USING GENETIC ALGORITHMS TO SCHEDULE MULTIPROCESSOR SYSTEMS UNDER LOGP MODEL

YU CHEN

A THESIS
IN
THE DEPARTMENT
OF
ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF APPLIED SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

JANUARY 2006

Your file Votre référence
ISBN: 0-494-14254-5
Our file Notre référence
ISBN: 0-494-14254-5

Abstract

Using Genetic Algorithms to Schedule Multiprocessor Systems under LogP Model

Yu Chen

In recent years, with the wide-spreading usage of computer technologies in various aspects of the modern world, the demand for more powerful computers has outmatched the yet rapid advancement in hardware development. LogP model is a practical model that reflects better the practical behavior of nowaday massively parallel computers

This thesis is dedicated to the design and evaluation of algorithms for multiprocessor system scheduling under the LogP model. The objective is to obtain a feasible schedule of input task graphs and corresponding LogP model with minimum *makespan*. Due to the NP-hard nature of the problem, we choose the genetic algorithms (GA) to effectively explore the huge solution space.

The approach consists of two main parts. The communication tasks under the LogP model are scheduled by a genetic algorithm with determined processor assignments. Another GA is used to optimize the processor assignments of computational tasks. The design issues in both GA algorithms are discussed in detail.

The evaluation of both parts of the algorithm over a set of benchmark task graphs shows an overall improvement over previous works within the LogP model.

Acknowledgments

I am indebted to the members of my thesis examining committee for their time and suggestions during the thesis defense.

It was a great pleasure to have Dr Nawwaf Kharma as my supervisor. He has been an unfailing source of knowledge, encouragement and support. He gave me much invaluable comments and also careful proof-reading. Without his supervision and expert knowledge in Genetic Algorithms, this thesis would not have been possible.

Thanks also go to Dr Skander Kort, my former supervisor, who introduced me to Multiprocessor Scheduling. He guided me through the early parts of my project and provided me a system SHELA to work on.

I would also like to thank Dr Yuke Wang, Dr Liang Chen, Dr Weiping Zhu, Dr Chunyan Wang and Dr Asim Al-Khalili for their suggestions and advices on my study and research. I am grateful for all the help I received from the members of the Department of Electrical and Computer Engineering.

Many thanks to my friends who provided help, advice and companionship during my study in Concordia University. I specially express my gratitude to Zhuoyan Li for his help. Finally, I thank my parents, my sister Linda Chen and my husband Yi Zhang for their encouragement and company. This work is dedicated to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Technological advances and users demand the evolution of computer architectures. The concept of multiprocessor was proposed as a means of exceeding the maximum performance of a single processor. Scheduling is the management of resources that have to be allocated to activities over time, subject to a number of constraints. The main multiprocessor scheduling problem concerns assigning partitioned tasks of application programs to processors.

An appropriate distribution of both computation and communication of the application programs is the key issue to achieving efficient execution in multiprocessor scheduling. Culler *et al.* [15] defined a realistic machine model which better reflects the practical behavior of massively parallel computers. Their LogP model considers dominant communication characteristics of each individual operation, as well as the limited capacity of communication networks.

Scheduling under the LogP model is NP-hard even for special cases such as join trees of *height*[1] one [30]. Only a few scheduling strategies (discussed in detail in Section 3.3) have been studied for this problem. Most of these strategies ignore the network capacity constraint of the LogP model and only deal with simple tree-like graphs.

In comparison, a large number of strategies for static scheduling under the standard delay model (SDM)[2] have been studied. Genetic Algorithms have successfully been applied to this problem and also have been superior to the list scheduling approaches.

This thesis presents genetic algorithms for scheduling general task graphs under the LogP model taking into account the limited number of P processors.

1.2 Outline of the Thesis

The rest of the thesis is organized as follows.

Section 2.1 and 2.2 in Chapter 2 give a brief description of the multiprocessor systems and parallel computing models addressed in this thesis. Section 2.3 and 2.4 present an overview of the scheduling problem, and the approaches proposed for solving them.

Section 3.1 in Chapter 3 formally defines the scheduling problem under the LogP model with a problem instance shown in Section 3.2. Previous work in related areas

[1]The *topological height* (height) is defined as the largest number of edges from an entry node to the node itself

[2]Standard delay models consider the latency for a message transmission is the only relevant communication parameters.

is briefly described in Section 3.3. Section 3.4 contains a summary of the major contributions of this thesis.

Chapter 4 describes in detail the genetic algorithm used for communication management when scheduling under LogP. Section 4.1 presents a method of extending task graphs under the LogP model, which generalizes the communications. Details of the genetic algorithm are described in the rest of the chapter.

Chapter 5 improves the algorithm proposed in Chapter 4 in two ways: the encoding method is modified in Section 5.1, and, in Section 5.2, the algorithm is adapted to more general LogP models, including models with significant gaps.

Chapter 6 describes processor allocations using genetic algorithms. The processor allocation problem is formalized in Section 6.2. Section 6.3 presents the algorithms in a step-wise fashion. Finally, Section 6.4 discusses further details of this algorithm.

Chapter 7 presents evaluation results to demonstrate the effectiveness of both GAs on 3 sets of benchmark task graphs. These graphs are described in Section 7.1. Section 7.2 presents the models used in the experiments. Section 7.3 and 7.4 describe the metrics used for evaluation. Section 7.5 is devoted to the five parts of the procedure.

Chapter 8 summarizes with some concluding remarks. Section 8.2 presents possible future directions for research.

Appendix A contains data on the benchmark graphs used in the simulations. This data includes detailed characteristics of the graphs.

Chapter 2

Preliminaries

This chapter contains background information about the multiprocessor scheduling problem.

2.1 Multiprocessor Systems

The term "multiprocessor" can label any system that applies more than one processor to perform desired applications. However, in most cases, it refers to general-purpose, asynchronous parallel machines with multiple instruction-steams and multiple data-streams (MIMD) [26].

Ideally, a system with n identical processors could offer n times the throughput of a single processor. But, the ideal case cannot be realized due to the effort involved in operation control and information transmissions among the processors.

A *homogeneous multiprocessor system* is composed of a set $\mathbf{P} = \{1, \cdots, m\}$ of m identical processors. These processors are fully inter-connected in a network, where all links are identical. A processor can communicate asynchronously through one or several of its links simultaneously.

2.2 Parallel Computing Models

Parallel computing refers to solving a problem faster by employing multiple processors simultaneously. Parallel computing models are key to the design of parallel programming environments. Various abstract models will be discussed in the following section.

Abstract models, also called computational models, are high-level machine descriptions that are referred to in algorithm design. They are used to describe the architectural classes in realistic terms. In an abstract model, the essential features of an architecture is identified and reflected in the model. In addition, an abstract model should be able to represent a large class of architectures, including possible future architectures. The requirements for a suitable abstract model have been formulated in [33] as:

- *Accuracy*

  The cost estimates should be close to the real time assumptions.

- *Simplicity*

  The cost model should depend on only a few parameters of program and architecture.

- *Monotonicity in the Architectural Parameters*

When the architecture is improved, the predicted costs should decrease.

2.2.1 The Models

In the following, we give a brief description of various abstract parallel computing models [33].

Network Models Network models are the models describing low-level features of the architecture, for instance the topology of the interconnection network and the fact of insecure message transmission.Network models focus on the interconnection topology. Most models assume synchronous operation. Figure 2.1 shows an example of network models.

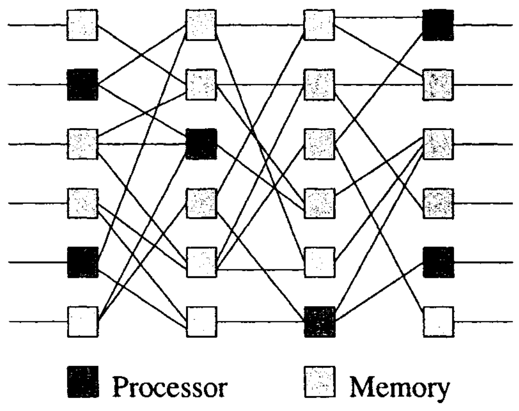


Figure 2.1: A network model

This kind of models have almost disappeared nowadays. Firstly, the models move whole packets from node to node: this is no longer sufficiently general. Second, the models are hard to use due to the excessive details of the architectures. Moreover, lacking portability is another serious drawback. Network models are now usually used for the design of elementary algorithms, such as routing, sorting and broadcasting.

PRAM Parallel random access machine (PRAM) is one of the most popular models in parallel algorithm design.

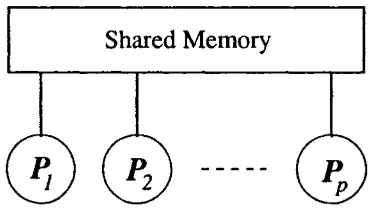


Figure 2.2: The PRAM model

A PRAM consists of p processors that have access to a global shared memory. Besides, each processor owns a small amount of local memory for registers and codes. It is a synchronous MIMD model. Every processor can access any memory location in one step regardless of the memory location.

The popularity of the PRAM is due to its simplicity. Generally, in PRAM models, the architectural details were ignored and the execution was supposed to be synchronous. PRAM has difficulties in dealing with highly data-dependent communication latencies as well as memory and network contention. The PRAM model is widely considered as not matching the essential features of existing parallel machines.

BSP Bulk synchronous parallel (BSP) models represent a the relatively new trend in modelling, which were discussed in [38]. As shown in Figure 2.3, BSP describes a computer as a collection of nodes, each consisting of a processor and memory. BSP models the interconnection network by means of a few parameters, instead

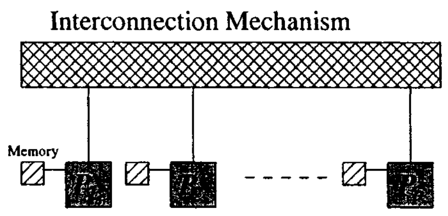


Figure 2.3: The BSP model

of the topology of a particular machine. It assumes the existence of a router and a barrier synchronization facility. One difference of BSP from PRAM is a larger step size.

BSP includes a cost model of three parameters:

- p, the number of processors
- l, the cost of a barrier synchronization, and
- g, a characterization of the available bandwidth.

Algorithm design for BSP is harder than for the PRAM, because of the additional parameters l and g. These parameters generally make algorithms less comparable. However, compared with network models, BSP is still much simpler. BSP accounts for the costs of communication and synchronization. It is suited to current architectures, in which communication costs are dominated by the time of exchanging messages with the network. However, it may prove inappropriate for future architectures with large numbers of processors.

SDM The *Standard Delay Model*(SDM) [45] is a representative model of distributed memory machines. The processors communicate with each other by transferring messages through an interconnected network. The most important concept introduced in SDM model is the *latency* of communications, the time spent for messages to go across the interconnections. Figure 2.4 shows communications under the SDM model. The latency depends on message size.

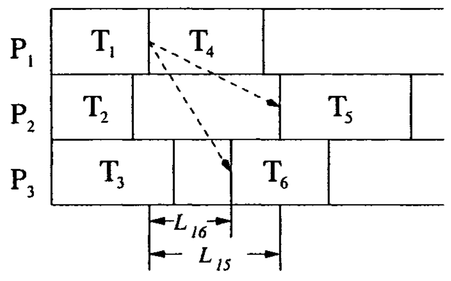


Figure 2.4: Communications under the SDM model

In this model, the *communication overheads* are ignored, since they have been shown not to work for many parallel architectures. In addition, the processor bandwidth does not have an upper bound. In reality, it is impossible for a processor to send/receive as many simultaneous messages as needed at a time, as implied in the SDM model.

LogP LogP improves on BSP with respect to authenticity. It also abstracts from the network topology and describes a machine in terms of a few parameters. The model does not deploy implicit synchronization. Instead, the processors work asynchronously and communicate via pairwise message exchanges.

LogP describes real architectures more detailedly and more accurately than BSP. It allows for the use of the overlap between computation and communication that is not represented by BSP. LogP does not enforce a specific program structure, and thus algorithms can be formulated more flexibly.

The LogP model will be described in detail since it is the model we are going to use in this thesis.

2.2.2 LogP model

LogP [15] is a model of a distributed-memory multiprocessor architecture in which processors communicate by point-to-point messages. The model only specifies the performance characteristics of the interconnection network, however, not the exact structure of the network.

The main parameters of the model are:

L : an upper bound on the latency, incurred in passing a message containing a word or many from a source processor to a target processor.

o : the overhead, defined as the length of time that a processor takes to transmit or receive a message; during this time, the processor cannot perform any other operations.

g : the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available communication bandwidth per processor .

P : the number of processors.

Thus, it is assumed that the capacity of the network is finite, such that at most $\lceil (L/g) \rceil$ messages can be transferred at a time. If a processor attempts to transmit messages beyond this limit, it stalls until the messages can be sent without exceeding the capacity limit.

The LogP model abstracts the communication network into three parameters. When the interconnection network is operating within its capacity, the time to transmit a message is $2o + L$. The available bandwidth per processor is determined by g, and the network capacity by $\lceil (L/g) \rceil$. The network is treated as a pipeline of depth L with initiation rate (i.e. one per clock cycle for individual instruction) g and a processor overhead of o on each end [15].

Figure 2.5 provides an illustrated example of the communication between processors in the LogP model. Consider two different processors P_1 and P_2. Assume

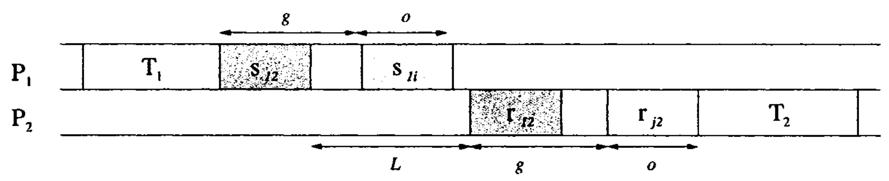


Figure 2.5: Communication between two processors in the LogP model

processor P_1 executes a task T_1 and the processor P_2, processes T_2, a successor of T_1. Then the result of the execution of T_1 must be transferred from processor P_1 to processor P_2 before T_2 can be executed. Assume the result of T_1 is sent twice for its successors, including T_2, on P_2. Figure 2.5 shows the communication between processors P_1 and P_2. The sending overheads are represented by s_{12} and s_{1i}; r_{12} is the receiving overheads corresponding to s_{12}. T_2 is also waiting for message from another task. r_{j2} is the receiving overhead.

As shown in Figure 2.5, messages are transmitted if a task and one of its successors are scheduled on different processors. Lengths of the overheads (s_{12}, s_{1i}, r_{12} and r_{j2}) are the same. It is also stated that a message can be received at least L time units after it has been submitted to the communication network, for example, from the end of s_{12} to the beginning of r_{12}. And, there is a delay of at least g time units between two send or receive overheads on the same processor. That is why s_{1i} does not start right after s_{12} finishes. Note that there need not be a delay between a send operation and a receive operation on the same processor.

The model is asynchronous, but is bounded above by L. Because of variations in latency, messages sent to a given target processor may not arrive in the same order as they were sent in. In some situations, it is possible to ignore one or more parameters and work with a simpler model: o can be increased to g ($o = g$), in which case g is no more relevant.

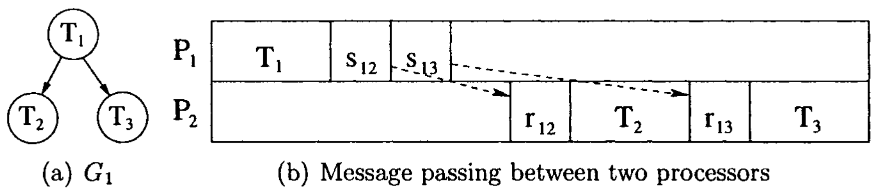
LogP is a machine-independent model for parallel computing. It tries to find out a tradeoff between the authenticity and simplicity when abstracting parallel computers. The model is sufficiently detailed to reflect the major practical issues in parallel algorithm design, yet simple enough to support detailed algorithmic analysis. At the same time, the model avoids specifying both programming style and communication protocols, being equally applicable to shared-memory, message passing and data parallel paradigms. Culler *et al.* [15] has applied optimal algorithms for broadcast and summation. It demonstrates how an algorithm may adapt its computational structure and communication schedule in response to each LogP parameter.

The classical LogP model only deals with short messages and does not adequately model machines with support for long messages. *LogGP* [2], the extension of LogP should be considered for long messages. However, in this thesis, the messages are assumed to be short enough to allow the use of the classical model. Actually, we may eliminate long messages by processing the related tasks on the same processor. In this case, there is no need to transmit the long messages. But, it is only practical when long messages are uncommon.

2.2.3 Communication Semantics

In both the LogP models and SDM models, data transfer between two tasks running on different processors are achieved by message passing. The requirements of message passing varies. Communication semantics define how requiring data are transferred from one processor to another. Finta and Liu [17] introduce two extreme cases of the communication semantics: independent data semantics and common data semantics.

Independent data semantics refers to independent data passing. Messages transferred to different successors are assumed to be different regardless the successors' positions [18]. In this case, the result of the execution of a task is sent separately to each of its successors regardless of the successor's position. Figure 2.6 provides an ex-



(a) G_1 (b) Message passing between two processors

Figure 2.6: Independent data semantics

ample. The precedence relationships among T_1, T_2 and T_3 are given by Figure 2.6(a). As shown in Figure 2.6(b), there are two separate messages containing T_1's results to be send from P_1 to P_2. Most of the work on scheduling parallel programs with communications is on the semantics of independent-data communication.

On the contrary, under *common data communication semantics*, results of a task can be sent to any processor at most once even the processor executes more than one successor of this task. Only in this case one can use broadcasting communication mechanisms. Figure 2.7 shows the communication among the same tasks in Figure 2.6

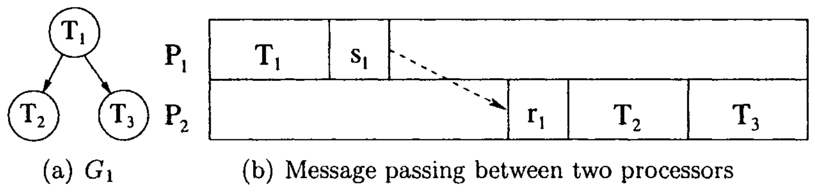


Figure 2.7: Common data semantics

in common data semantics. There is only one messages transmitting between processors. Note that there is no difference between independent data and common data if there is no task with more than one successor.

We consider independent data semantics in this thesis.

2.3 Scheduling Problems

2.3.1 Definitions

Tasks

Although scheduling problems arise in many application domains, a general model of scheduling may be applied to any application. The processes to be scheduled are made of complex activities. Such processes can always be modelled as tasks and relations among them [11]. For instance, a parallel program in multiprocessor scheduling problems can be divided into a set of computational tasks which are executed under certain precedence constraints. The tasks can often not be executed in an arbitrary order as the result of a task may be needed by other tasks.

A *directed acyclic graph* (DAG) $G = (V, E, w, l)$ can be used to represent the tasks and the precedence constraints. The vertices are the set $V = \{T_1, \cdots, T_n\}$ of n tasks, and each directed edge in E represents the precedence relation between two tasks, i.e. the processing order of the tasks is limited by this relation. The weight w_i associated with a task T_i presents the execution time of that task. If the result of task T_i is required by the execution of task T_j, $(T_i, T_j) \in E$. The length of the message sent by T_i to T_j is denoted by $l(T_i, T_j)$, and represents the weight of that edge. Hence, messages of this size are sent by T_i to all the tasks on other processors requiring its result before the execution of these tasks starts. When $(T_i, T_j) \in E$, T_i is said to be an *direct predecessor* of T_j, while T_j is the *direct successor* of T_i. We denote $Pred_G(T_i)$ as the direct predecessors of T_i in G, and $Succ_G(T_i)$ as the direct successors. Tasks without predecessors are called *source* nodes and are to be executed first. Tasks without successors are called *sink* tasks and executed last.

Scheduling

In general, scheduling is the process of assigning tasks to a set of resources. Scheduling problems are characterized by set $V = \{T_1, \cdots, T_n\}$ of n tasks and set $\mathbf{P} = \{P_1, \cdots, P_m\}$ of m processors. Therefore, scheduling means to the assignment of the tasks in V to processors from $\mathbf{P}$ under certain imposed constraints [11].

The constraints includes: *each task is processed by at most one processor at a time,* and *each processor can only process at most one task at a time.* These two constraints are applicable in classical scheduling theory, however the first may be relaxed in some new applications.

Schedules

A (feasible) schedule is an allocation of the resources to the tasks that satisfies all constraints defined by the scheduling problem. The objective of scheduling is finding a schedule that is optimal with respect to a certain objective function. The resources that have to be allocated and the constraints that have to be satisfied can be of various types. In mathematical terms, a scheduling problem is often solved as an optimization problem, with the objective of maximizing a measure of schedule quality [5].

A *schedule* S of task graph G maps a pair $(\sigma(T_i), \pi(T_i))$ to each task T_i in G, where $\sigma(T_i)$ presents the starting time of T_i's execution and $\pi(T)$ is the processor that T_i is assigned to.

Schedules may be represented by *Gantt charts*, a popular type of bar chart [11], as shown in Figure 2.8. A Gantt chart aims to show the timing of tasks as they occur

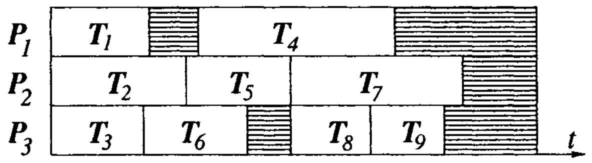


Figure 2.8: Gantt Charts

over time, while it does not (initially) indicate the relationships between tasks.

Generally, task $T_i \in V$ is bound by the following data except its execution time

w_i:

1. *Arrival time* art_i

2. *Due date* d_i

3. *Deadline* $\tilde{d}_i$

4. *Priority* q_i

Schedule S is said to be *feasible* iff. the following conditions are satisfied:

- on each processor, only one task can be processed at a time.,
- task T_i is processed in time interval $[art_i, \tilde{d}_i]$;
- all tasks in V have been scheduled;
- if tasks $T_i \prec T_j$, T_j must not start before T_j is finished;
- other constraints, if any, are satisfied.

Consequently, given a schedule, we can calculate the following parameters for each task $T_i, i = \{1, 2, \ldots, n\}$:

1. *completion time* $C_i = \sigma(T_i) + w(T_i)$;

2. *flow time* $F_i = C_i - art_i$;

3. *lateness* $D_i = C_i - d_i$.

2.3.2 Optimality Criteria

There are various *optimality criteria*, also called *performance measures*, for scheduling problems.

Makespan is defined as the duration between the start time of the first job and the finish time of the last executed job. Minimizing the makespan of the schedules is used most since it leads to both, the maximization of the processor utilization factor[1], and the minimum of the maximum completion time of the scheduled set of tasks. Makespan may also be important when a task set arrives periodically and is to be processed in the shortest time.

Another popular criterion *maximum lateness* involves due date. Maximum lateness

$$L_{max} = \max\{L_i\}, i \in \{1, 2, \dots, n\},$$

where n is the number of tasks. In real time environment, minimization of the maximum lateness leads to the construction of a schedule with all tasks finished on time.

From the user's viewpoint, the *mean flow time criterion* is also important, since minimizing mean flow time reduces mean completion time of the scheduled task set. Mean flow time

$$\bar{F} = \frac{1}{n}\sum_{i=1}^{n} F_i, i \in \{1, 2, \dots, n\}.$$

In this thesis, the objective of the problem is to minimize the schedule makespan.

[1]Processor utilization factor is the ratio of the processor's non-idle time in the given period, e.g. within the makespan C_{max}.

2.3.3 Analysis of Scheduling Problems

A scheduling problem could be either static or dynamic [3]. The difference between two types of scheduling problems is related to the predictability of structure of input jobs. Problems with a predictable structure are called *static* problems; on the contrary, in *dynamic* scheduling problems, the number of the tasks and the edges from the tasks are not known in advance. Only static scheduling problems are considered in this thesis.

2.4 Scheduling Strategies

2.4.1 List Scheduling

List scheduling is one of the most popular solutions to the multiprocessor scheduling problems. In list scheduling, each task is scheduled according to its priority. In addition, each task is scheduled as soon as all its predecessors have been scheduled.

Above all, there are two rules for designing a list scheduling algorithm (also called list scheduling algorithm). The first relates to the ranking of tasks. The second relates processor clustering, which assigns tasks to processors. Different settings of the two rules result in different algorithms. Also, it is necessary to define *ready* tasks, i.e. the tasks that can be scheduled.

A list heuristic builds a schedule step by step with the two rules. At each step, ready tasks are collected into a list which is sorted by rank. Among these ready tasks, the one T_i with the highest rank is chosen according to the first rule. Next, select a processor P_j, on which T_i is assigned according to the second rule. T_i starts on P_j immediately. This algorithm finishes when all tasks have been scheduled.

The particular assignment of priorities to tasks results in different schedules because tasks are selected for execution in different orders. HLF (Highest Level First), LNSF (Largest Number of Successors First) and LPTF (Largest Processing Time First) are three commonly used list scheduling heuristics. Here, HLF avoids delays in the *critical path* so as to minimize the execution time; LNSF maximizes the number of *ready* nodes by scheduling a task with the largest number of successors. And, the longest tasks are processed first in LPTF so that delays in long tasks are minimized. According to some empirical performance studies on List Scheduling, level-based heuristics are the best at approximating the optimal schedule.

2.4.2 Genetic Algorithms

Genetic Algorithms (GA) have been used as adaptive algorithms for solving practical problems and as computational models of natural evolutionary systems. Most GAs have at least the following elements in common [42]: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring.

The chromosomes in a GA population typically take the form of bit strings. Each chromosome can be thought of as a point in the search space of candidate solutions. The populations are replaced successively with another in GA. Usually, a fitness function is required to assign a fitness to each chromosome in the current population.

The fitness of a chromosome depends on how well that chromosome solves the problem at hand.

The simplest form of GA involves three types of operators:

Selection This operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce.

Crossover (single point) This operator randomly chooses a gene and exchanges the subsequences before and after that gene between two chromosomes to create two offspring.

Mutation This operator randomly flips some of the bits in a chromosome.

Given a clearly defined problem to solved and a bit string representation for candidate solutions, a simple GA works as follows [21]:

1. Start with a randomly generated population of n l-bit chromosomes (candidate solutions to a problem).

2. Calculate the fitness $f(x)$ of each chromosome x in the population.

3. Repeat the following steps until n offspring have been created:

   a. Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness.

   b. With probability p_c (crossover rate), cross over the pair at a randomly chosen point (chosen uniformly) to form two offspring.

c. Mutate the two offspring at each gene with probability p_m (mutation rate), and place the resulting chromosomes in the new population.

If n is odd, one new population member can be discarded at random.

4. Replace the current population with new population.

5. Go to step 2.

Such a procedure may also be accomplished as shown in Figure 2.9. Each iteration

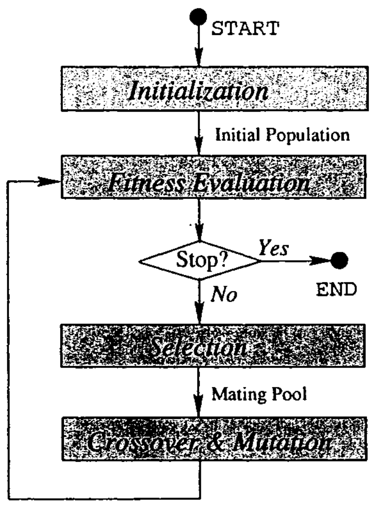


Figure 2.9: Process of Simple Genetic Algorithm

of such a process is called a *generation* and the entire set of generations is called a *run*. At the end of a run, there are often one or more highly fit chromosomes in the population.

The simple procedure just described is the basis for most applications of GAs. There are a number of details to fill in, such as the size of the population and crossover/mutation rates, and the success of the algorithm often depends greatly on these details.

Chapter 3

Problem Statement, Literature Review and Contributions

This thesis deals with static scheduling of general task graphs, representing parallel programs, on homogeneous multiprocessor systems, under the LogP model. Independent data semantics are assumed. The objective is to minimize the makespan of the parallel program. A formal description follows. We also present an overview of previous work in related areas and a summary of the contributions made by this thesis.

3.1 Problem Statement

Given a parallel program described by a DAG $G(V, E, w, l)$ and a LogP instance $M(L, o, g, P)$, we are looking for a schedule S of G on the multiprocessor system $\mathbf{P} = \{1, \ldots, P-1, P\}$ such that S is feasible under M **and** S has the minimum makespan among all feasible schedules of G. That is

$$S = argmin_{S' is feasible schedule of G}(makespan(S')).$$

A LogP-feasible schedule must have included both starting time and processor assignment for each in the extended task graph. A processor can not execute more than one task at a time. A task can only starts after all its predecessors have been finished. It takes at least a delay of L to transfer a message from one processor to another. In other words, the interval from completion of a send task to the beginning of its corresponding receive task must be equal or be greater than L. As for gaps, a delay of at least g is necessary between two consecutive send or receive operations. Note, there are not such delays required between a send task and a receive task on the same processor.

Denote the starting time and processor that have not been scheduled by $\perp$. v, v_i and v_j present any types of tasks in $G(V, E, w, l)$. s_{ij}, s_i and s_j are *send* tasks, and r_{ij}, r_i and r_j are *receive* tasks.Then, a *LogP-feasible* schedule can be defined by the following constraints:

1. $\sigma(v) \neq \perp$ and $\pi(v) \neq \perp$, $\forall v \in V$;

2. if $\pi(v_i) = \pi(v_j) \neq \perp$ and $v_i, v_j \in V$, then $\sigma(v_i) + w(v_i) \leqslant \sigma(v_j)$ or $\sigma(v_j) + w(v_j) \leqslant \sigma(v_i)$;

3. if $v_i \prec_G v_j$ and $v_i, v_j \in V$, then $\sigma(v_i) + w(v_i) \leqslant \sigma(v_j)$;

4. if coupled communications $s_{ij}, r_{ij} \in V$ and $(s_{ij}, r_{ij}) \in E$, then $\sigma(s_{ij}) + w(s_{ij}) +$

$L \leqslant \sigma(r_{ij})$;

5. if send tasks $s_i, s_j \in V$ and $\pi(s_i) = \pi(s_j) \neq \perp$, then $\sigma(s_i) + w(s_i) + g \leqslant \sigma(s_j)$ or $\sigma(s_j) + w(s_j) + g \leqslant \sigma(s_i)$;

6. if receive tasks $r_i, r_j \in V$ and $\pi(r_i) = \pi(r_j) \neq \perp$, then $\sigma(r_i) + w(r_i) + g \leqslant \sigma(r_j)$ or $\sigma(r_j) + w(r_j) + g \leqslant \sigma(r_i)$;

3.2 Problem Instance

Consider the task graph G_2 shown in Figure 3.10. Figure 3.11 are the feasible

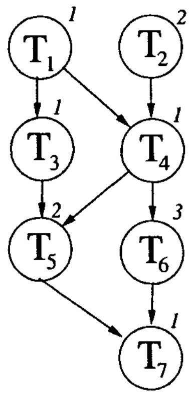


Figure 3.10: Task Graph G_2

schedule example of the task graph G_2 (Figure 3.10) under LogP instance $M(1,1,1,2)$.

1	2	3	4	5	6	7	8	9	10	11	12	13
T_1	s_{14}	T_3					r_{46}	T_6			r_{57}	T_7
T_2			r_{14}	T_4	s_{46}	T_5		s_{57}				

Figure 3.11: A Schedule $S(G_1)$ of G_1 feasible under $M(1,1,1,2)$

3.3 Literature Review

3.3.1 Approaches of Scheduling under SDM

The most common model used by the scheduling community for the scheduling problem is the SDM, where the *latency* for a message transmission is the only communication parameter.

