MODELLING AND SCHEDULING OF HETEROGENEOUS COMPUTING SYSTEMS

LIU GUOQUAN

NATIONAL UNIVERSITY OF SINGAPORE

2005

MODELLING AND SCHEDULING OF HETEROGENEOUS COMPUTING SYSTEMS

LIU GUOQUAN

(M. Eng., Tsinghua University)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF INDUSTRIAL AND SYSTEMS ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgements

I would like to express my heartfelt gratitude to:

My supervisors, Associate Professor Poh Kim Leng and Associate Professor Xie Min, for both their guidance of my research work and their personal care.

Associate Professor Ong Hoon Liong and Dr Lee Chulung, for their helpful advice about the topics in this dissertation.

Mr. Dai Yuan Shun, for his advice and suggestion.

Mr. Zeng Yi Feng, for his suggestion and help.

All the other faculty members in the Department of Industrial and Systems Engineering, from whom I have learnt a lot through coursework, discussions and seminars.

I would also like to thank my wife Xie Zhaojing, my son Liu Yiyang and other family members for their hearty support, confidence and constant love on me.

Table of Contents

Summary

For most distributed computing systems (DCS), distributed system reliability (DSR) and the completion time of an application are the two most important requirements. To meet these requirements, it is essential that appropriate algorithms are developed for proper program and file allocation and scheduling. This dissertation focuses on the development of algorithms to maximize DSR and/or minimize the completion time based on more practical DCS models.

In almost all current reliability-oriented allocation models program and file allocation has been considered separately, rather than simultaneously. In this study a reliability–oriented allocation model was proposed, which considered the program and file allocation together so as to obtain the highest possible DSR. Certain constraints were also taken into account to make the model more practical. The model is very comprehensive and can be reduced to some other existing models under certain conditions.

To solve the NP-hard problem of simultaneous program and file allocation formulated herein, a Genetic Algorithm (GA) was proposed. To gauge the suitability of Tabu Search (TS) and GA for solving this problem, a TS was proposed and the results of TS were compared with those of GA. GA and TS were both found to be capable of finding the optimal solutions in most cases when the solution space was small. However TS outperformed GA with shorter computing time and better solution quality for both small and large solution space. Further improvements in performance over that of the TS were obtained by using a parallel TS (PTS). Simulation results showed that the solution quality did not change significantly with increased number of processors whereas the speedup of the PTS basically grew linearly when the number of processor was not very large.

Extensive algorithms have been proposed for the NP-hard problem of scheduling a parallel program to a DCS with the objective of minimizing the completion time of the program. Most of these, however, assumed that the DCS was homogeneous. An iterative list algorithm was proposed in this dissertation to solve the scheduling problem for the more difficult heterogeneous computing systems. Simulation results showed that the proposed algorithm outperformed most existed scheduling algorithms for heterogeneous computing in terms of the completion time of the application.

To consider DSR and completion time simultaneously, a multi-objective optimization problem was formulated and a Tabu Search algorithm proposed to solve the problem. Two “lateral interference” schemes were adopted to distribute the Pareto optimal solutions along the Pareto-front uniformly. Simulation results showed that “lateral interference” could improve the “uniform distribution of non-dominated solutions” and was not sensitive to the different computation schemes of distances between the solutions.

In addition, a general centralized heterogeneous distributed system model was formulated and a solution algorithm developed to compute the distributed service reliability.

Keywords:

Task Scheduling, Distributed Computing System Reliability, Genetic Algorithm, Tabu Search, Multi-objective Optimization, Reliability Analysis

List of Tables

List of Figures

List of Acronyms

CHDS:	Centralized heterogeneous distributed systems;
DAG:	Directed Acyclic Graph;
DCS:	Distributed Computing Systems;
DPR:	Distributed program reliability;
DSR:	Distributed System Reliability;
GEAR:	Generalized Evaluation Algorithm for Reliability;
GA:	Genetic Algorithm;
MFST:	Minimal File Spanning Tree;
PTS:	Parallel Tabu Search;
SA:	Simulated Annealing;
SL:	Schedule Length;
TS:	Tabu Search;
VM:	Virtual Machine.

List of Notations

$A(t)$:	availability function of VM at time *t*;
$c_{i,j,k,l}$:	communication time from task v_i to task v_j when task v_i was assigned to processor p_k and task v_j was assigned to processor p_l;
$c_{i,j}^{s}$:	time-weight of the directed edge from task v_i to task v_j during the *s*-th iteration, which is used to compute the priorities of the tasks;
C_b:	budget limit;
C_j:	cost for a copy of program P_j;
C_t:	completion time limit;
$d_{i,j}$:	data transfer size (in bytes) from task v_i to task v_j;
DSR_i:	DSR of *i*-th sub-distributed system;
DSR_{best}:	DSR of x_{best};
$DSR_{tempbest}$:	DSR of $x_{tempbest}$;
e:	number of directed links among the tasks;
$e_{i,j}$:	directed link from *i*-th task to *j*-th task;
$EFT(v_i, p_j)$:	earliest computation finish time of task v_i on processor p_j;
$EST(v_i, p_j)$:	earliest computation start time of task v_i on processor p_j;
F_j:	*j*-th distributed files;
$l_{i,j}$:	direct link between processor p_i and processor p_j;
N_n:	number of nodes;
N_F:	number of files;

N_P:	number of programs;
NF_{ij}:	assignment of file F_j on the node N_i;
NN_{ij}:	link between nodes N_i and N_j;
NP_{ij}:	assignment of program P_j on node N_i;
p:	number of processors available in the system;
p_i:	*i*-th processor in the system;
P_j:	*j*-th computing program;
$P_0(t)$:	probability for the VM in working state at time *t*;
$P_1(t)$:	probability for VM in malfunctioning state at time *t*;
$\overset{\rightarrow}{PF}_j$:	set of files required by the *j*-th program P_j;
$r_{i,j}$:	communication rate (in bytes/second) between processor p_i and processor p_j;
R_i:	reliability of the processor p_i, which is the probability that processor p_i is functional;
$R_{i,j}$:	reliability of the directly link $l_{i,j}$ between processor p_i and processor p_j;
$R_s(t_b)$:	distributed service reliability function of t_b;
S:	current program and file set;
S_{best}:	program and file set where x_{best} was found;
S_j:	size of the *j*-th file F_j;
SC_i:	storage limit of node N_i;
t_b:	initial time for the service;
T_{bf}^{j}:	time point for the *j-th* programs need the files prepared in the VM;

T_{bp}^{k} :	beginning time when the *k-th* programs runs in VM;
T_{ex}^{k} :	execution time period for those programs in VM;
T_{ij}:	completion time of program P_j at node N_i;
TL_N :	Tabu List of program and file set;
UR :	distributed computing system unreliability;
v:	number of tasks in the application;
v_i :	*i*-th task in the application;
VM_i :	VM used in subsystem.
$w_{i,j}$:	computation time to complete task v_i on processor p_i ;
w_i^s :	time-weight of task v_i during the *s*-th iteration, which is used to compute the priorities of the tasks;
x :	current solution;
x_{best} :	the best solution found until now;
$x_{tempbest}$:	the temporarily best solution;
λ_i :	failure rate of processor p_i ;
$\lambda_{i,j}$:	failure rate of link $l_{i,j}$ between processor p_i and processor p_j .

Chapter 1

Introduction

A distributed computing system (DCS) consists of a collection of autonomous computers/processors linked by a network, with software designed to produce an integrated computing facility (Coulouris & Dollimore 2000). In such a system, an application consists of several tasks/programs. (In this dissertation, task and program, and computer and processor are used interchangeably for consistency with the literature.) The tasks may be executed on the different computers. Two communicating tasks executing on different computers communicate with each other using the system's network, thereby incurring communication cost. Communication costs are also incurred when some tasks need to access files on different computers.

Distributed computing has attracted more and more research effort over the last two decades as its performance-price ratio and flexibility exceeds that of supercomputers. The past decade has witnessed an ever-increasing demand for and the practice of high performance computing driven by powerful DCSs.

Compared with supercomputers, DCSs generally provide significant advantages, such as better performance, better reliability, better performance-price ratio and better scalability (Coulouris & Dollimore 2000). Performance (e.g., completion time) and reliability are essential requirements for most DCSs (Shatz *et al.* 1992), and to meet these requirements, it is important to employ a good algorithm for proper program and file allocation and scheduling.

In a homogeneous system, the computation times of a task on different processors are the same, and the communication times between two tasks on different processors are also the same. A heterogeneous DCS has several advantages over a homogeneous DCS. A heterogeneous DCS is a suite of diverse high-performance machines interconnected by high-speed links, so it can perform different computationally intensive applications that have diverse computational requirements. As the allocation and scheduling for a heterogeneous DCS are more difficult than that for a homogeneous one, most scheduling algorithms for DCSs assume that the distributed systems are homogeneous.

This dissertation focuses on scheduling, allocation algorithms for heterogeneous DCSs to meet certain criteria, for example maximum reliability and minimum completion time. At the same time, computing the DCSs' reliability is the prerequisite of reliability-oriented allocation and scheduling, so the computation and analysis of the reliability is also considered.

1.1 The problems & methodologies

Increasingly, DCSs are being employed for critical applications, such as aircraft control, banking systems and industrial process control. For these applications, ensuring system reliability is of critical importance. DCSs are inherently more complex than centralized computing systems, which could increase the potential for system faults. The traditional technique for increasing the distributed system reliability (DSR) is to provide hardware redundancy. However, this is an expensive approach. Moreover, many times, the hardware configuration is fixed. When the hardware configuration is fixed, the system reliability depends mainly on the assignment of various resources such as programs and files (Kumar *et al.* 1986, Raghavendra *et al.* 1988). Extensive program allocation or file allocation algorithms have been proposed to maximize the DSR. However most previous studies considered the program and file allocation problems separately rather than simultaneously as the optimum method. In addition, to make the allocation model more practical, certain constraints need to be taken into account.

In this dissertation, a more practical program and file allocation model was constructed by including constraints on program cost, file storage, and completion time. This model is very comprehensive and can degenerate to some other models in certain circumstances.

Reliability-oriented program allocation and file allocation are both NP-hard problems. Considering the program and file together and taking into account these constraints make the problem harder. A Genetic Algorithm (GA) was therefore proposed to solve the problem. GA's are inspired by Darwin's theory of evolution based on the survival of the fittest species as introduced by Holland (1977) and further described by Goldberg (1989). GA is a meta-heuristic that is easy to model and be applied to various optimization problems.

As this problem has constraints, the solution produced by GA is sometimes not feasible. Dealing with infeasible solutions needs extra computational effort and may impact the quality of solution. In this case, adjustments were applied to deal with the infeasible solutions.

Tabu Search (TS) (Glover 1989, 1990) is another meta-heuristic method used for many large and complex combinatorial optimization problems. This can usually produce quite good solutions although the algorithm is more complicated to implement. A TS was therefore proposed to solve the same problem and the results of TS were compared with those of GA. Simulation results show that TS outperforms GA in this case.

In practical situations, scheduling must be completed within a short time interval, and therefore a parallel TS was proposed to solve the problem and to further improve the performance of TS.

As the completion time is another important goal for distributed computing, the scheduling of parallel applications to minimize the completion time is very important in a DCS. An application consists of a number of tasks which may have dependencies. The scheduling problems are NP-hard in the general case (Gary & Johnson 1979); extensive heuristic scheduling algorithms have been proposed to minimize the completion time (schedule length) (Kwok & Ahmad 1999b). However, most of the existing task scheduling algorithms either assume that the DCSs are homogeneous or are high-complexity algorithms.

In this dissertation, a low-complexity algorithm for heterogeneous DCS was proposed to maximize the schedule length and the performance tested on randomly generated application graphs and some real world application graphs.

Maximizing the DSR and minimizing the schedule length are two major objectives of scheduling for DCSs. Most research has considered these two objectives separately although ideally they should be considered simultaneously. Some researchers proposed considering one of them as a constraint. However, it is very difficult to estimate a value for DSR or schedule length as the limitation. Hence, in this dissertation, Pareto's optimality concept was used to obtain a set of solutions rather than a single solution, and a TS algorithm was presented to solve the problem.

Analysis and computation of DSR is the prerequisite for reliability oriented allocation and scheduling. Several reliability measures have been studied by the researchers in the context of DCSs. For example, Raghavendra *et al* (1988) first introduced the distributed program reliability (DPR) and DSR. DPR is a measure of the probability that a given program can run successfully and be able to access all the required files from remote sites in spite of faults occurring in the processing elements and the communication links. DSR is the probability that all the given distributed programs can run successfully.

Most of these measurements cannot be simply implemented to analyze the service reliability of a centralized heterogeneous distributed system, designed and developed to provide certain important services, as it is affected by many factors including system availability and distributed program/system reliability. This dissertation studied the properties of centralized heterogeneous distributed systems and developed a general model for the analysis. Based on this model, an algorithm to obtain the service reliability of the system was also developed.

1.2 Contributions

This section briefly summarizes the major contributions of the work described in this dissertation.

