153
m350	774	280	741	209
m400	826	371	787	273
m450	898	462	862	339
m500	1042	589	1000	431

Table 5.2: Comparison of LGA and PLGA for task graphs generated by RPCG on 8-processor fully connected topology ($SP_s = P_s/W + 0.4 P_s$)

The performance of PLGA depends on the interval between two consecutive migrations, called migration frequency (MF). The setting of this parameter effect the running time of PLGA as well as the quality of the schedule obtained. Thus, we tested four migration frequencies and *no migration*. The comparison results are given in Table 5.3. Different migration frequencies show better results for different task graphs. This is normal given the random nature of subpopulations and generations. However, in all cases, results were better than *no migration* case. Even without migration, PLGA performed better than LGA for a number of task graphs. It is obvious from the table that migrating after every five generations gives better schedules for more task graphs. Therefore, we set the migration frequency to five in our experimentation.

Now we can summarize the interaction between scheduling processors (workstations) as follows. After every five generations, each workstation broadcasts copies of the best string in its subpopulation to all other workstations if the best string is updated since the last migration operation. Thus, in each migration operation, every workstation receives at most a total of $(W - 1)$ strings (W is the number of workstations in a parallel machine). When a string is received by a workstation, it replaces the worst string in its subpopulation by the received one. This is done only if the fitness value of the received string is higher than the fitness value of the worst string in the subpopulation.

5.2 Computational Environment

The level-based genetic algorithm (LGA), presented in chapter 4, was implemented on

Task Graph	LGA	PLGA				
		No Migration	$MF = 1$	$MF = 5$	$MF = 10$	$MF = 20$
m100	223	207	207	205	207	204
m150	310	320	306	302	300	310
m200	412	414	394	395	399	395
m250	536	527	514	504	520	514
m300	660	660	635	628	630	631
m350	774	753	741	729	747	748
m400	826	816	787	786	788	791
m450	898	886	862	861	856	848
m500	1042	1038	1000	1003	1009	1006

Table 5.3: Comparison of migration frequencies (MF) for task graphs generated by RPCG on 8-processor fully connected topology

SUN Ultra Sparc 10 workstation running Solaris operating system. Therefore, to show the correctness of the parallel version of the level-based genetic algorithm (PLGA) and to evaluate its performance relative to the performance of LGA, PLGA is implemented on a network of SUN Ultra Sparc 10 workstations. In order to simplify the usage of the network of SUN workstations as a parallel machine, a software package called *Parallel Virtual Machine* (PVM) is used. The development of PVM started in 1989 at Oak Ridge National Laboratory and is now an ongoing research project. An overview of this package is presented in the following subsection.

5.2.1 PVM Overview

The PVM is an integrated set of software tools and libraries that enables a network of heterogeneous Unix computers to be used as a single high-performance parallel machine. Therefore, large computational problems can be solved by using the aggregate power of many computers. PVM transparently handles all data conversion that may be required if two computers use different integer or floating point representation. It also handles all message routing and task scheduling across the network of incompatible computers.

The PVM computing model is based on the notion that an application program consists of several cooperating tasks. Each task is responsible for a part of the application's computational workload. These tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks as well as communication and synchronization between them.

PVM tasks may possess arbitrary control and dependency structures. Therefore, at any point in the execution of a parallel application, any PVM task may start or stop other tasks or add or delete computers from the parallel machine. In addition, any PVM task may communicate or synchronize with any other task. Therefore, the general form of MIMD parallel computation is possible.

The PVM system is composed of two parts. The first part is a daemon, called *pvmd*. The *pvmd* resides on all the computers making up the parallel virtual machine. To run a PVM application, *pvmd* must be executed on one of the computers that in turn starts up *pvmd* on all other computers making up the user-defined parallel virtual machine. Then, the PVM application can be started from a Unix prompt on any of these computers.

The library of PVM interface routines is the second part of the system. Users can call these routines for messages passing, processes spawning, tasks coordinating, and virtual machine modifying.

C, C++, and FORTRAN languages are currently supported by the PVM system. This is due to the widespread usage of these languages for large applications.

5.2.2 Implementation Issues

PLGA is based on the master-slave model of programming. The master process provides the user interface and manages all other processes called slave processes. The total number of processes is equal to the number of workstations in a parallel machine and each process is assigned to a different workstation. In this way, overloading the workstations is avoided.