Much research has been done in recent years on static scheduling task graphs under SDM-like models. Rayward-Smith [45] shows that it is NP-complete to find the minimum makespan under the model which was confined to unit communication times (UCT) and unit execution time (UET). He also presents a heuristic, called "generalized list scheduling", which adopts the same greedy strategy as Graham's list scheduling [23]: No processor remains idle if there is some task available that it could process. The ETF (Earliest Time First) algorithm [25] computes, at each step, the earliest start-times for all ready nodes and then selects the one with the earliest start-time. Here, the earliest start time of a node is computed by examining the start-time of the node on all processors exhaustively. When two nodes have the same value in their earliest start-times, the ETF algorithm breaks the tie by scheduling the one with the higher static priority. There are other ways to determine the priorities of nodes, such HLF (Highest Level First) [12]; LP (Longest Processing Time) [12]; LPT (Longest Processing Time) [19]; and CP (Critical Path) [51].

List scheduling algorithms have a low time complexity. For instance, the time complexity of ETF is $O(NP^2)$, where N is the number of the nodes in the given graph and P is the number of processors. However, all the list scheduling algorithms are sub-optimal on general task graphs. To improve these solutions, genetic algorithms have successfully been applied to the problem and the results reported have been superior to those of the list scheduling approaches.

The application of genetic algorithms to the multiprocessor scheduling problem can be classified into two categories: direct mapping and indirect mapping.

In the direct mapping, the phenotype is identical to the phenotype and the genetic operators manipulate the schedules directly. This scheme was first introduced in the HAR algorithm by Hou *et al.* [24]. In the HAR algorithm, each chromosome is a collection of lists, each of which represents the schedule on a distinct processor. Thus, each chromosome is a two-dimensional structure: one dimension is a particular processor index and the other is the ordering of tasks scheduled on the processor. The encoding scheme in HAR poses a restriction on the schedule being represented: the list of tasks within each processor in a schedule is placed in ascending order of their *height*. The genetic operators are based on the precedence relations between the tasks in the given graph. Their simulation study shows the HAR algorithm produces schedules within 20% from optimal solutions. This representational scheme of direct mapping is later modified by Corrêa *et al.* [14]. Their version called Lyon introduced an initialization method that generate the initial population via a list scheduling algorithm. The algorithm uses knowledge augmented operators.

In the indirect mapping scheme, the genotype is a set of symbols that are used by some decoding algorithm to build a schedule. Ahmad *et al.* [1] uses such a representational scheme. They employ a list of task priorities as a chromosome. A list of priorities is obtained from the input DAG first. The initial population is generated by randomly perturbing the original list. A list scheduling algorithm is used to determine the fittest chromosome. Therefore, the genetic search operates on the problem-space instead of the solution-space as is commonly done. They report to have better results than Hou *et al.*. Sandes and Megson's [47] improved approach is based on the implicit representation proposed by Ahmad *et al.*. It was extended to partially connected processor networks, through an extended chromosome representation. Auyeung *et al.* [4] introduced a different approach using indirect mapping. Their solution is to use the GA to find a combination of four heuristics. The chromosomes encode the weights of different heuristics in order to obtain an optimal combination of the priorities from the heuristics. Their results show that such an approach outperforms each one of the four list scheduling algorithms alone.

According to the comparisons carried out by Rebreyend [46], convergence times are similar for the two categories. Both approaches tends towards the same lower bounds. In general, the indirect mapping is simple and straightforward in structure. The direct mapping scheme requires more implementation effort, however it is more flexible. Note, the height restriction in the encoding of HAR and Lyon may prevent the search from obtaining the optimal solution; this is discussed in detail in Section 5.1.2.

3.3.2 Approaches of Scheduling under LogP

Besides latency, other communication characteristics have been shown to influence performance significantly, with the evolution of parallel architectures [39]. Some of these characteristics cannot be modelled as part of the message delay [15]. Therefore, using SDM may result in poor performance. On the other hand, the LogP model [15] (see Section 2.2.2) and its variants [2] consider the dominant communication characteristics of each individual communication, as well as the limited communication capacity. LogP can accurately model a variety of parallel machine [15, 36, 39], as well as clusters [15, 49].

It is NP-complete to find an optimal schedule under the LogP model, even for some special classes of task graphs: M. Middendorf *et al.* [41] assume all tasks have computation time c and assuming constant c, o and L. And they suggest that, for inverse trees[1], it is NP-complete to find a linear schedule (i.e. at least one of the predecessors of a task is computed on the same processor) with makespan greater than T_{max} even if $g \leqslant o$, Verriet [50] shows that the computation of a schedule of length at most D is NP-complete even for fork trees[2] when $g = o$. Zimmermann *et al.* [52] found that even for simple graphs, such as send-graphs and receive-graphs, it is NP-complete to decide whether there is a schedule under the LogP model with a makespan of at most D for any fixed number of processors.

Compared to research in multiprocessor scheduling under the SDM-like models, only a few scheduling strategies for the LogP models have been published [7, 8, 30, 31, 27, 34, 35, 36, 41, 50, 52]. These approaches include: (a) approximation algorithms; (b) replication-based algorithms; and (c) list scheduling algorithms.

Löwe *et al.* [37] generalize the result of Gerasoulis *et al.* [20] to the LogP models using linear schedules. Two polynomial-time algorithms were presented in [50] that

[1]A communication structure is an *inverse tree* if the communication structure with the same vertices and edges in inverse direction in a tree. [34]

[2]A fork tree consists of a node v_o and its k successors $v_1, \ldots, v_k$.

construct schedules for fork graphs; one is a 2-approximation algorithm for scheduling arbitrary fork graphs with unrestricted number of processors, the other constructs minimum-length schedules for fork graphs in which all sinks have the same execution length for the case that the number of processors is bounded. Further, Zimmermann *et al.* [53] consider k-linear scheduling[3] under the LogP model. They presented an algorithm to obtain optimal k-linear schedules for trees and tree-like task graphs with an unbounded number of processors, assuming $g = o$. This approach cannot be generalized easily to cases where $g > o$. Middendorf *et al.* [41] discussed some of those for linear schedules (i.e. $k = 1$). However, it is not yet clear whether these apply to k-linear schedules in general, and whether other normalization properties are required. There is no general approximation scheme with a constant performance ratio. For instance, Löwe [34] provides a performance ratio with parameters o and g. In short, approximation algorithms are not the solution to the scheduling problem of general task graphs under the LogP model.

Boeres *et al.* [6, 7, 8, 9] outlined a methodology to design replication-based clustering (i.e. obtaining processor assignments) algorithms for scheduling arbitrary task graphs with arbitrary costs, under the LogP model, onto a bounded/unbounded number of processors. Theoretical analysis presented in the paper shows that the algorithm generates linear schedules under small communication condition. However, due to lack of effective evaluations, conclusions based on experimental results may be limited to the graphs tested.

As for heuristics, two list scheduling algorithms presented by Kalinowski *et al.* [27]

[3]A k-linear schedule may map up to k directed paths of a task graph onto one processor

adapted the scheduling scheme of Hwang's [25] ETF algorithms to the LogP models. These approaches handle general task graphs, in cases where $g = o$. The proposed algorithms compute effective schedules if the communication time is not greater than the average computation time. In these algorithms, all send tasks are executed sequentially right after the computational tasks complete. Similarly, the receive tasks directly precede computation. Such a scheme reduces the number of possible task assignments and simplifies the algorithms, but possibly eliminates some good solutions. Details of the list scheduling algorithms under the LogP model are to be discussed in Section 6.1.

3.3.3 Summary

To the best of our knowledge, GAs have not been applied to scheduling problems under the LogP model, and all work of LogP scheduling only considered some special cases (except [6] which uses replication-based algorithms), such as specific classes of task graphs (except [27]), restricted LogP models (except [36]). This thesis presents a LogP scheduling methodology for general conditions from a practical viewpoint, and employs GAs extensively as part of its proposed methodologies.

Our approaches will follow the scheduling schemes of ETF algorithms presented by Kalinowski *et al.* [27]. The scheduling procedure is divided into two phases: First, it allocates tasks to processors. Second, it adds necessary communications to the results in the first phase. GAs are applied to each of the two phases, appropriately.

3.4 Contributions of the Thesis

In addressing the general problem of multiprocessor scheduling under the LogP model, distinct from that has been done before, this thesis makes the following specific contributions:

1. Task Graph Extension. Extending a task graph under the LogP model allows for the generalizations of communication tasks. With this task graph extension, general scheduling algorithms can be applied to the scheduling problem under the LogP model more easily.

2. Gap Handling. The effects of the gaps in a scheduling procedure are analyzed thoroughly. The decoders of GAs are modified to deal with significant gaps. In addition, the concept of gap filling is introduced to allow the algorithms to search for more solutions.

3. A novel GA based algorithm for communication management. The algorithm extends the ETF heuristics [27] by arranging the communication operations differently. A GA, with specific genetic operators, is presented for this purpose. A list scheduling algorithm is applied as the decoder. A novel encoding methodology that minimizes the representation space is proposed for this algorithm.

4. A novel GA-based algorithm for processor allocation. This algorithm allocates tasks to processors using a GA. Similarly, the chromosomes are decoded with a list scheduling algorithm. In this algorithm, the communications are arranged using fixed rules.

Chapter 4

Scheduling Communications using Genetic Algorithms

Existing scheduling algorithms under the LogP model assign communications to respective processors, after scheduling the computational tasks, using fixed rules. The objective of the rules is to obtain a feasible schedule under LogP, instead of to minimize the makespan. Most of these rules assign high priorities to the computational tasks.

2ETF from [27] is such an algorithm. It gets a schedule of the task graph under SDM and then derives another schedule which is feasible under LogP. During the second phase of 2ETF, the sending overheads from the same source of computational tasks are scheduled sequentially right after the source task. On the other side, the receiving overheads that have the same destination are grouped together and inserted before the common destination task.

The disadvantage of using fixed rules is illustrated by the following example. Task graph G_2 in Figure 4.1 is scheduled in two ways. Both schedules have identical

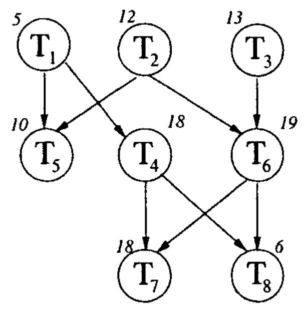
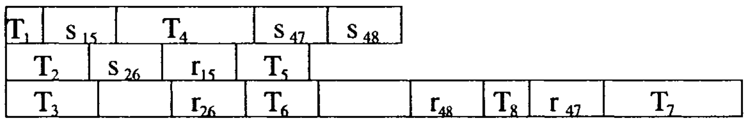
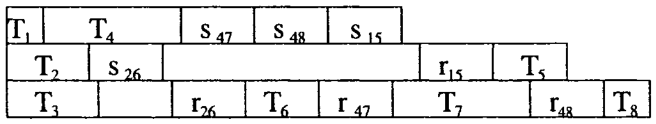


Figure 4.1: Task Graph G_2

allocations of the computational tasks. The difference between them is that Schedule S_1 in Figure 4.2(a) follows the rule applied by 2ETF while Schedule S_2 in Figure 4.2(b) does not. The critical path of the task graph is: $T_1 \longrightarrow T_4 \longrightarrow T_7$. If the sending



(a) Schedule S_1(2ETF)



(b) Schedule S_2

Figure 4.2: Different Communication Arrangements

overhead s_{15} is scheduled right after T_1, it will delay T_4 , the longest computational task in the task graph, and which is also on the critical path. Additionally, T_5 is the last task assigned to the second processor. Therefore, postponing s_{15}, and consequently postponing r_{15} and T_5, will not affect the overall processing time. On the contrary, it minimizes the makespan, like schedule S_2 does.

In brief, it is necessary to find a strategy to schedule the communication tasks in order to achieve better overall schedules. However, the exact set of communication tasks is not determined until the processor assignment is fixed. Thus, the objective is to schedule the communications with given processor assignments of computational tasks. The assignments come from one of the existing algorithms under SDM and the sequential communications are derived. The key step will be applying GA to find an optimal placement of both computations and communications.

A novel method to generalize the communications is introduced at first. With this method, we adapt the general list scheduling algorithm for LogP models. A genetic algorithm is proposed next, using the list scheduling algorithm as the decoder. The whole procedure of the GA is discussed in detail.

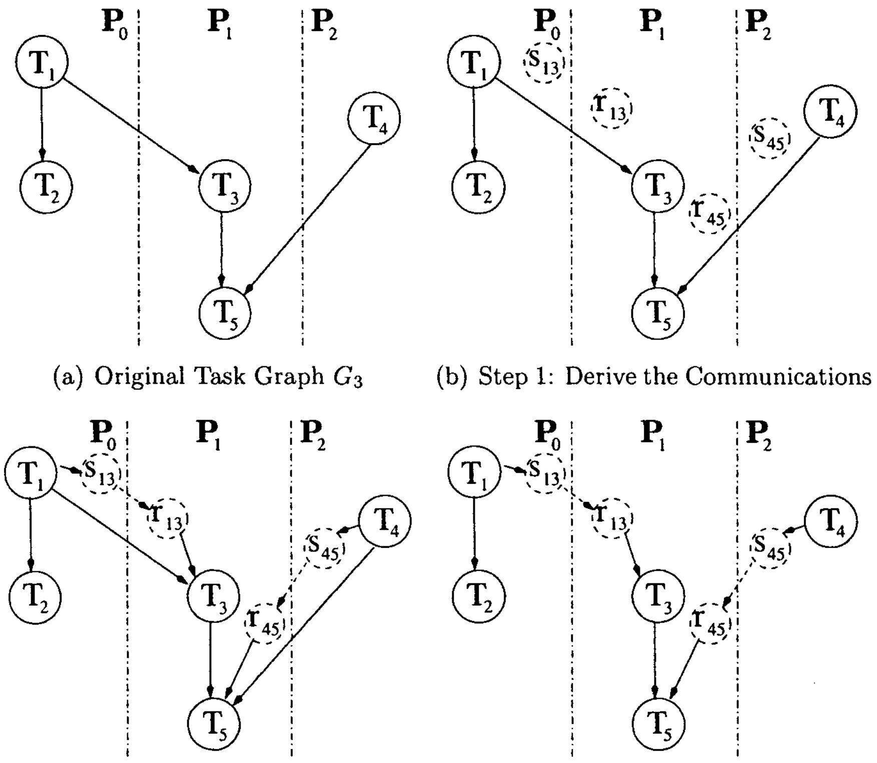
4.1 Extended Task Graph

According to the LogP model, no task can be scheduled during the overheads. In a sense, communications are similar to computational tasks, as both of them occupy processors. 2ETF manages communications after settling all the computational tasks. In this manner, communication is simply an extension of computation. But, now our focus are the communications. It is reasonable to allow communications more flexibility. Therefore, we treat the communication tasks the same as the computational tasks. Viewing communications as tasks also make possible the application of various optimization algorithms to this problem.

Such generalization can be achieved by extending the task graph. The main purpose of such extension is to insert all the communication tasks into the original graph given the processor assignment so that communication tasks can be considered together with the computational tasks. At the same time, the extension must maintain the partial order defined by the original graph.

Extending a task graph is a 3-step process. First, we generate the communication tasks. Whenever two directly connected nodes are assigned to different processors, there should be a pair of communications: a *send* task on the source processor and a *receive* task on the destination processor. Second, the edges of the graph are redefined taking into consideration the new nodes of the communication tasks. Clearly, a *send* task can not be scheduled until the source computational task is finished. Similarly, the *receive* task should finish before the destination computational task starts. Also, receiving a message can only happen after the message has been sent. Therefore, we should add 3 edges for each pair of communication tasks generated in last step. The three edges are: (a) an edge from the source task to the *send* task, (b) an edge from the *send* task to the *receive* task, and (c) an edge from the *receive* task to the destination task. At last, since the source task has been connected to the destination task through the communication tasks, the original edge between the two computational tasks is no longer necessary and is deleted. Removing this edge would not change the partial order of the original graph. As a result, this process expands each edge connecting two processors into three edges.

The process of extending a task graph G_3 of 5 nodes is illustrated in Figure 4.3. The nodes in the original graph have been distributed on 3 processors, and two edges connecting 2 different processors need to be extended. Consider the edge (T_1, T_3). After T_1 is finished on P_0, a message is passed from P_0 to P_1 so that P_1 can process T_3.



(a) Original Task Graph G_3 (b) Step 1: Derive the Communications

(c) Step 2: Connect the Communications (d) Step 3: Remove Redundant Edges

Figure 4.3: Extending a Task Graph G_3

Thus, we have a pair of communication tasks, s_{13} and r_{13}, to present the overheads. Figure 4.3(b) shows all the necessary communications to be added into this graph. After adding communication tasks, T_1, s_{13}, r_{13} and T_3 are reconnected to form a new path from T_1 to T_3, as in Figure 4.3(c). The final graph after extension is presented in Figure 4.3(d). Although all the edges are extended at the same time in the example, we usually extend only one edge at a time. An outline of the extension algorithm is given in Algorithm 1.

Algorithm 1 Task Graph Extension under LogP

```
Input:
    $G$, $w$, $l$: a precedence graph,
    $M = (L, o, g, P)$: a LogP instance,
    $\Pi(G) = \{\pi(T_i) | \forall T_i \in V\}$: an processor assignment of $G$.
Output:
    $G'$, $w'$, $l'$: an extended graph,
    $\Pi(G') = \{\pi(T_i) | \forall T_i \in V'\}$: an processor assignment of $G'$.

    Begin
    {Initially: $G' := G$, $\Pi(G') := \Pi(G)$ and $orgE := E'$.}
    for all $(T_i, T_j) \in orgE$ and $\pi(T_i) \neq \pi(T_j)$ do
        $s_{ij} := send(T_i, T_j)$;
        $r_{ij} := receive(T_i, T_j)$;
        $V' := V' \cup \{s_{ij}, r_{ij}\}$;
        $E' := E' \cup \{(T_i, s_{ij}), (s_{ij}, r_{ij}), (r_{ij}, T_j)\}$;
        $E' := E' / (T_i, T_j)$;
        $\pi(s_{ij}) := \pi(T_i)$;
        $\pi(r_{ij}) := \pi(T_j)$;
        $\Pi(G') := \Pi(G') \cup \pi(s_{ij}) \cup \pi(r_{ij})$;
    end for
    End
```

Note that the edges between coupled communication tasks are different from the others due to the latency of message passing. For instance, the ready time of a task is defined as the earliest moment that the task can be started regardless of the status of processors. It depends on accomplishment of all the tasks' predecessors. However, a *receive* task is not ready immediately after the completion of its predecessor, the corresponding *send* task. There must be an interval longer than L between the coupled communication tasks. Formally, the ready time of a task can be calculated as follows:

$$rt(T_i) = \begin{cases} \max_{T_j \in Pred(T_i)} (\sigma(T_j) + w(T_j)) & \text{if } T_i \text{ is not a } \textit{receive} \text{ task and } Pred_G(T_i) \neq \emptyset \\ \sigma(send(T_i)) + w(send(T_i)) + L & \text{if } T_i \text{ is a } \textit{receive} \text{ task} \\ 0 & \text{if } T_i \in source(G) \end{cases} \tag{4.1}$$

, where $\sigma(T_j)$ is the starting time of task T_j; $w(T_j)$ denotes the weight, i.e. execution time, of T_j and L presents the latency.

4.2 List Scheduling under LogP

Having all the information (including the processor assignments) of both computational tasks and communication tasks from the extended task graph, we then need to figure out how to arrange the tasks on each processor so as to obtain a schedule under LogP. Suppose we have the ranks of all the tasks. Then, list scheduling is one of the simplest methods to generate a LogP schedule.

As discussed in Section 2.4.1, list scheduling algorithms schedule tasks as early as possible. They apply two rules: one for processor assignments and another for task ranking. In our problem, the processor assignments come with the extended task graph. The ranks are assumed to be known. Therefore, a common list scheduling algorithm can be applied easily. In the rest of this section, we are going to describe our approach.

This algorithm accepts a extended graph G', a processor assignment $\Pi(G')$, a LogP instance M and a priority queue $Q(G')$ as inputs. Here, $Q(G')$ consists of the priority values for all tasks. Also, we assume $M = (L, o, g, P)$ satisfies $o = g$, allowing us to ignore gaps. The output of the algorithm should be a feasible schedule S of G' under the model M.

There are a few time variables involved in our approach. The completion time denoted by $ct(i)$ is associated with processor P_i, which presents the end of last task's execution on P_i. Current time CM is the moment when the algorithm makes decisions. We increase CM gradually starting from time 0. The increments of CM are determined by the values of future decision moments on all processors. $NM(i)$ refers to the *next decision moment* on processor i. It is the next earliest point of time when processor P_i is possibly going to start processing a task. In other words, at time $NM(i)$, there should be at least one task ready on processor P_i, while the processor is free for use at the same time. Hence, both the completion time of the processor and the ready time of each task on the processor influence the next decision moment. All these variables are initialized to 0 before the algorithm starts.

Furthermore, the algorithm works with two sets of nodes. One is the set U, containing the tasks that have not been scheduled yet. At the beginning, U is made up of all tasks in the extended task graph G'. Whenever a task is scheduled, it is removed from the set U. The other set of nodes is SN, the nodes that can be scheduled. This is not simply the set of ready nodes. The processors to which the nodes in SN are assigned are available at the moment.

At each iteration, we update SN. It is accomplished by scanning all the nodes in U. Then, we select the task T_i with the highest priority, in SN and schedule it to processor $\pi(T_i)$ at current moment t. T_i is then removed from U and SN. Since the processor $\pi(T_i)$ is no longer available at the current moment any more, any other tasks assigned to the same processor are deleted from SN. We also update the completion time of processor $\pi(T_i)$, as T_i becomes the last task on this processor at this moment. Besides, the ready time of all of T_i's direct successors are recomputed. Any tasks that can be scheduled at time t are scheduled before increasing CM at the next decision moment.

The details of this approach are described in Algorithm 2. Note that the ready time $rt(T_i)$ is calculated with Equation (4.1). It is necessary to keep in mind that the ready time of a receive task is computed differently.

Given the scheduling problem and a processor assignment, this list scheduling algorithm can transfer any integer strings into a LogP-feasible schedule. Therefore, it is used as the decoder of our GA.

4.3 Representation

This section introduces the representation of a feasible schedule. In the following, we are going to discuss the encoding strategy as well as the decoding methodology.

Algorithm 2 List Scheduling under LogP, with $g = o$

```
Input:
    $G'(V', E', w', l')$: an extended graph of $G(V, E, w, l)$,
    $M = (L, o, g, P)$: a LogP instance, such that $o = g$,
    $Q(G') = \{q(T_i)|\forall T_i \in V'\}$: a priority queue for tasks in $G$,
    $\Pi(G') = \{\pi(T_i)|\forall T_i \in V'\}$: an processor assignment of $G$.
Output:
    $S$: a feasible schedule of $G'(G)$ under $M$.

    Begin
    {Associate $ct(i)$ to the completion time of processor $P_i$, and $NMi$ to the next
    decision moment on the processor. .Denote the set of nodes that can be scheduled
    with $SN$. Initially:$SN = \emptyset$ $ct(i) = 0, \forall i, 0 \leqslant i < P$, the current moment
    $CM := 0$,the set of unscheduled nodes $U := V'$.}
    while $U \neq \emptyset$ do
        for all $T \in U$ do
            if $0 \leqslant rt(T) \leqslant CM$ and $ct(\pi(T)) \leqslant CM$
                and $T' \in \overline{U}, \forall T' \in Pred_{G'}(T)$ do
                $SN := SN \cup \{T\}$;
            end if
        end for
        while $SN \neq \emptyset$ do
            {Select a task $T_i$, such that $Q(T_i) = \min_{T \in U} Q(T)$;}
            $\sigma(T_i) := CM$;
            $U := U/T_i$;
            $ct(\pi(T_i)) := CM + w(T_i)$;
            $rt(T_i) := -1$;
            for all $T_j \in Succ_{G'}(T_i)$do
                {Update $rt(T_j)$;}
            end for
            for all $T \in SN$ do
                if $\pi(T) = \pi(T_i)$ do
                    $SN := SN/T$;
                end if
            end for
        end while
        for $i = 0$ to $P - 1$ do
            $NM(i) := \max(CM, ct(i), \min_{T \in \overline{U}}(rt(T)|\forall T, \pi(T) = i))$;
        end for
        $CM := \min_{i \in [0, P-1]} NM(i)$;
    end while
    End
```

4.3.1 Basics of Encoding

GA encoding is used to represent relatively complicated solutions in a simple way. The number of possible encodings may be very large, but, that does not mean that we can choose any encoding method. As a matter of fact, the encoding strategy affects every step of GA operations.

The following two principles introduced in [21] are often used for encoding design:

- Principle of *meaningful building blocks*;

- Principle of *minimal alphabets.*

The first principle states that substrings from the encoding may combine under crossover to produce better solutions. And, the second requires the smallest alphabet so as to maximize the number of exploitable schemata.

However, these two principles are sometimes difficult to apply due to the complexity of problems. In most cases, a capable encoding method meets a few requirements in the following manner. First of all, an encoding method must be accompanied by a decoder in order to generate the solution from the coding. A simple decoder can decrease the algorithm's complexity. And, the encoding itself should be simple. A chromosome must consists of building blocks that are able to deliver partial solutions. Without such meaningful building blocks, genetic operators are not able to produce solutions in a predictable way. At last, one encoded individual must present one, and only one solution. This prevents GAs from wasting time on meaningless individuals which degrades GA performance.

In next section, we are going to discuss an encoding strategy for our own problem, one that adheres to the limitations discussed above.

4.3.2 Encoding

First, the encoding method should allow the representation of a schedule to hold enough information while being simple enough for GA operators to act. As the processor assignments of both computational and communication tasks are known, the only thing we need, in order to obtain a feasible LogP schedule, is the starting time of each task. It is straightforward to give each task a priority so that we can schedule these tasks easily using list scheduling heuristics. Computational tasks and communication tasks are treated equally during the scheduling process. When more than one task is ready for the same processor at the same moment, the task with the highest priority is scheduled at first, regardless of the type of the task.

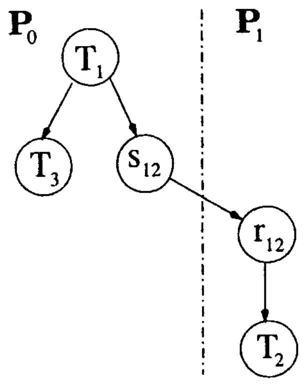


Figure 4.4: Extended Task Graph G_1'

Figure 4.4 shows an extended task graph G_1' with processor assignments. We can simply use a string of positive integers as the chromosome for our GA. Then an individual C is composed of n pairs $\{Q_1, Q_2, \ldots, Q_n\}$, where n is the number of vertices in the extended task graph, and a gene is a pair $Q_i = \{v_i, q_i\}$ ($v_i \in V$ and $q_i \geqslant 0$). Suppose we have two individuals

$$C_1 = \{\{T_1, 3\}, \{s_{12}, 4\}, \{r_{12}, 1\}, \underline{\{T_3, 5\}}, \{T_2, 2\}\},$$

and

$$C_2 = \{\{T_1, 3\}, \{s_{12}, 4\}, \{r_{12}, 1\}, \underline{\{T_3, 100\}}, \{T_2, 2\}\}.$$

It is easy to observe that T_1 should be scheduled on P_0 first. When task T_1 is finished on P_0 at time t_1 (t_1 is the execution time of T_1), T_3 and s_{12} will be ready simultaneously. Although C_1 and C_2 have different priority values for T_3, it does not affect the scheduling decision at time t_1, as both 5 and 100 are greater than the s_{12}'s priority, which is 4. In fact, while $q_{s_{12}}$ is set to 4, q_{T_3} could be any number greater than 4, so as to produce the same schedule. This is quite wasteful. Hence, we limit the value of priority to a small interval of integers, between $[0, n-1]$.

Let us look at another chromosome

$$C_3 = \{\{T_1, 3\}, \underline{\{s_{12}, 4\}}, \{r_{12}, 1\}, \underline{\{T_3, 4\}}, \{T_2, 2\}\}.$$

The priority values all satisfy $q_i \in [0, n-1]$. However, $q_{s_{12}} = q_{T_3} = 4$. At time t_1, we cannot decide which task to schedule based on priorities only. There are 2 alternatives: either s_{12} is scheduled at first, or T_3 is executed before s_{12}. If we choose one solution randomly whenever we decode the chromosome, it will be impossible to evaluate the chromosome consistently over time, since it represents 2 different solutions. On the other hand, we might lose one of the two schedules forever if we consistently select one interpretation over the other. Thus, we should not allow equal priority values for different genes in any one chromosome.

nodes:	T_1	s_{12}	r_{12}	T_3	T_2
priorities:	2	3	1	4	0
			$\Updownarrow$		
order:	0	1	2	3	4
permutation:	T_2	r_{12}	T_1	s_{12}	T_3

Figure 4.5: Example of an Encoded Chromosome

In conclusion, we set two restrictions on the priority values in the encoding. First, the priority of any node should be a non-negative integer not greater than $n - 1$, where n is the number of nodes in the graph. Second, each node should have a unique priority value, which is different from any other node's priority value. The two lines in Figure 4.5 show how such a chromosome would be like, when scheduling the extended task graph G'_1. Furthermore, priorities with these two restrictions can introduce a total order of all the nodes in the graph. A permutation of all the tasks, shown in the fourth line of Figure 4.5, is already capable of presenting a solution, and is much simpler too. And, the permutation can be easily transformed to the priority form, by assigning each node its rank in the permutation. Thereby, a permutation of n nodes can yield a feasible schedule using list scheduling, and conversely every feasible schedule can be represented by such a string.

4.4 Genetic Operations

Figure 4.6 shows the procedure of the algorithm.

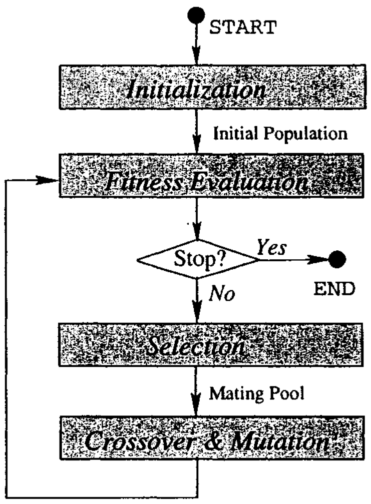


Figure 4.6: Algorithm Outline

4.4.1 Initial Population

Before evolution starts, we need an initial population. In the Simple GA (SGA) [21], individuals in the initial population are generated randomly. However, for an extended graph of n nodes, there are $n!$ individuals in our representation space. When n gets huge, it is even harder to start searching at a random point in the space. Therefore, it is helpful if we start our evolution at a population with better points than the randomly-generated ones.

We still generate most initial individuals randomly as in the SGA. However, one of the initial individuals is encoded from a known solution. This solution comes from an existing heuristic, such as 2ETF. Such a initialization procedure together with our elitism strategy guarantee that our GA generates individuals that are better or (at worst) of equal fitness than those generated by the 2ETF heuristic.

Most of our parameter values are determined empirically. The parameters in our GA are listed as below:

Crossover Rate & Mutation Rate Our crossover rate and mutation rate are relatively high, compared to those of the SGA. The crossover rate is 0.85 and the mutation rate is 0.15. The reason we are using relatively high rates is our elitism strategy. Since the best individual is always able to survive, higher rates are not going to destroy it. On the contrary, higher rates help us search the space more effectively, at least that is what we found to be true case.

Population Size The population contains 50 individuals.

4.4.2 Fitness Evaluation

Having an encoded chromosome described in Section 4.3.2, it is easy to extract a priority queue from the chromosome. The priority queue consists of the ranks of all nodes in the extended task graph. Such a queue can always be mapped onto a LogP-feasible schedule via the list scheduling algorithm (Algorithm 2). The goal is to minimize the makespan. Thus, we have to compute the makespan of each schedule introduced by the chromosomes.