The dissertation presents a more practical reliability–oriented allocation model, which considers the program and file allocation together and takes into account certain constraints such as program cost, file storage and completion time. This model, compared to previous models, is more practical, more comprehensive and can degenerate to some other models.

A GA is proposed to solve this NP-hard problem. Inappropriately dealing with unfeasible solutions may impact the quality of solutions. In this case, adjustments are applied to deal with the infeasible solutions. A TS is also designed to find optimal or near optimal solutions, and the results of GA and TS are compared to gauge their suitability for solving this problem. The numerical results show that in this case TS outperforms GA with shorter computing time and better solution quality. Comparison of results for this and other cases suggests that, if we have good knowledge of the state space, TS should be used; if not, then GA may be a better choice.

In certain practical situations scheduling must be achieved within a short time interval. Therefore to further improve the performance of the TS in this respect, a parallel TS is proposed to solve the same problem. The speedup of the parallel TS grows linearly with increase in number of processors without adversely affecting the solution quality, when the number of processors is not very large. This runs contrary to the common opinion that TS is not suitable for parallelization due to the sequential inherence of TS.

To minimize the completion time (schedule length), this dissertation proposes an iterative list scheduling algorithm for heterogonous DCSs. Simulation results, based on randomly generated application graphs as well as real applications, showed that in most cases the proposed algorithm obtained shorter schedule length compared with previous algorithms.

To maximize the systems reliability and minimize the schedule length simultaneously, a TS algorithm is used to obtain a set of solutions by means of the Pareto optimality concept. In addition, “lateral interference” is adopted to investigate two schemes to distribute the Pareto optimal solutions along the Pareto-front uniformly. The results show that “lateral interference” can improve the “uniform distribution of non-dominated solutions” and is neither sensitive to the different computation schemes nor to distances between the solutions.

To compute the distributed service reliability, a prerequisite for the reliability oriented allocation and scheduling, a centralized heterogeneous distributed system model and an algorithm, which first analyzes the service reliability of the system, are proposed.

1.3 Organization of the dissertation

This chapter has given a brief introduction to some basic concepts in allocation and scheduling for DCS, reviewed some major work related to the topics addressed in this dissertation and described the methodologies used.

The rest of this dissertation is arranged as the following:

Chapter 2 introduces related works involving DSR computation algorithms, reliability oriented program and file allocation algorithms, completion time oriented task scheduling algorithms, and multi-objective optimization.

Chapter 3 presents a reliability-oriented optimization model with storage, cost and completion time constraints in which program allocation and file allocation are considered together, and a GA is proposed to solve the problem.

Chapter 4 proposes a TS to solve the same problem presented in Chapter 3 and compares the results of TS and those of GA. In addition, to further improve the performance of the TS, a parallel TS (PTS) is proposed and the performance of PTS is analyzed by simulation.

Chapter 5 presents an iterative list scheduling algorithm to minimize the completion time which, together with DSR considered in Chapters 3 and 4, are the two most important requirements for heterogeneous DCSs. The proposed algorithm can obtain high quality solution with low time complexity.

Chapter 6 describes a scheduling model to maximize DSR and minimize the completion time, considered in Chapters 3 – 5, simultaneously. A TS algorithm was used to obtain a set of Pareto optimal solutions and a number of measurements adopted to distribute solutions along the Pareto surface uniformly.

Chapter 7 focuses on how to analyze and compute the reliability for centralized heterogeneous DCSs, this being a prerequisite for the reliability oriented allocation algorithms.

Chapter 8 summarizes this dissertation by discussing the contributions and limitations of the whole work. It also suggests some possible directions for future research.

Chapter 2

Literature Review

This chapter briefly surveys related work on distributed computing system reliability (DSR) evaluation, Reliability oriented task and file allocation, Completion time (Schedule length) oriented scheduling algorithms and Multi-objective optimization.

2.1 Distributed computing system reliability evaluation

Researchers have developed several reliability measures. Merwin & Mirhakak (1980) defined a survivability index S to measure survival in terms of the number of programs that remain executable in the DCS after some nodes or links become inoperative. The survivability index, however, is not applicable to large distributed systems because of the large computing time required (Martin & Millo 1986).

Aggarwal & Rai (1981) defined the network reliability for a computer-communication network and proposed a method based on spanning trees to evaluate the network reliability.

Satyanarayana (1982) proposed a source-to-multiple-terminal reliability (SMT Reliability), i.e. derived a topological formula to solve a variety of network reliability problems. The formula considered the unreliability of vertices and links, and with failure events s-independent or not. The formula, however, involves only non-cancelling terms although it explicitly characterizes the structure of both cancelling and non-cancelling terms in the reliability expression obtained by inclusion-exclusion.

Computer network reliability and SMT reliability are good reliability measures for computer communication network networks, but neither of them considers the effects of redundancy of programs and files in the distributed system. This issue was considered by Raghavendra *et al.* (1988) who developed an efficient approach based on graph traversal to evaluate distributed program reliability (DPR) and distributed system reliability (DSR).

DPR is the probability that a given program can run successfully and be able to access all the required files from remote sites in spite of faults occurring among the processing elements and the communication links. DSR is the probability that all the given distributed programs can run successfully.

Kumar *et al.* (1986) presented a Minimum File Spanning Trees (MFST) algorithm to compute DSR. The MFST is 2-step process:

- Step 1 computes all MFST,
- Step 2 converts these MFST's to a symbolic reliability expression.

The MFST's major drawback is that it is computationally complex and prior knowledge about multi-terminal connections is needed. To improve the MFST algorithm, Kumar *et al* (1988) developed an algorithm called Fast Algorithm for Reliability Evaluation (FARE) that does not require an a priori knowledge of multi-terminal connections for computing the reliability expression. The FARE algorithm uses a connection matrix to represent each MFST and proposes some simplified techniques for speeding up the analysis process.

Chen & Huang (1992) proposed the FST-SPR algorithm that further improved the evaluation speed by reducing the number of subgraphs generated during reliability evaluation. The basic idea of the FST-SPR is to make the subgraphs generated completely disjointed, so that no replicated subgraphs are generated during the reliability evaluation process. Chen *et al* (1997) proposed another algorithm: HRFST that does not need to search a spanning tree during each subgraph generation.

MFST's drawbacks were also alleviated by the Generalized Evaluation Algorithm for Reliability (GEAR) (Kumar & Agrawal 1993). GEAR is a 1-step algorithm that can compute the terminal-pair reliability, computer-network reliability, distributed program reliability and DSR. It is also more efficient than the MFST.

Chen & Lin (1994) presented an algorithm for computing the DSR - the Fast Reliability Evaluation Algorithm (FREA) that is based on a factoring theorem employing several reliability preserving reduction techniques. Compared with existing algorithms on various network topologies, file distributions, and program distributions, FREA is much more economical in both time and space.

Chang *et al.* (1999) proposed a polynomial-time algorithm to analyze the DPR of ring topology and showed that solving the DPR problem on a ring of trees topology is NP-hard. Later, Chang *et al.* (2000) developed a polynomially solvable case to compute DPR when some additional file distribution is restricted on the star topology which is NP-hard.

Lin (2003) presented two linear-time algorithms to compute the reliability of two restricted subclasses of DCSs with star topology. There are $|V|$ nodes and $|F|$ files in the DCS. The first algorithm runs in $O(|F|)$ when the file distribution is limited to being bipartite and non-separable. The second algorithm runs in O(|V|), when each file is allocated to no more than two distinct nodes and each node contains at most two distinct records. If the failure and working probabilities of every node are identical, then the computation can be accelerated to $O(\log|V|)$ time by means of the Fibonacci number and the Lucas number.

2.2 Reliability oriented task and file allocation

The reliability oriented task allocation problem can be stated as follows:

> Given an application consisting of *m* tasks and a DCS with *n* processors, allocate each of the tasks to one or more of the processors such that the system reliability is maximized subject to certain resource limitations and constraints imposed by the application or environment.

In the reliability oriented task allocation model, Bannister & Trivedi (1983) achieved optimization by balancing the load over a homogeneous system. However, their model does not consider failures of communication links and does not give an explicit system reliability measure. Hariri and Raghavendra (1986) considered that the reliability was maximized and the communication delay was minimized. They also considered the problem of task allocation for reliability by introducing multiple copies of tasks, but did not give an explicit reliability expression. In addition, their algorithm assumes that all the processors and communication links have the same reliability and each processor runs exactly one task.

Hwang and Tseng (1993) proposed a heuristic algorithm for reliability-oriented design of a distributed information system to the *k* copies of the distributed tasks assignment (k-DTA) problem. In Shatz *et al.* (1992)'s task allocation model, a cost function represents the unreliability caused by execution of tasks on processors of various reliability and by interprocessor communication. An A* algorithm is applied to do the state space search. This algorithm may be "trapped" in local minima which prevent the search from yielding an optimal solution. Kartik & Murthy (1995) further reduced the size of the search space by finding a set of mutually *s*-independent (non-communicating) tasks. Compared with the algorithm of Shatz *et al.* (1992) that of Kartik & Murthy (1997) can produce optimal allocations at all times and reduces the computations by using the ideas of branch-and-bound with underestimates and task independence.

The models of Shatz *et al.* (1992), Kartik & Murthy (1995) and Kartik & Murthy (1997) do not include the concept of a task requiring access to a number of data files. However this concept is considered in the model of Tom & Murthy (1998). Mahmood (2001) presented a least-cost branch-and-bound algorithm to find optimal task allocations and two heuristic algorithms to obtain sub-optimal allocations for realistically sized large problems in a reasonable amount of computational time.

Vidyarthi & Tripathi (2001) proposed a genetic algorithm based task allocation to maximize the reliability of the distributed system. The GA showed a better result than that of Shatz *et al.* (1992) in terms of the system reliability.

Chiu *et al.* (2002) developed a heuristic algorithm for *k*-DTA reliability oriented task allocation problem. The simulation shows that, in most test cases with one copy, the algorithm finds sub-optimal solutions efficiently. Even when the algorithm cannot obtain an optimal solution, the deviation is very small.

The distribution of data files can also impact on the reliability of distributed systems (Dowdy & Foster 1982). Pathak *et al.* (1991) developed a genetic algorithm (GA) to solve file allocation problems so as to maximize the reliability of distributed program(s). In this scheme, the different constraints are discussed, for example, the total number of copies of each file and the memory constraint at each node.

Pathak *et al.* (1991) also found that beyond a certain point, increasing the redundancy of files could not improve the reliability of the DCS. Kumar *et al.* (1995a) developed a genetic algorithm (GA) to solve the reliability oriented file allocation problem for distributed systems, and the proposed method was compared with optimal solutions to demonstrate the accuracy of the solution obtained from GA based methodology. Kumar *et al.* (1995a) also provided the relation between degree of redundancy of files and the maximum achievable reliability of executing a program. They showed that the redundancy is helpful in improving the reliability only up to a certain point. Beyond this point, no significant improvement in the reliability is achieved by increasing the redundancy of the files.

There are some file allocation problems with other objectives. Murthy & Ghosh (1993) formulated a file allocation model that sought to obtain the lowest cost file allocation strategy and to ensure the attainment of acceptable levels of response times during peak demand periods, for all on-line queries. Chang *et al.* (2001) addressed a files allocation problem in DCS's to minimize the expected data transfer time for a specific program that must access several data files from non-perfect computer sites.

In addition, there has been some research on increasing system availability (Lutfiyya *et al.* 2000). Goel and Soejoto (1981) first considered the performance of a combined software and hardware system. A generalized model has also been proposed in Sumita and Masuda (1986). Markov models are also implemented to analyze the system availability, combining both software and hardware failures and maintenance processes (Welke *et al.* 1995, Lai *et al.* 2002).

2.3 Schedule length oriented task scheduling algorithms

The general task scheduling problem includes the problem of assigning the tasks of an application to suitable processors and the problem of ordering task execution on each processor. When the parameters such as execution times of tasks, the data size of communication between tasks, and task dependencies, are known a priori, the problem is static scheduling.

2.3.1 Static scheduling

Static scheduling is utilized in many different types of analyses and environments. The most common use of static scheduling is for predictive analyses. Sometimes it is also used for post-mortem analyses. In static scheduling, information about the processor and about the tasks is assumed available. Extensive work has been done on static scheduling. The problem is known to be NP-hard in general form (Coffman 1976).

In the general form of a static task scheduling problem, an application can be represented by a directed acyclic graph (DAG) in which nodes denote tasks and directed edges denote data dependencies among the tasks. A task may have one or more inputs. When all inputs are available, the task is triggered to execute. After its execution, it generates its outputs. If there is a directed edge from task v_i to task v_j, task v_i is the parent of task v_j and task v_j is the child of task v_i. A task with no parent is called an entry task and a task with no child is called an exit task. Every task has a weight called the computation cost of the task, and every edge has a weight called communication cost of the edge. The communication cost is incurred if the two tasks are scheduled on different processors; otherwise the communication cost is zero.