The master process obtains from the user information about the problem to be solved. This information are the number of application processors, the number of application tasks, the precedence relations between tasks, the communication costs between tasks, the communication costs between application processors. It then starts slave processes. Once this is done, the master process sends each slave process the information obtained from the user about the problem. It also sends each slave process some information about other processes. Broadcasting this information helps slave processes to communicate with each other.

Each slave process sends the best schedule in its subpopulation to the master process before termination. When all slave processes have terminated, the master process outputs the best overall schedule. Master process also does the timing of the program. The implementation of the parallel level-based genetic algorithm is shown in Figure 5.2.

5.3 Experimental Results

Our hypothesis was that PLGA would generate schedules of quality better than or equal to the quality of schedules generated by LGA. Further, the running time of PLGA would be less than the running time of LGA.

The parameters used in PLGA are same as those used in LGA. The crossover probability and the mutation probability are set to 0.9 and 0.05, respectively. PLGA is set to run for a fixed number of generations, which is set to be 10 times the number of tasks in a task graph. In PLGA, each subpopulation is randomly seeded by using the *time* function of *C* plus an integer number. Each process uses a different integer number from

W = Number of scheduling processors (workstations).

Get task graph and system graph.

Get list of workstations.

For $j = 1$ to $(W - 1)$ do

 Begin

 Create a slave process.

 End.

For $j = 1$ to $(W - 1)$ do

 Begin

 Send task graph, system graph, and list of workstations.

 End.

{Execute the parallel level-based genetic algorithm (PLGA). (Figure 5.1)}

Receive the fittest string of each slave process.

Return the overall fittest string.

(a) Master process

Receive task graph, system graph, and list of workstations from the master process.

{Execute the parallel level-based genetic algorithm (LPGA). (Figure 5.1)}

Send the fittest string in the last subpopulation to the master process.

(b) Slave process

Figure 5.2: Implementation of the parallel algorithm

from other processes that is a unique integer number.

In order to evaluate the correctness and the effectiveness of PLGA, two experiments have been conducted and the performance of PLGA is compared with the performance of LGA. Schedule length and running time are the performance measures considered in the evaluation. In PLGA, the running time is measured from the start of the master process to its finish. All the experiments were performed when there is no load on the network.

In the first experiment, task graphs shown in Table 4.1 are scheduled on eight processors connected in two system topologies, namely, fully connected and mesh. The ratios of the lengths of schedules generated by PLGA using 2,4,8, and 16 workstations (W) to the lengths of schedules generated by LGA are shown in Table 5.4 and Table 5.5 for fully connected and mesh systems, respectively. In these tables, a ratio greater than 1.0 indicates a case where the length of the schedule generated by PLGA is larger than the length of the schedule generated by LGA. Out of the 72 comparisons shown in these tables, there were only three cases in which the schedules generated by PLGA are worse than or equal to those generated by LGA. However, the lengths of the schedules generated in these cases by PLGA are within 1% of those generated by LGA.

The speedup in running times of PLGA using 2,4,8, and 16 workstations over LGA are shown in Table 5.6 for the fully connected system and in Table 5.7 for the mesh systems. It is observed that the speedup increases as the number of workstations increases in the parallel machine.

The second experiment measures how good are the schedules generated by PLGA when it is implemented on various number of workstations. Therefore, the task graphs

Task Graph	PLGA/LGA			
	$W = 2$	$W = 4$	$W = 8$	$W = 16$
m100	0.91	0.92	0.91	0.91
m150	1.01	0.97	0.95	0.96
m200	0.98	0.96	0.96	0.97
m250	1.00	0.94	0.93	0.95
m300	0.97	0.95	0.95	0.93
m350	0.97	0.94	0.96	0.94
m400	0.96	0.95	0.93	0.95
m450	0.98	0.96	0.93	0.90
m500	0.97	0.96	0.95	0.94

Table 5.4: Ratios of schedule lengths generated by PLGA to those of LGA for task graphs generated by RPCG on 8-processor fully connected topology

Task Graph	PLGA/LGA			
	$W = 2$	$W = 4$	$W = 8$	$W = 16$
m100	0.98	0.97	0.90	0.91
m150	1.00	0.94	0.96	0.95
m200	0.94	0.93	0.92	0.89
m250	0.92	0.89	0.90	0.87
m300	0.92	0.91	0.90	0.92
m350	0.96	0.93	0.90	0.87
m400	0.96	0.93	0.92	0.89
m450	0.97	0.91	0.94	0.90
m500	0.96	0.95	0.92	0.90