Again, the makespan is the length of time a schedule takes. More specifically, it is the time span between the beginning of the first task and the completion of the last task. We need to find out the minimum completion time among all the tasks. Since tasks must be done before their successors, and there is only one latest task on each processor, we only need to calculate the completion time of the last computational task without successors on each processor. The maximum of these completion time is the makespan we are seeking.

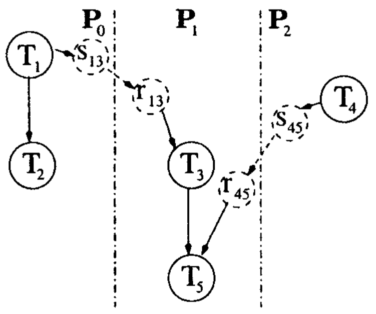


Figure 4.7: Extended Task Graph G_2'

An example is shown in Figure 4.8. Consider chromosome C as the permutation of all tasks in graph G_2' in Figure 4.7. Assume the computational tasks are all unit tasks, which take a unit of time each to execute. For the LogP model M, let $L = o = g = 1$, and $P = 3$. We arrange the tasks as schedule S, which is produced by our decoder. There are two computational tasks that have no successors: T_2 and T_5, which are on different processors. T_2 finishes at time $t = 3$, earlier than T_5. Therefore, the makespan of S should be the completion time of T_5, which is 7.

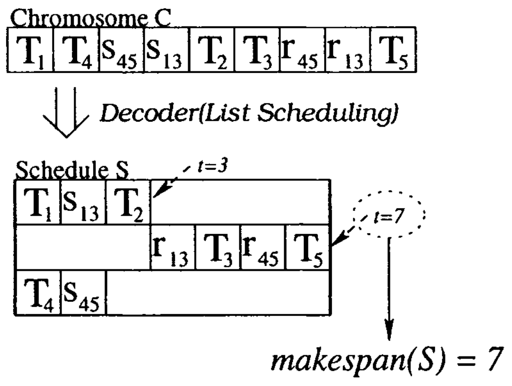


Figure 4.8: Evaluation of a chromosome C

There are two steps to evaluate an individual. The individual should be decoded into a feasible schedule at first. List scheduling is applied as the decoder of our GA. Then, the makespan is computed. The smaller the makespan, the better the individual is.

We evaluate an individual using the makespan of the schedule it represents. It is more convenient for us to put the makespan into a form that can be maximized. The differences between the makespans of different schedules are relatively small compared to the makespans themselves, especially when the graph is complicated. So, scaling is necessary in order to increase the differences between different makespans. This allows a GA to operate correctly.

Let $M(x)$ be the makespan of individual x, the fitness function is defined as:

$$f(x) = \frac{1}{M(x) - \min_{y \in W} M(y) + K} \tag{4.2}$$

where K is a scaling constant and $\min_{y \in W} M(y)$ represents the minimum makespan over all the generations to date.

As the key to fitness scaling, the variable K is problem-dependent. It influences the performance of the GA by affecting selection pressure, the primary component that determines the convergence rate of the GA. If K is too small, individuals with the highest fitness values, will dominate the population too rapidly, which would prevent the algorithm from searching other areas of the search space, hence, possibly leading to premature convergence to a sub-optimal schedule. If on the other hand K's value is too large, the search would proceed too slowly for it to reach the exact optimal solution. Therefore, the value of K must be chosen carefully.

K varies in each generation. Let E_{Bst} be the expected number of offspring of the best individual with a makespan M_{Bst} in current generation and E_{Wst} be the expected number of offspring of the worst makespan with M_{Wst}, respectively. We set

$$E_{Bst} = 2E_{Wst},$$

which implies that the best individual has only two times more chances to survive. According to the proportional selection method we are going to discuss later, the expected number of an individual's offspring should be proportional to its fitness. Hence, from Equation (4.2), it follows

$$\frac{1}{M_{Bst} - \min_{y \in W} M(y) + K} = 2 \times \frac{1}{M_{Wst} - \min_{y \in W} M(y) + K}.$$

On the other hand, according to our elitism strategy (to be discussed later),

$$\min_{y \in W} M(y) = M_{Bst}.$$

So, we can establish the equality

$$K = M_{Wst}. \tag{4.3}$$

This means K is set to the maximum makespan in each generation. As a result, our fitness function is

$$f_i(x) = \frac{1}{M(x) - \min_{y \in W} M(y) + \max_{z \in W_i} M(z)}, \tag{4.4}$$

where W_i stands for all the individuals in the population of current generation.

4.4.3 Termination Criteria

The evolution stops after 50 generations.

4.4.4 Selection

Proportional Selection

Proportional (or *roulette wheel*) selection is used to select those individuals which will produce or constitute (most of) the next generation. This method ensures the selection of an individual x with a probability p_x, which belongs to a uniform probability distribution.

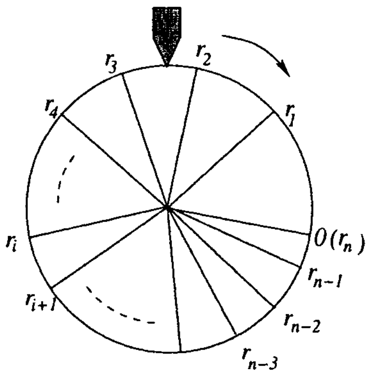


Figure 4.9: Roulette Wheel Selection

A brief description of this method is given below. Using Equation (4.4), we have the fitness $f(x_i)$ for each chromosome x_i, $i = \{1, 2, \ldots, n-1, n\}$, where n is the size of our population. Then, a number r_i is computed for each x_i:

$$r_i = r(x_i) = \frac{\sum_{k=1}^{i} f(x_i)}{\sum_{k=1}^{n} f(x_i)}$$

According to Equation (4.4), $f(x_i)$ is positive. Thus, $r(x_i) \in (0,1)$, $r_1 < r_2 < \ldots < r_{n-1} < r_n$ and $r_n = 1$. Then, we randomly generate a number s in the interval $[0,1)$. If $0 \leqslant s < r_1$, the first chromosome x_1 is selected. Or, if $r_i \leqslant s < r_{i+1}$, x_i is selected. It is like the spinning wheel in Figure 4.9. The wheel is divided into n sectors and r_i corresponds to the coordinates of the sectors on the circle. The dimension of a sector is proportional to the fitness values of its respective individual. Formally, the selection probability of a individual x_i is

$$sel(x_i) = \frac{f(x_i)}{\sum_{k=1}^{n} f(x_i)} = r_{i+1} - r_i.$$

The fixed pointer determines which sector is selected at each iteration. In the example, the pointer falls into the third sector on the wheel, so x_3 is copied to the mating pool in this round. A mating pool is used for stocking the new individuals preparing for crossover and mutation. After the selection procedure is repeated n' times, a mating pool of n' individuals is generated for later processing. Note, the size of the mating pool may be different from the population size.

The roulette wheel method has no bias. However, the better chromosomes may get selected more than once because of their higher selection probability. This allows better individuals to have more descendants than less fit ones. On the other hand, since our fitness values are always positive, any individual in the population may be selected for the mating pool.

Elitism

Elitism is the pass, without change, of the fittest individuals in one generation, to the next generation. The number or percentage of individuals involved is usually a variable set by the GA user. This ensures that a GA does not accidentally loose a highly fit individual due to crossover or mutation, which are probabilistic operations (see Section 4.4.5), which may effect any individual in the mating pool.

There are N individuals in our population. In selection, we set up a mating pool having 1 individual less than the original population. Such a mating pool goes through the other genetic operations. But the best individual is inserted directly into the next population. Thus, the mating pool is only made up of $N - 1$ individuals which are selected from all N individuals in the original population. As other genetic operators do not affect the size of the population, the number of individuals after the application of the genetic operators remains $N - 1$. These $N - 1$ chromosomes are added to the fittest chromosome in the original population, to build a new population of size N.

4.4.5 GA Operator

Crossover

Crossover is the process of combining the genes of one chromosome with those of another to create offspring that inherit traits of both parents. It is employed to reach the most promising regions. During the process, it is always trying to search (new) neighboring region. The new individuals generated from recombination should inherit part of information from their parents while being different.

According to the encoding method in Section 4.3.2, the information preserved by the chromosome is a feasible schedule which is determined by the order of genes. The operator should deal with the order instead of the values. The descendants obtained by crossover have to collect all the n genes when getting the partial order of genes from both their parents. Also, each gene could appear only once in the descendant so as to satisfy the encoding restraints.

As the result, a special two-point crossover operator is proposed for our LogP scheduling problem. Let n be the length of the chromosomes, i.e. the number of all tasks in the extended task graph. Two crossover points, integers c_1 and c_2, are chosen randomly such that $c_i \in \{0, 1, \ldots, n-1\}, i \in \{1, 2\}$ and $c_1 < c_2$. The points indicate the positions within the chromosome where the string is divided into segments. When $c_1 = 0$ or $c_2 = n - 1$, it becomes a one-point crossover. The segment between the crossover points on one parent is replace with the same genes in a different order. The new order is determined by the order that these genes appear in the other parent.

To illustrate the procedure of the crossover, let us consider two chromosomes:

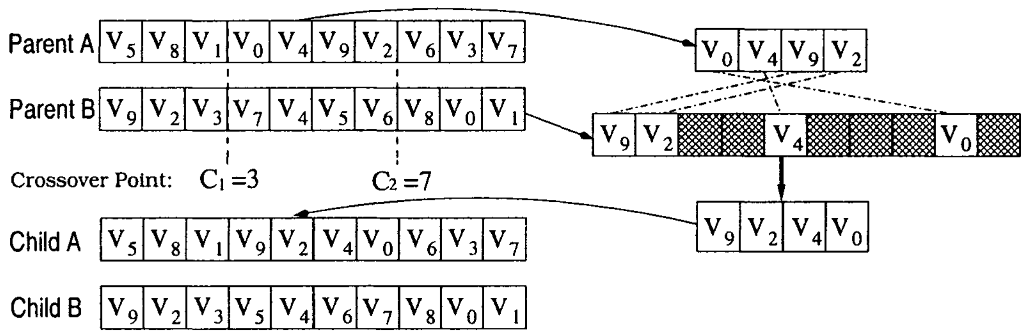
$$X_i = x_1 x_2 \ldots x_n$$

and

$$Y_i = y_1 y_2 \ldots y_n,$$

where i is the number of current generation.

The first step is to cut the chromosome X_i at the points c_1 and c_2 and to remove the sub-string x' between the cross points $x_{c_1}x_{c_1+1}\dots x_{c_2-1}$. Search the chromosome Y_i from the beginning for all the elements in x' and generate permutation x'' of x', where $x'' = x'_{c_1}x'_{c_1+1}\dots x'_{c_2-1}$, such that $\{x_{c_1}, x_{c_1+1}, \dots, x_{c_2-1}\} = \{x'_{c_1}, x'_{c_1+1}, \dots, x'_{c_2-1}\}$. At last, x' in X_i is replaced by x'' so as to form a new individual $X_{i+1} = x_1x_2\dots x_{c_1-1}x'_{c_1}x'_{c_1+1}\dots x'_{c_2-1}x_{c_2}\dots x_n$.



Parent A
V5 V8 V1 V0 V4 V9 V2 V6 V3 V7
Parent B
V9 V2 V3 V7 V4 V5 V6 V8 V0 V1
Crossover Point: C1 =3 C2 =7
Child A
V5 V8 V1 V9 V2 V4 V0 V6 V3 V7
Child B
V9 V2 V3 V5 V4 V6 V7 V8 V0 V1
V0 V4 V9 V2
V9 V2 V4 V0
V9 V2 V4 V0

Figure 4.10: Crossover Operation

Figure 4.10 explains the whole process of the recombination operation executed between 2 individuals, *parent A* and *parent B*, each of which has 10 genes. The crossover points are respectively $c_1 = 3$ and $c_2 = 7$. That means the genes from the 4^{th} to the 7^{th} in each parent chromosome will be re-ordered according to the other parent. When we modify the first parent individual *parent A*, for example, the substring "$v_0v_4v_9v_2$" is replaced by another permutation of the same genes following the order of these genes in *parent B*. *parent B* is scanned for these genes. Consequently, the order of the four genes (v_0, v_2, v_4 and v_9) in the *parent B* is "$v_9v_2v_4v_0$". To generate a new individual *child A*, we need to insert "$v_9v_2v_4v_0$" in the place where we removed "$v_0v_4v_9v_2$". Repeat these steps on *parent B*, and another offspring individual is generated: *child B* in the figure. As a matter of fact, this method allows the offspring to collect every gene only once so as to maintain the characteristics defined by the encoding.

Mutation

The goal of mutation is not to obtain any better solutions, but to introduce a measure of divergence into a converging population. Such an operation generally enhances the diversity of a population. Unlike crossover, mutation should always try to destroy the permutation of nodes so as to reach some new area of search space that could be far away from the area of current focus of search.

As for our problem, the results of mutation should still satisfy the encoding restrictions. The only thing we can change is the order of the genes, but not the values. In this way, the operation will lead to a legitimate schedule.

We employ a mutation operator to reconstruct a continuous part of the chromosome. Let n be the total number of all the tasks in the extended task graph. Two integers m_1 and m_2 are selected *randomly*, such that $0 \leqslant m_1 \leqslant (m_2 - 1) \leqslant (n-2)$. m_1 and m_2 present the position of the segment to be reordered in the chromosome. When mutating a chromosome, we extract the substring between the m_1^{th} and $(m_2 + 1)^{th}$ gene and shuffle it *randomly*. At last, we put the string back to get the new individual. When $m_1 = 0$ and $m_2 = n - 1$, the whole chromosome will be rearranged, like initializing a totally new individual randomly. The segment must contain at least 2 genes for the reconstruction to be carried out successfully. That is why m_1 and m_2 must satisfy $m_1 < m_2 - 1$.

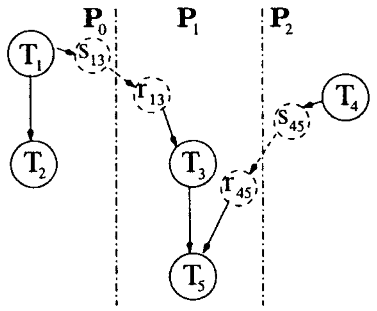


Figure 4.11: Extended Task Graph G_2'

The extended graph G_2' in Figure 4.11 was the one we had in Section 4.1. Figure 4.12 illustrates how the mutation works. The first string A beside the graph is an eligible chromosome waiting for mutation. Assume that $L = o = g = w(T_i) = 1, \forall i \in 1, \ldots, 5$. Then, the schedule S corresponds to A. It is selected to shuffle the segment from the 4th gene to the 7th. A possible result is the chromosome A', shown in Figure 4.12. S' is the schedule obtained from A'.

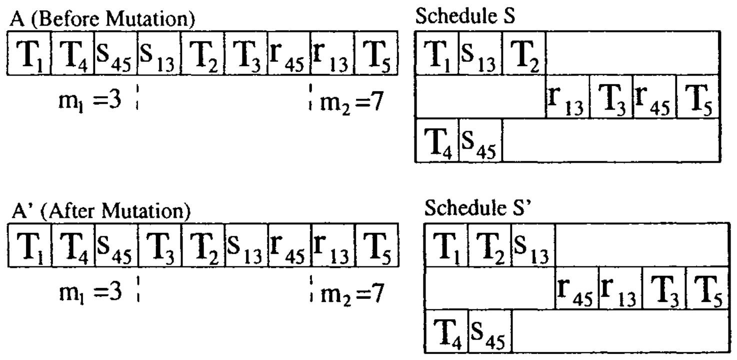


Figure 4.12: Mutation Procedure

4.5 Discussion

In this chapter, the problem of communication managements under LogP models is introduced. The communications are re-arranged with known processor assignment so as to achieve a shorter makespan. A novel genetic algorithm is proposed for this problem. We also have discussed every detail of the algorithm, including the encoding method and the genetic operators.

The algorithm's framework is set up as the SGA with a new encoding method. Not only the decoder, but also the genetic operators are specifically designed for the problem. This GA is simple and practical.

Meanwhile, there are some drawbacks. The encoding method discussed in Section 4.3.2 produces integer strings, each of which corresponds to a single solution. But, a schedule can be represented by more than one string if encoded in this way. In a sense, the search space is still much larger than the solution space. When the graphs are more complicated, this may present an obstacle to the GA in exploring the space effectively.

In addition, $g = o$ is assumed in this chapter. Although this assumption is widely used in past research, the condition $g > o$ has become a more common issue due the lagging of network developments as compared to the rapid progress on processors. It is also necessary to take this condition into account.

The next chapter is going to discuss these drawbacks in detail and to improve the GA.

Chapter 5

More Communication Scheduling under LogP

This chapter is devoted to improving the algorithm proposed in the previous chapter. New restrictions are added to the encoding method in order to minimize the representation space. In addition, the decoder is adapted for more general models.

5.1 Modified Encoding

5.1.1 Disadvantages of the Original Design

The original encoder is capable of generating chromosomes representing any and all permutations of tasks. The decoder transforms these permutations into feasible schedules. A GA using this encoding method has produced better schedules than existing algorithms. It is also superior to the encoding method that uses random numbers for priority values.

However, there are still drawbacks affecting the GA's performance. The order of the tasks in a schedule are determined not only by their priorities, but also by some other constraints such as partial orders on the set of the tasks. There may be more than one chromosome representing the same solution. For instance, a task v_i proceeding another task v_j can start if a path exists from node v_i to v_j, regardless of their priority values.

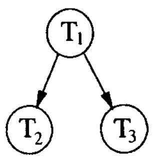


Figure 5.1: Simple Task Graph G

When scheduling the tasks of graph G in Figure 5.1 on the same processor, no communications are required. The algorithm is capable of dealing with this class of problems, as long as the processor assignments are known. According to the original encoding method, there are 6 chromosomes in total.

$$C_1 \quad (T_1, T_2, T_3);$$
$$C_2 \quad (T_1, T_3, T_2);$$
$$C_3 \quad (T_2, T_1, T_3);$$
$$C_4 \quad (T_2, T_3, T_1);$$
$$C_5 \quad (T_3, T_1, T_2);$$
$$C_6 \quad (T_3, T_2, T_1).$$

T_1 is the predecessor of both T_2 and T_3. Hence, T_1 has to start first even if it has the lowest priority. Only the order of T_2 and T_3 is determined by the priorities. There are two feasible schedules for this problem:

$$S_1: \; T_1, T_2, T_3;$$

$$S_2: \; T_1, T_3, T_2.$$

C_1, C_3 and C_4 all represent the same solution S_1, while the others are mapped to S_2. All 6 chromosomes must be evaluated before determining which schedule is the best. It takes much more time to search such a relatively larger representation space than to search the search space itself.

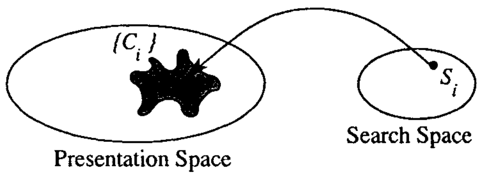


Figure 5.2: Mapping between representation space and search space

Figure 5.2 illustrates such a mapping between the representation space and the search space. More than one chromosomes represent one schedule. When the mapping is not one-to-one, the representation space is much larger than the search space. In fact, the problem constraints are more important than task priorities, as they must be satisfied before the priorities are taken into account. The proportion of representation space to search space become more significant when the graph is more complicated.

With a representation space much larger than the search space, it is difficult to find the optimal solution, as much more computations are required. There is also a high risk of converging to a local optima, since it is possible for a GA to get stuck around different points in the presentation space, which actually stand for one or more sub-optimal schedules.

5.1.2 Encoding Restrictions

The representation space needs to be minimized in order to avoid redundant search. One straightforward way to do this is by placing restrictions on the representational scheme.

Nevertheless, restrictions may introduce bias into the algorithm which may result in the exclusion of part of the search space. Both [24] and [14] state that nodes in their chromosomes are sorted by their heights. Although [24] has proved that a schedule is legal if it satisfies the height-ordering condition, it is not sufficient to suggest a new technique that minimizes the representation space without ensuring that it does not exclude valid solutions. In fact, height-ordering restrictions limit the search to particular solutions, which form a proper subset of all feasible solutions. An

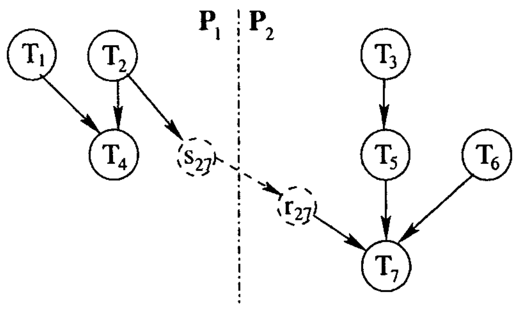


Figure 5.3: Extended Task Graph G'_h

extended task graph G'_h with a pair of communication tasks is illustrated in Figure 5.3. A set partition of node set V'_h can be $\{\{T_1, T_2, T_3\}, \{T4, s_{27}, T_5, T_6\}, \{r_{27}\}, \{T_7\}\}$, in which nodes in each subset have the same height. According to the height-ordering restriction, nodes with lower heights must be placed before nodes with higher heights. For example, T_1 has to be put in a position before T_6. Obviously, this has introduced a non-existing precedence relationship into the problem, e.g. T_6 is prior to T_7. Such additional relationships prevent the algorithm from generating solutions produced by chromosomes that do not satisfy the height-ordering condition, such as

$$C_1(G'_5) = (T_2, s_{27}, T_1, T_4, T_3, r_{27}, T_5, T_6).$$

If the optimal solution is among these solutions, a GA is not able to find it regardless of the operators' performance. This condition is surely undesirable.

Instead of the heights, we sort the nodes by the precedence relationships. The *linear extensions*[1] of the given graph are used as chromosomes. Thereby, the partial order is encapsulated in the chromosomes. A chromosome is still a fully permutation of the tasks. Meanwhile, the chromosome must satisfy the partial order of the graph. Whenever precedence exists between two nodes, their ranks in a chromosome must reflect such a relationship. A node has to come after all its predecessors, which may not be directly connected with the node. Such a restriction allows chromosomes to preserve problem specifications while providing enough information for GA processing. Besides C_h above,

$$C_2(G'_5) = (T_3, T_6, T_2, T_1, s_{27}, T_5, r_{27}, T_7)$$

is also a valid chromosome that satisfies the partial order of the graph.

[1]A *linear extension* of a partially ordered set X is a permutation of the elements x_1, x_2, ... of X such that $x_i < x_j$ implies $i < j$. For example, the linear extensions of the partially ordered set ((1,2),(3,4)) are 1234, 1324, 1342, 3124, 3142, and 3412, all of which have 1 before 2 and 3 before 4. [10]

5.1.3 Initialization

The initialization method needs modification to take into account the added restrictions. First, the new initialization process should be able to produce only linear extensions of the extended task graph. The algorithm can be improved if initialization can only generate legal individuals. Second, the individuals generated in this step should bear no bias. Individuals in the initial population are expected to be mapped to points which are distributed uniformly in the solution space.

We denote the population size by N. Given a partial order $\mathbb{P}$, which is defined by the extended task graph $G'(V', E', w', l')$, over the set of nodes V', initialization is expected to generate a set $R = \{\mathbb{F}_1, \mathbb{F}_2, \ldots, \mathbb{F}_i, \ldots \mathbb{F}_N\}$ of total order $\mathbb{F}_i$ as initial sequences, such that:

$$\mathbb{P} \subseteq \mathbb{F}_i, \ \forall\ \mathbb{F}_i \in R.$$

It is straightforward to apply a repair mechanism to transform randomly generated permutations into strings that necessarily meet the restriction. Denote the randomly generated linear order by $\mathbb{F}'$. If $\mathbb{P} \not\subseteq \mathbb{F}'$, $\mathbb{F}'$ needs repair. $\mathbb{F}$, the result of repairing $\mathbb{F}'$, should still be a total order, such that $\mathbb{P} \subseteq \mathbb{F}$. Furthermore, it should be a combination of $\mathbb{P}$ and $\mathbb{F}'$. But it is not a simple union operation, as conflicts may exist between $\mathbb{F}'$ and $\mathbb{P}$ when $\mathbb{P} \not\subseteq \mathbb{F}'$. $\mathbb{P}$ is the dominant constraint in the problem. Therefore, when one relation $(x_i, y_i) \in \mathbb{P}$ conflicts with $(y_i, x_i) \in \mathbb{F}'$, we always follow (x_i, y_i) in $\mathbb{P}$. Otherwise, $\mathbb{F}$ would not satisfy $\mathbb{P} \subseteq \mathbb{F}$.

Unfortunately, the repair mechanism has shortfalls regardless of its implementation. Combining $\mathbb{P}$ and $\mathbb{F}'$ to get $\mathbb{F}$ is similar to updating $\mathbb{F}'$ with relations from $\mathbb{P}$. As a linear extension of partial order $\mathbb{P}$, $\mathbb{F}$ must has transitivity ($a \leqslant b$ and $b \leqslant c$ implies $a \leqslant c$). One replacement may result in a chain of conflicts. Some other relations have to be changed in order to maintain the transitivity. There could be more than one way to adjust conflicting relations. But, any method of modification is bound to introduce some bias into the resulting population.

Let set $V = a, b, c$ and the partial order $\mathbb{P} = \{(c, a)\}$. $\mathbb{F}' = (a, b, c)$ is a linear order over set V and $\mathbb{P} \not\subseteq \mathbb{F}'$. $\mathbb{F}'$ needs repair. $\mathbb{F}'$ includes 3 binary relations, (a, b), (b, c) and (a, c). The last one conflicts with (c, a) in $\mathbb{P}$. Then, $\mathbb{F}'$ is updated to $\{(a, b), (b, c), (a, c)\}$. However, $a \prec b$ and $b \prec c$ lead to $a \prec c$. This means either (a, b) or (b, c) has to be changed. In terms of permutations, a has to be placed before c according to $\mathbb{P}$. There are 3 positions for b: before a: $(b, a), (b, c)$; between a and c: $(a, b), (b, c)$; after c: $(a, b), (c, b)$. None of these positions can be presented by a subset of $\mathbb{F}'$. A decision on which position can not be determined by $\mathbb{F}'$ alone. Always arranging the involved elements according to a fixed rule introduces bias into the results. On the other hand, it is almost impossible to make such choices flexible, on a random basis, due to the exponential complexity of the operation.

An approach similar to a classical algorithm for topological sorting [29] is applied to generate linear extensions with minimal bias. Legal individuals are generated directly instead of producing illegal ones, and then repairing them. Various topological sorting algorithms are provided by existing computing libraries, such as LEDA [40]. But such libraries usually produce only one topological sorting for a certain graph, but not a set of uniformly distributed strings. The initialization algorithm is shown by Algorithm 3.

Algorithm 3 GA Initialization

```
Input:
    $G'(V', E', w', l')$: an extended task graph of $G(V, E, w, l)$.
Output:
    $C(G') = (v_{i_0}, v_{i_1}, ...., v_{i_{n-1}})$: a legal chromosome for scheduling $G'$.

    Begin
    {Denote the set of nodes that have no predecessors with $SN$
    and source node of edge $e_i$ with $src(e_i)$.
    Initially, $SN = source(G')$, $C(G') = \emptyset$ .}
    while $SN \neq \emptyset$ do
        {Randomly select a node $v_i$, such that $v_i \in SN$ ;}
        $C(G') := C \cup (v_i)$;
        $V' := V'/v_i$;
        for all $e_i \in E'$ do
            if $src(e_i) = v_i$ do
                $E' := E'/e_i$;
            end if
        end for
        $SN := source(G')$;
    end while
    End
```

An example of the application of Algorith 3 is shown in Figure 5.4. A chromosome $C_1(G'_6) = (T_1, T_2, T_3, T_4)$ is generated for extended task graph G'_6. Such a process is illustrated by Figure 5.5. Various choices in each iteration might result

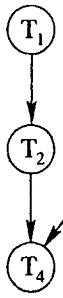


Figure 5.4: Extended Task Graph G'_6

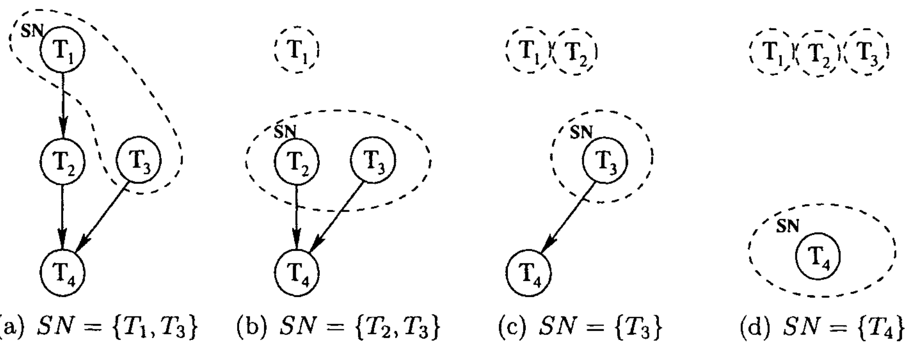


Figure 5.5: Initialization Procedure

in different chromosomes. Table 5.1 lists 3 individuals that may be generated during initialization. Table 5.2 lists all possible permutations using all 4 nodes. Only 3

Iteration 1	Iteration 2	Iteration 3	Iteration 4	Results
Take T_1	Take T_2	Take T_3	Take T_4	(T_1, T_2, T_3, T_4)
	Take T_3	Take T_2	Take T_4	(T_1, T_3, T_2, T_4)
Take T_3	Take T_1	Take T_2	Take T_4	(T_3, T_1, T_2, T_4)

Table 5.1: All possible results by initializing GA population for G

of 24 permutations are legal, which are exactly the 3 individuals in Table 5.1. As

$(T_1, T_2, T_3, T_4)\surd$	(T_2, T_1, T_3, T_4)	$(T_3, T_1, T_2, T_4)\surd$	(T_4, T_1, T_2, T_3)
(T_1, T_2, T_4, T_3)	(T_2, T_1, T_4, T_3)	(T_3, T_1, T_4, T_2)	(T_4, T_1, T_3, T_2)
$(T_1, T_3, T_2, T_4)\surd$	(T_2, T_3, T_1, T_4)	(T_3, T_2, T_1, T_4)	(T_4, T_2, T_1, T_3)
(T_1, T_3, T_4, T_2)	(T_2, T_3, T_4, T_1)	(T_3, T_2, T_4, T_1)	(T_4, T_2, T_3, T_1)
(T_1, T_4, T_2, T_3)	(T_2, T_4, T_1, T_3)	(T_3, T_4, T_1, T_2)	(T_4, T_3, T_1, T_2)
(T_1, T_4, T_3, T_2)	(T_2, T_4, T_3, T_1)	(T_3, T_4, T_2, T_1)	(T_4, T_3, T_2, T_1)

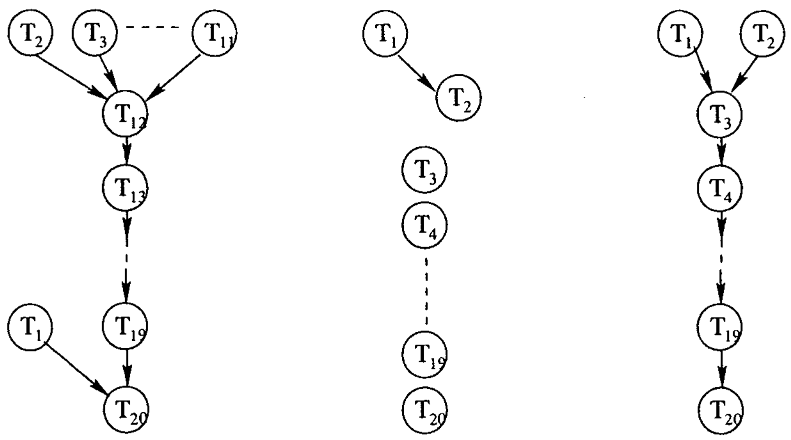
Table 5.2: All linear orders over set of nodes

long as the random numbers in each step are generated using a uniform distribution, it is possible to generate any strings that satisfy the partial order. Such an initialization method produces valid chromosomes only, and it is able to generate all valid candidate solutions.