Some researchers used graph theory methods (Bokhari 1979; Bokhari 1981; Stone 1977; Stone 1978; Stone & Bokhari 1978). Chu *et al.* (1980), and Chern *et al.* (1989) used the integer 0-1 programming techniques to solve the resource allocation problem. However, heuristic methods are the most prevalent ones to solve task scheduling. Typical heuristic approaches include:

2.3.1.1 List scheduling algorithms

List scheduling algorithms include Insertion Scheduling Heuristics (Hwang *et al.* 1989), Modified Critical Path (Wu & Gajski, 1990), Mapping Heuristics (El-Rewini & Lewis 1990), Dynamic Critical Path (Sih & Lee 1993), Hybrid Mapper (Matheswaran & Siegel 1998), and Heterogeneous Earliest Finish Time (Topcuoglu *et al.* 2002), *etc.*

The basic idea of list scheduling is to assign priorities to the tasks and to place the tasks in a list arranged in descending order of priorities. The task with a higher priority is scheduled before a task with a lower priority. The task is assigned to a suitable processor to minimize a predefined cost function.

t-level (top level) and *b-level* (bottom level) are two major attributes for assigning priorities. The *t-level* of a task v_i is the length of the longest path from an entry task to v_i in the DAG (excluding v_i). Here, the length of a path is the sum of all the task and edge weights along the path. The *b-level* of a task v_i is the length of the longest path from task v_i to an exit task. Because the edge weight may be zero when the two tasks are assigned to the same processor, the *t-level* and *b-level* of a task are dynamic attributes. Some scheduling algorithms do not take into account the edge weights in computing the b-level, which is referred to as *static b-level* or simply *static level*. Another important concept is *critical path* (CP), which is a path from an entry node to an exit node whose length is the maximum.

The *t-level* and *b-level* attributes are used in various ways to assign a task a higher priority. A higher priority can be a smaller *static level* (El-Rewini & Lewis 1990), a smaller *t-level*, a larger *b-level* (Topcuoglu *et al.* 2002), a larger (*b-level* - *t-level*) or a smaller (*t-level* - *b-level*) (Wu & Gajski 1990).

During the processor selection phase, task v_i is assigned to the suitable processor so that the earliest start time (Wu & Gajski 1990) or earliest finish time of task v_i (Topcuoglu *et al.* 2002) is minimized. The earliest start time of task v_i on processor p_j is decided by two items: the ready-time of task v_i and the earliest available time of processor p_j. The ready-time of task v_i is the time when all data needed by task v_i have arrived at processor p_j. When determining the earliest available time of processor p_j, some algorithms only consider scheduling a task after the last task on processor p_j. Some algorithms also consider the idle time slots on processor p_j and may insert a task between two already scheduled tasks (Topcuoglu *et al.* 2002), which still satisfy the data dependency.

Some algorithms just order the ready tasks instead of whole tasks. The ready tasks are those that whose parent tasks have been scheduled. The Earliest Time First (ETF) algorithm (Hwang *et al.* 1989) computes the earliest start times for all ready tasks and then selects the one with the smallest start time. The earliest start time of a task is the smallest one among the start time of the task on all processors. This algorithm uses the *static level* to break the tie of two tasks.

The following algorithms have been developed for heterogeneous environments:

Mapping Heuristic (MH) (El-Rewini & Lewis 1990) initializes a ready task list ordered in decreasing *static level* and each task is scheduled to a processor that allows the earliest start time. The algorithm takes into account the heterogeneity during the scheduling process, but assumes that the environment is homogeneous when computing the computation time of tasks and the communication time. When communication contention is considered, the time complexity is $O(v^2 p^3)$ for v tasks and p processors; otherwise, it is $O(v^2 p)$.

Dynamic Level Scheduling (DLS) algorithm (Sih & Lee 1993) computes the dynamic levels (DL) for all ready tasks. DL is the difference between the *static level* of a task and its earliest start time on a processor, so every task has several DL's. At each step, the ready task-processor pair that maximizes DL is chosen for scheduling. When it computes the *static level*, the computation time of a task is the median value of the computation times of a task on the processors. The time complexity is $O(v^3 p)$ for v tasks and p processors.

Levelized-Min Time (LMT) algorithm (Iverson *et al.* 1995) uses the so-called level to sort tasks. A task in a lower level has higher priority than a task in a higher level. Within the same level, the task with higher computation time has higher priority. Then, the algorithm assigns the task to a processor so that the summation of the task's computation time and transfer time taken by all the required data for this task is minimum. For a fully connected DAG, the time complexity is $O(v^2 p^2)$ for v tasks and p processors.

Heterogeneous Earliest-Finish-Time (HEFT) algorithm (Topcuoglu *et al.* 2002) significantly outperforms DLS, MH and LMT in terms of average schedule length ratio, speedup, etc. The HEFT algorithm selects the task with the highest *b-level* value at each step and assigns the selected task to the processor that minimizes its earliest finish time with an insert-based approach. When computing the priorities, the algorithm uses the task's average computation time on all processors and the average communication rates on all links. The time complexity is $O(ep)$ for e edges and p processors. For a dense graph, the time complexity is $O(v^2 p)$ for v tasks and p processors.

2.3.1.2 State space search reduction algorithms

Shen &Tsai (1985) treated the task assignment as a graph-matching problem and used a state-space search method – A^* algorithm to solve it. However, their model did not consider the precedence relations between tasks. Wang & Tsai (1988) consider the precedence relations between tasks into the model. Ajith & Murthy (1999) also used a state space technique – A^* algorithm to obtain an optimal allocation designed to minimize the total turnaround time of all tasks. In Tom & Murthy (1999)'s method, the state space search can be drastically reduced by scheduling independent tasks last.

2.3.1.3 Metaheuristic algorithms

Tripathi *et al.* (1996) presented a genetic task allocation algorithm for DCS. In this algorithm, how to improve the initial population structures of GA's is discussed by finding that the incorporation of the problem specific knowledge into the construction. Budenske *et al.* (1997) presented a GA for real-time on-line input-data dependent remappings of the tasks to the processors in the heterogeneous hardware platform using previously stored and off-line statically determined mappings. Kwok & Ahmad (1997) proposed a parallel GA-based algorithm with an objective to simultaneously meet the goals of high performance, scalability, and fast running time. Ignatius & Murthy (1997) presented an efficient heuristic algorithm based on simulated annealing (SA) for solving the task allocation problem in DCSs.

2.3.1.4 **Clustering algorithms** (Sarkar 1989, Wu & Gajski 1990, Gerasoulis & Yang 1992, Kim & Yi 1994, Yang & Gerasoulis 1994, Kwok & Ahmad 1996, Palis *et al.* 1996, Srinivasan & Jha 1999)

This group of algorithms maps the tasks to an unlimited number of clusters (UNC). The basic idea of clustering algorithms is that, at the beginning of the scheduling process, each node is considered as a cluster. In the subsequent steps, two clusters are merged if the merging reduces the completion time. This merging procedure continues until no cluster can be merged. The rationale behind the algorithms is that they can take advantage of using more processors to further reduce the schedule length. However, the clusters generated by the algorithm may need a post-processing step for mapping the clusters onto the processors because the number of processors available may be less than the number of clusters.

2.3.2 Dynamic scheduling

The dynamic scheduling heuristics can be grouped into two categories: on-line mode and batch-mode heuristics. Both on-line and batch mode heuristics assume that estimated expected task execution times on each machine in the computing system are known (Ghafoor & Yang 1993, Kafil & Ahmad 1998).

Chen *et al.* (1988) proposed a heuristic search algorithm called "dynamic highest level first/most immediate successors first" (DHLF/MISF) to find a fast but sub-optimal schedule. In this algorithm, the A* algorithm coupled with an efficient heuristic function is bound to achieve a minimum-schedule length.

Sih & Lee (1993) presented a technique to use dynamically-changing priorities to match tasks with processors at each step, and schedules over both spatial and temporal dimensions to eliminate shared resource contention.

Zomaya & Teh (2001) developed a dynamic load-balancing genetic algorithm to search optimal or near-optimal task allocations during the operation of the parallel computing system. The algorithm considers other load-balancing issues such as threshold policies, information exchange criteria, and interprocessor communication.

2.3.3 Genetic Algorithm, Tabu Search and Simulated Annealing and their applications

There is one class of combinatorial optimization algorithms: general iterative algorithms. Because of their ease of implementation and robustness in solving various problems, more and more researchers use this kind of method to solve the combinatorial optimization problems. We introduce three popular iterative algorithms: Genetic Algorithm, Tabu Search and Simulated Annealing.

Genetic Algorithm (GA) is a search algorithm inspired by the mechanism of evolution and natural genetics (Holland 1975, Goldberg 1989). GA starts with initial *population* and an individual in the *population* is a string of symbols and is an abstract representation of the solution. The symbol is called a *gene* and each string of genes is termed a *chromosome*. The individuals in the population are evaluated by some *fitness* measure. The population of chromosomes evolves from one *generation* to the next through the use of two types of genetic operators: (1) *mutation* which alters the genetic structure of a single chromosome, and (2) *crossover* which obtains a new individual by combining genes from two selected parent chromosomes. Based on the *fitness* value, two individuals (*parents*) are selected from the population. The genetic operators (*crossover and mutation*) are applied to the selected parents with some probability to generate new possible solutions called *offsprings*. The performance of GA depends largely on: 1) the representation of the solution to the problem, 2) parameter selection (population size and probabilities of crossover, mutation), 3) crossover and mutation mechanism. Genetic Algorithm (GA) has the following features. Firstly, GA guides its search by evaluating the fitness of each solution instead of the optimization function. Hence, we can implement GA's to some problems the state space of which we are not familiar with. Secondly, the algorithm is a multi-path approach that searches many peaks in parallel, hence reducing the possibility of local minimum trapping. Thirdly, GA explores the search space where the probability of finding improved performance is high.

Kumar *et al.* (1995b) developed a GA for network topology design to maximize the network reliability under different network constraints. The problems solved here deal with optimizing those network parameters that characterize the network reliability. Sena *et al.* (2001) presented a parallel version of a Genetic Algorithm and implemented it on a cluster of workstations to obtain optimal and/or sub-optimal solutions to the well-known Traveling Salesman Problem.

Tabu Search (TS) is a higher-level method for solving combinatorial optimization problems (Glover 1989, 1990). TS starts from an initial feasible solution, makes several neighborhood *moves* and then selects the *move* producing the best solution while keeping track of the regions of the solution space which have already been searched so as not to repeat a search near these areas. In other words, TS uses the memory to preserve a number of previously visited states along with a number of states that might be considered unwanted. This information is stored in a Tabu List. The performance of TS depends largely on 1) encoding of state space, 2) choice of the neighborhood structure, 3) length of the Tabu list. These parameters are usually difficult to select. In addition to the above Tabu parameters, two extra parameters are often used: Intensification and Diversification. Intensification is to encourage move combinations and solution features historically found good. They may also initiate a return to attractive regions to search them more thoroughly. Diversification adds randomness to this otherwise deterministic search.

Pierre & Elgibaoui (1997) used a Tabu Search to search the sub-optimal solutions for network topology design to minimize the total communication cost with performance and reliability constraints. Jozefowska *et al.* (2002) used Tabu Search for scheduling jobs on parallel, identical machines with an additional continuous resource to minimize the makespan.

Simulated Annealing (SA) is an iterative search method inspired by the annealing of metals (Kirkpatrick *et al.*, 1983, Cerny 1985). Starting with an initial solution SA tries to minimize a cost function by making "moves", which are occasionally accepted solutions of higher values of the cost function with the probability controlled by a parameter called temperature. One of the salient features of SA is that the probability of acceptance of moves that increase the cost function exponentially decreases as temperature decreases. The process ends as soon as temperature is low enough that no further improvement can be expected. At high temperature, the search is almost random, while at low temperature the search becomes almost greedy. At zero temperature, the search becomes totally greedy, i.e., only good moves are accepted (Kirkpatrick *et al.* 1983, Cerny 1985).

In a large combinatorial optimization problem, an appropriate move mechanism, cost function, solution space, and cooling schedule are required in order to find an optimal solution with SA. Kim *et al.* (2002) presented a scheduling problem for unrelated parallel machines with sequence-dependent setup times, using SA. Baykasoglu (2002) presented a SA algorithm developed to solve the Flexible Job Shop Scheduling Problem.

These algorithms have several similarities (Sait & Youssef, 1999):

1. They are approximation (heuristic) algorithms, i.e., they do not guarantee finding an optimal solution.
2. They are blind, in that they do not know when they have reached an optimal solution. Therefore they must be told when to stop.
3. They have "hill climbing" property, i.e., they occasionally accept uphill (bad) moves.
4. They are general, i.e., they can easily be engineered to implement any combinatorial optimization problem; all that is required is to have a suitable solution representation, a cost function, and a mechanism to traverse the search space.
5. Under certain conditions, they asymptotically converge to an optimal solution.

In general, GA is easy to model and be applied to any type of optimization problems. However, the quality of solution is usually not as good as those from TS and SA. SA is ranked next (Pierre & Elgibaoui 1997, Jozefowska *et al.* 1998**,** Augugliaro *et al.* 1999, Fu & Su 2000, Youssef *et al.* 2001) and TS highest as it can usually produce quite a good solution although the algorithm is more complicated to implement. The reason that TS outperforms the GA and SA may be that TS has a memory mechanism.