Table 5.5: Ratios of schedule lengths generated by PLGA to those of LGA for task graphs generated by RPCG on 8-processor mesh topology

Task Graph	LGA/PLGA			
	$W = 2$	$W = 4$	$W = 8$	$W = 16$
m100	1.05	1.37	1.57	1.57
m150	1.07	1.40	1.63	1.75
m200	1.07	1.41	1.66	1.79
m250	1.07	1.42	1.62	1.79
m300	1.08	1.44	1.70	1.79
m350	1.07	1.43	1.71	1.77
m400	1.08	1.40	1.72	1.88
m450	1.07	1.30	1.73	1.90
m500	1.07	1.43	1.72	1.91

Table 5.6: The speedup in the running times of PLGA over LGA for 8-processor fully connected topology

Task Graph	LGA/PLGA			
	$W = 2$	$W = 4$	$W = 8$	$W = 16$
m100	1.04	1.44	1.64	1.64
m150	1.06	1.32	1.63	1.69
m200	1.07	1.45	1.68	1.74
m250	1.08	1.43	1.69	1.78
m300	1.07	1.44	1.70	1.79
m350	1.08	1.45	1.72	1.85
m400	1.08	1.44	1.72	1.87
m450	1.07	1.45	1.73	1.90
m500	1.07	1.44	1.72	1.90

Table 5.7: The speedup in the running times of PLGA over LGA for 8-processor mesh topology

generated by OPCG and shown in Table 4.2 are scheduled by PLGA. The performance of PLGA is compared to the performance of LGA, and the comparison results are shown in Table 5.8. As shown in the table, there is only one case in which LGA performed better than PLGA.

As noticed from the experiments, the performance of PLGA is unpredictable that is increasing the number of workstations does not guarantee improving the quality of the schedule generated. This is because PLGA is stochastic and its performance depends on the seeds used by the subpopulations. However, the schedules generated by PLGA are better than or equal to those generated by LGA. Even if the schedule generated by LGA is better than the schedule generated by PLGA, the difference in the quality is insignificant such that the difference will be less than 1%.

On the other hand, the speedup achieved by PLGA increases as the number of workstation increases. This is due to the division of the computation over the workstations. It also indicates that the communication overhead in PLGA is small comparing to the computation. This is because of the following two reasons. Firstly, only one string is broadcasted, this is the best string in a subpopulation. Secondly, the best string is broadcasted only if it is updated since last migration operation.

At the end, we would like to offer the following caution about the results that we have presented. Each result is stochastic; that is it depends on the particular seed used by the population. We liked to be able to present the results as averages for each test problem obtained over a large number of trails. However, at the time we did this work, workstations' time on the network were in demand.

Task Graph	LGA	PLGA			
		$W = 2$	$W = 4$	$W = 8$	$W = 16$
on140m5	85.63	86.11	92.54	88.32	91.71
on165m7	80.17	84.66	86.97	88.31	88.04
on191m6	84.84	86.40	88.03	89.07	88.65
on219m4	88.79	92.33	93.62	93.04	91.91
on293m7	86.87	89.50	88.71	92.29	91.79
on329m5	87.08	88.13	90.07	92.69	92.22
on350m4	91.76	92.20	92.92	94.94	95.32
on373m3	89.10	91.87	94.88	95.62	96.73

Table 5.8: Comparison of LGA and PLGA in terms of the quality of the schedules for task graphs generated by OPCG

Chapter 6

Conclusion

In this thesis, the primary objective has been to design a fast and efficient parallel genetic algorithm for scheduling parallel tasks on multiprocessor systems. Inter-task and inter-processor communication overhead is assumed part of the problem formulation. To achieve this objective, an efficient sequential genetic scheduling algorithm was designed upon which the parallel genetic scheduling algorithm was developed. The sequential algorithm and the parallel algorithm were denoted by LGA and PLGA, respectively.

PLGA was designed for loosely coupled multiprocessor systems and implemented on a network of SUN workstations. To simplify the usage of these workstations as a parallel machine, a software package called Parallel Virtual Machine (PVM) was used. PVM is a collaborative venture between Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University and is now an ongoing research project.

The main conclusions of this thesis are the following.