We can not deny that some bias still exists. As the choices are made randomly, the probabilities associated with the generation of chromosomes are unequal. It is 0.5 for the permutation (T_3, T_1, T_2, T_4) and 0.25 each for the rest. Though, the effect of such a distinction becomes less significant when the graph gets larger. Most of the graphs we work with have at least 20 nodes. Also, the number of nodes on the critical path is usually less than half of the total number of nodes. Suppose there are 10 nodes on the critical path of a graph with 20 nodes (Figure 5.6(a)). Then, the highest probability of generation for a single chromosome is

$$Pr_{max} \leqslant \frac{1}{10} \times \frac{1}{9} \approx 1\%.$$

Pr_{max} is even lower if there are more nodes. For a graph with 20 nodes, the number of all permutations is 20!. If there is only 1 edge in the graph, e.g. G_2 in Figure 5.6(b), the number of valid chromosome will be $\frac{20!}{2}$. As for G_3 in Figure 5.6(c), there is only one pair of nodes that have no relation between them. Thus, the number would be 2.



(a) G_1:$Pr_{max} = \frac{1}{90}$ (b) G_2:1 edge (c) G_3:2 strings

Figure 5.6: Various Graphs

Both cases are rare, and for most graphs, the number of legal strings is between $\frac{20!}{2}$ and 2. Therefore, it is reasonable to assume that the average lies somewhere in the vicinity of

$$\frac{2 + 10 \times 19!}{2} \approx 5 \times 19! > 10^{17}.$$

The size of our populations was kept to 100 or less, due to the complexity of the decoder. Therefore, $Pr_{max} \lesssim \frac{1}{90}$ should be acceptable when selecting less than 100 strings out of 10^{17}.

5.1.4 GA Operator

As initialization has been adapted to new restrictions, GA operators should also produce legal chromosomes. Repair mechanisms are not recommended as they could slow down the algorithm and introduce bias. Reproduction does not modify individuals. Therefore, the reproduction method described in Section 4.4.4 is still valid. As such, only crossover and mutation are discussed in this section.

The original crossover operator cuts strings at two points and replaces the middle part with the same genes but in probably a different order. Such a crossover operator still works for the new encoding.

Let

$$X_i = (x_1, x_2, \ldots, x_n),$$

and

$$Y_i = (y_1, y_2, \ldots, y_n)$$

be two chromosomes involved in a crossover operation, where i refers to the current generation and n stands for the total number of nodes. It is clear that X_i and Y_i satisfy the partial order $\mathbb{P}$ defined by the graph. That is, $\mathbb{P} \subset X_i$ and $\mathbb{P} \subset Y_i$. Assume the crossover points are c_1 and c_2, such that $c_1 < c_2$ and $c_1, c_2 \in [1, n]$. This divides the nodes in X_i into 3 sets: $N_{1x} = \{x_1, x_2, \ldots, x_{c_1}\}$; $N_{2x} = \{x_{c_1+1}, x_{c_1+2}, \ldots, x_{c_2}\}$; $N_{3x} = \{x_{c_2+1}, x_{c_2+2}, \ldots, x_n\}$. Hence, for any $x_{m_1} \in N_{1x}$, $x_{m_2} \in N_{2x}$ and $x_{m_3} \in N_{3x}$,

$$x_{m_1} \prec_{X_i} x_{m_2} \prec_{X_i} x_{m_3}. \tag{5.1}$$

Also, there are 3 total orders $\mathbb{F}_{1x} : N_{1x} \times N_{1x}$; $\mathbb{F}_{2x} : N_{2x} \times N_{2x}$; $\mathbb{F}_{3x} : N_{3x} \times N_{3x}$, satisfying $\mathbb{F}_{1x}, \mathbb{F}_{2x}, \mathbb{F}_{3x} \subset X_i$. The crossover operation is looking for a full order $\mathbb{F}'_{2x} : N_{2x} \times N_{2x}$, such that $\mathbb{F}'_{2x} \subset Y_i$, and $\mathbb{F}_{2x}$ is replaced with $\mathbb{F}'_{2x}$, to obtain a new chromosome X_{i+1}, such that $\mathbb{F}_{1x}, \mathbb{F}'_{2x}, \mathbb{F}_{3x} \subset X_{i+1}$.

For $(x_j, x_k) \in \mathbb{P}$:

1. If $x_j \in N_{1x}$ and $x_k \in N_{1x}$ then $(x_j, x_k) \in \mathbb{F}_{1x}$. Thus, $(x_j, x_k) \in X_{i+1}$. In the same way, $(x_j, x_k) \in X_{i+1}$ if $x_j \in N_{3x}$ and $x_k \in N_{3x}$;

2. If $x_j \in N_{1x}$ and $x_k \in N_{3x}$, according to Equation (5.1) $x_j \prec_{X_i} x_k$. This is true since the positions of x_j and x_k remains the same in X_{i+1},$x_j \prec_{X_{i+1}} x_k$. $(x_j, x_k) \in X_{i+1}$.

3. If $x_j \in N_{1x}$ and $x_k \in N_{2x}$, $x_j \prec_{X_i} x_k$, then, $x_j \prec_{X_{i+1}} x_k$. This also exists if $x_j \in N_{2x}$ and $x_k \in N_{3x}$.

4. If $x_j \in N_{2x}$ and $x_k \in N_{2x}$, then $(x_j, x_k) \in Y_i$ as $(x_j, x_k) \in \mathbb{P}$. Therefore, $(x_j, x_k) \in \mathbb{F}'_{2x}$. Meanwhile, $\mathbb{F}_{1x} \subset X_{i+1}$, $(x_j, x_k) \in X_{i+1}$.

In conclusion, $(x_j, x_k) \in X_{i+1}$, $\forall$ $(x_j, x_k) \in \mathbb{P}$. That is

$$\mathbb{P} \subset X_{i+1}.$$

That means X_{i+1} also satisfies $\mathbb{P}$, the partial order derived from the graph. The chromosomes generated by this crossover method are still legal strings as defined in Section 5.1.2.

The original mutation operator shuffles the whole chromosome. Obviously, there is a high chance for such a mutation operator to interrupt the partial order embedded in a valid chromosome.

We propose a new mutation operator that only swaps two neighboring genes. Furthermore, there must be no precedence relationship between the two chosen genes. Otherwise, the output string would not satisfy the graph's partial order.

Consider the chromosome

$$X_i = (x_1, x_2, \ldots, x_n)$$

which is selected for mutation. i is the generation number and n is the number of nodes. we denote the graph partial order by $\mathbb{P}$. A point m is selected, such that $1 \leqslant m \leqslant (n-1)$ and $(x_m, x_{m+1}) \notin \mathbb{P}$. We switch x_m and x_{m+1} to obtain a new chromosome

$$X_i' = (x_1, x_2, \ldots, x_{m-1}, x_{m+1}, x_m, x_{m+2} \ldots, x_n).$$

The switch does not affect any binary relationship that involves x_m or x_{m+1}. Since $(x_m, x_{m+1}) \notin \mathbb{P}$, (x_{m+1}, x_m) does not violate $\mathbb{P}$. Therefore,

$$X_i' \supset \mathbb{P}.$$

This operator keeps the graph partial order.

5.1.5 Discussions

Let N be the number of nodes in the extended task graph, M be the number of edges. We denote the number of points in the presentation space by K. When the individuals are encoded using the method of Section 4.3.2, there would be a total of totally $K_{org} = N!$ legal strings in the representation space. As for the modified encoding method, K_{mod}, the size of the new representation space varies with the graph's structure.

Assume $M < N$. K_{mod}'s upper bound occurs when all M edges have a common source node (or target node). In that case, maximal $\max(K_{mod}) = \frac{N!}{M+1}$. When the edges are on the same path, minimal $K_{mod} = \frac{N!}{M!}$ is obtained. Thus, the proportion k of K_{mod} to K_{org} satisfies

$$\frac{1}{M!} \leqslant k \leqslant \frac{1}{M+1},$$

when $M < N$. Obviously, k is even smaller if M is increased. In brief, the representation space with modified encodings is at least $(M + 1)$ times smaller than the original space, while representing the same set of solutions.

However, the mapping of the modified method is not yet one-to-one. There are other problem constraints, such as, the processor assignments. But it costs too much in terms of computation to achieve such a mapping. The sub-optimal mapping achieved by the modified method is efficient and practically sufficient.

5.2 Scheduling Communications with Gaps

Gaps, denoted by g in the LogP model, define the minimum time interval between two consecutive transmission or reception events, on one processor. That is , the processor is just not available for one type of communications during a gap. It may be possible to schedule computational tasks as well as other types of communications on this processor, if there are not any tasks executing on it. The length of a gap is determined by the capacity of the network. The network capacity is finite, with an upper bound of $\lceil \frac{L}{g} \rceil$. A processor can not transmit any messages that would exceed this limit.

One or more of the four parameters of a LogP model can be eliminated. It is possible to ignore some parameters by using certain approximations [15] based on certain assumptions. For instance, 2ETF [27] ignores the gap g. This can be achieved by increasing o so as to include g.

Scheduling procedure can be simplified using approximations. But, an approximation technique works for only one class of machines. Since our goal is to find a general approach to schedule under LogP model, it is important that the algorithm should also be able to handle the cases when $g > o$, so that it can be applied to any machine.

5.2.1 Problem Description

The problem remains the same as the one described in Chapter 4, except the model. The problem covers all LogP instances. The constraint that $g = o$ is no longer necessary. We still assume *independent data semantics.* To generate all these necessary communication tasks, an extended task graph $G'(V', E', w', l')$ is constructed based on the original task graph $G(V, E, w, l)$ and the processor assignment $\Pi(G)$. Respectively, $\Pi(G')$ is the processor assignment after graph extension. With this processor assignment, G' is scheduled under model $M(L, o, g, P)$.

A LogP-feasible schedule must include both starting time and processor assignment for each node in the extended task graph. A processor can not execute more than one task at a time. A task can only start after all its predecessors have finished. The interval from completion of a send task to the beginning of its respective receive task must be equal or greater than L. As for gaps, a delay of at least g exists between any two consecutive send or receive operations. Note, there are not such delays required between a send task and a receive task on the same processor.

We designate the starting time and processor that have not been scheduled by $\perp$. v, v_i and v_j are the tasks in G'. s_{ij}, s_i and s_j are *send* tasks, and r_{ij}, r_i and r_j are *receive* tasks. Then, a *LogP-feasible* schedule can be defined by the following constraints:

1. $\sigma(v) \neq \perp$ and $\pi(v) \neq \perp$, $\forall v \in V'$;

2. if $\pi(v_i) = \pi(v_j) \neq \perp$ and $v_i, v_j \in V'$, then $\sigma(v_i) + w(v_i) \leqslant \sigma(v_j)$ or $\sigma(v_j) + w(v_j) \leqslant \sigma(v_i)$;

3. if $v_i \prec_{G'} v_j$ and $v_i, v_j \in V'$, then $\sigma(v_i) + w(v_i) \leqslant \sigma(v_j)$;

4. if coupled communications $s_{ij}, r_{ij} \in V'$ and $(s_{ij}, r_{ij}) \in E'$, then $\sigma(s_{ij}) + w(s_{ij}) + L \leqslant \sigma(r_{ij})$;

5. if send tasks $s_i, s_j \in V'$ and $\pi(s_i) = \pi(s_j) \neq \perp$, then $\sigma(s_i) + w(s_i) + g \leqslant \sigma(s_j)$ or $\sigma(s_j) + w(s_j) + g \leqslant \sigma(s_i)$;

6. if receive tasks $r_i, r_j \in V'$ and $\pi(r_i) = \pi(r_j) \neq \perp$, then $\sigma(r_i) + w(r_i) + g \leqslant \sigma(r_j)$ or $\sigma(r_j) + w(r_j) + g \leqslant \sigma(r_i)$;

Consider the extended task graph G'_3 shown in Figure 5.7. There is a total of 4 computational tasks assigned to 2 processors, with 3 pairs of (derived) communication tasks. We assume $w(T_i) = 1$, where $i \in \{1, 2, 3, 4\}$. This graph is scheduled under

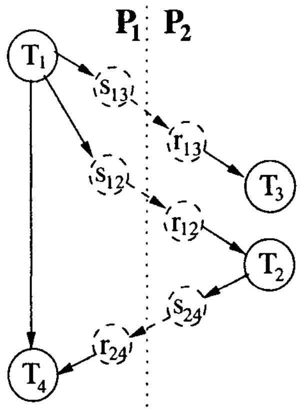


Figure 5.7: Extended Task Graph G'_3

a LogP model $M_g = (1, 1, 2, 2)$, in which the gap is greater than the overhead. We should consider gaps whenever a communication task is scheduled. $S_{G'_3}$ in Figure 5.8 is a feasible schedule for G_1 under M_g. The earliest communication task in S_{G_1} is s_{12} on processor P_1. As soon as s_{12} starts, no other send tasks can be put on P_1 during the gap, from the moment $t = 1$ to $t = 3$. That is why s_{13} can only start at time $t = 3$, but not earlier than $t = 2$. When processor P_2 starts receiving the message for T_2 from P_1, a gap from $t = 4$ to $t = 6$ for receive tasks is generated too. Consequently, r_{13}, another receive task on P_2, can only be scheduled after the gap, i.e. after $t = 6$. T_2 is executed immediately after r_{12} at time $t = 5$, because the gap does not prevent the processor from performing other types of tasks. Similarly, s_{24} starts right after r_{13} since they are different types of communication tasks.

5.2.2 List Scheduling with $g > o$

So far, a GA has been successfully applied to the problem when $g = o$. We continue to use GA, to solve the general case. Model specifications are only involved

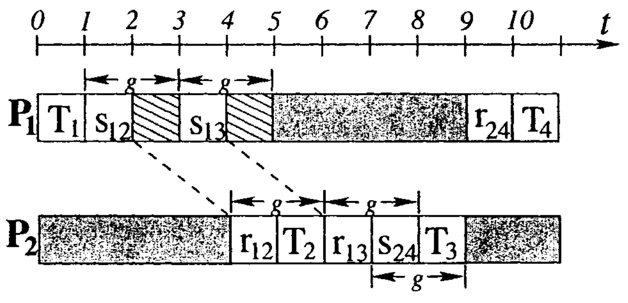


Figure 5.8: Feasible Schedule S_{G_1} of G_1, under model $L = 1, o = 1, g = 2, P = 2$

in the decoder part of the original GA. Hence, we focus on accommodating our list scheduling algorithm to the case where $g > o$. Having such a decoder, we only need some trivial modification to the GA itself. The modified GA is described in the next section.

Task ranking and processor assignments are given as the inputs. Therefore, we only need to determine the starting time of the tasks, in cases where g is not necessarily equal to o. Above all, it is important to find out how a dominant gap influences the scheduling rules. First, a gap results from limited network capacity; it occurs as soon as a processor starts processing a communication task. Second, a gap can only affect the scheduling rules between two consecutive send tasks or receive tasks. These two tasks must be the same type. Accordingly, a gap always accompanies with a communication task. And the type of this task determines the type of task that will be delayed. So, in most cases, the scheduling strategy should remain the same as Algorithm 2. The only situation to resolve is that when a communication task is to be carried out before the end of the gap.

Suppose there is a send task s_{ij} finishing on processor P_m at time t_{CM}, as shown in Figure 5.9, and the next step is to select a task to start at t_{CM}, if possible. Processor

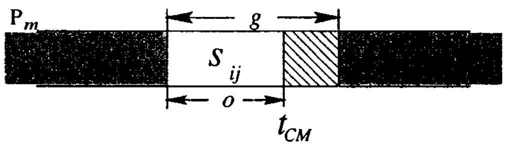


Figure 5.9: A communication task s_{ij} finishes on P_m at t_{CM}

P_m is idle at this moment, as s_{ij} is done. With $g > o$, no send operations can be performed before $(t_{CM} + g - o)$. Meanwhile, both computational tasks and receive tasks can be scheduled as long as they are ready before t_{CM}. Among the ready nodes, the one with the highest priority starts at this moment. Hence, for the period from t_{CM} to $t_{CM} + g - o$, the processor is still unavailable for send tasks, but available for other types of tasks.

To minimize the modification to Algorithm 2, the concept of *available time* of a processor is introduced in place of the *completion time. Available time* indicates the earliest moment when a processor becomes available for tasks of a certain type. We denote it by $avt(P_i, j)$, where i is the processor number and j specifies the task type. Values of available times for different types on the same processor at the same moment may vary. But at any moment the following conditions must be satisfied:

$$avt(P_i, \text{SEND}) \geqslant avt(P_i, \text{COMP})$$
$$avt(P_i, \text{RECV}) \geqslant avt(P_i, \text{COMP})$$

for $\forall i \in \{0, 1, \ldots, n-1\}$, where n is the number of processors. Whenever a task is scheduled, all three available times on the corresponding processor must be updated at the same time. However, the updates are carried out differently. It is clear that the available time of a processor for computational tasks is exactly the processor's completion time, i.e. $avt(P_i, \text{COMP}) = ct(P_i)$, at any time. It is more complicated to update the available time for communication tasks. In fact, we explain how this is done for one type of communication; the other one is done the same way. Consider Figure 5.10. v_j is the last task executing on processor P_i, at $t = CM$. Obviously, it

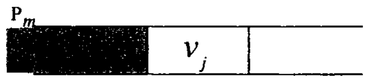


Figure 5.10: v_j, the last task scheduled on Pi at $t = CM$

finishes at $\sigma(v_j) + w(v_j)$. Denote the available time before v_j starts for send tasks on P_i by $avt_{\text{before } v_j}(P_i, \text{SEND})$. Assume v_j is a computational task. In this case, we do not update $avt(P_i, \text{SEND})$, if $avt_{\text{before } v_j}(P_i, \text{SEND}) > (\sigma(v_j) + w(v_j))$. A gap must be taken into account for all send tasks until $avt_{\text{before } v_j}(P_i, \text{SEND})$. The execution of a computational task has no impact on this. When $avt_{\text{before } v_j}(P_i, \text{SEND}) \leqslant (\sigma(v_j) + w(v_j))$, $avt_{\text{after } v_j}(P_i, \text{SEND})$ should be v_j's completion time. The condition that v_j is a receive task should be the same as the one v_j is a computational task. If v_j is a send task, then no send task can start until the end of v_j's gap. The end of this gap is the available time of this processor for send tasks. All of these conditional computation are represented by Equation (5.2).

$$avt(P_i, x) = \begin{cases} \sigma(v_j) + w(v_j) & \text{if } x = \text{COMP}, \\ \sigma(v_j) + g & \text{if } x \neq \text{COMP and } x = \text{type}(v_i), \\ \max\{avt_{\text{before } v_j}(P_i, x), \sigma(v_j) + w(v_j)\} & \text{otherwise.} \end{cases} \tag{5.2}$$

A task v_j is schedulable at time t iff. $avt(\pi(v_j), type(v_j)) \leqslant t$ **and** $rt(v_j) \leqslant t$.

Available time takes the place of completion time. Thus, it also acts as a key for determining the next decision moment (NM) in each iteration, just like completion time does in Algorithm 2. Since a task can only start when it is ready and the assigned processor is free for it, the earliest moment an unscheduled task can be scheduled is defined as:

$$st(v_i) = \max(rt(v_i), avt(\pi(v_i), type(v_i))), s.t. v_i \in U. \tag{5.3}$$

After all tasks that can start have been scheduled in each iteration of list scheduling, we step forward to the earliest moment when there are some ready tasks can start. The earliest time that an unscheduled task can be scheduled is calculated with Equation (5.3). NM can be easily decided using Equation (5.4).

$$NM = \min_{v_i \in V}\{st(v_i) | st(v_i) > CM\}. \tag{5.4}$$

With different ways of defining what task can start and determining the next decision moment, the rest of the list scheduling algorithm remains the same as Algorithm 2. Tasks that can be scheduled are included into the set SN at each iteration. These tasks are scheduled in sequence based on their priorities. The higher priority a task has, the earlier it starts. Whenever a task is scheduled, SN as well as ready time of all this task's successors must be updated. It moves on to the next iteration whenever there are no more tasks that can be scheduled at the current moment. And, the scheduling stops when all the tasks have been scheduled.

With chromosome

$$X = \{T_3, s_{15}, T_1, T_2, r_{26}, T_5, s_{26}, T_4, r_{15}, T_6\}$$

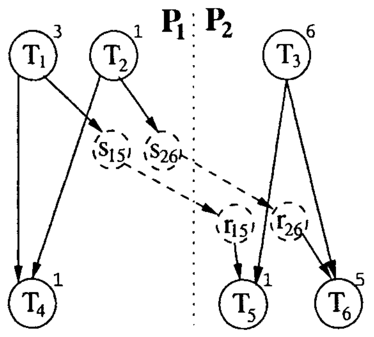


Figure 5.11: Extended Task Graph G'_4

as input, scheduling the extended task graph G'_4 in Figure 5.11 is done using model $(1, 1, 3, 2)$. The resultant schedule is shown in Figure 5.12. Its makespan is 13.

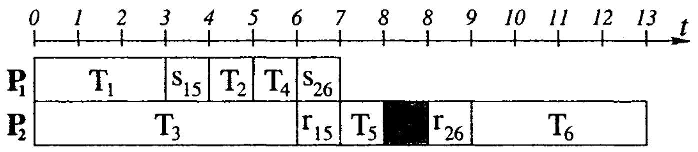


Figure 5.12: Schedule $S_{G'_4}$ introduced by X

Gaps lead to different values of available times for three types of tasks. A communication task can not start during a gap generated by a communication task of the same type. Furthermore, available time for the type is not affected by gaps generated by another type. Also, there is no difference between available times for all types if $g = o$. In this case, available time works exactly the same way as completion time in Algorithm 2. In conclusion, the modified algorithm produces a feasible schedule without increasing the computational complexity of the original algorithm.

5.2.3 Filling Gaps

Like any list scheduling algorithms, the decoder schedules a task as soon as possible. Meanwhile, gaps delay one type of communication. Sometimes, the delayed communication task is not able to compete with other ready tasks, at the same moment, even if the communication task has a higher priority.

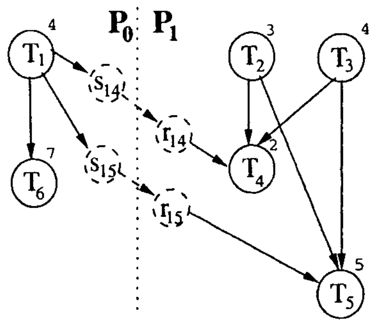


Figure 5.13: Extended Task Graph G'_5 under LogP model $(1, 1, 2, 2)$

Consider the extended task graph G'_5 in Figure 5.13, which is derived under a model instance $M_g = (2, 1, 2, 2)$. Each computational task in G'_5 is labelled with its cost, i.e. length of execution time. And there are two pairs of communications, which are from the same source task T_1. Both *send* tasks, s_{14} and s_{15} are going to be ready at the same moment. Due to the gap which is longer than the overhead, the difference between the starting times of these two *send* tasks is at least g, and the same for *receive* tasks. Gaps of the first *send* and the first *receive* task are critical. Different decisions regarding these gaps lead to different results. Figure 5.14 demonstrates the

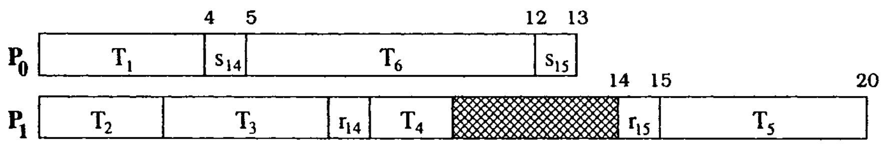


Figure 5.14: Feasible schedule S_1 when all gaps are filled. $makespan(S_1) = 24$

schedule S_1 generated by Algorithm 4. T_6 starts immediately after s_{14} since it is the only task that can be scheduled on P_0 at time $t = 5$. The process call *filling a gap* occurs when a task is scheduled on a processor, where there is at least one gap. T_6

is so long that s_{15} is put off for much longer than a gap. r_{15} and T_5 are also delayed. To avoid this, S_2 in Figure 5.15 leaves the idling period unfilled between s_{14} and s_{15}.

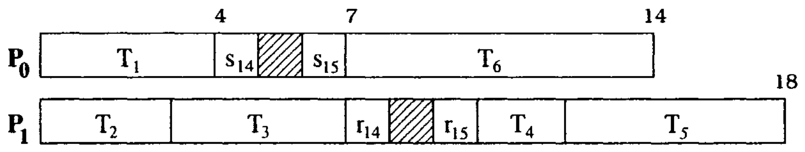


Figure 5.15: Feasible schedule S_2 when not filling the gaps. $makespan(S_2) = 19$

In the same way, T_4 is not inserted right after r_{14}. S_2's makespan is less than S_1's. S_2 is still not the shortest. Figure 5.16 shows the optimal solution S_3 for G_5'. The

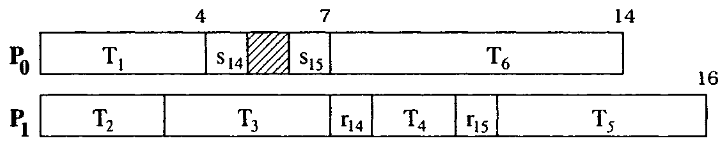


Figure 5.16: Feasible schedule S_3 when filling a gap sometime. $makespan(S_3) = 18$

only difference between S_2 and S_3 is that T_4 starts between r_{14} and r_{15}. The filling has eliminated an idle period on P_1 so as to reduce the makespan. Using the original GA and the list scheduling decoder results in losing some solutions. It might not be able to reach the optimal solution, S_3.

As a matter of fact, the effect of filling a gap is determined by characteristics of the specific gap, especially the attributes of each task that is ready either before or during the gap. For instance, if a filling does not exceed the gap, it is better to fill the gap. However, the situation becomes more complicated when there is not such a task. As it is implied in the example, it is almost impossible to conclude with a simple strategy. In fact, finding such rules is a separate optimization problem.

Benefiting from the robustness of GAs, we embed the gap filling decision into the encoding instead of setting-up a whole raft of rules. There can be more than one chromosome corresponding to one solution, given the encoding method of Section 4.5. It remains the same when applied to the problem with $g > o$. Basically, we split the representations of the same solution into a few groups. Different groups fill gaps on different positions. Whether the gap is filled or not is determined by comparison of priorities. In this way, a GA can explore the possibility of gap filling while seeking the optimal order of all tasks.

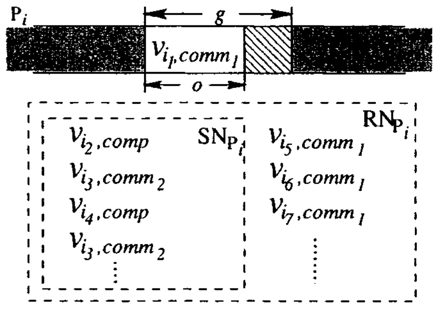


Figure 5.17: General Situation Requiring Gap Filling Decision

Figure 5.17 shows a general situation. A filling decision is required on processor P_i, when communication task $v_{i,comm_1}$ is finished at time $t = \sigma(v_{i,comm_1})+o$. In the figure, the first subscript of a node is the rank of the task, i.e. the priority, while the second refers to the task's type. $comm_1$ and $comm_2$ can either be a *send* or a *receive* task. And, *comp* represents a computational task. Also, at this moment, there are a group of ready tasks assigned to P_i, which are included in the set of RN_{P_i}. However, only types other than $comm_1$ can be scheduled immediately. This part of RN_{P_i} belongs to the set of SN_{P_i}, which stands for *schedulable nodes.* The list scheduling algorithm only picks a task from SN_{P_i} with the highest priority, regardless any characteristics of tasks of type $comm_1$ in RN_{P_i}. Such an operation aims to fill $v_{i,comm_1}$'s gap. If we are unable to fill the gap, no task should be scheduled until $t = \sigma(v_{i,comm_1}) + g$. At

$t = \sigma(v_{i_1,comm_1}) + g$, tasks of $comm_1$ type can also be scheduled. In this case, tasks of all types are compared, in terms of their priorities. Furthermore, it is very important that the type of any tasks starting at $t = \sigma(v_{i_1,comm_1}) + g$ be $comm_1$, if the gap is left unfilled. Otherwise, the interval from $(\sigma(v_{i_1,comm_1}) + o)$ to $(\sigma(v_{i_1,comm_1}) + g)$ becomes another unnecessary idling time, if task of another type is scheduled at $t = \sigma(v_{i_1,comm_1}) + g$.

Whether the gap is filled or not, is determined as follows. At time $t = \sigma(v_{i_1,comm_1}) + o$ when a comm task is done, we update the set of ready nodes RN_{P_i} and the set of schedulable nodes SN_{P_i} on P_i. Select the node $v_{i,type_i}$ with the highest priority from SN_{P_i} and $v_{j,type_j}$ from RN_{P_i}. If $v_{i,type_i} = v_{j,type_j}$, that is $v_{i,type_i}$ and $v_{j,type_j}$ are exactly the same node, $v_{i,type_i}$ is scheduled at $t = \sigma(v_{i_1,comm_1}) + o$. Otherwise, node task is scheduled on P_i at this moment. More generally, at any moment when there is a gap on the processor and there is no task processing on that processor, only the task with the highest priority in both sets of ready nodes and schedulable nodes can be scheduled on the processor.

As a matter of fact, this approach is comparing schedulable tasks to tasks delayed by the gap. The priority of a task can be considered as the measurement of its urgency. The higher the priority is, the earlier the task should be scheduled. Therefore, it is reasonable to schedule the delayed task before other tasks if it has the highest priority. The finalized decoder is listed in detail by Algorithm 4.