2.4 Multi-objective optimization

Many real-world optimization problems inherently involve multiple non-commensurable and often competing objectives. For this kind of problem, optimizing one of the objectives often means that other objectives have to be compromised.

There are three broad categories in solving multi-objective (MO) optimization problems. Conventional MO optimization methods often combine these multiple objectives into a single scalar, by using addition, multiplication or other combinations of arithmetical operations. If the combination is possible, this approach is the simplest one and the most computationally efficient one. However, it is very difficult to devise such a method, because accurate scalar information on the range of objectives must be known to avoid some objectives dominating others.

Alternatively, only one objective is optimized and the others are treated as constraints (Erschler *et al.* 1976, Fox 1987). However, there is one potential pitfall in that the algorithm may not be able to find a feasible solution because the problem is over-constrained.

The third category is the Pareto based method, which uses the concept of Pareto's optimality to obtain a set of solutions instead of just one. Some related definitions are described as follows:

> A general multi-objective minimization optimization problem is a problem in which the n objective functions $f_k, k = 1,2,\cdots,n$ are simultaneously minimized.

It is likely, however, that the objective functions are a nonlinear vector function F of a general decision variable s in the whole solution space S, where $F(s) = (f_1(s), f_2(s), \cdots, f_n(s))$.

Definition 2.1 (Pareto dominance): A given vector $u = (u_1, u_2, \cdots, u_n)$ is said to dominate another vector $v = (v_1, v_2, \cdots, v_n)$ iff

$$\forall i \in \{1,2,\cdots,n\},\ u_i \le v_i \ \wedge\ \exists i \in \{1,2,\cdots,n\},\ u_i < v_i \tag{2.1}$$

Definition 2.2 (Pareto-optimal): Given a set of solutions $S = \{s_1, s_2, \cdots s_m\}$, a solution $s_i \in S$ is said to be Pareto-optimal iff no solution $s_j \in S$ dominates solution s_i.

Pareto-optimal solutions are also called non-dominated, efficient, and non-inferior solutions. The corresponding objective vectors are referred to as non-dominated. The set of all non-dominated vectors is called as the non-dominated set, or tradeoff surface, of the problem.

Definition 2.3 (Pareto front): Given a multi-objective optimization function $F(s)$ and a set of Pareto-optimal solution Ω, the Pareto-front is:

$$\{\bar{u} = F(s) = (f_1(s), \ldots, f_n(s)) \mid s \in \Omega) \tag{2.2}$$

Evolutionary algorithms are particularly suitable to solve multi-objective optimization problems because they can produce a set of solutions simultaneously instead of just one. Even in the late 1960s, Rosenberg (1967) considered the possibility of using genetic-based search to solve the multi-objective optimization. Later, an early GA application on multi-objective optimization by Schaffer (1985) opened a new avenue of research in this field.

As evolutionary algorithms need scalar fitness information to work, most research on evolutionary multi-objective optimization are related to the fitness assignment. There are three broad categories (Fonseca & Fleming 1995): aggregating function based approaches, population based non-Pareto-based approaches, and population based Pareto-based approaches.

2.4.1 Aggregating function based approaches

Given evolutionary algorithms that need a scalar fitness value to work, it is logical to combine the multiple objectives into a single one using either an addition multiplication or any other combination of arithmetical operations. The function of combining objectives into one is normally referred to as *aggregating function*. The main advantage of this method is that, if the combination of objectives is possible, it is very simple and very efficient because no further interaction with the decision maker is required. However the disadvantages of this method are very obvious. Some accurate information on the range of the objectives has to be known to avoid having one of them dominate the others. However, obtaining such information on each objective is very computationally expensive. The following section introduces the most popular aggregating approaches.

2.4.1.1 Weighted sum approach

The basic idea of this method is the addition of all the objective functions together using different weighting coefficients for each one of them. The main strength of the method is its computational efficiency which enables the generation of a strongly non-dominated solution that can be used as an initial solution for other techniques. The main weakness of the method is that the optimization process is sensitive to the weights (Coello 1996) and that it is very difficult to choose appropriate weights without prior knowledge on the shape of the search space. It also suffers from the disadvantage of excluding the concave portions of the trade-off curve (Coello 1996).

Gen *et al*. (1995, 1997) added fuzzy logic to handle the uncertainty involved in the decision making process. A weighted sum is still used in this approach, but the coefficients of the objectives are represented with fuzzy numbers reflecting existing uncertainty regarding their relative importance.

2.4.1.2 Goal attainment

In this approach, the decision maker first provides a set of goals and weights.

For a *k* objectives minimization problem:

Minimize α

subject to: $b_i + \alpha w_i \geq f_i(\bar{x}) \quad i = 1,2,\cdots,k$ (2.3)

where b_i is the ith goal;

α is a scalar variable unrestricted in sign;

w_i is the weight for ith objective, which is normalized, i.e., $\sum_{i=1}^{k} | w_i | = 1$; $f_i(\bar{x})$ is the ith objective function.

The main advantage is that it is very simple to implement and is computational efficient. The main disadvantage is the misleading selection pressure (Wilson & MacLoud 1993). The selection pressure is the degree to which the better individuals are favored; the higher the selection pressure, the more the better individuals are favored. Hence, the higher the selection pressure, the faster the convergence rate.

2.4.2 Population-based non-Pareto approaches

To overcome the difficulties of the aggregating approaches, several alternative algorithms have been proposed. This section introduces some of the approaches.

2.4.2.1 Vector evaluated genetic algorithm (VEGA)

The basic idea of the VEGA is that the main population is divided into sub-populations and selection is performed according to each objective function in each sub-population (Schaffer 1985). These sub-populations would be shuffled together to obtain a new population. The main advantage of this approach is its simplicity. The obvious disadvantage is that it only manages to find certain extreme solutions along the Pareto tradeoffs. In addition, the shuffling and merging of all sub-populations correspond in fact to fitness averaging for each of the objective components. It therefore suffers from the same disadvantage as does the linear combination of the objectives, i.e., the inability to produce Pareto-optimal solutions in the presence of non-convex search spaces (Richardson *et al.* 1989).

To deal with the first disadvantage, Schaffer (1985) suggested some heuristics. For example, to use a heuristic selection preference approach for non-dominated individuals in each generation or to crossbreed among the sub-population.

Cvetkovic *et al.* (1998) also proposed several approaches to overcome VEGA's problems. For example, to wait for a certain number of generations before shuffling together the population, or to copy or migrate a certain number of individuals from one sub-population to another. Interestingly, however, the nominally disadvantageous tendency of VEGA to favor certain solutions can be of advantage when handling constraints, because in this case favoring a solution that does not violate any constraint over those which do is just what is needed. Surry *et al*. (1995), for instance, used

VEGA to model constraints in a single-objective optimization problem to avoid the need for a penalty function.

As an extension of VEGA, Lis and Eiben (1997) proposed a multi-sexual genetic algorithm, where each individual has an additional feature of sex or gender and one individual from each sex is used in recombination. There are as many sexes as the optimization criteria and each individual is evaluated according to the optimization criteria related to its sex.

2.4.2.2 Lexicographic ordering

The basic idea is that the objectives are ranked in order of importance by the designer and then the optimum solution is obtained by minimizing the objective functions, starting with the most important one and proceeding according to the assigned order of importance of the objectives (Fourman 1985). In another version of this algorithm (Fourman 1985), which apparently worked quite well, an objective was randomly selected at each run. In the algorithm of Kursawe (1991), one of the objectives was also selected randomly at each step according to a probability vector. The main advantage of this method is that in some cases it overcomes the disadvantage of a weighted sum of objectives enabling VEGA to see, as convex, a concave trade-off surface. This does however depend on the distribution of the population and on the problem itself. The main weakness in this approach is that it will tend to favor more certain objectives when many are present in the problem, and so will not obtain the whole trade-off surface.

2.4.2.3 Weighted Min-Max approach

The basic idea of this approach is to solve the optimization problems for each objective separately to obtain the extremes of the objective functions. The desirable solution is then the one which gives the smallest values of the relative increments of all the objective functions. The idea is taken from game theory which deals with solving conflicting situations. The min-max approach to a linear model was proposed by Solich (1969), and was further developed by Osyczka (1978, 1984), Rao (1986) and Tseng and Lu (1990). The main advantage is its simplicity and computational efficiency because it does not require checking for non-dominance. The main disadvantage is that it can create a very high selection pressure if certain combinations of weights are produced at early stages of the search (Coello 1999).

Hajela & Lin (1992) included the weights of each objective in the chromosome to generate a set of Pareto-optimal solutions by varying the weighting coefficients. Coello (1996) proposed two variations of the weighted min-max strategy used by Hajela & Lin (1992). Ishibuchi & Murata (1996) proposed an algorithm similar to the weighted min-max technique (Hajela & Lin 1992), which maintains adequate diversity (Bently & Wakefield 1997). However, in some case, sharing techniques or a local research technique have to be used to keep the diversity (Coello 1996).

2.4.3 Pareto based approaches

Goldberg (1989) proposed an idea of using Pareto-based fitness assignment to solve the problems of Schaffer’s approach (Schaffer 1985). The basic idea is that all the non-dominated individuals are assigned the highest rank and eliminated from the further contention. Then, from the rest, another set of non-dominated solutions are assigned the next highest rank. The process continues until the all individuals are ranked. The Pareto method was found to outperform the Vector Evaluated Genetic Algorithm (VEGA) method in some cases (Hilliard *et al.* 1989, Liepins *et al.* 1990, Ritzel *et al.* 1994).

The main advantage of this method is that it is not sensitive to the shape or continuity of the Pareto front, these two issues being of serious concern in mathematical programming techniques (Coello 1999). The main disadvantage is that there is no efficient algorithm to check for non-dominance in a set of feasible solutions (Coello 1996).

Goldberg (1989) also suggested using "niching" techniques to keep the individuals' diversity. i.e., to prevent the individuals from converging to a single point on the front. Hence, "niching" techniques can also prevent the GA becoming trapped in local optima.

"Niching" techniques are based on the mechanism of natural ecosystems. An ecosystem consists of several subspaces and different species. Every subspace supports different species which compete to survive and are capable of interbreeding among themselves but are unable to breed with individuals outside their groups. The search space can be viewed as the ecosystem, a niche as a subspace and a group of individuals with similar metrics as a species. For each niche, the resources are finite and must be shared among the individuals of that niche. A niche is commonly referred to as an optimum of the domain, the fitness representing the resources of that niche. By analogy, niching methods tend to achieve a natural emergence of niches and species in the system.

A well-known niching technique is the sharing method which was originally introduced by Holland (1975) and improved by Goldberg and Richardson (1987). Fitness sharing modifies the search landscape by reducing the payoff in densely populated regions. Typically, the shared fitness $f_i^{'}$ of an individual i with fitness f_i is simply

$$f_i^{'} = \frac{f_i}{m_i} \tag{2.4}$$

where m_i is the niche count which can be calculated as the following equation:

$$m_i = \sum_{j=1}^{N} sh(d_{ij}) \tag{2.5}$$

where N is the population size and $sh(d_{ij})$ is the sharing function, representing the similarity level between the individuals i and j . The most widely used sharing function is given as following:

$$sh(d_{ij}) = \begin{cases} 1-(d_{ij}/\sigma_s)^{\alpha}, & if \quad d_{ij} < \sigma_s \\ 0, & otherwise \end{cases} \tag{2.6}$$

where d_{ij} is the distance between individual i and j ; σ_s denotes the threshold of dissimilarity and α is a constant parameter which regulates the shape of the sharing function.

Sharing methods tend to encourage search in unexplored regions of the solution space and benefit the formation of stable niches. However, setting the threshold of dissimilarity σ_s requires *a priori* knowledge of how far apart the optima are. For real world problems, the information on the distance between the optima is generally not available. Various empirical formulas have been proposed to set the threshold of dissimilarity but this problem remains the major flaw in the method (Mahfoud 1995).

On the other hand, a fixed threshold of dissimilarity is for all individuals. Hence the sharing method can fail to work in cases where the optima are not equidistant or the estimated distance between two peaks is incorrect.

Another drawback of the sharing scheme is that the computation of niche counts is very expensive. Clustering analysis (Yin & Germay 1993) and dynamic niching (Miller & Shaw 1996) can reduce computational complexity and increase the sharing effectiveness. In addition, when the computational time to compute the fitness of individuals is far more expensive than the computational cost of sharing scheme, standard sharing can be implemented with only a small increase in the computational requirements.

Goldberg (1989) suggested a method to overcome the disadvantage of VEGA by means of a non-dominated sorting procedure in conjunction with a sharing technique. This non-dominated sorting procedure uses a ranking selection method to emphasize good individuals and a niche method to maintain stable subpopulations of good individuals.