1. Some feasible schedules cannot be generated for some scheduling problems if height based ordering technique is used to maintain the precedence constraints between tasks assigned to the same processor. In contrast, level based ordering technique does not have this severe drawback. Thus, schedules generated by HGA are worse than those generated by LGA.
2. Generally, the schedules generated by PLGA are better than or equal to the schedules generated by LGA. Even if the schedule generated by LGA is better than that generated by PLGA, the difference in the quality will be insignificant.
3. Although increasing number of workstations in a parallel machine increases the total number of strings in the whole population, it does not guarantee improving the quality of the schedule generated by PLGA. This is because the performance of PLGA depends on the seeds used by the subpopulations.
4. The speedup in the running time achieved by PLGA over LGA increases as the number of workstations in a parallel machine increases. This is because the computation is distributed over workstations. In addition, this is an indication that the communication overhead in PLGA is small.

A number of interesting areas for future research are:

1. The termination strategy used, stopping after a specified number of generations, is not viable in practice. Thus, an effective termination strategy is needed. One approach might be to stop when the best schedule has not changed for a specified number of generations.

2. The proposed scheduling algorithm is a pure genetic algorithm. It may be improved by adding some knowledge about the scheduling problem to it. This can be done in the initialization or in the crossover or mutation operations.
3. Additional work to determine good choices for the parameters of PLGA is needed. These parameters are the migration frequency, the migration paths between subpopulations, the number of string migrated, and the strategy used to select strings for migration.
4. The current implementation of PLGA is synchronous; that is a workstation does not continue executing GA until it receives migrated strings from other workstations. An asynchronous implementation is also possible. In this case, a workstation periodically checks its incoming messages queue for migrated strings that have been sent from other workstations, and if any are found, the workstation can insert them into the subpopulation.

References

- [1] T. L. Adam, K. Chandy, and J. Dickson, "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, 17(12): 685-690, Dec. 1974.
- [2] I. Ahmad and M. K. Dhodhi, "Task assignment using a problem-space genetic algorithm," *Concurrency: Practice and Experience*, 7(5): 411-428, August 1995.
- [3] I. Ahmad and M. K. Dhodhi, "Multiprocessor scheduling in a genetic paradigm," *Parallel Computing*, 22: 395-406, 1996.
- [4] I. Ahmad and M. Kafil, "A parallel algorithm for optimal task assignment in distributed systems," *Proceedings of Advances in Parallel and Distributed Computing*, pages 284-290, 1997.
- [5] N. Ansari, M.-H. Chen, and E. S. H. Hou, "Point pattern matching by genetic algorithm," *16th Annual Conference on IEEE Industrial Electronics*, pages 1233-1238, 1990.
- [6] M. S. T. Benten and S. M. Sait, "Genetic scheduling of task graphs," *International Journal of Electronics*, 77(4): 401-415, 1994.

- [7] R. Bettati and J. W.-S. Liu, "End-to-end scheduling to meet deadline in distributed systems," *12th International Conference on Distributed Computing Systems*, pages 452-459, 1992.
- [8] M. Bozyigit and A. A. Abdulghani, "Parallel scheduling algorithm for parallel Applications," *Parallel Algorithms and Applications*, 6: 303-316, 1995.
- [9] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, 14(2): 141-154, February 1988.
- [10] A. Chipperfield and P. Fleming, "Parallel genetic algorithms," *Parallel and Distributed Computing Handbook*, A. Y. Zamaya (Ed.), McGraw-Hill, pages 1118-1143, 1996.
- [11] T. Chockalingam and S. Arunkumar, "A randomized heuristics for the mapping problem: the genetic approach," *Parallel Computing*, 18: 1157-1165, 1992.
- [12] T. Chockalingam and S. Arunkumar, "Genetic algorithm based heuristics for the mapping problem," *Computers and Operations Research*, 22(1): 55-64, 1995.
- [13] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, John Wiley, 1976.
- [14] R. C. Correa, A. Ferreira, and P. Rebreyend, "Scheduling multiprocessor tasks with genetic algorithms," *IEEE Transactions on Parallel and Distributed Systems*, 10(8): 825-837, August 1999.

- [15] L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.
- [16] E. Dekel and S. Sahni, "Binary trees and parallel scheduling algorithms," *IEEE Transactions on Computers*, C-32 (3): 307-315, March 1983.
- [17] E. Falkenauer and A. Delchambre, "A genetic algorithm for bin packing and line balancing," *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1186-1192, 1992.
- [18] T. C. Fogarty and R. Huang, "Implementing the genetic algorithm on transputer based parallel processing systems," *Parallel Problem Solving from Nature*, H.-P. Schwefel and R. Manner (Eds.), pages 145-149, 1991.
- [19] B.-P. Gan and S.-Y. Huang, "Scheduling dynamically evolving parallel programs using the genetic approach", *Proceedings of the fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, pages 290-295, 2000.
- [20] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine-A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.
- [21] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