The following example gives details on the decoding procedure. The same graph

Algorithm 4 List Scheduling under LogP ($g > o$)

```
Input:
    $G'(V', E', w', l')$: an extended graph of $G(V, E, w, l)$,
    $M = (L, o, g, P)$: a LogP instance,
    $Q(G') = \{q(T_i)|T_i \in V'\}$: a priority queue for tasks in $G$,
    $\Pi(G') = \{\pi(T_i)|T_i \in V'\}$: an processor assignment of $G$.
Output:
    $S$: a feasible schedule of $G'(G)$ under $M$.

    Begin
    {Associate $avt(i, j)$ to the available time of processor $P_i$ for tasks of $j$ where $j \in$ {COMP, SEND, RECV}, and $NM$ to the next decision moment .Denote the set of nodes that can be scheduled with $SN$, the set of ready nodes that are blocked by gaps with $RN$. Initially:$avt(i, j) = 0, \forall i, 0 \leqslant i < P$, and $\forall j \in$ {COMP, SEND, RECV};$SN = \emptyset$;$RN = \emptyset$; the current moment $CM := 0$;the set of unscheduled nodes $U := V'$.}
    while $U \neq \emptyset$ do
        for all $T \in U$ do
            if $0 \leqslant rt(T) \leqslant CM$ and $avt(\pi(T), \text{COMP}) \leqslant CM$ do
                if $avt(\pi(T), type(T)) \leqslant CM$ do
                    $SN := SN \cup \{T\}$;
                if $avt(\pi(T), type(T)) > CM$ do
                    $RN := RN \cup \{T\}$;
        while $SN \neq \emptyset$ do
            {Select a task $T_i$, such that $q(T_i) = \min_{T \in U} q(T)$;}
            if $avt(\pi(T_i), \text{SEND}) \neq avt(\pi(T_i), \text{RECV})$ do
                for all $T_j \in RN$ do
                    if $\pi(T_j) = \pi(T_i)$ and $q(T_j) > q(T_i)$ do
                        for all $T \in SN$ do
                            if $\pi(T) = \pi(T_i)$ do
                                $SN := SN/T$;
                                $rt(T) := avt(\pi(T_j), type(T_j))$;
                        Goto next inner while loop;
            $\sigma(T_i) := CM$;
            $U := U/T_i$;
            $rt(T_i) := -1$;
            for all $k \in$ {COMP, SEND, RECV} do
                {Update $avt(\pi(T), k)$};
            for all $T_j \in Succ_{G'}(T_i)$ do
                {Update $rt(T_j)$;}
            for all $T \in SN$ do
                if $\pi(T) = \pi(T_i)$ do
                    $SN := SN/T$;
        end while
        {Update $NM$};
        $CM := NM$;
    end while
    End
```

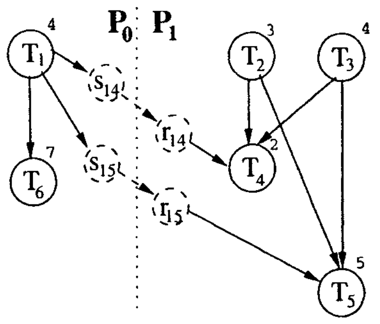


Figure 5.18: Extended Task Graph G_5' under LogP model $(1,1,2,2)$

G_5' (Figure 5.18) under the same model instance $(1,1,2,2)$. Assume there is a chromosome

$$X_g = \{T_1, T_4, T_2, r_{15}, s_{15}, T_3, s_{14}, T_5, T_6, r_{14}\}.$$

Figure 5.19 shows the whole scheduling procedure of G' based on X_g.

It is clear that gaps are taken into account during the scheduling procedure. And, gap fillings at various positions are determined by the genes.

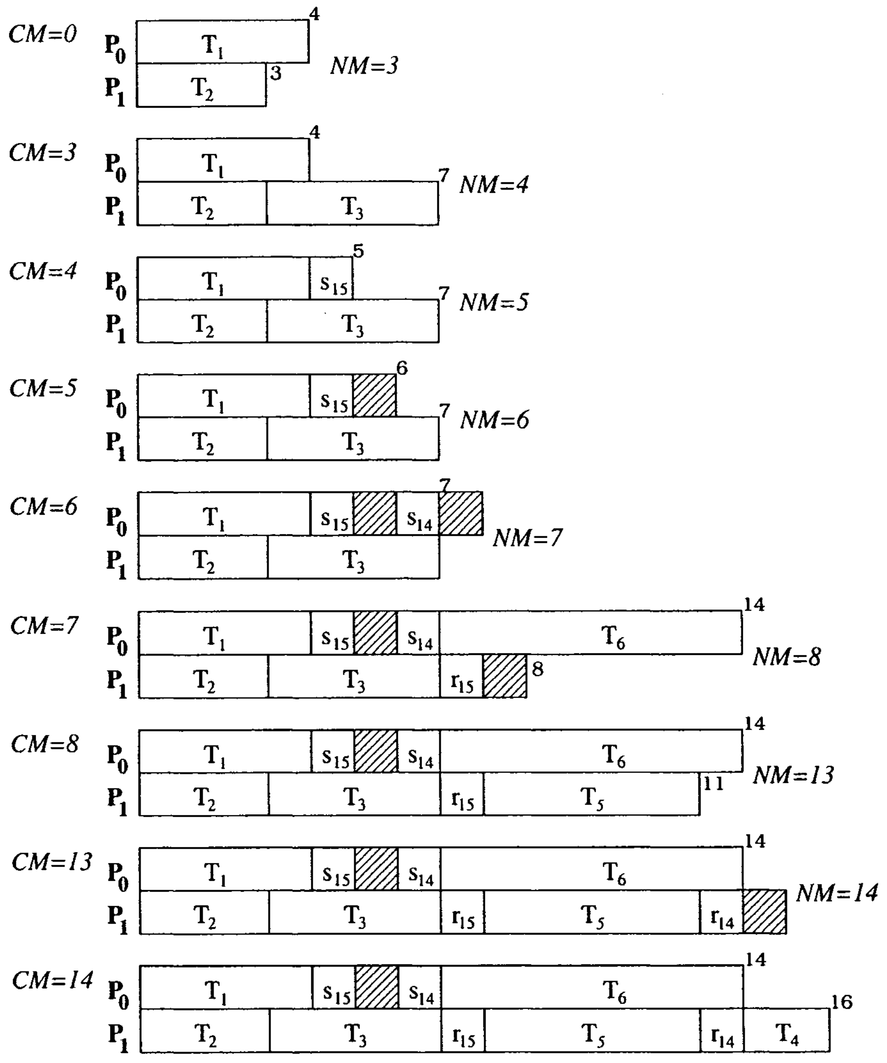


Figure 5.19: Decoding X_g

Chapter 6

Processor Allocation Using GA

In previous chapters, we have developed a new way to schedule communications under the LogP model. The GA based method is designed only for situations where processor allocation is known. The allocation must be generated by some other heuristics before our approach can be applied. Some of these heuristics are described in the following section.

6.1 Early-Task-First Strategies

There are two list scheduling algorithms for the LogP model introduced in [27]. One is the two-pass ETF heuristic (i.e. 2ETF) and the other is ETF with reservation heuristic (i.e. ETFR). As schedules generated by ETFR always contain unnecessary idling slots, a garbage collector is run to remove these slots. ETFR with garbage collector is called ETFRGC in [27]. Although ETFRGC produces more efficient schedules than ETFR, there is no conceptional distinction between the two algorithms.

Therefore, ETFR will refer to ETFR with garbage collector from now on.

Both 2ETF and ETFR are extensions of the Early-Task-First (ETF) Algorithm [25], which is a typical list scheduling algorithm for scheduling under SDM. ETF schedules a task as soon as possible. Meanwhile, it utilizes as many processors as possible. Thus, there are two sets involved: the set of available tasks and the set of free processors. Usually, tasks which become available earlier are scheduled earlier. Algorithm 5 provides a detailed description of this approach.

The ready time of T is denoted by $rt(T)$, which is the latest completion time of T's predecessors. $st(T, P)$ denotes the earliest starting time of task T on processor P. Due to communication requirements, $st(T, P)$ may vary from processor to processor. It is determined not only by T's ready time and P's completion time, but also by the positions of T's predecessors. A task can start only after all its predecessors have terminated. That is,

$$rt(T) \leqslant CM.$$

Such a task is called a *ready task.* A processor is considered free at a given moment only if there are no tasks being processed on that processor at that moment.

$$ct(P) \leqslant CM.$$

Obviously, ETF is always trying to fill all free processors with ready tasks at every moment. From the point of view of tasks, ETF assigns task to the processor which would allow the task to start earliest.

Algorithm 5 Earlier Task First under SDM

```
Input:
    $G(V, E, w, l)$: a precedence graph,
    $M$: an SDM instance.
Output:
    $S$: a feasible schedule of $G$ under $M$.

    Begin
    {Associate $ct(P_i)$ to the completion time of processor $P_i$. Denote the set of
    available nodes with $RN$ and the set of free processor with $FP$.
    Initially:$RN = \emptyset$ $ct(i) = 0, \forall i, 0 \leqslant i < P$, the current moment $CM := 0$ and the
    set of unscheduled nodes $U := V'$.}
    while $U \neq \emptyset$ do
        for all $T \in U$ do
            if $0 \leqslant rt(T) \leqslant CM$ and $T' \in \overline{U}, \forall T' \in Pred_{G'}(T)$ do
                $RN := RN \cup \{T\}$;
            end if
        end for
        while $RN \neq \emptyset$ and $FP \neq \emptyset$ do
            {Select a task $T_i$ and a processor $P_j$
            s.t. $st(T_i, P_j) = \min_{T \in RN} \min_{P \in FP} st(T, P)$;}
            $\pi(T_i) := P_j$;
            $\sigma(T_i) := CM$;
            $rt(T_i) := -1$;
            $ct(j) := CM + w(T_i)$;
            $RN := RN/T_i$;
            $U := U/T_i$;
            $FP := FP/P_j$;
            for all $T_j \in Succ_{G'}(T_i)$ do
                {Update $rt(T_j)$;}
            end for
        end while
        {update $CM$};
        $FP := \{P | ct(P) \leqslant CM\}$;
    end while
    End
```

Both 2ETF and ETFR are composed of two scheduling phases: processor allocation and communication embedding. The two algorithms applied different strategies in the first phase.

As a relatively simple extension to ETF, 2ETF simplifies the LogP model and transfers it to an alternative SDM instance. Communication overheads are considered as a part of communication costs but they do not occupy processors. Thus, the cost of a communication in the SDM model is set to be $2o + L$.

ETFR emphasizes communications during the first phase. It is always assumed that send/receive operations are necessary between a task and any of its successors. According to the LogP model, whether a message needs to be transmitted or not is determined only by the processor assignments of the origin task and the destination task. If these tasks have been allocated to the same processor, there is no need to transmit a message. However, ETFR reserves idle slots for communication tasks before the processor assignment is known. These slots are always large enough for any potential overheads. It is ensured that interval between a task T_i and the immediate following task on the same processor is longer than $n_{succ}(T_i) \times o$, where $n_{succ}(T_i)$ refers to the number of T_i's successors.

As for the communications, the ETF extensions apply a simple scheme, which assumes that each computational task is preceded by a sequence of incoming communications, and followed by outgoing communications. Both algorithms arrange sending tasks from the same origin sequentially without any interruption. Also, communication operations that are receiving messages for the same destination computational task are scheduled sequentially.

The original 2ETF and ETFR algorithm presented in [27] can only deal with models that satisfy $g = o$. We extend them so that they are able to handle significant gaps. For example, Algorithm 6 lists the procedure of improved ETFR, which is suitable for the situation where $g > o$.

2ETF and ETFR can be considered as two different strategies of computational task arrangement, as they handle communications in the simplest way, which places communication tasks between their origins and successors. ETFR produces better schedules in more cases than 2ETF. But, whether ETFR is superior to 2ETF is dependent on the specific nature of the task graphs.

Although, appropriately inserting communications between (allocated) computational tasks may lead to a significant reduction in the overall makespan, processor assignment of computational tasks is a significant factor that should be taken into account. Therefore, we are going to seek a better way of allocating processors.

6.2 Problem Description

The processor assignment should lead us to the shortest makespan schedule under a specific instance of the LogP model.

The computer program is described using a directed acyclic graph $G(V, E, w, l)$. $w : V(G) \rightarrow \mathbb{Z}^+$ is a function that assigns an execution length to each task of G and $l : E(G) \rightarrow \mathbb{N}$ is a function that specifies the number of messages needed to send through each edge of G.

A *processor assignment* Π for a problem instance (G, M) is a function π, such

Algorithm 6 ETFR with $g > o$

Input:
    $G(V, E, w, l)$: a task graph,
    $M = (L, o, g, P)$: a LogP instance,
Output:
    S: a feasible schedule of G under M.

```
    Begin
    {Associate $ct(i)$ to the completion time , $ct_s(i)$ to the available time of send tasks,
    $ct_r(i)$ to the available time of recv tasks of processor $P_i (0 \leqslant i < P)$.
    Initially: $ct(i) = 0, ct_s(i) = 0, ct_r(i) = 0, \forall i, 0 \leqslant i < P$, }
    while $SN \neq V$ do
        while $RN \neq \emptyset$ and $FP \neq \emptyset$ do
            {Compute $e_s(T, i)$ for every $T \in RN$ and for every processor $i \in FP$};
            {Select a task $\hat{T}$ and a processor $\hat{i}$ such that $\hat{T} \in RN$, $\hat{i} \in FP$
            and $\hat{e_s}(\hat{T}, \hat{i}) = \min_{T \in RN} \min_{i \in FP} e_s(T, i)$ };
            if $\hat{e_s}(\hat{T}, \hat{i}) \leqslant NM$ then
                for each $T' \in Pred(\hat{T})$ such that $\pi_S(T') \neq \hat{i}$ do
                    $\pi_S(send(T', \hat{T})) := \pi_S(T')$;
                    $\sigma_S(send(T', \hat{T})) := \max\{NSM(T'), ct_s(\pi_S(T'))\}$;
                    $ct_s(\pi_S(T')) := \sigma_S(send(T', \hat{T})) + g$;
                    $\pi_S(recv(T', \hat{T})) := \pi_S(\hat{T})$;
                    $\sigma_S(recv(T', \hat{T})) :=$
                    $\max\{ct(\hat{i}), ct_r(\pi_S(\hat{T})), \sigma_S(send(T', \hat{T})) + L + o\}$;
                    $ct_r(\pi_S(T')) := \sigma_S(recv(T', \hat{T})) + g$;
                    $ct(\pi_S(T')) := \sigma_S(recv(T', \hat{T})) + o$;
                end for
                $\pi_S(\hat{T}) := \hat{i}$;
                $\sigma_S(\hat{T}) := \hat{e_s}$;
                $NSM(\hat{T}) := \sigma_S(\hat{T}) + w(\hat{T})$;
                {Let $reserved = \sum_{T'' \in Succ(\hat{T})} g$};
                $ct(\hat{i}) := \sigma_S(\hat{T}) + w(\hat{T}) + reserved$;
                $RN := RN/\hat{T}$;
                $FP := FP/\hat{i}$;
                $SN := SN \cup \{\hat{T}\}$;
                if $ct(\hat{i}) < NM$ then
                    $NM := ct(\hat{i})$;
                end if
            else
                {exit the innermost while loop}.
            end if
        end while
        {Update $CM$,$NM$, $RN$,$FP$};
    end while
    End
```

that

$$\pi : V(G) \to \{1, 2, \ldots, P\} \cup \{\perp\}.$$

π assigns a processor to the tasks in V. The value $\perp$ denotes the task has not been allocated to any processor yet. A processor assignment is called *valid* if

$$\pi(v) \neq \perp, \forall v \in V.$$

That is, each task has been assigned to a processor. The assignments involved in our problem must be *valid.*

Any feasible schedule of task graph G under a model M corresponds to exactly one processor assignment. But, more than one schedule may have the same processor assignment. The set of all the feasible schedules with the same processor assignment Π is denoted with $\mathbb{S}_\pi$. Then, there exists at least one schedule $S_{min}(\Pi)$ such that

$$\forall S \in \mathbb{S}_\pi, C_{max}(S_{min}(\Pi)) \leqslant C_{max}(S).$$

We define S_{min}'s makespan $C_{max}(S_{min}(\Pi))$ as the *appropriateness* of processor assignment Π, which is represented by $app(\Pi)$.

Accordingly, the objective is to find a processor assignment $\Pi(G)$ with the minimum appropriateness $app(\Pi)$.

6.3 Processor Assignment using GA

6.3.1 Algorithm Outline

The GA-based algorithm described here is essentially the same as the GA in Chapter 4, except for the encoding and the genetic operators. Figure 6.1 shows an outline of the algorithm.

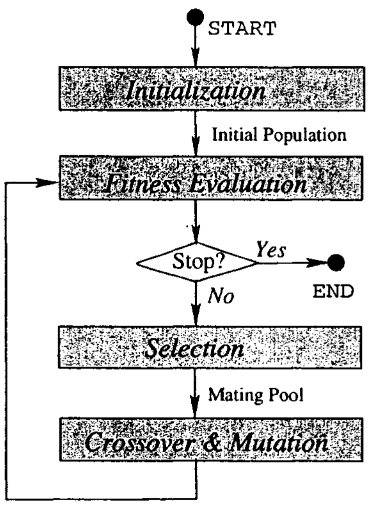


Figure 6.1: Algorithm Flow Chart

This GA also starts with an initial population made up of randomly generated individuals, in addition to some specific chromosomes. In our case, two specific chromosomes correspond to processor assignments generated by 2ETF and ETFR respectively. The size of the population is fixed. Due to the complexity of the decoder, we use a relatively small population. There are 40 chromosomes in each generation, 38 of which are generated at random. For each random initial individual, each computational task in the graph is randomly allocated to a processor.

Elitism is also used. The best individual in a generation enters the next generation without alteration. 39 (potentially repeating) individuals are chosen using fitness proportional selection from the current generation, and placed in a mating pool. The mating pool is subjected to crossover and mutation. As a result, a group of 39 chromosomes are generated. The new population is composed of these 39 chromosomes plus the single elite individual. The evolutionary run is halted after 50 generations.

6.3.2 Encoding

Similar to any other GA design, the first step is to invent a way of presenting a candidate solution of the problem, using a chromosome. It is important that the encoding contains building blocks that are carefully matched with appropriate crossover and mutation operators.

As for the processor allocation problem, real-valued genes are used to encode a solution directly. Each gene corresponds to the processor assignment for a given task. Note that the subject of this problem are the *original* graphs, which have not been extended yet. Only the assignment of computational tasks are encoded. Therefore, all the chromosomes have the same (fixed) length, which is equal to the number of computational tasks in the corresponding task graph.

For example, when scheduling task graph G in Figure 6.2 on 2 processors, P_0 and P_1, the chromosome

$$C = \{(T_1, 0), (T_2, 0), (T_3, 1), (T_4, 0), (T_5, 1), (T_6, 0), (T_7, 0), (T_8, 1)\}$$

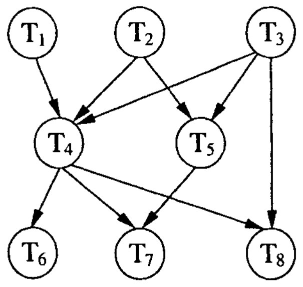


Figure 6.2: Task Graph G

represents a processor assignment, where tasks T_1, T_2, T_3, T_6 and T_7 are assigned to processor P_0, with the others placed on P_1.

Information about tasks may be removed from the chromosome to simplify the representation. The genes only keep processor numbers, with each gene corresponding to exactly one task. Such gene-task correspondence is decided according to certain sorting rules. In our implementation, it is determined by the topological sorting used by LEDA [40]. For instance, according to LEDA [40], the nodes in G in Figure 6.2 should be sorted in this sequence:

$$T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8.$$

Then, the chromosome C can be simplified to

$$C' = \{0, 0, 1, 0, 1, 0, 0, 1\}.$$

In short, the processor assignment is encoded as a string of integers, each of which represents the number of the processor that the corresponding task is assigned to.

6.3.3 Decoder

Given a certain processor assignment, looking for the optimal schedule is an NP-hard problem. Hence, it is necessary to evaluate processor assignments in a manner which would keep the computational complexity of the algorithm as low as possible.

We adopt the task arrangement method used in 2ETF/ETFR. The decoding procedure is also divided into two phases: computational task arrangement and communication embedding.

The first phase still follows the Earlier-Task-First logic as in Algorithm 5. The difference is that there is no need to calculate the ready time of one task for each free processor any more. Instead, each task is always bound to one processor. Note that during the first phase, a standard delay model is applied, so as to attain a lower complexity. The procedure of *Earlier Task First with known Processor Assignment (or ETFPA)* is described in Algorithm 7.

As for communications, we apply the strategy used in 2ETF, which is named algorithm LogP-Feasible-Schedule by Kalinowski *et al.* [27]. In order to form a feasible schedule under LogP, communication tasks are inserted according to the following rules:

- A *send* task should be on the same processor as its source computational task, while a *recv* task should be on the same processor as its destination task.
- At first, *send* tasks from the same source computational task must be scheduled sequentially.

Algorithm 7 Earlier Task First with known Processor Assignments

```
Input:
    $G(V, E, w, l)$: a precedence graph,
    $\Pi(G) = \{\pi(T_i) | \forall T_i \in V\}$: a processor assignment of all nodes in $G$,
    $M$: an SDM instance.
Output:
    $S$: a feasible schedule of $G$ under $M$.

    Begin
    {Associate $ct(P_i)$ to the completion time of processor $P_i$. Denote the set of
    available nodes with $RN$ and the set of free processor with $FP$.
    Initially:$RN = \emptyset$ $ct(i) = 0, \forall i, 0 \leqslant i < P$, the current moment $CM := 0$ and the
    set of unscheduled nodes $U := V'$.}
    while $U \neq \emptyset$ do
        for all $T \in U$ do
            if $0 \leqslant rt(T) \leqslant CM$ and $T' \in \overline{U}, \forall T' \in Pred_{G'}(T)$ do
                $RN := RN \cup \{T\}$;
            end if
        end for
        while $RN \neq \emptyset$ and $FP \neq \emptyset$ do
            {Select a task $T_i$ and a processor $P_j$
            s.t. $st(T_i, P_j) = \min_{T \in RN} \min_{P \in FP} st(T, P)$;}
            $\pi(T_i) := P_j$;
            $\sigma(T_i) := CM$;
            $rt(T_i) := -1$;
            $ct(j) := CM + w(T_i)$;
            $RN := RN/T_i$;
            $U := U/T_i$;
            $FP := FP/P_j$;
            for all $T_j \in Succ_{G'}(T_i)$ do
                {Update $rt(T_j)$;}
            end for
        end while
        {update $CM$};
        $FP := \{P | ct(P) \leqslant CM\}$;
    end while
    End
```

- In the same manner, *recv* tasks corresponding to the same destination computational task must be grouped together; no other task may be inserted into such a group.

- *send* tasks are handled *immediately* after their origin computational task is finished. There should not be any delay between the computational task and the first message transmission from that task.

- A computational task must start *immediately* after all tasks that receive input data for it have been done.

- *send* tasks from the same source computational task are sorted using a topological sorting routine, in the order of their destination tasks; this topological sort is fixed for the whole procedure.

- Likewise, *recv* tasks in the same group are sorted according to their sources.

- According to the definition of the LogP model, a message is received at least L time units after it has been submitted to the network. That is, the interval between a pair of send and recv operations must be longer than L.

6.3.4 Crossover

Crossover is a procedure that randomly selects two chromosomes from the mating pool and then recombines parts of them to produce two new chromosomes. The goal is to mix genetic information of fit individuals, in order to occasionally produce new offspring that join the best features of their parents, and hence exceed their fitness.

Algorithm 8 LogP Feasible Schedule with $g > o$

```
Input:
    $G(V, E, w, l)$: a task graph,
    $M = (L, o, g, P)$: a LogP instance,
    $S$: a schedule of $G$ under SDM.
Output:
    $S'$: a feasible schedule of $G$ under $M$.

    Begin
    {Associate $ct(i)$ to the completion time of processor $P_i$, $ct_s(i)$ to the available
    time of send task of processor $P_i$, $ct_r(i)$ to the available time of recv task of
    processor $P_i (0 \leqslant i < P)$.
    Initially: $ct(i) = 0, ct_s(i) = 0, ct_r(i) = 0, \forall i, 0 \leqslant i < P$, S.CurrentTasks(C);}
    while $C \neq \emptyset$ do
        for each $T_j \in C$ do
            $\pi_{S'}(T_j) := \pi_S(T_j)$;
            for each $T_k \in Pred(T_j)$ do
                if $\pi_S(T_k) \neq \pi_S(T_j)$ do
                    $i = \pi_{S'}(T_j)\ \pi_{S'}(recv(T_k, T_j)) := i$;
                    $\sigma_{S'}(recv(T_k, T_j)) := \max\{ct(i), \sigma(send(T_k, T_j)) + o + L, ct_r(i)\}$;
                    $ct(i) := \sigma_{S'}(recv(T_k, T_j)) + o$;
                    $ct_r(i) := \sigma_{S'}(recv(T_k, T_j)) + g$;
                end if
            end for $\sigma_{S'}(T_j) := ct(\pi_{S'}(T_j))$;
            $ct(i) := \sigma_{S'}(T_j) + w(T_j)$;
            for each $T_k \in Succ(T_j)$ do
                if $\pi_S(T_k) \neq \pi_S(T_j)$ do
                    $\pi_{S'}(send(T_j, T_k)) := i$;
                    $\sigma_{S'}(send(T_j, T_k) := \max\{ct(i), ct_s(i)\}$;
                    $ct(i) := \sigma_{S'}(send(T_j, T_k) + o$;
                    $ct_s(i) := \sigma_{S'}(send(T_k, T_j)) + g$;
                end if
            end for
        end for
        S.CurrentTasks(C);
    end while
    End
```

Uniform crossover operator [48] is used in our GA. Two parents are selected from the current generation to generate two new individuals. Crossover points are not fixed in advance. They are determined probabilistically at each gene position. Let p_x be the crossover probability at each gene position, and N be the number of genes in a chromosome. The recombination operation can be described as follows:

```
UniformCrossover($a, b$)
    $c \leftarrow a$;
    for $i \leftarrow 0$ to $N$
        do sample $u \in U(0,1)$;
            if $u > p_x$
                do $c_i \leftarrow b_i$;
    return $c$;
```

This procedure generates only one individual from two chromosomes. So, it must be run twice for a pair of offspring.

According to Ackley [16], p_x is set to 0.5. When $p_x = 0.5$, both parents have the same chance of passing on their genetic material to their offspring. In this case, the uniform crossover operation can be simplified, by constructing a mask. Each bit of the mask determines the parent that the child should receive the corresponding gene from. The crossover probability (p_c) is set to 0.85.

The following example shows how the crossover operator works for a problem instance of scheduling G (in Figure 6.3), under a LogP model with $P = 4$. G is composed of 8 nodes. Hence, chromosomes in the evolution are strings of 8 integers.

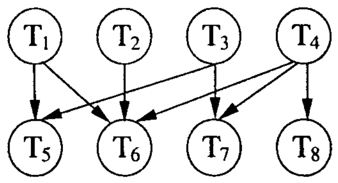


Figure 6.3: Task Graph G

And, each gene should be a integer not less than 0 and not greater than 3. The sequence of tasks resulting from a default topological sorting routine, applied to G is: $(T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8)$.

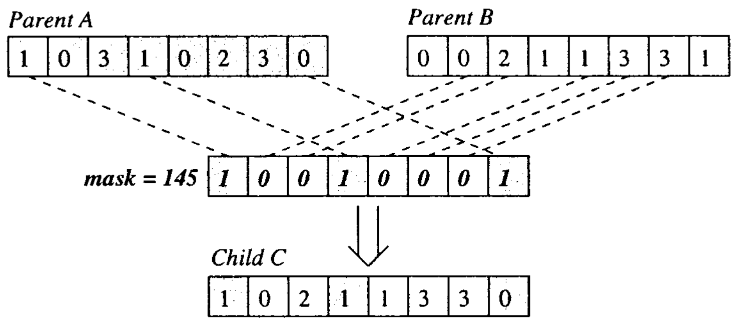


Figure 6.4: Uniform Crossover

Suppose there are two chromosomes A and B involved in the crossover operation illustrated in Figure 6.4. As there are 8 nodes in the graph, the upper bound of the crossover mask is $2^8 - 1 = 256$. Thus, the first step is to generate an integer between 0 and 255, which is 145 in this example. This integer is converted to its binary equivalent 10010001, which is a 1-dimensional vector of bits. The vector has the same length as a chromosome. The value of a vector bit determines the source of the corresponding bit in the child chromosome. Specifically, 1 in the first bit of the vector means that the first gene of the child chromosome is copied from the first gene of parent A. In the same manner, the second bit of the child is taken from B, as the second bit of the vector is 0.

6.3.5 Mutation

Mutation is relatively simpler than crossover, but certainly not of negligible significance. Evolution with only mutation and selection but no crossover can still be effective, though crossover accelerates the search process. The objective of mutation is to maintain genetic diversity lost during crossover and selection, both of which place relentless pressure on the population to converge to an ever-shrinking pool of genes. Individuals are only changed slightly in mutation. This means the processor assignments are mildly perturbed, with some small probability.

Our GA has employed two types of mutation operators: 1-bit mutation and swap mutation. 1-bit mutation is implemented by selecting a gene at random and changing its value randomly as well. The new value of the mutated gene usually differs from its original setting. On the other hand, swap mutation is performed on two genes instead of one. It randomly selects a pair of genes and then swaps them. It is important that the selected pair of genes have different values or else the swap will be useless. These two operators are applied alternately during mutations. The mutation rate p_m is 0.1.

From the programmatic point of view, 1-bit mutation moves a task from one processor to another so that the search may reach a new point. On the other hand, new solutions can also be introduced by switching two tasks on different processor, which is achieved by swap mutation. Although both the mutation operators can bring diversity to the population, they accomplish it in different ways. 1-bit mutation maintains the positions of all the tasks but one, while swap mutation maintains the numbers of tasks on both each processor. The search may be led to distinct directions by these different mutation operators. There are more chances of exploring a wider space of valid schedules, if both mutations are applied, which is the case here.

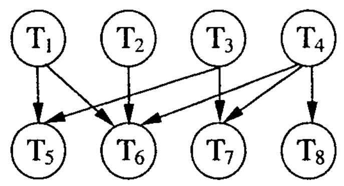


Figure 6.5: Task Graph G

The mutation example uses G of Figure 6.5. Generally, mutations are performed in two steps. One is to select the mutating position(s), and the other is to perform the modification at the genes. Figure 6.6 shows how 1-bit mutation works on chromosome

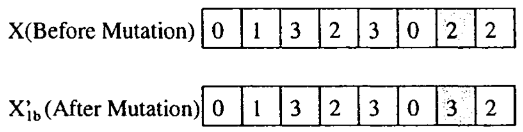

X(Before Mutation) 0 1 3 2 3 0 2 2
X'1b (After Mutation) 0 1 3 2 3 0 3 2

Figure 6.6: 1-bit Mutation

X. The 7th gene x_7 is selected for mutation. To ensure this gene is modified to a new value, we apply:

$$x_i' = (x_i + (r \bmod P)) \bmod P,$$

where r is an integer generated randomly. In this case, the possibility of all new values are equal. Assume that $r = 1$ in this example. This result in a new value of $(2 + (1 \bmod 4)) \bmod 4 = 3$. And X'_{1b} is the result of the 1-bit mutation.

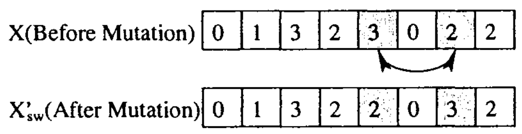

X(Before Mutation) 0 1 3 2 3 0 2 2
X'sw(After Mutation) 0 1 3 2 2 0 3 2

Figure 6.7: Swap

As for swap mutation, the first step is still to select mutating genes. The selected genes must have different values, for example, the 5^{th} and the 7^{th} genes of X (see Figure 6.7). Afterward, the two genes are swapped resulting in X'_{sw}.

Processor assignments represented by X, X'_{1b} and X'_{sw} are illustrated in Figure 6.8. 1-bit mutation (see Figure 6.8(b)) has moved task T_7 from processor P_2 to P_3. On

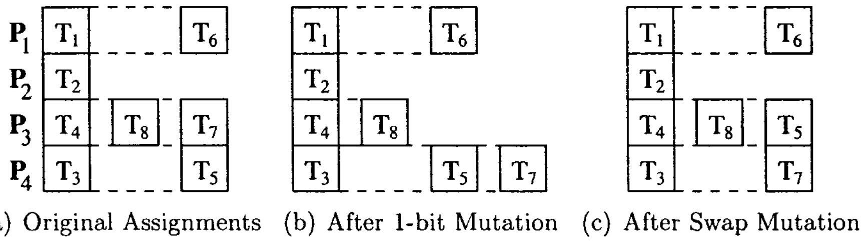


Figure 6.8: Processor Assignments before/after Mutation

the other hand, swap mutation switches T_5 and T_7, as shown in Figure 6.8(c).

6.4 Discussion

In this chapter, a novel genetic algorithm is proposed for the processor allocation problem under LogP models. The concept of appropriateness is introduced as a measure for the evaluation of processor assignments. The algorithm has been discussed in detail.