Several algorithms based on this idea, have been proposed, including Multiple Objective Genetic Algorithm (MOSA) (Fonesca & Fleming 1993) and Non-Dominated Sorting Genetic Algorithm (NSGA) (Srinivas & Deb 1994). These methods implemented Goldberg's suggestion in different ways. The following section briefly discusses the these algorithms and a Tabu-Based Exploratory Evolutionary Algorithm (EMOEA) (Tan *et al.* 2003).

2.4.3.1 Multiple objective genetic algorithm (MOGA)

Fonseca and Fleming (1993) proposed to rank a certain individual according to the number of the individuals that dominate it. All non-dominated individuals are assigned rank 1, and the rank of a dominated one is one plus the number of the individuals that dominate it. The fitness assignment is performed in the following way (Fonseca and Fleming 1993):

1. Sort the individuals according to the rank.

2. According to the method proposed by Goldberg (1989), assign fitness to individuals by interpolating from the best rank 1 to worst rank n. Usually the interpolation is linear, but not necessarily.

3. Average the fitness of the individuals with same rank, so all the individuals with same rank are assigned to same fitness.

In this method, a number of individuals have the same fitness. This type of blocked fitness assignment tends to cause premature convergence (Goldberg & Deb 1991). To avoid this, a sharing on objective domain was used to distribute the population over the Pareto-optimal region. The main advantages of the method are that it is efficient and relatively easy to implement (Coello 1996). The main disadvantage is that its performance is highly dependent on the selection of the threshold of dissimilarity.

2.4.3.2 Niched Pareto genetic algorithm

Horn & Nafpliotis (1994) proposed a tournament selection scheme based on Pareto dominance instead of a non-dominated sorting and ranking selection method in solving multi-objective optimization problems. Tournament selection is one of the most widely implemented selection techniques for GA's and involves selection of the best individual from a subset comprising individuals randomly chosen from the current population.. In this scheme (Horn & Nafpliotis 1994), two candidates for selection and a comparison set of individuals are picked randomly from the population. Each of the candidates is then compared with each individual in the comparison set. If one candidate is dominated by the comparison set and the other is not the latter is selected for reproduction. If neither or both are dominated by the comparison set, i.e., there is a tie, then sharing is used to choose a winner.

The main advantage of this method is that it is very fast because this approach applies Pareto selection to a subset of the population instead of to the entire population at each run. It can produce good non-dominated fronts that can be kept for a large generation (Coello 1996). The main disadvantage is that the approach needs to select a suitable size for the tournament to perform well (Coello 1999).

2.4.3.3 Non-dominated sorting genetic algorithm

Srinivas and Deb (1994) proposed the Non-dominated Sorting Genetic Algorithm (NSGA). Before selection is performed, the population is ranked on the basis of non-domination. Then all non-dominated individuals are classified into a first front. All these individuals are assigned to the same dummy fitness to provide them with an equal reproductive potential.. These classified individuals are then *shared* with their dummy fitness value to maintain the diversity of the population. After sharing these individuals are ignored temporarily and the rest of the individuals are dealt with by the same process to form second front. The process continues until all individuals in the population are classified into different fronts. The dummy fitness value should be smaller than the minimum shared dummy fitness value of the previous front.

A stochastic remainder proportionate selection was used for this approach. Since individuals in the first front have the maximum fitness value, they always get more copies than the rest of the individuals. This results in quick convergence of the population toward non-dominated regions and sharing helps to distribute it over this region. The efficiency of NSGA lies in the way in which multiple objectives are reduced to a dummy fitness function using a non-dominated sorting procedure. With this approach any number of objectives can be solved and both maximization and minimization problems can be handled.

The main advantage of this method is that it can handle any number of objectives and allows multiple equivalent solutions to exist. The main disadvantage is that it is inefficient compared to MOGA (Coello 1996).

2.4.3.4 Tabu-based exploratory evolutionary algorithm (EMOEA)

Tan *et al.* (2003) proposed an exploratory multi-objective evolutionary algorithm (EMOEA) that integrates the features of Tabu Search and evolutionary algorithm for multi-objective optimization. In addition, a new “lateral interference” is presented to distribute non-dominated individuals along the discovered Pareto-front uniformly. The lateral interference can be performed without the need for parameter settings, unlike many niching or sharing methods, and can be flexibly applied in either the parameter or objective domain. Besides the lateral interference, the Tabu restriction also helps to maintain the diversity of the solutions, which in turn helps to prevent the search from becoming trapped in local optima as well as concurrently promoting the evolution towards the global trade-offs. In addition, EMOGA offers a competitive behavior to escape from local optima in a noisy environment.

Chapter 3

A Reliability Oriented Genetic Algorithm for Distributed Computing Systems

Distributed computing systems (DCS) are common today as they provide cost-effective design to achieve system reliability, availability and performance requirements (Kumar *et al.* 1986, Raghavendra *et al.* 1988, Shatz *et al*. 1992, Kumar & Agrawal 1993, Yeh & Chiu 2001, Wong & Dillon 2000). When the topology of a DCS is fixed, the distributed system reliability (DSR) depends mainly on the assignment of various resources such as programs and files (Kumar *et al.* 1986, Raghavendra *et al.* 1988). For systems with long mission times or with a large number of processors, an improved program allocation can increase the system reliability dramatically (Shatz *et al*. 1992).

DSR, which was first studied by Kumar *et al.* (1986), is defined as the probability for all the distributed programs to work successfully. Note that the reliability concept here is a general one depending on the definition of failure and other performance requirements. Kumar *et al.* (1986) also presented an algorithm called MFST (Minimal File Spanning Tree) to compute the DSR. A drawback of the MFST algorithm is that it is computationally complex and prior knowledge about multi-terminal connections is needed. Later an algorithm called GEAR (Generalized Evaluation Algorithm for Reliability) was developed (Kumar & Agrawal 1993) to resolve the problem. Other related algorithms have also been presented (Chen & Huang 1992, Chen & Lin 1992, Chen & Lin 1994).

Reliability oriented task allocation problems have been studied extensively (Wang & Shatz 1988, Shatz *et al.* 1992, Hwang & Tseng 1993, Kartik & Murthy 1997, Tom & Murthy 1999, Vidyarthi & Tripathi 2001, Mahmood 2001).

The distribution of data files can also have impact on the reliability of distributed systems (Dowdy & Foster 1982). Consequently, there has been research done on reliability oriented file allocation (Pathak *et al.* 1991, Kumar *et al.* 1995a). Most previous studies however considered the program allocation and file allocation problems separately, whereas both should be optimally allocated simultaneously to achieve the highest level of system reliability.

Chari (1996) presented a heuristic procedure for DCS design along with the location of data files and programs, but the procedure aimed at minimizing the DCS cost. Chiu *et al.* (2002) developed a reliability oriented heuristic algorithm to assign the files and tasks for DCS with memory space (storage) constraints. However, speeding up computation is one of the main reasons to build the DCS. If the storage capacity of one processor is large enough for the storage of all programs and all files, the heuristic algorithm of Chiu *et al.* (2002) assigns all programs and files to this processor, which is not rational for a DCS. In fact, the assignment of programs and files is often subjected to the completion time constraint and the cost constraint (Shatz *et al.* 1992).

In this chapter it is proposed to first construct a reliability oriented optimization model with the storage constraints, cost constraint and completion time constraint by considering both program allocation and file allocation together. Considering the program and file allocation together and taking into account these constraints makes the problem difficult to solve. It is this computational problem that is the subject of the current study.

The problem has important practical applications. For example, many organizations want to implement a common distributed information system by using their existing computer and network resources. The system reliability is an important requirement in this case. The topology of the system is fixed, so the system reliability depends mainly on the program and file allocation. Generally the organizations have many computers and are willing to offer these resources albeit with several resource constraints.

The chapter is organized as follows. Section 3.1 describes the DCS design problem and an optimization model for the program and file allocation. Section 3.2 presents an exhaustive search algorithm and a genetic algorithm to solve the optimization problem. Section 3.3 presents two numerical examples to compare the two algorithms. Then, some sensitivity studies are carried out for the investigation of important parameters. Section 3.5 concludes this chapter.

3.1 Optimization model

3.1.1 Structure of the system

A typical DCS can be depicted by a graph as in Figure 3.1, where nodes denote the processing elements, which are linked by the network. Each node includes a set of programs and a set of files. In the DCS, successful execution of programs is dependent on the successful access of necessary files distributed throughout the system.

Recall that Kumar *et al.* (1986) defined distributed program reliability (DPR) as the probability that a given program can run successfully and be able to access all the required files from remote sites in spite of some faults occurring among the processing elements and the communication links.

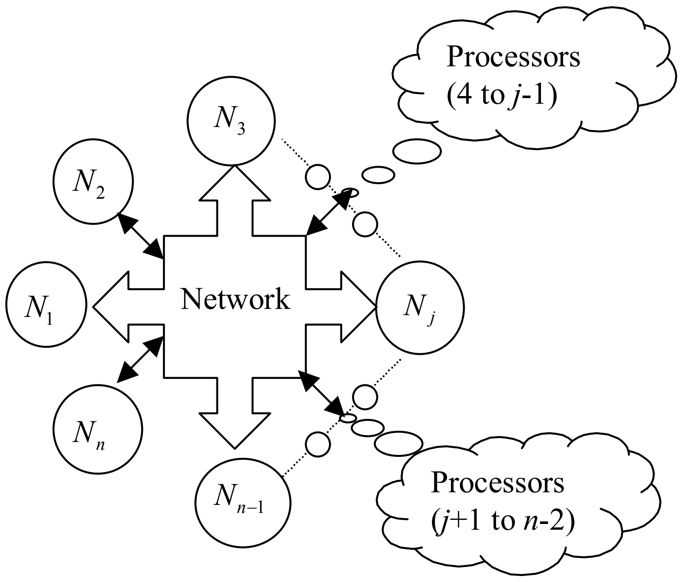


Figure 3.1: *n* processors of a distributed system

Similarly, the DSR is defined as the probability that all programs perform successfully in a DCS. To explain DSR, we first introduce a concept – file spanning tree (FST) (Kumar *et al.* 1986) which is a tree that connects the root node, where the program runs, to other nodes that hold all the files required for executing the given program. A minimal file spanning tree (MFST) (Kumar *et al.* 1986) is an FST that contains no subset FST.

The DPR is defined as

$$\text{DPR} = \Pr(\text{at least one MFST for the given program is working})$$

This can be written as

$$DPR = \Pr(\bigcup_{i=1}^{N_{MFST}} MFST_i) \tag{3.1}$$

where N_{MFST} is the number of the MFST for the given program.

Similarly, the subgraph which provides all the required connections for executing all the programs in the distributed system is referred to as a forest (Raghavendra *et al.* 1988). The minimal file spanning forest (MFSF) (Raghavendra *et al.* 1988) is defined as the forest which contains no subset forest. Correspondingly, the DSR is given as

$$DSR = \Pr(\bigcup_{j=1}^{N_{MFSF}} MFSF_j) \tag{3.2}$$

where N_{MFSF} is the number of the MFSF for the given distributed system.

Different assignments of programs and files on a given distributed system lead to different DSRs. The more copies there are of the programs the higher will be the DSR. However, the budget for purchasing programs is usually limited. As each copy of the programs requires a fixed amount of resource, the number of copies of each program is also limited. Similarly, each node has a limited storage capacity and therefore the number of copies of each file is also limited. A further constraint is that the completion time for a transaction cannot be delayed beyond a certain deadline.

In order to maximize the DSR's, subject to the above constraints, we need to determine the optimum number of redundant copies of programs and files, and to then assign them to various locations in the DCS. The optimization model is presented below.

3.1.2 Modelling and optimization of system reliability

Notations:

C_b:	budget limit;
C_j:	cost for a copy of program P_j;
C_t:	completion time limit;
F_j :	j-th distributed file;
N_F :	number of files;
N_n :	number of nodes;
N_P :	number of programs;
NF_{ij} :	assignment of file F_j on the node N_i ;
NN_{ij} :	link between nodes N_i and N_j;
NP_{ij} :	assignment of program P_j on node N_i;
P_j :	j-th computing program;
$\overrightarrow{PF}_j$:	set of files required by the j-th program P_j ;
S_j:	size of the j-th file F_j;
SC_i :	storage limit of node N_i;
T_{ij}:	completion time of program P_j at node N_i.

The topology of the distributed system is denoted by $NN_{ij}\ \ (i, j = 1,2,\ldots, N_n)$, which has the value 1 or 0: $NN_{ij} = 1$ if there is a link between node N_i and node N_j, else 0.

Each copy of the program P_j spends the expected cost of C_j (j=1,2,…, N_P). The assignment of these programs is denoted by NP_{ij} (i=1,2,…, N_n; j=1,2,…, N_P), which has the value 1 or 0: NP_{ij}=1 if program P_j is assigned to node N_i, else 0. Thus, the total cost for all the programs can be calculated by

$$C = \sum_{i=1}^{N_n} \sum_{j=1}^{N_P} C_j NP_{ij} \tag{3.3}$$

There is usually a project budget for preparing the programs, which restricts the total cost to be no greater than the budget limit denoted by C_b, i.e. $C \leq C_b$.