- [22] V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, McGraw-Hill, 1996.
- [23] B. Hamidzadeh, D. J. Lilja, and Y. Atif, "Dynamic scheduling techniques for heterogeneous computing systems," *Concurrency: Practice and Experience*, 7(7): 633-652, October 1995.
- [24] R. Hauser, R. Manner, and M. Makhaniok, "NERV: a parallel processor for standard genetic algorithms," *Proceedings of 9th International Parallel Processing Symposium*, pages 782-789, 1995.
- [25] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.
- [26] E. S. H. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed Systems*, 5(2): 113-120, February 1994.
- [27] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Research*, 9: 841-848, May 1961.
- [28] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal of Computing*, 18(2): 244-257, April 1989.

- [29] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Transactions on Computers*, C-33 (11): 1023-1029, November 1984.
- [30] K. Kojima, W. Kawamata, H. Matsuo, and M. Ishigame, "Network based parallel genetic algorithm using client-server model," *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 244-250, 2000.
- [31] A. Kumar, A. Srivastava, A. Singru, and P. K. Ghosh, "Robust and distributed genetic algorithm for ordering problems," *Proceedings of 5th IEEE International Symposium on High Performance Distributed computing*, pages 253-262, 1996.
- [32] Y.-K. Kwok and I. Ahmad, "Dynamic critical path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, 7(5): 506-521, May 1996.
- [33] S. Mohan and P. Mazumder, "Wolverines: standard cell placement on a network of workstations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(9): 1312-1326, September 1993.
- [34] D. Moldovan, *Parallel Processing: From Applications to Systems*, Kaufmann, 1993.
- [35] M. A. Palis, J.-C. Liou, and D. S. L. Wei, "Task clustering and scheduling for distributed memory parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, 7(1): 46-55, January 1996.

- [36] C. A. G. Pico, "Dynamic scheduling of computer tasks using genetic algorithms," *Proceedings of the first IEEE Conference on Evolutionary Computing*, pages 829-833, 1994.
- [37] H. Pirkul and E. Rolland, "New heuristic solution procedures for the uniform graph partitioning problem: extensions and evaluation," *Computers and Operations Research*, 21(8): 895-907, 1994.
- [38] C. P. Ravikumar and A. K. Gupta, "Genetic algorithm for mapping tasks onto a reconfigurable parallel processor," *IEE Proceedings-Computers and Digital Technology*, 142(2): 81-86, 1995.
- [39] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, 9: 138-153, 1990.
- [40] H. El-Rewini, T. G. Lewis, and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Prentice Hall, 1994.
- [41] K. Schwan and H. Zhou, "Dynamic scheduling of hard real-time tasks and real-time threads," *IEEE Transactions on Software Engineering*, 18(8): 736-748, August 1992.
- [42] R. Sethi, "Scheduling graphs on two processors," *SIAM Journal of Computing*, 5(1): 73-82, March 1976.

- [43] H. Tamaki and Y. Nishikawa, "A parallel genetic algorithm based on a neighborhood model and its application to the jobshop scheduling," *Parallel Problem Solving from Nature*, R. Manner and B. Manderick (Eds.), pages 573-582, 1992.
- [44] R. Tanese, "Parallel genetic algorithm for a hypercube." *Proceedings of the second International Conference on Genetic Algorithms*, pages 177-183, 1987.
- [45] P.-C. Wang and W. Korfhage, "Process scheduling using genetic algorithms," *Proceedings of seventh IEEE Symposium on Parallel and Distributed Processing*, pages 638-641, 1995.
- [46] D. Whitley, "A genetic algorithm tutorial," *Statistics and Computing*, 4: 65-85, 1994.
- [47] S. H. Woo, H. S. Lee, S. B. Yang, T. D. Han, and S. D. Kim, "Multiprocessor scheduling with genetic algorithms in heterogeneous environment," *PDPTA '97*, pages 928-931, 1997.
- [48] S.-H. Woo, S.-B. Yang, S.-D. Kim, and T.-D. Han, "Task scheduling in distributed computing systems with a genetic algorithm," *HPC Asia '97*, pages 301-305, 1997.
- [49] W. Xiao, P. Hao, S. Zhang, and X. Xu, "Hybrid flow shop scheduling using genetic algorithms," *Proceedings of the 3rd World Congress on Intelligent Control and Automation*, pages 537-541, 2000.