The algorithm is based on the SGA. The potential solutions are encoded using positive integer strings that represent the processor assignments of the computational tasks. The encoding methodology used in this algorithm assures that a coded string only represents one solution. The decoder uses a list scheduling algorithm to build a

LogP-feasible schedule. As for the genetic operators, uniform crossover, 1-bit mutation and swap mutation are applied.

Still, this algorithm has similar drawbacks to the communication management algorithm (Section 4.5). As the problem is on homogeneous multiprocessor systems, all the processors are identical. The encoding method gives each processor a number which makes the processor unique. Consequently, different strings may correspond to the same solution. For instance, there is no difference between assigning all the tasks to processor P_1 and assigning them to processor P_2 as P_1 and P_2 are identical. The representation space is $P!$ times larger than the solution space, where P is the number of the processors.

Chapter 7

Experiments

7.1 Benchmark Graphs

7.1.1 Characteristics of Task Graphs

The objective of the experiments is to compare the new algorithms to existing heuristics under various conditions that scheduling algorithm may encounter. Therefore, it is necessary to classify DAGs by certain characteristics so as to evaluate the performance more efficiently. A DAG is presented by $G(V, E, w, l)$. The following characteristics are representative and are widely used by researchers.

General Characteristics When a graph's size is concerned, there are a group of characteristics involved. Note, graphs here refers to the original task graphs without any extensions.

- *Number of Nodes*: Total number of nodes in a graph is the mostly used measurement for a graph's size. It is denoted with N_v.

- *Number of Edges*: N_e refers to the number of edges in the graph.

- *Average Node Weight*: N_w

$$N_w = \frac{\sum_{v \in V(G)} w(v)}{N_v}.$$

- *Height*: A layering of a DAG is a partition of its node set into subsets, called layers. The height of a graph is the number H of layers.

- *Width*: The width W of a graph is defined as the largest number of non-precedence-related nodes in the DAG.

- *Sequential Time*: If the program is executed on a single processor, the makespan would be the sum of costs of all computational tasks. Such a sum is called *sequential time* t_{seq}. It is sometimes used as a baseline for experiments in parallel scheduling algorithms.

$$t_{seq}(G) = \sum_{v \in V(G)} w(v).$$

Anchor Out-degree The number of out-going edges from a node is called the node's *out-degree*. [28] defines *anchor out-degree*, also called *anchor*, as the mode of out-degrees of the nodes in the graph. Anchor reflects the branching factor of a program, one of the factors that affect the graph's complexity.

Denote the out-degree of node v in a graph G with $d_{out}(v)$. Then, the anchor of G is

$$A_{out}(G) = \frac{\sum_{v \in V(G)} d_{out}(v)}{N_v}, \tag{7.1}$$

where N_v is the number of the nodes in G. The anchor here is equal to the average number of successors of all the nodes.

Critical Path The critical path is the longest path in a directed acyclic graph. Formally, a *chain* in a DAG is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that there is a directed edge from v_1 to v_2, a directed edge from v_2 to v_3 and so on, up to and including a directed edge from v_{n-1} to v_n. The longest chain in a DAG is called the *critical path.*

For a weighted DAG, *critical path* is usually defined to be the path from a source node to a sink node with the greatest weight. The weight of an edge is not considered in critical path analysis under LogP. Therefore, *path weight* refers to the sum of the weights of all the nodes on the path. Its length is

$$t_{CP}(G) = \sum_{v \in CP} w(v).$$

Parallelism Kwok [32] defines parallelism as a parameter determined by the width of the graph. A parallelism of 1 means the average width of the graph is $\sqrt{N_v}$, a value of 2 means the graph has an average width of $2\sqrt{N_v}$; and so on. Accordingly, the parallelism of a graph is

$$\rho(G) = \frac{W(G)}{\sqrt{N_v}}. \tag{7.2}$$

7.1.2 Sample Graphs

Specific groups of graphs are used in our experiments. Brief descriptions of these graphs follow.

Small-Scale Random Graphs (SSG)

Small-scale random graphs (SSG) are randomly generated and small in size. They are made up of 14 graphs. The number of nodes in these graphs varies from 5 to 70 with increments of 5. The average weight of a node in each graph is approximately 10, which is relatively small. Detailed characteristics of the graphs are listed in Table A.1

Random Graphs with 20 Layers (R20L)

These are random graphs we generated according to the experiments described by Kalinowski *et al.* [27]. They have the same features, such as anchors and average weights, as the graphs used in [27], but not exactly the same sets. Each of the graphs has 20 layers, thus, this group is called *Random Graphs with* 20 *Layers* (R20L). This suite of random graphs consists of two subsets with different parameters. In the first subset (Graph1-10), the graphs' anchors are close to 2, while the second subset (Graph11-20) has an average anchor out-degree of approximately 8. Average execution time for the first class of graphs is approximately 30, compared 90 for the second class. In addition, both nodes and edges are generated uniformly.

Random Graphs without Optimal Schedules (RGNOS) [32]

The fourth set of graphs, referred to as random graphs with no known optimal solutions (RGNOS), are a part of benchmark graphs generated by Kwok [32]. Kwok's RGNOS consists of 5 sets of graphs with different *communication-to-computation ratios* (CCR). The communication costs under the LogP model are not determined by CCR but by the model. Hence, we only select one set of 50 randomly generated graphs from Kwok's RGNOS for the experiments. These graphs are much larger than SSG, with from 50 nodes to 500 nodes each, with increments of 50. Parallelism varies among the 50 graphs. Five different values of parallelism were chosen: 1, 2, 3, 4, and 5.

Traced Graphs [32]

The last set of test graphs is called *traced graphs*, which simulate some parallel programs obtained via a parallelizing compiler. We only randomly select one graph from Kwok's traced graphs. The graph is used to verify reliability of our algorithms.

7.2 Models

The algorithms we have proposed are general schedulers for scheduling problems under LogP. Therefore, it is necessary to experiment under various circumstances to evaluate the algorithms thoroughly. However, it is impossible to enumerate every instance. The best way is to classify the instances, then select typical examples of each class for the testing.

We first consider a group of models that satisfy $g = o$. Consequently, models belonging to this class are selected for experiments. This group is listed in Table 7.1. In these model instances, two values of P were considered: $P = 4$ and $P = 8$. When

Model	L	o	g	P
M_1	10	1	1	4
M_2	10	10	10	4
M_3	1	10	10	4
M_4	10	1	1	8
M_5	10	10	10	8
M_6	1	10	10	8

Table 7.1: Models with $g = o$

$P = 8$, there may be more communications to be handled than $P = 4$. In addition, we consider 3 pairs of values for L and o (g is ignored): $(1, 10)$, $(10, 1)$ and $(10, 10)$. These pairs correspond to different ratios between L and o. When $L = 1$ and $o = 10$, communication overheads are much more significant than latency. On the other hand, latency dominates network communication costs if $L = 10$ and $o = 1$. Finally, $(10, 10)$ presents cases where values of latency and overhead are similar. In this case, latency and overhead are both important to the overall communication costs.

Since models listed in Table 7.1 have already covered various possible comparisons between L and o, we only consider the condition where $L < o$ in the experiments that handle significant gaps. Therefore, all the LogP model instances used here have the same values for the latency and the overheads: $L = 5$ and $o = 10$. There are two groups of models, both using has the same number of processors.

The value of the gap varies in the 6 models suited to the condition with distinct gaps. In addition, the value is determined by characteristics of graphs used in this experiments.

It has already been established in Chapter 5 that we are exploring every possibility of *gap filling*. Whether filling is a good strategy or not, depends on the costs of the computational tasks, as well as the size of the slots to be filled. By that we mean the difference between the gap and the overhead ($g - o$). We choose the first class of graphs from the $R20L$ graph set to evaluate the algorithm when $g > o$. The average weight N_w for every one of these graphs is 30 time units. There are three possible relationships between N_w and $g - o$ reflected in the experiments: $g - o \ll N_w$, $g - o = N_w$ and $g - o \gg N_w$. Table 7.2 shows the three values of gaps: 15, 40 and 80.

Model	L	o	g	P
M_1'	5	10	80	4
M_2'	5	10	40	4
M_3'	5	10	15	4
M_4'	5	10	80	8
M_5'	5	10	40	8
M_6'	5	10	15	8

Table 7.2: Models with $g > o$

Finally, when applying ETF algorithm under the SDM model, the communication cost t_c is assumed to be $2 \times o + L$.

7.3 Performance Measures

7.3.1 Makespan

As it has been emphasized several times in previous chapters, the objective is to find a LogP-feasible schedule with minimum makespan. Obviously, makespan is the most important measure used in our experiments. Formally, the makespan of schedule S is denoted by $C_{max}(S)$, which is

$$C_{max}(S) = \max_{v \in V(G)} (\sigma_S(v) + w(v)). \tag{7.3}$$

7.3.2 Speedup

The specific characteristics of a given graph have directly impact on the makespan of the optimal schedule for that graph. To eliminate such effects, we also use another measure to evaluate an algorithm: *speedup*.

In parallel computing, speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. Speedup of algorithm A is defined by the following formula:

$$\delta_A = \frac{t_{seq}}{C_{max}(S_A)}, \tag{7.4}$$

where t_{seq} is the sequential time of the graph, while C_{S_A} refers to the makespan of the schedule S_A obtained by algorithm A. A desirable parallel algorithm should have speedups greater than 1 in all cases.

7.3.3 Pairwise Comparison

It is impossible to set an absolute standard of speedup for parallel algorithms. As a matter of fact, a *Speedup* > 1 only indicates that the algorithm is superior to the serial algorithm that schedules all tasks on the same processor. This is obviously not a comprehensive measure of algorithm performance.

As a result, the new algorithms are also compared to some existing algorithms. In a pairwise comparison, the improvement achieved by a new algorithm A over an existing baseline algorithm B is denoted by α_{AB}, which is defined as

$$\alpha_{A,B} = \frac{C_{max}(S_B) - C_{max}(S_A)}{C_{max}(S_B)} \times 100\%. \tag{7.5}$$

$\alpha_{A,B}$ is expected to be positive for any improved algorithm A.

7.4 Methodology

7.4.1 Implementations

All the algorithms have been implemented in C++ and were run under both Windows and Linux. The program uses graph data structures from LEDA [40]. Table 7.3 provides further details of OS platform, compiler type and libraries used.

Platform	Compiler	Utility Library
Windows XP	Microsoft Visual C++ 6.0	LEDA-4.4-windows
Fedora Core 4 i386	gcc 2.95.3	LEDA-4.2-linux

Table 7.3: Implementation Environments

The program accepts 3 inputs: a task graph, a model instance and an algorithm name. It prints the makespan of the final schedule and the schedule itself of the given graph under the given model produced by the requested algorithm. When running a GA-based algorithm, the average makespan and the best makespan in each generation are recorded. Scripts are used to execute the programs on different problem sets, without having to go back to the experimenter.

As most other GAs, our own GAs use random functions. Hence, the algorithm can not guarantee to return the same result every time. In other words, result from a single run of the algorithm can hardly reflect the algorithm's performance. Thus, we use 50 runs for *each case*. The average of the results of all 50 runs is considered to be the final makespan of the algorithm in the corresponding case. The speedups and the improvements are calculated based on such makespans.

7.4.2 Baseline Heuristics

As mentioned in Section 7.3.3, pairwise comparison is the most comprehensive way to evaluate our algorithms. There is not yet a standard benchmark for scheduling problems under the LogP model. The new algorithms are compared to some existing algorithms, which are considered, from now on as the benchmarks for our own tests.

Both 2ETF and ETFR [27] are selected as the baselines in the experiments as they are both algorithms for general scheduling problems under LogP. The communication scheduling algorithms described in Chapter 4 and Chapter 5 are all based on processor assignments produced either by 2ETF or ETFR. Hence, it is interesting to investigate whether the new algorithms can find a better arrangement of communications, or not.

7.5 Results and Analysis

7.5.1 Communication Scheduling Heuristics

The decoder has the greatest effect on the complexity of our GA-based schedules. As a matter of fact, the decoder used in this thesis is itself a heuristic. Therefore, it is important to examine it for both complexity and accuracy. Such a decoder must be able to transfer a chromosome into a feasible schedule.

The experiments are carried out in 2 parts, each of which corresponds to a different existing list scheduling algorithm. The two parts are called **2ETFList** and **ETFRList** after the original algorithms.

It starts with the results of the existing list heuristics. Both the processor assignments and the task ordering by the starting times are used as the inputs of our algorithm. Such a full order is equivalent to the ranks of all tasks. Having the processor assignments and the ranks, we can execute our list scheduling algorithm, Algorithm 2, on the respective problem again.

Both algorithms are tested on 3 sets of graphs (SSG, R20L and RGNOS) and under 6 model instances (M_1-M_6), which assume $g = o$ (see Table 7.1). The makespans of output schedules are shown in Appendix A. At the same time, the improvements of our list scheduling algorithms compared to the corresponding ETF algorithm are listed in Table 7.4, Table 7.5, Table 7.6, Table 7.7, Table 7.8 and Table 7.9.

Obviously, such an approach works for both 2ETF and ETFR. The improvements are always non-negative, and in most cases, positive. Makespans can be improved by up to 17%. The list scheduling algorithm is always able to transfer a priority queue

Graphs	M_1	M_2	M_3	M_4	M_5	M_6
n15	1.190	4.128	4.306	1.190	4.128	4.306
n20	0.000	5.389	0.000	1.111	3.681	0.000
n25	1.493	6.289	4.795	1.493	6.289	4.795
n30	2.454	3.505	0.242	4.487	4.380	2.660
n35	2.564	2.867	1.370	5.172	10.115	7.882
n40	2.500	6.329	6.421	4.118	2.024	1.255
n45	4.082	6.506	3.362	4.717	8.741	7.099
n50	1.880	12.715	13.852	3.196	5.956	2.530
n60	3.548	7.810	6.064	3.937	5.482	9.040
n65	2.586	9.787	11.183	4.444	10.105	8.043
n70	5.645	11.036	8.497	4.514	6.267	6.191
Avg.	2.540	6.942	5.463	3.489	6.106	4.891

Table 7.4: Improvements of 2ETFList on SSG, compared to 2ETF (%)

Graphs	M_1	M_2	M_3	M_4	M_5	M_6
n15	0.000	0.000	0.000	0.000	0.000	0.000
n20	0.000	0.000	5.076	0.000	4.878	5.348
n25	0.000	0.000	12.291	0.000	0.000	0.000
n30	0.645	0.450	2.469	0.000	3.900	3.591
n35	1.111	4.826	0.000	0.000	0.716	2.710
n40	2.538	6.397	6.076	0.000	1.296	2.407
n45	0.889	2.315	3.592	0.000	8.546	7.629
n50	1.195	2.997	4.564	0.000	1.486	5.975
n60	1.684	5.242	1.905	0.405	4.090	0.000
n65	1.690	1.877	2.878	0.000	4.472	5.066
n70	3.966	2.297	0.159	0.000	7.094	7.172
Avg.	1.247	2.400	3.546	0.037	3.316	3.627

Table 7.5: Improvements of ETFRList on SSG, compared to ETFR (%)

graph	M_1	M_2	M_3	M_4	M_5	M_6	graph	M_1	M_2	M_3	M_4	M_5	M_6
1	0.444	3.708	4.287	0.779	3.352	4.194	11	1.348	8.678	6.091	2.561	10.462	11.272
2	0.382	0.986	1.104	0.502	2.520	1.957	12	1.130	4.198	2.212	1.402	9.312	8.955
3	0.460	0.618	2.373	0.207	3.123	3.018	13	2.159	4.282	4.102	2.381	9.923	7.729
4	0.360	1.024	2.168	0.417	3.011	4.035	14	1.254	4.566	1.989	2.407	9.691	9.342
5	0.735	3.314	4.709	0.928	3.230	4.645	15	1.474	5.764	1.534	2.218	5.505	9.746
6	0.675	1.716	3.747	0.568	4.068	2.787	16	1.325	4.977	2.623	2.621	5.920	8.659
7	0.310	0.882	0.728	0.571	1.713	1.696	17	0.699	5.542	4.173	1.433	14.695	6.355
8	0.278	2.552	0.195	0.800	0.383	2.987	18	2.225	5.462	7.745	2.405	7.287	8.164
9	0.826	4.722	4.469	1.478	0.682	3.172	19	1.587	3.904	3.785	1.892	9.006	10.695
10	0.455	6.505	1.634	0.402	3.278	2.442	20	1.215	4.333	4.036	2.022	8.356	8.921
Avg.	0.493	2.603	2.541	0.665	2.536	3.093	*Avg.*	1.442	5.171	3.829	2.134	9.016	8.984

Table 7.6: Improvements of 2ETFList on R20L, compared to 2ETF (%)

G.	M_1	M_2	M_3	M_4	M_5	M_6	G.	M_1	M_2	M_3	M_4	M_5	M_6
1	0.091	0.974	0.889	0.000	0.399	0.143	11	1.378	3.670	2.932	1.747	3.563	6.995
2	0.280	0.000	0.644	0.102	0.000	0.624	12	1.126	2.550	1.971	0.747	3.648	5.545
3	0.095	1.306	0.582	0.000	0.661	0.000	13	0.540	0.679	1.202	1.828	5.359	3.469
4	0.179	1.213	0.627	0.206	0.000	0.837	14	0.585	1.199	1.407	1.316	6.188	5.048
5	0.189	0.415	0.393	0.104	0.000	0.000	15	0.924	1.878	2.369	1.348	4.989	2.932
6	0.000	0.797	0.590	0.096	0.353	0.000	16	1.098	1.771	1.726	1.226	4.578	5.009
7	0.000	2.423	0.434	0.000	0.843	0.412	17	0.901	1.073	0.291	1.964	3.571	3.387
8	0.276	0.000	0.982	0.398	0.512	0.000	18	0.740	2.288	1.478	1.182	5.448	4.394
9	0.000	0.300	0.131	0.000	0.000	0.216	19	1.406	2.601	2.852	1.230	5.928	5.236
10	0.000	1.206	0.566	0.000	0.000	0.501	20	0.852	2.743	2.571	1.412	3.620	5.717
Avg.	0.111	0.863	0.584	0.091	0.277	0.273	*Avg.*	0.955	2.045	1.880	1.400	4.689	4.773

Table 7.7: Improvements of ETFRList on R20L, compared to ETFR (%)

Graphs	M_1	M_2	M_3	M_4	M_5	M_6
r1p50	1.166	0.000	0.000	0.601	0.000	1.394
r1p100	3.040	5.704	0.490	4.615	13.408	11.999
r1p150	0.000	3.280	5.351	1.754	0.000	8.826
r1p200	0.555	7.583	5.106	1.826	15.319	19.157
r1p250	1.108	2.978	5.692	4.247	12.902	11.874
Avg.	1.174	3.909	3.328	2.609	8.326	10.650
r3p50	2.007	1.956	6.469	2.595	8.879	10.291
r3p100	3.003	11.197	4.771	4.789	5.524	5.553
r3p150	1.362	5.492	7.601	2.812	12.804	12.846
r3p200	2.861	4.886	8.040	4.762	17.332	3.957
r3p250	0.554	5.949	8.669	4.379	18.776	8.338
Avg.	1.957	5.896	7.110	3.867	12.663	8.197
r5p50	2.144	3.184	11.734	1.031	5.142	1.061
r5p100	1.131	8.968	4.283	3.871	6.675	10.388
r5p150	3.573	14.864	14.116	3.598	6.485	8.375
r5p200	2.214	8.728	5.731	4.678	12.387	9.957
r5p250	4.639	10.658	10.288	4.611	9.741	10.934
Avg.	2.740	9.280	9.230	3.558	8.086	8.143

Table 7.8: Improvements of 2ETFList on RGNOS, compared to 2ETF (%)

Graphs	M_1	M_2	M_3	M_4	M_5	M_6
r1p50	0.000	0.000	0.000	1.370	3.330	2.444
r1p100	1.887	0.000	0.000	2.450	4.799	3.903
r1p150	0.000	0.000	0.000	0.000	0.096	0.000
r1p200	1.339	0.340	0.952	3.288	0.000	0.000
r1p250	1.616	0.880	0.400	2.775	2.397	1.177
Avg.	0.968	0.244	0.270	1.977	2.124	1.505
r3p50	1.773	0.000	7.107	0.550	5.061	8.553
r3p100	2.480	0.000	1.402	2.136	9.995	3.671
r3p150	0.287	2.576	1.510	1.495	4.258	4.437
r3p200	1.987	0.866	1.599	1.389	8.885	4.971
r3p250	0.324	0.279	0.000	2.780	1.505	1.002
Avg.	1.370	0.744	2.324	1.670	5.941	4.527
r5p50	1.185	0.000	5.672	0.267	1.360	1.243
r5p100	0.883	3.917	2.546	2.111	7.518	5.015
r5p150	2.299	3.369	2.846	2.442	4.539	7.353
r5p200	2.014	2.449	1.424	1.680	4.798	2.810
r5p250	0.253	0.739	1.092	3.196	4.504	4.903
Avg.	1.327	2.095	2.716	1.939	4.544	4.265

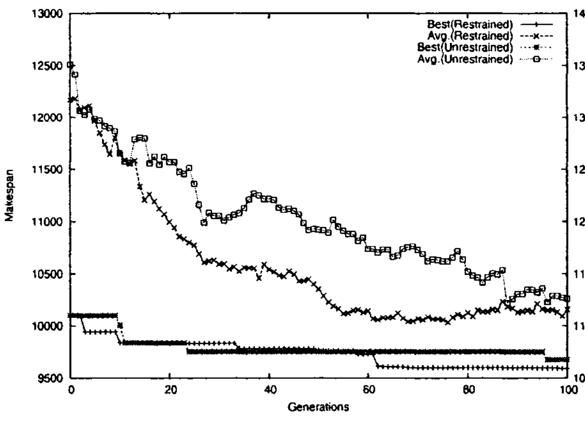
Table 7.9: Improvements of ETFRList on RGNOS, compared to ETFR (%)

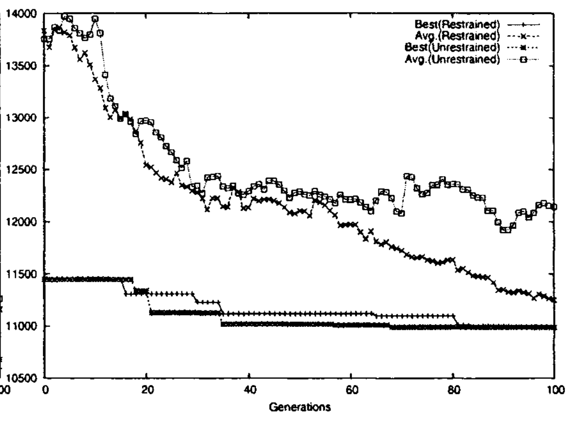
into a relatively better schedule, compared to the existing algorithms. Hence, this list scheduling approach should be an efficient decoder for processor assignments.

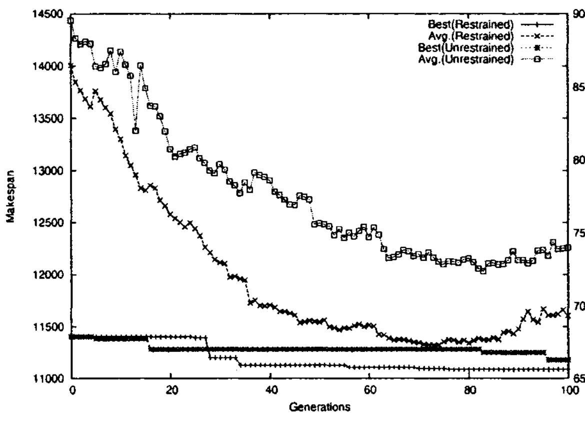
7.5.2 Different encodings in communication scheduling

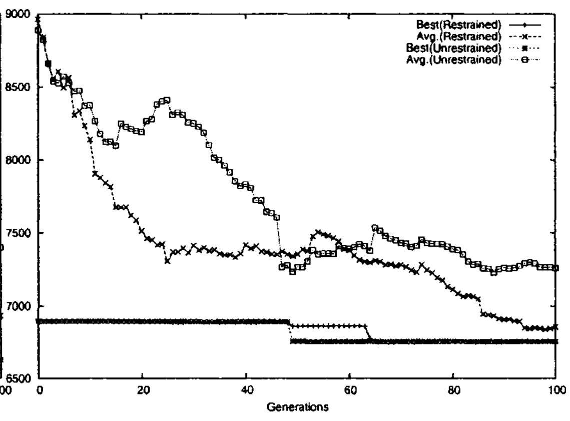
This section compares the novel encoding method proposed in Section 5.1.2 to the original encoding method described in Section 4.3.2. In the experiments, GAs with both encodings are run on a traced graph representing the compilation of Gaussian elimination algorithm [32]. The same sets of genetic operators and the same values of parameters are applied. Average makespan and the best makespan in each generation are logged for the comparisons. We observe 100 generations for each experiment. Moreover, the experiments are carried out for processor assignments generated from both 2ETF and ETFR.

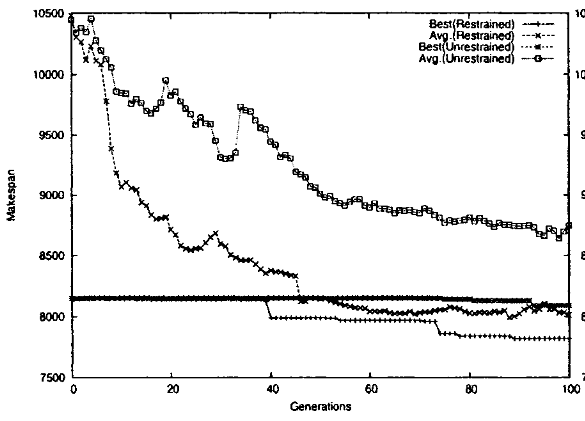
Figure 7.9 presents the comparisons under different models. In the figures, each pair of curves corresponding to the different genotypes has close starting points. The difference between the average makespans of the two initial populations is less than 3%. The size of the solution space is much larger than the population size, which is 50, even though the restrained encoding is supposed to minimize the representation space. That is why the GA tends to start with similar initial populations regardless of the encoding method. But the final populations are distinct. The restrained encoding leads to better results. The average makespan of the final population generated with the improved encoding is shorter by up to 13.54% (Figure 7.9(f)). This encoding results in an improvement of up to 3.34% compared to the unrestricted encoding


(a) GAComm2ETF, M_1


(b) GAComm2ETF, M_2


(c) GAComm2ETF, M_3


(d) GACommETFR, M_4


(e) GACommETFR, M_5

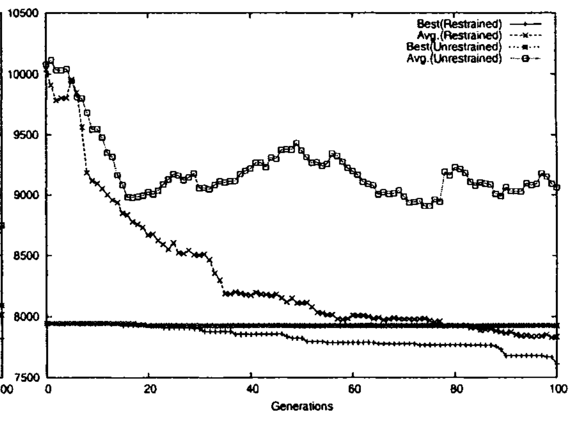

(f) GACommETFR, M_6

Figure 7.9: Comparison of Two Encoding Method

(Figure 7.9(e)). As a matter of fact, the populations encoded with constraints evolve more regularly and converge faster.

All the results indicate the modified genotype is better than the primitive one. The operators collaborate more efficiently with the restrained encoding. With such an encoding method, the algorithm can produce a better result.

7.5.3 Scheduling communications with GA

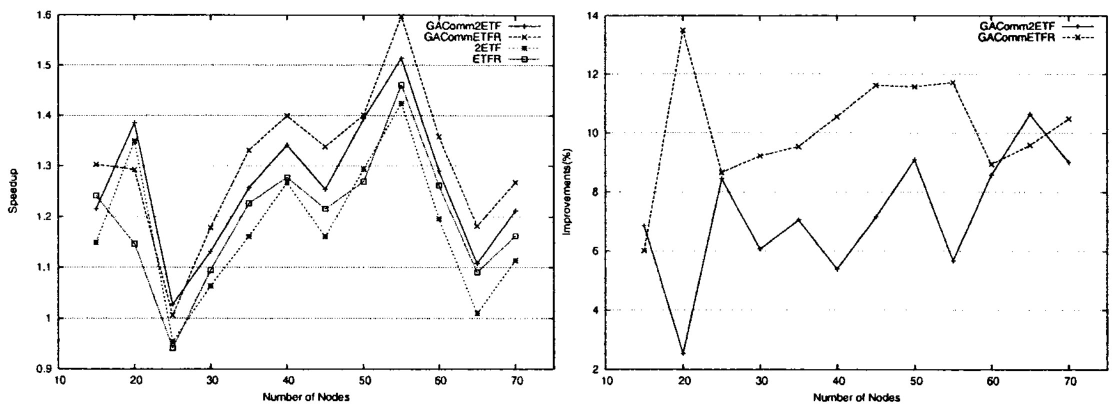
The experiments of the algorithm which is proposed in Chapter 3 are divided into 2 parts based on the source of processor assignments. As this algorithm deals with communication arrangements, the two parts of the experiments are called **GAComm2ETF** and **GACommETFR**, respectively. The test sets used in this section includes SSG, R20L and RGNOS.

The experiments are accomplished in a few steps. First, a processor assignment is obtained by running an existing list heuristic, either 2ETF or ETFR. With such a processor assignment, the corresponding graph is scheduled again using the GA (Chapter 4). Only the tasks on the same processors are reordered by GA. The processor assignment of obtained schedules is not changed. The GA is run 50 times on each single problem. Minimal makespan from each run are recorded. The values for the same problem (with the identical graph and the same model) are averaged for the final results.

Table B.1 lists the speedups of both GAComm2ETF and GACommETFR on each graph in SSG. The average speedup of GAComm2ETF over SSG is 1.26, and average speedup of GACommETFR is 1.31. Although GACommETFR is more likely to generate a better schedule than GAComm2ETF, neither of them is guaranteed to be superior in all cases. Figure 7.10(a) compares our GA to the existing heuristics using average speedups with respect to the number of nodes. GAComm2ETF achieves better schedules than GACommETFR in only 2 graphs (graph $n20$ and $n25$) out of the 12 graphs in SSG. 2ETF also generate better schedules than ETFR when scheduling graph $n20$ and $n25$. In fact, the shape of the GA's curve is similar to the curve of the original algorithm, which produces the corresponding processor assignment. As described in Chapter 6, 2ETF and ETFR use the same technique to arrange communications by simply inserting them. It is indicated by the results that the GA tends to retain the advantage/disadvantage of the various processor assignment algorithms.

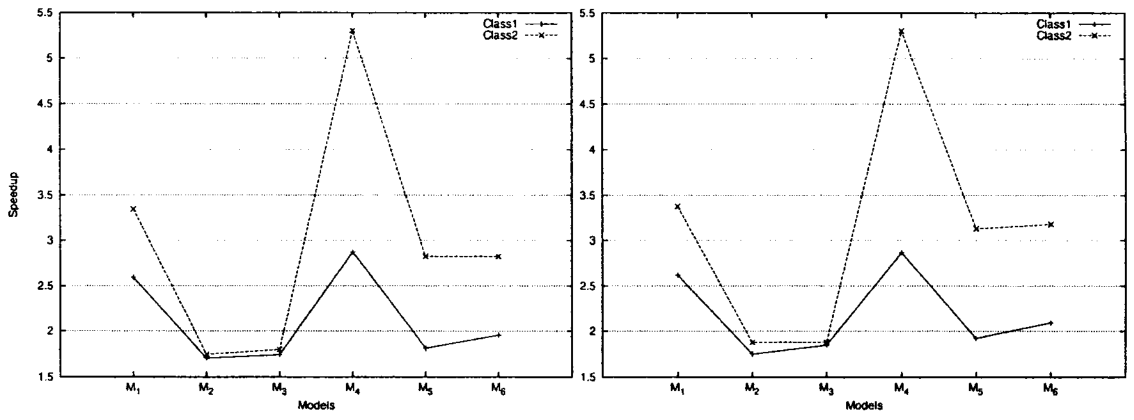
Table B.2 shows the improvements of GA in all cases of scheduling SSG. The GA improves the final results, as the improvements are always *positive.* Average improvements of GA on processor assignments originated from 2ETF is 7.21%, and from ETFR is 10.12%. It can be seen in Figure 7.10(b) that when the GA is applied to the processor assignments from ETFR, the improvements are higher than those achieved over 2ETF. This is true in most cases (except graph n05 and n65).