A given program P_j executed at node N_i requires a completion time of T_{ij}, i=1,2,…, N_n and j=1,2,…, N_P. Thus, the total completion time for the processing node N_i can be computed by

$$T_i = \sum_{j=1}^{N_P} T_{ij} NP_{ij} \text{ , } i=1,2,\ldots, N_n \tag{3.4}$$

For a project, there usually exists a time constraint for all the nodes to complete their own programs, such that the completion time of any node N_i cannot exceed the completion time limit (C_t), i.e. $T_i \leq C_t$ for all i.

Each copy of the required file F_j has its own file size S$_j$, j=1,2,…, N_F.

The assignment of these copies of files is denoted by NF$_{ij}$, i=1,2,…, N_n and j=1,2,…, N_F, which has the value either 1 or 0: NF_{ij}=1 if file F_j assigned to node N_i, else 0. Thus, the total size of files stored in the processing node N_i can be computed by

$$Z_i = \sum_{j=1}^{N_F} S_j NF_{ij} \text{ , } i=1,2,\ldots, N_n \tag{3.5}$$

As the storage device of a node N_i has a limited capacity, the total size of files stored in the node N_i cannot exceed its storage limit (SC$_i$), i.e. $Z_i \leq SC_i$ for all i.

The DCS consists of at least one copy of all the computing programs P_j, j=1,2,…, N_P and at least one copy of all the required files F_j, j=1,2,…, N_F, which is denoted by the basic constraints, i.e. $\sum_{i=1}^{N_n} NP_{ij} > 0$, j=1,2,…,N_P and $\sum_{i=1}^{N_n} NF_{ij} > 0$, j=1,2,…, N_F.

Based on the above conditions, our objective is to maximize the DSR. The DSR can be computed by an algorithm called GEAR (Kumar & Agrawal 1993), which is a generalized algorithm for evaluating DSR. In order to compute DSR, the following variables are required to be known: the topology of the distributed system NN_{ij}, $i = 1,2,\ldots,N_n$ and $j = 1,2,\ldots,N_n$;the computing programs $\{P_1, P_2 \ldots\ldots P_{N_P}\}$ and the program allocation NP_{ij}, $i = 1,2,\ldots,N_n$ and $j = 1,2,\ldots,N_P$; the files required by each program $\overrightarrow{PF}_j$, $j = 1,2,\ldots,N_P$ and the file allocation NF_{ij}, $i = 1,2,\ldots,N_n$ and $j = 1,2,\ldots,N_F$; and the reliability of the links.

GEAR (Kumar & Agrawal 1993) evaluates all the sub-networks that satisfy the file requirement for given programs. The inputs of the algorithm include the topology of the network, the program allocation, file allocation and the reliability of the links. In each step, the algorithm updates four vectors: program vector (PV), file vector (FV), reliability vector (RV) and loop vector (LV). PV keeps the information that which required programs have been found; correspondingly, FV keeps the information that which required files have been found. RV maintains the information about the links in the subnetwork and is used to computer the DSR. LV keeps the information about the nodes that have been traversed in a subnetwork. Form a root node, GEAR searches the adjacent nodes in parallel and updates PV, FV, RV, and LV. RV is updated in such a way that all the terms in the RV are disjoint, so they can be added directly to get the reliability expression.

There are two termination rules of a search path. One is that all programs and the required file have been found, as indicated by the PV and FV. Another is that all nodes have been traversed, as indicated by LV. The RV of each terminal node that satisfies the first termination rule, is the reliability term. The DSR is the sum of all these reliability terms. More details of the evaluation process can be found in the paper of Kumar & Agrawal (1993).

In our optimization model, the topology of the distributed system NN_{ij}, $i = 1,2,\ldots,N_n$ and $j = 1,2,\ldots,N_n$, the computing programs $\{P_1, P_2,\ldots,P_{N_P}\}$ and the files required by the programs $\{\vec{PF}_1,\vec{PF}_2,\ldots,\vec{PF}_{N_p}\}$ are assumed to have already been obtained. The optimization model for program and file assignment can be stated as following.

Decision variables: NP_{ij}, i=1,2,…, N_n; j=1,2,…,N_P,

NF_{ij}, i=1,2,…, N_n; j=1,2,…,N_F.

Object function:

$$\textbf{Maximize } DSR = f(NN_{ij}, P_k, \vec{PF}_k, NP_{ik}, NF_{il}) \tag{3.6}$$

where $i, j = 1,2,\ldots,N_n$; $k = 1,2,\ldots,N_P$; $l = 1,2,\ldots,N_F$

Constraints:

Basic Constraints:

$$\sum_{i=1}^{N_n} NP_{ij} > 0 \text{ , } j=1,2,\ldots,N_P, \tag{3.7}$$

$$\sum_{i=1}^{N_n} NF_{ij} > 0,\ j=1,2,\ldots,N_F, \tag{3.8}$$

Cost Constraint:

$$\sum_{i=1}^{N_n}\sum_{j=1}^{N_P} C_j \cdot NP_{ij} \le C_b \tag{3.9}$$

Completion Time Constraint:

$$\sum_{j=1}^{N_P} T_{ij} \cdot NP_{ij} \le C_t,\ i=1,2,\ldots, N_n \tag{3.10}$$

Storage Constraints:

$$\sum_{j=1}^{N_F} S_j \cdot NF_{ij} \le SC_i,\ i=1,2,\ldots, N_n \tag{3.11}$$

3.2 Solution algorithms

In order to obtain the solution for the above optimization model, an exhaustive search algorithm and a genetic algorithm are presented. The exhaustive algorithm can guarantee the optimal solution but is computationally complex. The genetic algorithm can effectively find a good solution but cannot guarantee the optimal result every time.

3.2.1 Exhaustive search algorithm

The exhaustive search algorithm consists of the following steps:

Step 1: Set maximum DSR (DSR_{Max}) to 0 and set optimum program and file allocation set $\{PFL_{Opt}\}$ to Φ.

Step 2: Select a new program and file allocation $PFL_{New}=\{NP_{ij}, NF_{mn}\}$ from $\{0,0,\ldots,0\}$ to $\{1,1,\ldots,1\}$ by changing 0 to 1 one by one.

Step 3: If the PFL_{New} fits the basic constraints, cost constraint, completion time constraint and storage constraints, go to step 4. Otherwise, go to step 2.

Step 4: Use GEAR algorithm to calculate the new allocation's DSR (DSR_{New}).

Step 5: If $DSR_{Max} < DSR_{New}$, set $\{PFL_{Opt}\} = \Phi$ and add PFL_{New} into $\{PFL_{Opt}\}$. Otherwise if $DSR_{Max} = DSR_{New}$, add PFL_{New} into $\{PFL_{Opt}\}$.

Step 6: Are all the allocations from {0,0,…,0} to {1,1,…,1} searched? If yes, go to step 7. Otherwise, go to step 2.

Step 7: Output the optimum allocation set $\{PFL_{Opt}\}$ and the maximum DSR (DSR_{Max}).

The result of $\{PFL_{Opt}\}$ contains all the optimal assignments that can obtain the maximum DSR (DSR_{Max}) without violating any of the constraints.

Although this exhaustive search algorithm can guarantee the finding of all the optimal assignments, the effectiveness of the algorithm is low. A more effective algorithm will be developed in the following subsection.

3.2.2 Genetic algorithm implementation

A genetic algorithm is an evolutionary optimization technique, which is a rapidly growing area of artificial intelligence. Genetic algorithms are inspired by Darwin's theory of evolution based on the survival of the fittest species as introduced by Holland (1975) and further described by Goldberg (1989). Some genetic operators for this optimization problem are first introduced and then the general procedures are presented.

3.2.2.1 Operators

Chromosome encoding

Since the purpose of the chromosome is to represent whether the program or file is placed on the corresponding node or not, every chromosome is a list of binaries corresponding to $\{NP_{ij}, NF_{mn}\}$. Hence, the length of the chromosome is $N = N_n * N_P + N_n * N_F$.

Selection

We implement Roulette Wheel Selection here. Parents are selected according to their fitness. Fitter chromosomes have large fitness value, more chance the random number falls into their scope. Hence, the fitter the chromosomes are, the more chances they have of being selected. The basic process can be described as follows:

1. Calculate the sum of all chromosome fitnesses in population – S_{all}.

2. Generate a random number from the interval (0, S_{all})-r.

3. Go through the population and sum the fitness from 0 - sum s. When the sum s is greater then r, stop and the current chromosome is selected.

For example, suppose that there are four chromosomes in the population and their fitnesses are 0.8, 0.9, 0.85, 0.95 respectively. The Roulette Wheel Selection procedure is as follows:

1. The sum is 0.8 + 0.9 + 0.85 + 0.95 = 3.3

2. The random number is 2

3. (0.8 + 0.9) < 2 < 0.8 + 0.9 + 0.85, the third chromosome is selected.

If we use the DSR as the fitness, the difference is very small between the chromosomes and therefore it does not provide enough discrimination between chromosomes with very close DSR. This is especially so when most chromosomes are close to the optimal solution. To overcome this problem we use the following fitness function:

$$Fitness = \frac{-1}{\ln(DSR)} \tag{3.12}$$

Since *DSR* is between 0 and 1, ln(*DSR*) will tend to 0 and the *Fitness* will tend to infinity as the *DSR* tends to 1. Thus, Equation (3.12) can enlarge the difference between the chromosomes and give a chromosome, whose DSR is closer to 1 and just a little larger than the others, more chance of being selected.

Crossover

In crossover, two new chromosomes are formed by swapping the sets of genes of two parent chromosomes. In a program and file assignment problem, crossover diversifies the population by swapping parts of the two parent chromosomes selected randomly. A chromosome resulting from a crossover may comprise a combination of program and file assignment that is infeasible requiring an adjustment to be made as below.

Mutation

To avoid convergence to a local optimum as the population size increases, the mutation operation is used more frequently. Every bit of the chromosome has the predefined probability – mutation rate to be selected as the mutation site. If the corresponding bit is 1, it is changed to 0, and vice versa. The mutated chromosome must satisfy all the constraints or else adjustment is required as below.

Adjustment

There are various methods of handling constraints, for example, penalty function, adding repair operators, or just discarding the infeasible solution. In the current case, it is difficult to find an effective and efficient penalty function. The method of discarding the infeasible solution is most suited to the case where the infeasible solutions appear infrequently. However, if infeasible solutions appear frequently but can be adjusted to feasible solutions without too much computational cost, repair operators are efficient and effective.

Our algorithm is of this type and therefore we have selected the repair operators to handle constraints. In this section, repair operators are referred to as adjustment.

1) Basic Constraint Adjustment:

If the number of copies of any program P_j is 0, i.e. $\sum_{i=1}^{N_n} NP_{ij} = 0$, a random node N_i is selected and a copy of this program is put on it, i.e. NP_{ij} is changed to 1. The allocation, however, is not allowed to violate the completion time constraint.

If the number of copies of a file F_j is 0, i.e. $\sum_{i=1}^{N_n} NF_{ij}$, a random node N_i is selected and a copy of this file is put on it, i.e. NF_{ij} is changed to 1. The allocation, however, is not allowed to violate the storage constraint.

2) Cost Constraint Adjustment

If the chromosome does not satisfy the cost constraint, a random copy of the program that has the largest number of copies is inversed to 0. This adjustment is repeated until the chromosome satisfies the cost constraint.

3) Completion Time Constraint Adjustment

If some nodes do not meet the completion time constraint, there are two possible adjustments depending on the conditions. First, if every program presenting on the infeasible nodes has just one copy in the DCS, i.e. $\sum_{i=1}^{N_n} NP_{ij} = 1, j \in (1,2,\ldots N_P)$, randomly select a program presenting on the infeasible nodes and transfer it to another feasible node. Otherwise, any program presenting on the infeasible nodes that has the largest number of copies is inversed to 0. The above adjustments are repeated until the adjusted chromosome satisfies the completion time constraint.

4) Storage Constraint Adjustment

If some nodes do not meet the storage constraints, then in a similar method to that used for completion time adjustment, the adjustment is done in one of two ways depending on the conditions applying. If every file presenting on the infeasible nodes has just one copy in the DCS, i.e., $\sum_{i=1}^{N_n} NF_{ij} = 1, j \in (1,2\ldots N_F)$, randomly select a program presenting on the infeasible nodes and transfer it to another feasible node. Otherwise, any file presenting on the infeasible nodes that has the largest number of copies is inversed to 0. The above adjustments are continued until the adjusted chromosome satisfies the storage constraint.

3.2.2.2 Implementation procedure

Based on the above operators, the general procedures for the GA to solve this optimization problem are as below:

Step 1: Encode the solutions into chromosomes;

Step 2: Generate an initial population;

Step 3: Selection;

Step 4: Crossover;

Step 5: Mutation;

Step 6: Do the chromosomes satisfy the constraints? If yes, go to step 8; and if no, go to step 7;

Step 7: Adjustment;

Step 8: If the number of generations reach a predetermined value (termination criteria), stop GA; and if no, go to step 3;

This optimization problem can be effectively solved by GA according to the above procedures. Some numerical examples are shown in the next section.

3.3 Numerical examples

In this section, two numerical examples are illustrated. The first one is a four-node system and the result of the GA is compared with that of the exhaustive search algorithm. The second example is a ten-node system. All algorithms have been implemented in VC++ 6.0 on a Pentium II 266 MHZ processor with 128 M of RAM. It is shown that GA can provide good solution within acceptable time but the exhaustive search algorithm cannot finish the execution within an acceptable time.

3.3.1 A four-node distributed computing system

The topology of the four-node DCS is depicted in Figure 3.2.

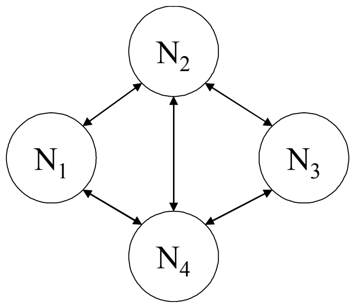


Figure 3.2: Topology of a four-node DCS

There are three computing programs to be executed in the DCS (P1, P2, P3). Files needed for program execution, link reliability used, and completion times of programs are shown Tables 3.1-3.3. The costs of each program are 1, 6, 4, respectively, and there is a cost limit of 15. The size of each file is 3, 5 and 2, respectively. The storage constraints are assumed to be 5, 5, 5, 1 for each node.

Table 3.1: Required files for program execution

Programs	Needed Files
P_1	F_1,F_2
P_2	F_1,F_3
P_3	F_3

Table 3.2: Link reliabilities of a four-node distributed system

Node	1	2	3	4
1	1	0.9		0.7
2	0.9	1	0.75	0.9
3		0.75	1	0.6
4	0.7	0.9	0.6	1

Table 3.3: Completion time of each program and the completion time constraint

Program \ Node	1	2	3	4
1	2.0	3.0	1.5	2.0
2	4.0	6.0	3.1	4.0
3	6.0	9.0	4.5	6.0
Completion Time Constraint	6			

An exhaustive search algorithm was used to search for the optimal assignment of the four-node DCS and two optimum solutions were found. The maximized reliability of the four-node DCS is 0.8745. The optimum assignments are shown in Table 3.4.

Table 3.4: Optimum allocation for the four-node DCS

Solution1	Nodes	N_1	N_2	N_3	N_4
	Programs	P_1,P_2		P_3	P_3
	Files	F_1,F_3	F_2	F_2	
Solution2	Nodes	N_1	N_2	N_3	N_4
	Programs	P_1,P_2		P_3	P_3
	Files	F_2	F1,F3	F_1,F_3	

The GA was also implemented to solve the same problem. The parameters of GA are given below: Population size: 50; Mutation probability: 1.0%; Crossover probability: 70%; and Generation: 40. The GA was repeated 10 times, and results are given in Table 3.5.

Table 3.5: The result of the GA algorithm for the optimum allocation

Solutions	Frequency	DSR	Nodes	N_1	N_2	N_3	N_4
1	4	0.8745	Programs	P_1,P_2		P_3	P_3
			Files	F_1,F_3	F_2	F_2	
2	3	0.8745	Programs	P_1,P_2		P_3	P_3
			Files	F_2	F_1,F_3	F_1,F_3	
3	2	0.864	Programs	P_1	P_2	P_3	P_3
			Files	F_1,F_3	F_2	F_2	
4	1	0.8115	Programs	P_1,P_2		P_3	
			Files	F_2	F_3	F_1,F_3	

Table 3.5 shows the GA cannot guarantee optimum solutions but most results are optimal or near optimal. The average computing duration was 7.8s which was far less than that of the exhaustive search algorithm. With the same parameters, the GA ran 1000 times and the result statistics are presented in Table 3.6:

Table 3.6: The result statistics of the GA

DSR	0.8745	0.864	0.8115
Frequency	408	344	248
Computing duration	6.303		

Table 3.6 shows that, in this case, the probability of the GA finding the optimum solution is the highest (40.8%), which means that the GA can obtain the optimum solution most often when the state space is small. The computing duration of the exhaustive search algorithm is about 60 seconds while the average computing duration of the GA is about 6.303 seconds. It is obvious that GA can obtain the optimal or sub-optimal solution in less time than the exhaustive search algorithm does. For some complex DCS's, the exhaustive search algorithm may not therefore be an acceptable method.

3.3.2 A ten-node distributed computing system

The topology of the ten-node distributed system is depicted by Figure 3.3. The DCS involves ten programs and twelve files. Files needed for program execution, link reliabilities and constraints are shown in Table 3.7 – 3.12. The parameters of GA are as follows: Population size: 50; Mutation probability: 1.0%; Crossover probability: 70%; and Generation: 50.

The ten solutions obtained by the GA are listed in Table 3.13 below.

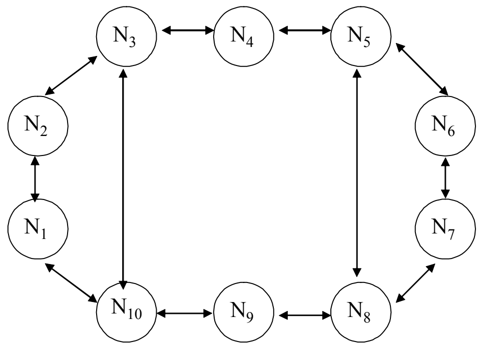


Figure 3.3: Topology of a ten-node DCS

Table 3.7: Needed files for program execution

Programs	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Needed Files	F_1, F_2, F_4	F_2, F_4	F_3, F_5	F_4	F_5, F_3	F_6, F_{11}	F_7, F_{12}	F_8	F_9	F_{10}

Table 3.8: Link reliabilities of a ten-node distributed system

Nodes	1	2	3	4	5	6	7	8	9	10
1	1.00	0.90								0.95
2	0.90	1.00	0.80							
3		0.80	1.00	0.95						0.99
4			0.95	1.00	0.90					
5				0.90	1.00	0.80		0.95		
6					0.80	1.00	0.85			
7						0.85	1.00	0.95		
8					0.95		0.95	1.00	0.90	
9								0.90	1.00	0.85
10	0.95		0.99						0.85	1.00

Table 3.9: Cost of each program and the cost constraint

Programs	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Cost	1	2	4	2	3	5	4	2	3	2
Cost Constraint	35									

Table 3.10: Completion time of each program and the completion time constraint

Program \ Node	1	2	3	4	5	6	7	8	9	10
1	2.0	3.0	4.0	2.0	3.0	4.0	6.0	3.0	2.5	1.0
2	3.0	4.5	6.0	3.0	4.5	6.0	9.0	4.5	3.8	1.5
3	2.6	3.9	5.2	2.6	3.9	5.2	7.8	3.9	3.3	1.3
4	4.0	6.0	8.0	4.0	6.0	8.0	12.0	6.0	5.0	2.0
5	5.0	7.5	10.0	5.0	7.5	10.0	15.0	7.5	6.3	2.5
6	3.4	5.1	6.8	3.4	5.1	6.8	10.2	5.1	4.3	1.7
7	2.2	3.3	4.4	2.2	3.3	4.4	6.6	3.3	2.8	1.1
8	3.2	4.8	6.4	3.2	4.8	6.4	9.6	4.8	4.0	1.6
9	3.8	5.7	7.6	3.8	5.7	7.6	11.4	5.7	4.8	1.9
10	2.8	4.2	5.6	2.8	4.2	5.6	8.4	4.2	3.5	1.4
Completion Time Constraint	10									

Table 3.11: Size of each file

Files	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}
Size	3	5	2	3	4	2	5	3	2	1	3	2

Table 3.12: Size constraint of each node

Node	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}
Storage Constraint	5	5	5	1	10	15	7	2	5	4

Table 3.13: Solution for the ten-node DCS by GA

Solutions	DSR	Duration (s)
1	0.915	381
2	0.913	429
3	0.917	460
4	0.915	412
5	0.915	415
6	0.901	373
7	0.908	365
8	0.910	360
9	0.911	374
10	0.921	422

Table 3.14: One of the best assignments (DSR=0.921) among the ten solutions

Nodes	Programs	Files	Nodes	Programs	Files
N_1	P_2,P_3	F_2	N_6		F_1,F_6,F_9,F_{10},F_{11},F_{12}
N_2		F_1,F_6	N_7		F_1,F_{10},F_{11}
N_3	P_4	F_5	N_8	P_7	F_9
N_4	P_1,P_6		N_9	P_5	F_3,F_4
N_5	P_9	F_7,F_8,F_9	N_{10}	P_7,P_{10}	F_6,F_{12}

From Table 3.13, the maximum reliability that GA can find in ten iterations is 0.921 with one of the assignments as shown in Table 3.14. As no results have ever been presented for a model which considers both program and file allocation, there are no benchmarks available in the literatures to assess the quality of the GA solution for large problems. Also, there are no relaxation techniques available to obtain some upper-bound benchmarks for this kind of problems. In this situation, we can only use statistical data (mean and standard deviation) to assess the quality of the GA solution for large problems.

The mean of reliability was 0.913 and the standard deviation was 0.0055. The average computing duration was 399.1 seconds. We believe that these results confirm the effectiveness of GA in finding a good enough assignment in an acceptable execution time.

3.4 Sensitivity analysis

In our optimization model, some of the parameters such as the costs of programs and completion times usually have to be estimated from a source that may not be accurate. Sensitivity analysis have therefore been carried out to determine the effect of these parameters on the optimal assignment of programs and files and the consequent effect on DSR. Such analyses should be helpful to practitioners in determining the accuracy required for this data

3.4.1 Sensitivity to the expected cost of programs

The expected cost C_j of a copy of program P_j is affected by the price fluctuation of software market. A sensitivity analysis of the expected cost parameter C_2 is shown in this section, and similar studies can also be implemented in analyzing other cost parameters.

Taking the example of four-node computing system, let the expected cost C_2 of each copy of P_2 change from 0 to 10 to see the influence of the cost on the optimal assignment of programs and files when other parameters are fixed. Table 3.15 shows the results of the sensitivity analysis.

Table 3.15: Sensitivity analysis of the program cost parameter

C_2	One of the assignment				DSR	Number of Copies			
	N_1	N_2	N_3	N_4		P_1	P_2	P_3	Total
0, 1, 2, 3	P_3 F_1,F_3	P_2 F_2	P_1,P_2 F_2	P_1	0.8745	2	2	1	5
4, 5, 6	$P_1,P_2,$ F_1,F_3	F_2	$P_3,$ F_2	$P_3,$ F_3	0.8745	1	1	2	4
7, 8, 9	P_3 F_1,F_3	P_2 F_2	P_1 F_1	P_1	0.864	2	1	1	4
10	P_1,P_2 F_1,F_3		P_3 F_2		0.8115	1	1	1	3

In general, as the expected cost of program P_2 increases, the DSR decreases due to the number of redundant copies of the programs being reduced by the limitation of budget. From Table 3.13, we can observe that there are four different solutions of assignment when C_2 changes from 0 to 10:

1) $0 \le C_2 \le 3$, the budget is sufficient to purchase 2 copies of P_1 and P_2 and one copy of P_3 (totally 5 copies). With this selection, the maximum DSR can reach 0.8745 by means of one of the assignments given in Table 3.13;

2) $4 \le C_2 \le 6$, the budget is sufficient to purchase 1 copy of P_1 and P_2 and 2 copies of P_3 (totally 4 copies), and the maximum DSR can also reach 0.8745;

3) $7 \le C_2 \le 9$, the budget is sufficient to purchase 2 copies of P_1 and 1 copy of P_2 and P_3 (totally 4 copies), and the maximum DSR is reduced to 0.864;

4) Finally, when $C_2 = 10$, the budget is only sufficient to purchase one copy of all the programs (totally 3 copies), so the DSR decreases to 0.8115 according to the optimal assignment presented in Table 3.15.

The sensitivity analysis shows that, with C_2 in the range 0 to 6, the maximum DSR is robust to the fluctuation of program prices its value being maintained at 0.8745.

The second range of C_2 with a total of 4 copies of programs has the same DSR as the first range with a total of 5 copies of programs. This shows that having too many redundant copies of programs in a distributed system may sometimes be inutile.

3.4.2 Sensitivity to the completion time

The predetermined completion time constraint (C_t) may fluctuate in practice for reasons such as customer requirement changes. Hence the system designer may also want to know what the effect of the C_t fluctuation has on the solved optimal assignment of programs and files.

A sensitivity approach is proposed here to analyze the influence of C_t on the optimal solution. Again taking the four-node example in Section 3.3.1, let C_t change from 4 to 13 with the other parameters unchanged. The results for the sensitivity analysis are given in Table 3.16.