The algorithm is also tested with the graph set R20L. GA speedups are listed in Table B.3. The average speedup is 2.6. The speedups are computed for two classes of graphs in R20L with respect to models. Such comparisons are shown in Figure 7.11. The curves in Figure 7.11(a) are similar to the curves of Figure 7.11(b). Curves of similar shapes are expected for any other processor assignments. Average speedup



(a) Average Speedups (b) Average Improvements

Figure 7.10: Comparison of GA on different processor assignments when scheduling graphs in SSG



(a) Average speedup of GAComm2ETF (b) Average speedup of GACommETFR

Figure 7.11: Comparison of speedup on graphs with different anchors

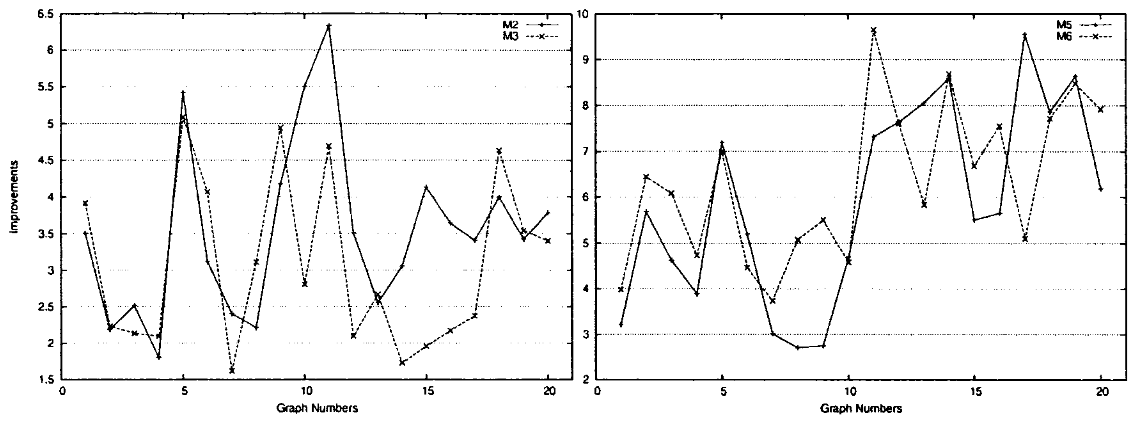
values for the second class of graphs is greater than those of the first class, by 41.9%. In fact, the algorithm always gets higher average speedups when scheduling the graphs with higher anchor values (the second class). When the latency is decreased (from M_2 to M_3 and from M_5 to M_6), there are only slight improvements in speedup. And, speedup values drop when overhead is increased (from M_1 to M_2 and from M_4 to M_5). Moreover, the algorithm tends to return higher speedup values if the graphs are scheduled on more processors. For instance, the speedups under M_1 ($P = 4$) is much lower than the speedups under M_3 ($P = 8$). As a result, the higher speedup value always occurs when scheduling under a model with a longer L, a smaller o and a greater P.

Table B.4 shows the improvements of the algorithm when scheduling graphs of R20L. We use the following group of figures to analyze the relationships between the model parameters and GA's improvements. The average improvement

$$\alpha_{GAComm}(G) = \frac{\alpha_{GAComm2ETF,2ETF}(G) + \alpha_{GACommETFR,ETFR}(G)}{2} \tag{7.6}$$

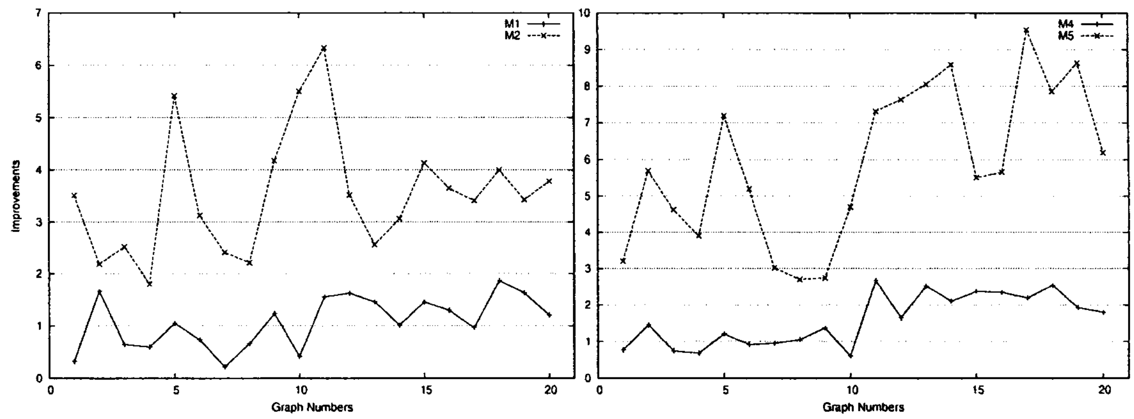
is used here, as the measure of improvement.

Figure 7.12 compares two sets of models: (M_2, M_3) and (M_5, M_6). Models in the same set have only different latencies. The chaotic curves show that there is no reliable improvement in GA performance when latency is changed. In contrast, increasing overheads leads to significant improvements. Models with different overheads are compared in Figure 7.13. When o is increased from 1 to 10, the GA returns higher improvement values by 252%. Likewise, the GA achieves better performance when



(a) $P = 4$ (b) $P = 8$

Figure 7.12: Average Improvement of GA performance on Graphs with different Latencies



(a) $P = 4$ (b) $P = 8$

Figure 7.13: Average Improvement of GA performance on Graphs with different Overheads

there are more processors. Figure 7.14 plots the average improvement of the GA with respect to the number of processors P. The improvement increases 80% when the

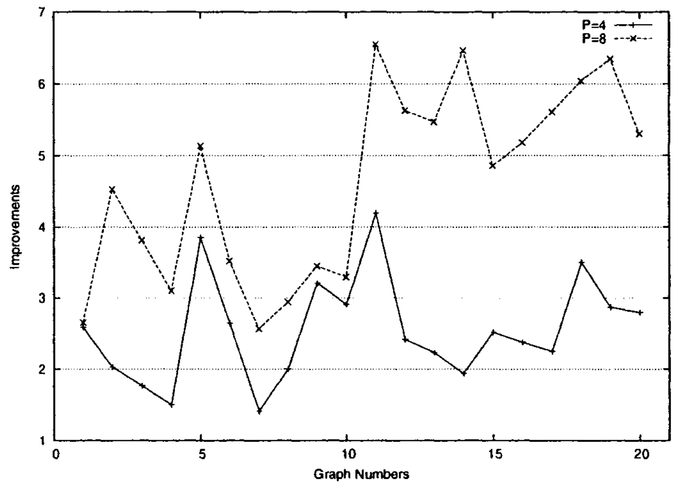
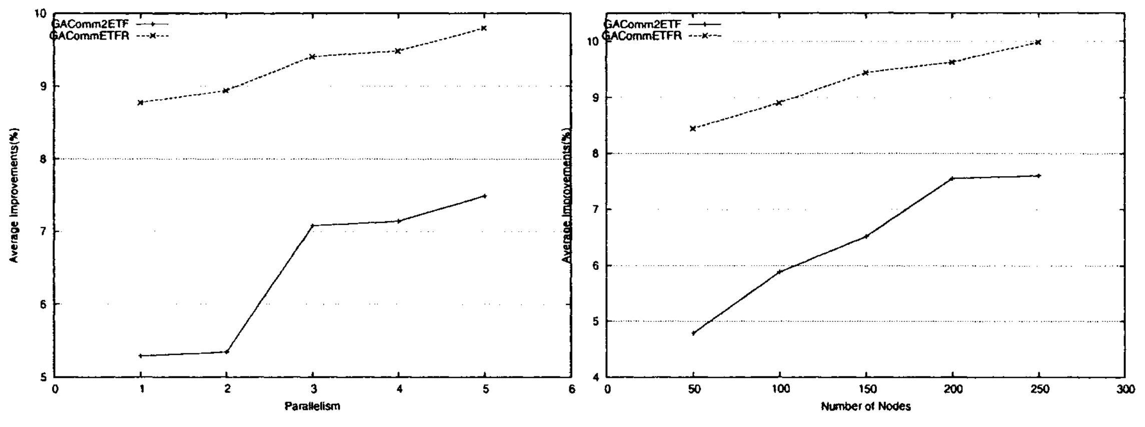


Figure 7.14: Average improvement on Graphs with different Numbers of Processors

number of processors is doubled.

The algorithm is also run on the graph set of RGNOS, which contains several subsets of graphs with different values of parameters. Table B.6 consists of the improvement values of GA. The GA's improvements compared to existing ETF heuristics are non-negative in all cases. The average is 8.17%. And, as in previous cases, the algorithm tends to return higher improvement values on processor assignments generated by ETFR, rather than by 2ETF. That can be seen in Figure 7.15. In addition, the algorithm is likely to achieve more improvements over graphs with higher parallelism. Average improvements on graphs with $\rho = 5$ is 20% higher than that on graphs with $\rho = 1$. Meanwhile, the algorithm also performs better on graphs with more nodes. As it is shown in Figure 7.15(b), 36.54% more improvements are achieved, on average,



(a) (b)

Figure 7.15: Comparison of Improvements on RGNOS

when the number of nodes are increased from 50 to 250.

In summary, the genetic algorithm works for all types of graphs in the experiments. It can always improve a schedule generated by one of the existing ETF heuristics by finding a better re-arrangement of the communication tasks. There are a few parameters of graphs and models which have a significant impact on the results: the overheads, the number of processors, the number of nodes in a graph, the anchor out-degree of a graph and the parallelism of the graph. From the results in this section, we may safely draw the conclusion that the GA performs better when the delays caused by communication overheads are more significant.

7.5.4 Scheduling communications when $g > o$

In this section, the algorithm is evaluated under models with significant gaps (Table 7.2). The first class of graphs in R20L is used. The experiments are the same as those described in Section 7.5.3.

The algorithm succeeds in taking large gaps into account. Table B.7 shows GA speedups in various situations. The values of GA speedups are less than the speedups when $g = o$ (Table B.3) by 43.75%. As it is shown in Figure 7.16, GA speedups still have the same tendency as as the corresponding ETF heuristics. There exist fractional

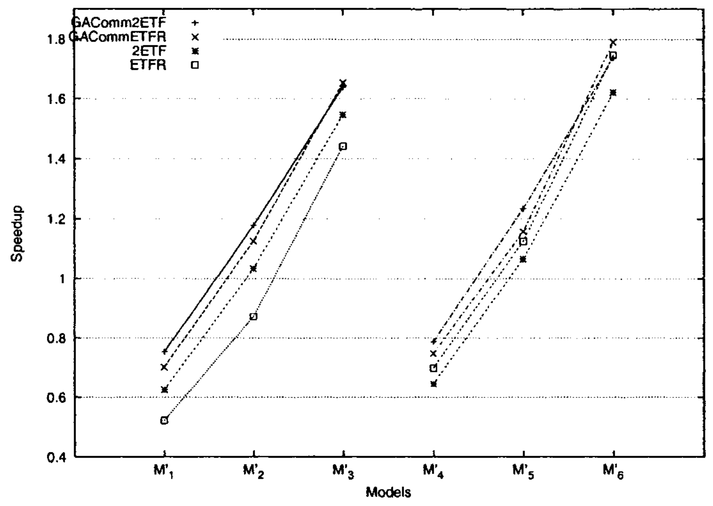
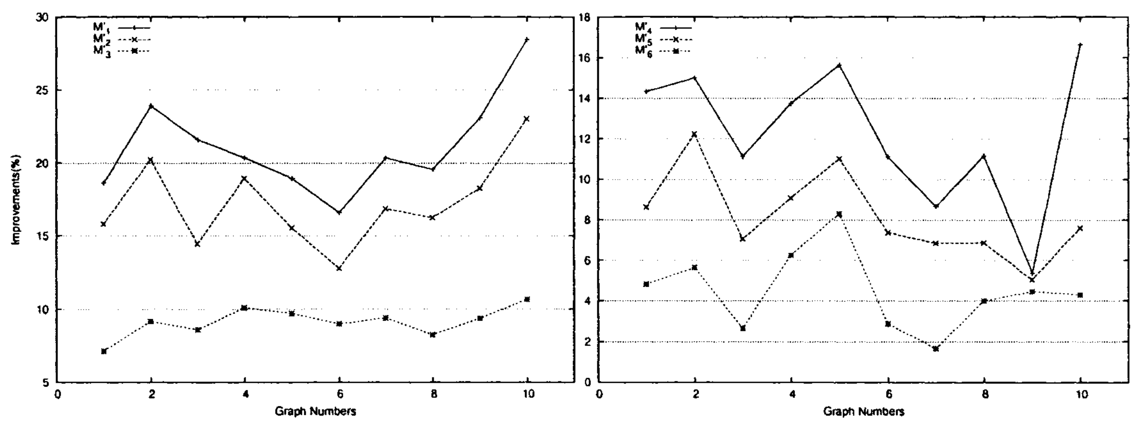


Figure 7.16: Average speedup with respect to number of nodes in graphs

speedups of GA when ETF speedups are also fractional, i.e. the final schedule is longer than its sequential time equivalent. This happens when $g \gg o$, mostly under model M_1' and M_4'. That is because the existing ETF heuristics do not take gaps into account. Such an approach may generate too many communication tasks for the same source. Consequently, idling periods brought by gaps are unavoidable and hence lengthen the schedule. Obviously, speedup decreases with decreasing gaps.

Table B.8 lists GA's improvements compared to corresponding ETF heuristics. The GA achieves positive improvements in every test case with $g > o$. The average improvement is 15.85%. Figure 7.17 makes this point clear. This figure compares



(a) $P = 4$ (b) $P = 8$

Figure 7.17: Average Improvements of GA performance under different models ($g > o$)

GA's average speedups (Equation (7.6)) on each graph under different models. As a matter of fact, the length of gaps is one of the factors that impact a GA's performance. When the gap is increased from 15 to 80, GA's improvement is increased by 142.88%, on average. Furthermore, curves corresponding to different values of gaps are distributed separately without any intersections. According to the results, the algorithm is expected to gain more improvements when the gaps are larger. This is verified by all 10 graphs. Still, GA performs better when the delays caused by communication overheads are more significant when $g > o$.

7.5.5 Processor assignments

The algorithm proposed in Chapter 6 is tested on two sets of graphs: SSG and RGNOS. Average makespan are calculated after 50 runs of each problem instance. We do not insert any specific individuals into the initial generations so as to evaluate the genetic algorithm thoroughly.

According to the speedups of all algorithms when scheduling SSG, GA is always superior to ETF heuristics.

Note, there exists $\delta_{GA} = 1$ especially under models (M_2 and M_5) with large communication costs. When $\delta_{GA} = 1$, the speedups of the ETF heuristics are less than 1. The principle of ETF heuristics is to make use of as many processor working as possible. Tasks are always distributed to different processors. In this case, it is impossible for these heuristics to generate a schedule which uses only one processor. However, such a processor assignment may be optimal for some graphs with relatively complicated structures, as separating two consecutive tasks could bring in unnecessary delays via extra communications. This makes it difficult to yield a schedule with a speedup greater than 1 by simply re-arranging the communications. But, the processor assignment GA is always able to return a schedule with a speedup greater than 1.

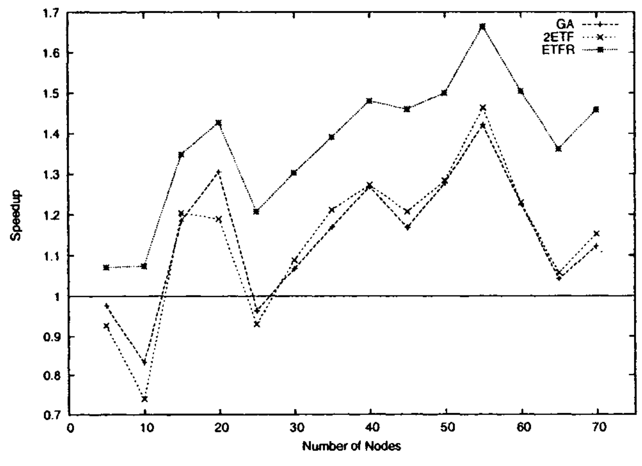
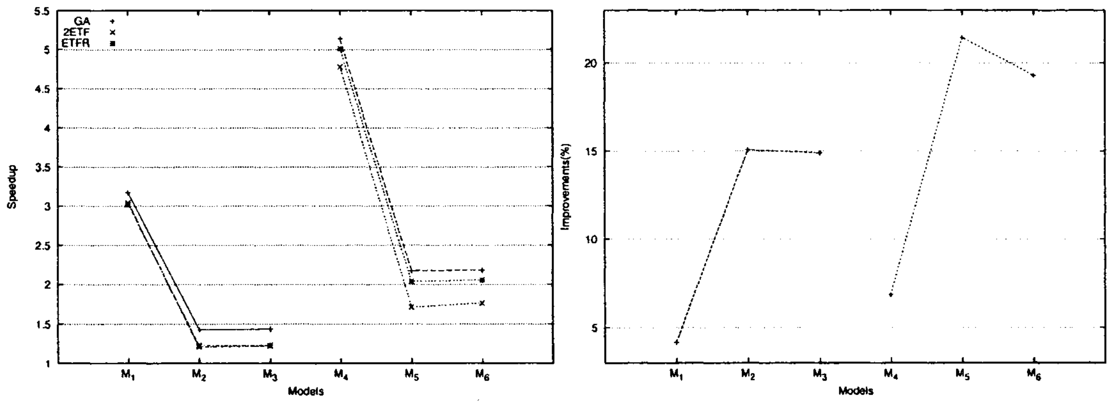


Figure 7.18: Average speedup of different algorithms when scheduling SSG

The comparison of the average speedups of three algorithms with respect to different graphs is illustrated in Figure 7.18. GA gains 20.6% higher speedups than 2ETF.

We also apply the GA to schedule RGNOS, to observe how the graph characteristics and the model parameters impact the algorithm's performance. GA generates better schedules than the ETF heuristics. On average, GA speedups are 13% better than 2ETF's.

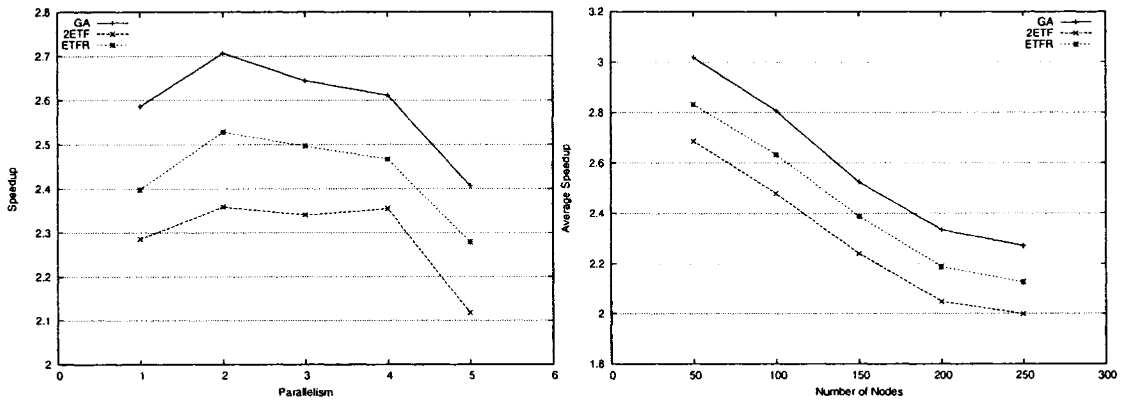


(a) Average speedup (b) Average improvement compared to 2ETF

Figure 7.19: Comparisons between models

Figure 7.19(a) compares average speedup with respect to 6 models ($M_1, M_2, \ldots, M_6$). It can be seen that the greatest speedup was achieved under model M_4, while the lowest was under M_2. Higher speedup values are obtained when there are more processors or when there are lower communication costs, i.e. smaller latency or smaller overheads. The difference between GA and 2ETF is shown in terms of average improvement in Figure 7.19(b). The best improvement is achieved when L, o and P are all at their highest (M_5). In cases where communication cost dominates, the difference between GA and 2ETF becomes more significant.

The tendency of speedup decreasing with increasing ρ is observed (with one exception at $\rho = 1$) in Figure 7.20(a). GA speedup increases by 11.1% when parallelism decrease from 5 to 2. In the same way, it is shown in Figure 7.20(b) that GA speedup decreases for graphs with more nodes. The average speedup decreases by 24.8% when the number of nodes increases by a factor of 5. In brief, the GA gains higher speedup



(a) With respect to ρ (b) With respect to N_v

Figure 7.20: Comparisons of average speedups

values on less complicated graphs.

According to the results in this section, the algorithm proposed in Chapter 5 is able to find a better schedule than both ETF heuristics. It is possible for this algorithm to reach the optimal processor assignment which is not reachable by other heuristics. Furthermore, various settings of the scheduling problems leads to different performances of this algorithm.

7.6 Summary

This section has introduced the basic methodology of the experiments. The benchmark sets of graphs are discussed. We also list all LogP model instances used in the experiments. In addition, three performance measures are explained.

There are 2 genetic algorithms proposed in this thesis. These algorithms are evaluated in 4 parts. All the results are recorded and analyzed in detail.

The results indicates that both algorithms have advantages over the existing heuristics for scheduling under LogP. The algorithms perform differently in various scheduling environments.

The average speedup value gained by the communication arrangement algorithm (among three sets of test graphs) is higher than the corresponding list scheduling algorithm by 7.60%, when the gap is ignored. GA improves the schedules by 12.14%, when dealing with communications with significant gaps. As for processor assignment, GA achieves an average improvement of 11.42% when $g = o$. Both algorithms performs better when the delay caused by communication overheads are more significant.

Chapter 8

Conclusions and Future Work

In Chapter 3, we said that the goal of this thesis is to schedule general graphs, on homogeneous multiprocessor systems, under the LogP model. The major contributions of this thesis are two approaches to scheduling using GAs:

1. Communication management;

2. Processor allocation.

In each case, the scheduling problem were formalized as a single objective optimization problem. The objective functions for communication management and processor allocation were both based on the makespan of the schedules described in Chapter 2.

Now that all the details have been presented in Chapter 4, 5 and 6, it is time to wrap up. Section 8.1 lists the essential results obtained in this thesis. Section 8.2 suggests possible directions for future work.

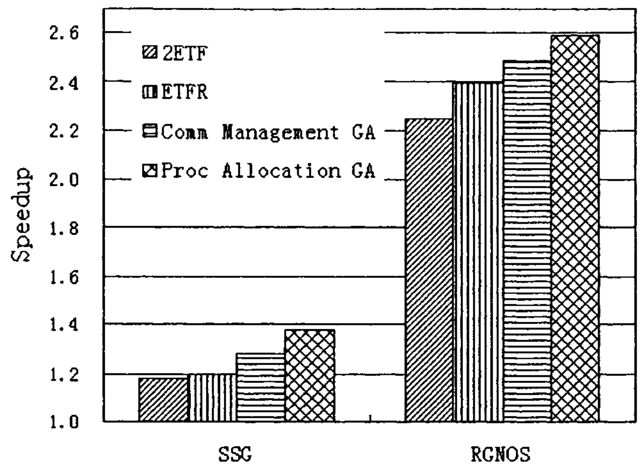


Figure 8.1: Comparison of Speedups

8.1 Fundamental Results

This thesis has presented work and progress mainly relevant to the application of evolutionary computation in static multiprocessor scheduling. Essential results are:

- A novel approach, the task graph extension under the LogP model, has been presented which generalizes communication overheads as tasks. It allows application of more general scheduling algorithms to be applied to the problem.

- GAs have been applied to multiprocessor scheduling problems under the LogP model. The GAs presented in this thesis combine established and new techniques in a unique manner. Both GAs provide better solutions than existing list scheduling algorithms. Figure 8.1 shows the average speedups of four algorithms over two sets of test graphs. GAs clearly outperform the list scheduling algorithms on the test graphs.

- A new concept of gap filling has been introduced to handle significant gaps under the LogP model. It allows search for more possible schedules especially when significant gaps occurs.

- A comprehensive experimental methodology to quantitatively compare the GAs to the existing approaches has been presented. In particular, several quantitative performance measures as well as sets of test graphs have been proposed. The experimental results show that both measures and test graphs can reveal differences in performance.

8.2 Future Directions of Research

This thesis provides a starting point for further work in scheduling general task graphs under the LogP model, using GAs. Some of the possible directions in future work are described in this section.

Improving the Genetic Algorithms

Since the scheduling problems for communication management and processor allocations are NP-hard, we had to resort to the use of heuristics (list scheduling algorithms) and meta-heuristics (genetic algorithms). The algorithms presented in this thesis work reasonably well in practise, but there is scope for improvement in the followings:

1. One-to-one Encoding

The encoding methods used in this thesis only assure that identical chromosomes correspond to identical candidate solutions. But, one solution is represented by more than one chromosomes. It would be desirable to design encoding methods which utilize one-to-one mapping. For instance, we can add restrictions on the encoding methods. The new encoding should remain unbiased.

2. Simpler Decoder

   Current decoders use a list scheduling algorithm to generate the exact schedule in order to evaluate each individual. Most computation time of the genetic algorithms are spent in the decoders. One way to improve the decoder is to find another measure that can be obtained without the exact schedule and can reflect the schedule's makespan.

3. Adaptive Parameters

   The GA are based on the SGA. Most parameters are determined prior to evolution. For example, the crossover and mutation rates and the number of generations are all set in advance. These parameters can actually be changed according to the convergency rate or diversity of the population.

Combining Communication Management and Processor Allocation

Throughout this thesis, we have presented communication management and processor allocation algorithms as two distinct algorithms, operating one after the other. It may be possible to combine them into a new algorithm. In this case, the new algorithm will have be independent of ETF strategies.

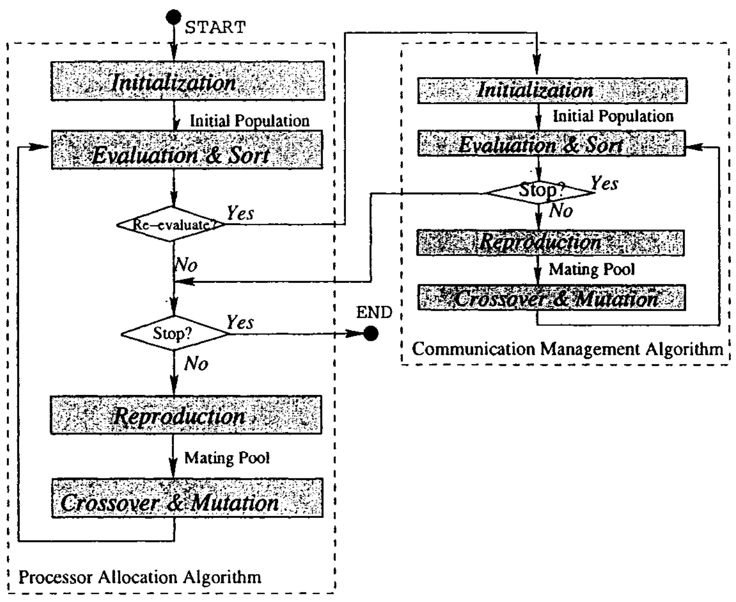


Figure 8.2: Combined GAs for Scheduling under the LogP model

For instance, these two algorithms can be combined into a two-level hierarchy. Figure 8.2 shows the framework of such a hierarchy. Communication management algorithm cannot be used to as an alternative to the evaluation of the processor allocation algorithm. Then, the communication tasks of each individual within a specific generation of the processor assignment algorithm are re-arranged to generate better schedules. The new makespans are used in the reproduction procedures. However, such an evaluation can be executed at each generation due to the complexity of the algorithm. In the same manner, the population size and the number of generations of the communication management algorithm would be lower than usual.

An interesting combination would introduce another representation methodology. The new encoding should allow an individual to include both processor assignments and task ranks. The key issue of developing such an encoding is the embedding of the task ranks, which are usually determined after the processor assignment is set.

Multi-objective Genetic Algorithms

Decreasing either communication costs or computational costs may leads to a better schedule. However, the communication costs and the computational costs for a given graph conflict with each other. Using these costs as two objectives, we are able to apply a multi-objective genetic algorithm to the scheduling problem under the LogP model.

The challenge of such an approach is to determine the objective function. The two objective functions should be easy to obtain. In addition, it must be proved that the functions are able to evaluate the makespans of the potential schedules. As a matter of fact, some straightforward objective functions may not work for this problem.

Other Heuristics

While the genetic algorithms are clearly well suited to the scheduling problem under LogP and extending a task graph helps to simplify the problem, it would be interesting to try other heuristics as well.

List scheduling algorithm is a promising candidate due to its lower complexity. One possible extension is to find new task ranking rules based on the results produced by the GAs. The difference between the new list scheduling algorithm and the existing algorithm is that the new one treats the communication tasks with the same seriousness as the computational tasks, by extending the task graph. This is a much more ambitious approach with potentially greater rewards due the reduced computation time.

Thorough Evaluations

The experiments have proved that the algorithms presented in this thesis can handle various conditions of scheduling under the LogP model. However, the experiments are not as organized as the experiments suggested in [32].

The test graphs used in this thesis are from different researchers. There are not any specific benchmarks for scheduling under the LogP model. A comprehensive test set is desired.

Instead of developing an entirely new test set, another interesting approach is to adapt the algorithms to SDM models. Most parts of the algorithms are unrelated to the model except for the decoder. In that case, the algorithms can be examined on some widely used test sets, such as Kwok's benchmarks [32].

Besides, a big challenge for the evaluation is the comparison of schedules obtained for program graphs against program execution times on a contemporary machine preserving properties of the LogP model.

Appendix A

Benchmark Graphs

Graph	N_v	N_e	N_w	H	A_{out}	t_{CP}	t_{seq}
n05	5	4	14.600	3	1.333	66	73
n10	10	20	9.800	3	3.333	63	98
n15	15	28	10.800	4	2.333	88	162
n20	20	29	8.600	5	1.933	105	172
n25	25	49	8.160	6	2.722	140	204
n30	30	77	9.500	6	3.080	158	285
n35	35	98	10.457	6	3.379	157	366
n40	40	117	10.450	6	3.545	161	418
n45	45	142	10.844	7	4.057	215	488
n50	50	147	11.140	8	3.675	231	557
n55	55	171	11.455	7	3.886	212	630
n60	60	197	10.633	8	4.477	243	638
n65	65	235	9.908	10	4.196	296	644
n70	70	300	10.314	9	5.172	280	722
graph1	92	149	31.098	20	1.987	1070	2861
graph2	94	160	29	20	2.105	1010	2726
graph3	94	154	30.957	20	2.081	1008	2910
graph4	93	152	31.108	20	2	1012	2893
graph5	88	148	30.920	20	2.085	990	2721
graph6	98	155	32.194	20	2.214	1085	3155
graph7	95	163	26.621	20	2.012	919	2529
graph8	79	125	30.354	20	1.923	1024	2398
graph9	84	135	30.762	20	1.957	1003	2584
graph10	96	164	32.833	20	2.158	1035	3152
graph11	243	1730	91.671	20	8.084	3510	22276
graph12	244	1677	97.004	20	7.588	3491	23669
graph13	259	1988	90.127	20	8.148	3324	23343
graph14	253	1848	90.375	20	8.105	3398	22865
graph15	243	1768	97.189	20	7.789	3504	23617
graph16	229	1618	91.624	20	7.632	3297	20982
graph17	289	2250	88.363	20	8.621	3427	25537
graph18	228	1593	89.785	20	7.622	3430	20471
graph19	244	1685	96.086	20	7.590	3549	23445
graph20	246	1754	89.423	20	7.901	3297	21998
r1p50	50	206	54.480	4	5.024	336	2724
r1p100	100	681	48.300	6	10.984	468	4830
r1p150	150	1541	50.453	6	12.230	560	7568
r1p200	200	1964	50.945	10	11.353	687	10189
r1p250	250	3731	51.776	8	20.277	598	12944
r2p50	50	115	47.240	5	5.227	331	2362
r2p100	100	378	50.850	7	4.447	614	5085
r2p150	150	884	50.707	8	9.609	589	7606
r2p200	200	2760	52.900	9	16.331	613	10580
r2p250	250	3100	53.260	9	16.848	732	13315
r3p50	50	164	56.600	5	5.125	388	2830
r3p100	100	556	49.650	7	6.318	544	4965
r3p150	150	1173	51.760	9	10.200	667	7764
r3p200	200	1655	49.810	10	9.851	787	9962
r3p250	250	2406	49.996	10	11.680	719	12499
r4p50	50	182	50.840	6	4.667	435	2542
r4p100	100	507	52.730	8	5.633	672	5273
r4p150	150	1017	51.233	8	8.071	681	7685
r4p200	200	1483	45.805	10	10.669	697	9161
r4p250	250	2294	50.664	10	11.356	833	12666
r5p50	50	149	55.300	7	5.731	524	2765
r5p100	100	478	44.960	8	6.208	538	4496
r5p150	150	1077	47.960	8	12.239	480	7194
r5p200	200	1757	50.875	9	9.346	637	10175
r5p250	250	2215	48.572	10	10.649	684	12143

Table A.1: Characteristics of Benchmark Graphs

Appendix B

Experiment Results

Graphs	M_1		M_2		M_3		M_4		M_5		M_6	
	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2
$n15$	1.976	1.976	0.814	0.930	0.856	1.013	1.976	1.976	0.814	0.905	0.856	1.019
$n20$	1.954	1.954	1.096	0.961	1.096	0.920	1.977	1.977	1.096	0.901	1.096	1.042
$n25$	1.578	1.557	0.730	0.675	0.774	0.676	1.578	1.557	0.729	0.735	0.775	0.833
$n30$	1.840	1.892	0.690	0.690	0.690	0.770	1.938	1.996	0.790	0.847	0.840	0.879
$n35$	1.969	2.074	0.635	0.756	0.635	0.777	2.322	2.345	0.970	0.959	1.010	1.078
$n40$	2.152	2.245	0.719	0.764	0.724	0.777	2.683	2.660	0.886	0.966	0.886	0.987
$n45$	2.167	2.242	0.629	0.723	0.629	0.727	2.478	2.551	0.811	0.872	0.811	0.915
$n50$	2.160	2.323	0.731	0.797	0.815	0.807	2.676	2.618	0.977	0.906	1.002	0.953
$n55$	2.466	2.689	0.675	0.819	0.815	0.849	3.223	3.159	0.905	1.031	1.000	1.038
$n60$	2.156	2.225	0.647	0.688	0.654	0.695	2.671	2.686	0.756	0.903	0.857	0.952
$n65$	1.955	1.901	0.581	0.650	0.596	0.618	2.215	2.279	0.639	0.800	0.661	0.849
$n70$	2.074	2.135	0.500	0.568	0.500	0.575	2.717	2.745	0.732	0.777	0.753	0.808

$\delta_1 : \delta_{GAcomm2ETF}$ $\delta_2 : \delta_{GAcommETFR}$

Table B.1: GA Speedup of Communication Arrangements of SSG ($g = o$).

Graphs	M_1		M_2		M_3		M_4		M_5		M_6	
	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2
$n15$	2.381	1.205	8.716	12.870	9.498	11.111	2.381	1.205	8.716	7.223	9.478	2.472
$n20$	2.200	5.333	5.988	18.265	0.000	17.621	3.333	4.396	3.681	15.111	0.000	20.290
$n25$	3.522	0.750	12.097	12.426	9.760	17.962	3.515	0.758	11.969	11.380	9.832	8.649
$n30$	4.975	3.455	3.505	13.065	0.242	17.769	5.712	2.864	12.200	7.767	9.787	10.431
$n35$	4.667	6.154	2.867	16.204	1.370	12.959	9.431	4.258	13.276	8.921	10.744	8.737
$n40$	2.895	8.281	8.003	16.805	7.384	17.059	8.347	4.758	4.453	7.118	1.255	9.272
$n45$	8.102	8.563	6.506	14.122	3.362	15.491	7.099	2.878	10.812	16.084	7.099	12.579
$n50$	3.053	9.509	12.727	18.190	16.974	17.823	4.950	3.277	10.636	9.769	6.221	10.893
$n55$	5.030	6.276	4.111	15.772	7.311	18.818	6.905	2.722	3.589	14.659	7.080	12.046
$n60$	4.548	4.721	11.532	15.881	6.064	14.754	5.953	5.367	7.457	8.248	15.915	4.651
$n65$	5.362	8.446	12.510	16.894	13.056	12.774	7.711	2.880	12.235	7.651	12.972	8.917
$n70$	6.401	9.343	11.036	16.092	8.497	14.847	7.733	3.993	10.381	9.386	10.041	9.262

$\alpha_1 : \alpha_{GAComm2ETF,2ETF}$ $\alpha_2 : \alpha_{GAComm2ETF,2ETF}$

Table B.2: Improvements of GA on SSG, compared to the original heuristic that produces the corresponding processor assignments

Graphs	M_1		M_2		M_3		M_4		M_5		M_6	
	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2
1	2.556	2.600	1.759	1.790	1.836	1.878	2.821	2.794	1.855	1.943	1.939	2.110
2	2.644	2.584	1.629	1.714	1.712	1.792	2.799	2.784	1.816	1.878	1.866	2.059
3	2.706	2.778	1.825	1.878	1.966	1.917	3.041	3.016	2.003	1.997	2.192	2.221
4	2.619	2.606	1.680	1.783	1.744	1.839	3.040	2.990	1.792	1.968	1.888	2.114
5	2.539	2.593	1.667	1.681	1.765	1.867	2.852	2.866	1.819	1.901	1.940	2.071
6	2.694	2.752	1.803	1.854	1.941	1.925	3.024	3.041	1.974	1.947	2.096	2.098
7	2.625	2.733	1.617	1.761	1.707	1.852	2.925	2.958	1.713	2.004	1.931	2.174
8	2.235	2.226	1.473	1.518	1.593	1.639	2.429	2.405	1.562	1.591	1.711	1.768
9	2.409	2.420	1.652	1.582	1.678	1.747	2.608	2.616	1.624	1.787	1.777	1.965
10	2.886	2.882	1.940	1.958	2.047	2.031	3.200	3.197	1.995	2.214	2.235	2.372
11	3.311	3.287	1.739	1.895	1.732	1.890	5.173	5.229	2.810	3.183	2.814	3.127
12	3.356	3.388	1.837	1.980	1.827	1.991	5.211	5.266	2.961	3.258	2.892	3.382
13	3.368	3.430	1.663	1.814	1.670	1.809	5.616	5.380	2.683	3.089	2.772	3.143
14	3.344	3.451	1.711	1.862	1.681	1.865	5.195	5.451	2.809	3.086	2.647	3.183
15	3.333	3.354	1.815	1.910	1.770	1.910	5.308	5.301	2.809	3.182	2.957	3.224
16	3.355	3.382	1.762	1.903	1.766	1.889	5.414	5.222	2.904	3.180	2.861	3.136
17	3.401	3.417	1.672	1.787	1.680	1.775	5.356	5.502	2.710	3.016	2.671	3.088
18	3.222	3.265	1.707	1.845	1.738	1.840	4.951	4.823	2.726	3.044	2.788	3.114
19	3.411	3.348	1.815	1.941	1.791	1.963	5.406	5.327	2.933	3.204	3.063	3.232
20	3.350	3.440	1.763	1.858	1.776	1.878	5.411	5.527	2.899	3.068	2.784	3.158

$\delta_1 : \delta_{GAcomm2ETF}$ $\delta_2 : \delta_{GAcommETFR}$

Table B.3: GA Speedup of Communication Arrangements of R20L ($g = o$)

Graphs	M_1		M_2		M_3		M_4		M_5		M_6	
	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2
1	0.489	0.159	4.292	2.729	4.562	3.281	1.249	0.284	4.263	2.154	4.828	3.137
2	1.630	1.701	2.995	1.384	2.361	2.102	2.307	0.609	5.403	5.975	4.697	8.189
3	0.959	0.317	1.407	3.623	2.450	1.827	0.847	0.635	5.481	3.770	6.848	5.329
4	0.473	0.716	2.011	1.602	2.797	1.395	0.782	0.573	4.678	3.115	4.883	4.585
5	1.486	0.616	6.719	4.122	5.729	4.448	1.632	0.784	7.096	7.297	6.918	7.072
6	1.175	0.297	3.178	3.071	4.855	3.284	1.215	0.615	5.759	4.630	4.650	4.276
7	0.362	0.072	1.444	3.370	2.017	1.212	1.181	0.717	2.734	3.289	3.392	4.087
8	0.540	0.783	3.368	1.064	2.055	4.180	1.292	0.804	1.996	3.422	4.832	5.333
9	1.515	0.958	6.515	1.843	6.985	2.900	2.397	0.345	1.328	4.175	5.879	5.141
10	0.720	0.114	8.121	2.885	3.237	2.385	0.897	0.303	5.860	3.545	4.308	4.868
11	1.392	1.723	8.748	3.921	6.193	3.195	3.279	2.075	10.926	3.721	12.253	7.047
12	1.626	1.633	4.428	2.603	2.231	1.971	2.042	1.258	10.968	4.319	9.612	5.592
13	2.171	0.755	4.440	0.679	4.102	1.240	2.980	2.071	10.734	5.384	8.182	3.501
14	1.427	0.603	4.926	1.199	1.989	1.471	2.805	1.416	10.559	6.635	11.803	5.574
15	1.490	1.424	6.243	2.029	1.553	2.369	3.230	1.520	5.729	5.292	10.117	3.255
16	1.332	1.265	5.453	1.848	2.623	1.726	3.258	1.441	6.593	4.719	10.052	5.062
17	0.929	0.998	5.751	1.073	4.459	0.291	2.395	2.006	15.408	3.685	6.803	3.399
18	2.510	1.239	5.611	2.390	7.765	1.508	3.433	1.651	10.135	5.588	10.782	4.658
19	1.733	1.537	4.136	2.722	4.147	2.947	2.298	1.554	10.573	6.709	11.199	5.755
20	1.477	0.935	4.825	2.743	4.239	2.571	2.154	1.443	8.507	3.875	9.981	5.862

$\alpha_1 : \alpha_{GAComm2ETF,2ETF}$ $\alpha_2 : \alpha_{GAComm2ETF,2ETF}$

Table B.4: Improvements of GA on R20L, compared to the original heuristic that produces the corresponding processor assignments

Graphs	M_1		M_2		M_3		M_4		M_5		M_6	
	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2
r1p050	1.166	3.034	0.000	9.950	0.000	9.950	4.008	2.921	0.000	7.973	1.394	7.171
r1p100	3.105	5.519	5.704	14.342	0.490	14.342	6.154	4.592	14.804	10.013	11.999	10.479
r1p150	0.000	5.033	3.280	13.443	5.351	13.443	1.754	3.223	0.000	8.606	8.826	7.546
r1p200	0.555	5.286	7.583	14.113	5.106	14.126	1.826	5.263	15.526	6.013	19.157	5.972
r1p250	1.108	6.939	3.142	15.619	6.204	15.695	4.493	4.907	14.166	9.417	11.874	8.384
r3p050	2.361	3.596	1.956	11.696	6.469	15.590	3.979	1.639	16.355	13.383	18.641	14.650
r3p100	3.525	4.046	11.197	14.215	4.771	13.288	4.789	2.535	5.524	12.702	5.553	6.957
r3p150	1.362	3.603	5.772	13.680	7.601	15.161	3.259	2.493	14.157	10.342	15.386	10.107
r3p200	3.391	5.544	4.886	13.358	8.300	14.555	5.121	3.551	17.996	13.397	4.354	9.284
r3p250	0.698	5.192	6.220	14.533	8.669	14.180	4.379	4.254	19.666	7.936	8.338	6.792
r5p050	2.358	2.345	3.184	11.854	11.734	11.888	1.289	0.930	5.142	11.461	1.061	9.669
r5p100	1.264	3.189	9.232	15.591	4.283	15.353	4.530	2.797	7.403	10.794	11.956	9.573
r5p150	3.611	5.135	14.864	15.473	14.116	14.466	4.353	3.925	6.485	11.172	8.375	13.219
r5p200	2.418	5.971	8.728	14.839	5.731	14.462	4.929	2.784	12.698	10.128	10.879	8.753
r5p250	4.830	4.641	10.658	14.139	10.814	14.484	6.616	4.797	9.741	10.184	10.934	10.489

$\delta_1 : \delta_{GAcomm2ETF}$ $\delta_2 : \delta_{GAcommETFR}$

Table B.5: GA Speedup of Communication Arrangements of RGNOS ($g = o$)

Graphs	M_1		M_2		M_3		M_4		M_5		M_6	
	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2
r1p050	1.166	3.034	0.000	9.950	0.000	9.950	4.008	2.921	0.000	7.973	1.394	7.171
r1p100	3.105	5.519	5.704	14.342	0.490	14.342	6.154	4.592	14.804	10.013	11.999	10.479
r1p150	0.000	5.033	3.280	13.443	5.351	13.443	1.754	3.223	0.000	8.606	8.826	7.546
r1p200	0.555	5.286	7.583	14.113	5.106	14.126	1.826	5.263	15.526	6.013	19.157	5.972
r1p250	1.108	6.939	3.142	15.619	6.204	15.695	4.493	4.907	14.166	9.417	11.874	8.384
r3p050	2.361	3.596	1.956	11.696	6.469	15.590	3.979	1.639	16.355	13.383	18.641	14.650
r3p100	3.525	4.046	11.197	14.215	4.771	13.288	4.789	2.535	5.524	12.702	5.553	6.957
r3p150	1.362	3.603	5.772	13.680	7.601	15.161	3.259	2.493	14.157	10.342	15.386	10.107
r3p200	3.391	5.544	4.886	13.358	8.300	14.555	5.121	3.551	17.996	13.397	4.354	9.284
r3p250	0.698	5.192	6.220	14.533	8.669	14.180	4.379	4.254	19.666	7.936	8.338	6.792
r5p050	2.358	2.345	3.184	11.854	11.734	11.888	1.289	0.930	5.142	11.461	1.061	9.669
r5p100	1.264	3.189	9.232	15.591	4.283	15.353	4.530	2.797	7.403	10.794	11.956	9.573
r5p150	3.611	5.135	14.864	15.473	14.116	14.466	4.353	3.925	6.485	11.172	8.375	13.219
r5p200	2.418	5.971	8.728	14.839	5.731	14.462	4.929	2.784	12.698	10.128	10.879	8.753
r5p250	4.830	4.641	10.658	14.139	10.814	14.484	6.616	4.797	9.741	10.184	10.934	10.489

$\alpha_1 : \alpha_{GAComm2ETF,2ETF}$ $\alpha_2 : \alpha_{GAComm2ETF,2ETF}$

Table B.6: GA Improvements on Communication Arrangements of RGNOS ($g = o$)

Graphs	M'_1		M'_2		M'_3		M'_4		M'_5		M'_6	
	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2
1	0.764	0.753	1.178	1.211	1.616	1.670	0.837	0.775	1.273	1.198	1.771	1.822
2	0.733	0.660	1.152	1.122	1.578	1.622	0.795	0.758	1.240	1.194	1.661	1.770
3	0.915	0.782	1.379	1.159	1.858	1.723	0.862	0.804	1.339	1.226	1.865	1.876
4	0.687	0.650	1.099	1.095	1.595	1.645	0.669	0.672	1.104	1.088	1.686	1.826
5	0.738	0.645	1.147	1.082	1.596	1.650	0.760	0.742	1.204	1.127	1.674	1.781
6	0.797	0.720	1.260	1.123	1.747	1.746	0.813	0.800	1.291	1.187	1.836	1.838
7	0.663	0.644	1.074	1.072	1.544	1.630	0.709	0.690	1.134	1.077	1.611	1.809
8	0.685	0.615	1.053	0.974	1.437	1.420	0.725	0.632	1.112	1.017	1.568	1.500
9	0.692	0.701	1.085	1.050	1.538	1.538	0.721	0.714	1.147	1.091	1.659	1.643
10	0.861	0.851	1.347	1.373	1.882	1.882	0.989	0.902	1.496	1.365	2.038	2.034

$\delta_1 : \delta_{GAcomm2ETF}$ $\delta_2 : \delta_{GAcommETFR}$

Table B.7: GA Speedup of Communication Arrangements of R20L ($g > o$)

Graphs	M'_1		M'_2		M'_3		M'_4		M'_5		M'_6	
	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2
1	11.147	26.079	6.342	25.306	3.753	10.557	18.839	9.810	14.154	3.129	7.762	1.899
2	19.625	28.178	15.548	24.967	5.722	12.676	22.588	7.412	20.187	4.311	9.198	2.132
3	14.889	28.280	7.875	21.061	4.170	13.036	19.093	3.157	12.936	1.164	5.264	0.056
4	18.648	22.054	14.984	22.935	7.247	12.976	20.564	6.928	15.482	2.719	7.675	4.861
5	16.050	21.844	11.714	19.430	7.133	12.338	23.988	7.287	20.418	1.654	11.743	4.918
6	13.697	19.480	11.579	14.047	6.433	11.601	16.730	5.486	13.154	1.622	5.534	0.220
7	12.794	27.913	9.300	24.488	3.059	15.780	14.198	3.143	10.267	3.448	2.879	0.481
8	18.682	20.419	13.996	18.518	6.037	10.488	14.419	7.903	9.120	4.619	4.852	3.165
9	18.818	27.368	13.940	22.648	7.351	11.431	8.539	0.401	6.324	4.406	4.502	5.610
10	24.563	32.374	16.955	29.114	5.531	15.788	20.086	13.180	13.382	1.828	7.013	1.603

$\alpha_1 : \alpha_{GAComm2ETF,2ETF}$ $\alpha_2 : \alpha_{GACommETFR,ETFR}$

Table B.8: GA Improvements on Communication Arrangements of R20L ($g > o$)

Graphs	M_1		M_2		M_3		M_4		M_5		M_6	
	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2
n05	6.250	6.250	12.048	27.000	1.351	16.092	6.250	6.250	12.048	27.000	16.092	1.351
n10	12.088	8.046	23.438	56.054	21.600	47.594	12.088	8.046	23.438	56.054	47.594	21.600
n15	1.190	0.000	28.899	22.500	25.837	13.889	1.786	0.602	28.899	19.689	4.908	25.837
n20	3.333	6.452	8.281	30.059	5.471	34.621	3.333	4.396	6.368	32.169	30.918	8.917
n25	5.224	3.788	35.849	40.870	30.137	44.565	7.463	6.061	35.849	34.824	23.881	30.137
n30	6.515	2.321	33.411	40.000	31.159	36.667	6.429	0.701	30.657	21.918	21.271	24.202
n35	10.256	6.915	38.280	36.678	37.329	32.348	6.897	0.613	15.862	12.649	4.188	9.852
n40	8.000	9.360	33.861	36.474	32.905	35.593	5.882	3.030	15.385	10.300	10.493	12.552
n45	11.837	9.244	41.205	37.913	39.228	38.539	7.972	0.964	27.704	26.837	20.000	24.691
n50	11.278	10.943	36.197	34.778	32.321	33.690	3.653	4.091	12.696	18.209	15.386	6.396
n55	13.011	6.400	35.252	30.997	24.460	31.072	6.257	3.971	14.457	13.740	15.548	14.053
n60	8.710	5.980	43.986	43.376	39.942	42.061	7.169	6.060	30.263	17.403	11.238	29.492
n65	5.460	11.081	49.171	46.018	48.190	46.109	7.863	0.265	43.902	26.147	22.689	42.449
n70	7.527	7.775	55.487	52.375	54.217	51.051	5.556	0.730	34.423	29.561	26.701	32.270

$\alpha_1 : \alpha_{GAall,2ETF}$ $\alpha_2 : \alpha_{GAall,ETFR}$

Table B.9: Improvements of GA on Processor Allocations when Scheduling SSG ($g = o$)

Graphs	M_1	M_2	M_3	M_4	M_5	M_6
r1p050	3.586	2.003	2.001	6.221	3.147	3.130
r1p100	3.221	1.415	1.415	5.836	2.226	2.257
r1p150	3.111	1.053	1.053	5.716	1.771	1.749
r1p200	3.112	1.080	1.085	5.718	1.818	1.842
r1p250	2.777	0.800	0.796	4.993	1.338	1.314
r2p050	3.413	2.253	2.294	4.690	2.922	3.006
r2p100	3.522	2.055	2.065	6.100	3.051	3.070
r2p150	3.329	1.519	1.497	5.681	2.479	2.459
r2p200	2.877	0.857	0.850	5.007	1.411	1.391
r2p250	2.987	0.927	0.920	5.470	1.548	1.564
r3p050	3.492	2.108	2.136	5.164	3.028	3.114
r3p100	3.351	1.591	1.570	5.103	2.422	2.482
r3p150	3.178	1.285	1.300	5.237	1.997	2.049
r3p200	3.100	1.192	1.196	5.009	1.978	1.949
r3p250	3.117	1.084	1.086	5.455	1.804	1.754
r4p050	3.356	1.920	1.963	4.509	2.669	2.683
r4p100	3.292	1.675	1.665	5.089	2.525	2.531
r4p150	3.276	1.435	1.389	5.119	2.179	2.120
r4p200	3.192	1.216	1.250	5.336	2.029	1.959
r4p250	3.036	1.126	1.135	4.951	1.879	1.830
r5p050	3.052	1.999	1.984	3.739	2.427	2.548
r5p100	3.075	1.586	1.630	3.982	2.165	2.227
r5p150	2.942	1.282	1.299	4.486	1.860	1.872
r5p200	3.090	1.150	1.147	4.605	1.802	1.803
r5p250	3.107	1.119	1.135	5.252	1.915	1.916

Table B.10: GA Speedup on Processor Allocations when Scheduling RGNOS ($g = o$)

Graphs	M_1		M_2		M_3		M_4		M_5		M_6	
	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2	α_1	α_2
r1p050	1.671	7.876	14.613	15.622	13.146	15.199	12.084	1.416	16.457	17.259	13.870	16.206
r1p100	3.027	2.649	19.751	15.465	12.186	15.747	8.923	4.845	24.142	7.036	20.383	9.180
r1p150	4.683	5.753	10.583	14.369	12.891	14.591	2.737	0.258	16.806	6.429	12.917	6.608
r1p200	4.548	4.492	17.322	13.521	15.131	13.863	9.493	4.195	27.395	6.319	29.775	6.418
r1p250	4.288	6.175	11.727	15.258	16.840	15.376	8.853	3.465	25.869	7.216	23.710	5.360
r3p050	4.309	5.974	12.490	12.775	8.816	13.575	5.017	0.543	12.658	1.522	11.759	3.516
r3p100	3.296	4.849	22.361	14.699	14.269	13.353	8.649	1.330	17.930	7.984	23.372	10.200
r3p150	2.175	3.298	15.142	13.810	15.649	14.685	5.133	1.743	23.767	0.054	22.973	3.273
r3p200	5.223	6.708	13.304	13.331	16.636	14.355	10.647	3.259	27.381	12.491	21.916	4.138
r3p250	3.480	4.922	12.940	13.696	18.932	13.635	8.780	3.478	25.743	6.164	13.510	2.612
r5p050	2.905	3.422	13.639	12.768	18.651	11.411	4.696	1.785	15.098	0.317	17.708	1.340
r5p100	2.708	2.837	16.896	13.657	15.604	14.080	6.993	1.302	16.004	2.973	20.869	0.902
r5p150	6.074	4.159	20.659	12.052	23.418	12.714	6.932	3.167	23.075	3.349	22.239	1.341
r5p200	4.081	5.917	14.370	13.609	12.534	14.392	7.709	0.789	20.072	7.014	19.947	5.179
r5p250	6.552	5.535	15.606	13.348	19.555	13.687	7.289	4.375	21.401	9.540	23.101	8.326

Table B.11: Improvements of GA on Processor Allocations when Scheduling SSG ($g = o$)

Bibliography

[1] I. Ahmad and M. K. Dhodhi. Multiprocessor scheduling in a genetic paradigm. *Parallel Computing*, 22:395–406, 1996.

[2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[3] A. J. Anderson. *Multiple processing : a system overview.* Prentice/Hall International, 1989.

[4] A. Auyeung and a. H. Iker Gondra. Multi-heuristic list scheduling genetic algorithm for task scheduling. In *The* 18^{th} *Annual ACM Symposium on Applied Computing*, pages 721–724, 2003.

[5] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling Computer and Manufacturing Processes.* Springer-Verlag, Heidelberg, second edition, 2001.

[6] C. Boeres, G. N. da Cunha, and V. E. F. Rebello. On minimising the processor requirements of logp schedules. In *Euro-Par*, pages 156–165, 2001.

[7] C. Boeres, A. Nascimento, and V. E. F. Rebello. Scheduling arbitrary task graphs on logp machines. In *Euro-Par*, pages 340–349, 1999.

[8] C. Boeres, A. Nascimento, and V. E. F. Rebello. Towards an effective task clustering heuristic for logp machines. In *IPPS/SPDP Workshops*, pages 1065–1074, 1999.

[9] C. Boeres, V. E. F. Rebello, and D. B. Skillicorn. Static scheduling using task replication for logp and bsp models. In *Euro-Par*, pages 337–346, 1998.

[10] R. A. Brualdi. *Introductory Combinatorics.* Elsevier, New York, 1997.

[11] P. Brucker. *Scheduling Algorithms.* Springer-Verlag, Berlin, third edition, 2001.

[12] E. Coffman. *Computer and Job-Shop Scheduling Theory.* Wiley, New York, 1976.

[13] A. L. Corcoran and R. L. Wainwright. A parallel island model genetic algorithm for the multiprocessor scheduling problem. In *Selected Areas in Cryptography*, pages 483–487, 1994.

[14] R. Corrêa, 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] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[16] D.Ackley. An empirical study of bit vector function optimization. In *Genetic Algorithms and Simulated Annealing*, pages 170–215. Morgan Kaufmann, 1987.

[17] L. Finta and Z. Liu. Scheduling of parallel programs in single-bus multiprocessor systems. Technical Report RR-2302.

[18] L. Finta and Z. Liu. Complexity of task graph scheduling with fixed communication capacity. *International Journal of Foundations of Computer Science*, 8(1):43–, 1997.

[19] D. K. Friesen. Tighter bounds for lpt scheduling on uniform processors. *SIAM J. Comput.*, 16(3):554–560, 1987.

[20] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):686–701, 1993.

[21] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., 1989.

[22] M. Golub and S.Kasapovic. Scheduling multiprocesso tasks with genetic algorithms,. In *The International Conference Applied Informatics*, pages 273–278, Insbruck, Austria, February 18-21 2002.

[23] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[24] 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, Febuary 1994.

[25] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989.

[26] K. Hwang and F. A. Briggs. *Computer architecture and parallel processing.* McGraw-Hill, New York, 1984.

[27] T. Kalinowski, I. Kort, and D. Trystram. List scheduling of general task grphs under logp. *Parallel Computing, Special Issue on Scheduling Parallel and Distributed systems*, 26:1109–1128, 2000.

[28] A. A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *International Conference on Parallel Processing*, pages 243–250, 1994.

[29] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Longman Publishing Co., Inc., 3^{rd} edition, 1997.

[30] I. Kort and D. Trystram. Scheduling fork graphs under logp with an unbounded number of processors. In *Euro-Par*, pages 940–943, 1998.

[31] I. Kort and D. Trystram. Some results on scheduling flat trees in logp model. *Journal of Information Systems and Operational Research (INFOR)*, 37(1), 1999.

[32] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[33] C. Leopold. *Parallel and distributed computing.* Wiley-Interscience Publication, New York, N.Y., 2001.

[34] W. Löwe and W. Zimmermann. On finding optimal clusterings of task graphs. In *Parallel Algorithms/Architecture Synthesis pAs '95*, pages 241–247, 1995.

[35] W. Löwe and W. Zimmermann. Scheduling iterative programs onto logp-machine. In *Euro-Par*, pages 332–339, 1999.

[36] W. Löwe and W. Zimmermann. Scheduling balanced task-graphs to logp-machines. *Parallel Computing*, 26(9):1083–1108, 2000.

[37] W. Löwe, W. Zimmermann, and J. Eisenbiegler. On linear schedules for task-graphs for generalized logp-machines. In *Euro-Par*, pages 895–904, 1997.

[38] B. Maggs, L. Matheson, and R. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. of the 28th Hawaii International Conference on System Sciences (HICSS)*, volume 2, pages 61–70, Jan 1995.

[39] R. P. Martin, A. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *ISCA*, pages 85–97, 1997.

[40] K. Mehlhorn, S. Naher, and C. Uhrig. The LEDA platform of combinatorial and geometric computing. In *Automata, Languages and Programming*, pages 7–16, 1997.

[41] M. Middendorf, W. Löwe, and W. Zimmermann. Scheduling inverse trees under the communication model of the logp-machine. *Theor. Comput. Sci.*, 215(1-2):137–168, 1999.

[42] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, Massachusetts, 1998.

[43] K. Plateau, B. Bouvry, and P. Trystram. Andes: Evaluating mapping strategies with synthetic programs, 1997.

[44] S. C. S. Porto, J. P. F. W. Kitajima, and C. C. Ribeiro. Performance evaluation of a parallel tabu search task scheduling algorithm. *Parallel Computing*, 26(1):73–90, 2000.

[45] V. J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18(1):55–71, 1987.

[46] P. Rebreyend, F.E.Sandnes, and G.M.Megson. Static multiprocessor task graph scheduling in the genetic paradigm: A comparison of genotype representations. Technical report, Êcole Normale Supêrieure de Lyon, 1998.

[47] F. E. Sandnes and G. M. Megson. Improved static multiprocessor scheduling using cyclic task graphs: A genetic approach. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany*, volume 12, pages 703–710, Amsterdam, 1998. Elsevier, North-Holland.

[48] G. Sywerda. Uniform crossover in genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 2–9, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[49] A. T. C. Tam and C.-L. Wang. Realistic communication model for parallel computing on cluster. In *IWCC*, pages 92–, 1999.

[50] J. Verriet. Scheduling tree-structured programs in the logp model. Technical Report UU-CS-1997-18, Department of Computer Science, Utrecht University, Utrecht, the Netherlands, June 1997.

[51] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.

[52] W. Zimmermann, W. Löwe, and D. Trystram. On scheduling send-graphs and receive-graphs under the logp-model. *Information Processing Letters*, 82(2):83–92, 2002.

[53] W. Zimmermann, M. Middendorf, and W. Löwe. On optimal k-linear scheduling of tree-like graphs for logp-machines. In *Euro-Par*, pages 328–336, 1998.