Table 3.16: Results for the sensitivity to the changes of completion time constraint

C_t	DSR	Assignment				Number of Nodes Free from Executing Programs
		N_1	N_2	N_3	N_4	
4	0					
5	0.8115	P2 F1,F3	P1 F2	P3 F2	P1	0
6	0.8745	P1,P2 F1,F3	F2	P3 F2	P3	1
7	0.8745					
8	0.9315	P1,P3 F1,F3	P2 F2	P3 F2		1
9	0.942	P1,P2 F1,F3	P3 F2	P3 F2		1
10	0.942	P2,P3 F1,F3	P1 F2	P1 F2		1
11	0.942					
12	0.942	P1,P2,P3 F1,F3	F2	F2		3
13	0.942					

As shown in Table 3.16, over the range in C_t from 4 to 9, the more time there is available for all the programs (C_t), the higher the DSR that can be obtained according to the optimal assignment. With $C_t \geq 12$ there is no influence of time constraint (C_t) on the DSR when all the programs can be completed on a single node, such as N_1. On the other hand, if $C_t < 4.5$, the DSR will always be 0 because the least completion time of program P_3 is 4.5.

3.5 Discussions

Some related work is worth mentioning here. Kumar *et al.* (1995a) developed a genetic algorithm (GA) to solve a file allocation scheme. In their scheme, the objective function was to maximize the distributed program reliability (DPR) when the topology of the system, program distribution, files needed for program execution and reliability parameters were given. From the viewpoint of system level, the distributed systems reliability (DSR) can describe the system better than the DPR. Hence, the objective of our optimization model was to maximize the DSR. When the number of programs is set to 1, the objective to maximize DSR is the same as maximizing DPR, the objective function in Kumar *et al.* (1995a).

At the same time, when the topology of the DCSs is given, the DSR depends mainly on the allocation of various resources such as the assignment of programs and needed files (Kumar *et al.* 1986, Raghavendra *et al.* 1998). Hence, to maximize the DSR, the file allocation and program allocation should be considered together. Our algorithm deals with both the program allocation and file allocation to maximize the DSR. When the program allocation is fixed, the models will degenerate to the file allocation problem discussed by Kumar *et al.* (1995a).

In the optimization model of Kumar *et al.* (1995a) the different constraints, for example, the total number of copies of each file and the memory constraint at each node, are discussed. In our model, some additional constraints such as the cost constraint and completion time constraints are considered. Although more constraints make the GA more difficult to implement, they make the optimization model more practical.

Kartik & Murthy (1997) presented a heuristic algorithm to solve the program allocation problems for maximizing the DSR. In their algorithm, the network topology of the computer system was assumed cycle-free, which means that there exists one unique path between any pair of nodes.

Our optimization model does not limit the topology because GEAR (Kumar & Agrawal 1993) used by our model does not limit the network topology of the computer system. In addition our optimization model permits redundancies, which is a more general situation and considers both the program allocation and file allocation together.. Our optimization model is therefore more general and practical than that of Kartik & Murthy (1997).

This chapter presented an optimization model that considered file allocation and program allocation together. Two solution algorithms were developed to solve the problem. The first is an exhaustive search algorithm, which can guarantee to find all the optimum solutions but at the expense of long run-time. The second is a genetic algorithm, which is more effective in run time than the first (especially for some complex DCS's). The genetic algorithm is therefore strongly recommended when the DCS is too complex to be solved by the exhaustive algorithm in an acceptable time,.

A sensitivity analysis was also conducted, which showed that extending completion time might improve the DSR and release more computers for alternative tasks without sacrificing the DSR.

Chapter 4

A Reliability Oriented Tabu Search for Distributed Computing Systems

In Chapter 3, an optimization model for program and file assignment with the objective to maximize the distributed system's reliability (DSR) was presented and a Genetic Algorithm (GA) was proposed to solve the problem. For many combinatorial optimization problems, Genetic Algorithm can provide excellent results. For example Vidyarthi & Tripathi (2001) used a simple GA to optimize the reliability of a DCS with task allocation, which provided better results than that of Shatz *et al.* (1992). However, GA is a population-based search, and requires the evaluation of multiple prospective solutions (i.e., a population) over many generations. Hence, for some complex problems, GA's may need significant amount of computational effort.

Tabu Search (TS) is a competing meta-heuristic method for many of the same large and complex combinatorial optimization problems. Beginning with an initial feasible solution, successive "moves" to superior solutions are made within a neighborhood. To avoid convergence to a local optimum, particular moves are temporarily deemed to be "tabu".

Combined with other methods, TS can provide even better results. Budenbender *et al.* (2000) propose a hybrid Tabu Search/Branch-and-Bound algorithm to solve a transportation network design problem. Chen & Lin (2000) combined Tabu search and noising method to solve a special version of the task allocation problem that included both capacity constraint and number of task constraints along with the inclusion of fixed cost.

Unlike GA's, TS is not population-based but successively moves from solution to solution. This offers some potential for improved efficiency if it also provides the same or improved quality of solutions for the same execution time. Pierre & Elgibaoui (1997) applied a Tabu Search to the topological design of computer networks with a reliability constraint and this provided better solutions than GA and Simulated Annealing (SA). Balicki & Kitowski (1993) considered three evolutionary algorithms to solve a three-criteria optimization problem of finding a set of Pareto-optimal task assignments, and finally recommended the algorithm with Tabu mutation. Subrata & Zomaya (2003) used GA, TS, and ant colony algorithm (ACA) to solve the reporting cells planning problem, and TS showed the best performance.

In some cases, however, GA outperforms TS. Mayer *et al.* (1998) concluded that TS had methodological flaws when applied to multi-dimensional systems with continuous independent variables, and GA's were found to be both efficient and successful for this kind of problem. Braun *et al.* (2001) compared 11 different heuristics including GA, TS, SA and A^* for mapping a class of independent tasks onto heterogeneous DCSs. The results showed that GA consistently gave the best results.

As there are the widely differing views on the efficiency of available methods, this chapter compares GA and TS to gauge their suitability for solving the program and file allocation problem.

The chapter is organized as follows. Section 4.1 describes a TS algorithm to solve the optimization problem. Section 4.2 reports on two numerical examples to compare GA and TS. Section 4.3 presents a parallel TS to further improve the performance of the TS. Section 4.4 reports the computational results of the parallel TS. Finally, Section 4.5 presents the conclusions of the chapter.

4.1 A TS algorithm

The optimization model has been presented in Chapter 3. In order to obtain the solution for this optimization model, an exhaustive search algorithm and a genetic algorithm were presented. In this chapter, a TS algorithm is proposed to solve the same problem. The exhaustive algorithm can guarantee the optimal solution but is computationally complex and, for large scale problems cannot obtain the optimal solution in an acceptable time. Therefore, for small-scale problems, the results of the exhaustive algorithm were used to evaluate the results of TS and GA. For large scale problems, the results of the TS and the GA were compared with each other.

TS, developed by Glover (1989, 1990), is a general purpose heuristic technique. It has been successfully applied to a variety of combinatorial problems. An important feature of TS is the Tabu list (also called the short-term memory) which records those solution states that are not permitted at the current iteration. Restricting the next move to only non-Tabu state solutions can prevent cycling and help to overcome local optimality. However, this may result in rejecting some worthwhile moves. Therefore, a solution state remains “tabu” only for a number of iterations.

In this research, TS combined with branch-and-bound is employed and particular features such as “back-tracking” and “restarting” are incorporated. At the same time, a greedy algorithm was used to generate an initial solution.

4.1.1 Basic initial solution

We attempt to construct an initial feasible, and hopefully good, solution by a greedy algorithm as follows:

- Compute the priority of each node and sort them

  First, compute the priorities of the nodes which is defined as the following equation:

$$\Pr(N_i) = 1 - \prod_{j=1}^{l_i} (1 - R_{i,t(j)}) \tag{4.1}$$

  where

  N_i is the node i;

  $R_{i,t(j)}$ is the reliability of the link between node N_i and node $N_{t(j)}$;

  $t(j)$ is the jth node which is directly linked with node N_i ;

  l_i is the number of the nodes which are directed linked with node N_i.

  After computing the priorities of the processors, sort the processors into the processor sequence according to their priorities in descending order.

- Compute the priority of each program and sort them. The priority of the program is the number of the file that the corresponding program needs. The programs are sorted into the program sequence according to their priority in descending order

- Compute the priority of each file and sort them. The priority of the file is the number of the programs that need the file. The files are sorted into the file sequence according to their priority in descending order.

- Assign the programs and files to the processor

  Programs are assigned in the program sequence to the processors until Completion Time Constraint is violated. After this, the next processor in the processor sequence is chosen and the procedure is repeated until all the programs are assigned.

  Files are assigned in the file sequence to the processors until Storage Constraints are violated. After this, the next processor in the processor sequence is chosen and the procedure is repeated until all the files are assigned. This method can guarantee that the solution is feasible.

As the feasible solutions obtained by the greedy algorithm can be far away from optimal in terms of the objective function value, we designed an improved algorithm that attempts to improve the initial solution. It combines the branch-and-bound optimization algorithm with TS. The whole solution space can be partitioned into subsets according to the number of the copies of the programs and files. Where the number of copies of the program and file are at maximum we refer to this as the *saturated state* case. In this case, different copies of the program or file are treated as different elements. We refer to the different copies of the program P_i as $P_i\,(j)$; and different copies of the file F_i as $F_i(j)$.

Proposition 1: Given two program and file sets A and B, if the number of copies of each program or file in set A is not less than the number of copies of the corresponding program or file in set B, then the reliability of the optimal allocation of the set A is not worse than that of the optimal allocation of the set B.

Proof:

Given program and file sets A and B, where the copies of each program or file in set A are not less than corresponding copies of program or file in set B, i.e., $B \subseteq A$.

Let C denotes the set which consists of the different elements of set A and set B;

DSR_b^* denote the DSR when the set B is optimally allocated;

DSR_a^* denote the DSR when the set A is optimally allocated;

S_b^* denote the set of all the sub-networks that satisfy the program requirement and file requirement when the set B is optimally allocated;

S_a denote the set of all the sub-networks that satisfy the program requirement and file requirement when the set B is optimally allocated and the set C is randomly allocated;

S_a^* denote the set of all the sub-networks that satisfy the program requirement and file requirement when the set A is optimally allocated.

If $A = B$, then $DSR_a^* = DSR_b^*$.

If $B \subset A$,

Because the DSR is computed by evaluating all the subnetworks(or sub-trees) that satisfy the program requirement and file requirement (Kumar & Agrawal 1993), then

$DSR_b^* = f(S_b^*)$

$$DSR_a = f(S_a) = f(S_b^* + S_c) = 1 - (1 - DSR_b^*)(1 - \alpha) = DSR_b^* + (1 - DSR_b^*)\alpha$$

where S_c is the increase of the sub-networks due to the set C.

α is the reliability of the extra subnetworks set S_c.

It is evident that: $S_c \geq 0$, so $\alpha \geq 0$;

$DSR_b^* \leq 1$; so $(1 - DSR_b^*) \geq 0$; so $(1 - DSR_b^*)\alpha \geq 0$;

So $DSR_a \geq DSR_b^*$

And it is evident that: $DSR_a^* \geq DSR_a$.

So $DSR_a^* \geq DSR_b^*$.

■ QED

The above result suggests that we only need to search the program and file sets which are in *saturated states*.

4.1.2 Neighborhood and candidate list

Because the neighborhood significantly affects the solution quality, it is necessary to clarify how the neighborhood is defined.

The neighborhood of current solution is a subset of the whole solution space, in which each solution can be reached from current solution by an operation called a *move.*

A common way to explore a neighborhood is to generate a candidate list of the possible moves and then to evaluate each one until an acceptance criterion is met. However, the neighborhood to be examined can be quite large for the problem. An answer to this is to use an appropriate candidate list strategy which selects a *subset* of the available moves to narrow down the examination of the elements of the neighborhood. The simplest of these strategies is to simply pick this subset at random (Reeves 1993) — a random subset candidate list strategy. Because the neighborhood is totally random, we therefore adopt the random candidate list strategy.

4.1.3 Definition of moves

Adaptation of TS to a specific problem mainly relies on the definition of moves. The moves describe how to explore the neighborhood. In this case, there are eight kinds of moves: add program, add file, reduce program, reduce file, exchange programs, exchange files, move program, and move file.

- Add program: Randomly select a program. This program should not belong to the “Tabu List of Added Programs” and adding one more copy of this program should not violate the cost constraint. Then add one copy of this program to a randomly selected processor that should satisfy completion time constraints.

- Add file: Randomly select a file. This file should not belong to the “Tabu List of Added Files”. Then add one copy of this file to a randomly selected processor that should satisfy storage constraints.

- Reduce program: Randomly select a program which has more than one copy, then delete it.

- Reduce file: Randomly select a file which has more than one copy. Then delete it.

- Exchange programs: Two different programs on different processors exchange their locations. The new solution should not belong to the “Tabu List of

Solutions". After exchanging, the corresponding two processors should satisfy the completion time constraint.

- Exchange files: Two different files on different processor exchange their locations. The new solution should not belong to the "Tabu List of Solutions". After exchanging, the corresponding two processors should satisfy the storage constraints.

- Move program: Move a randomly selected program on a randomly selected processor to another processor. The new solution should not belong to the "Tabu List of Solutions". After moving, the processor should satisfy the completion time constraint.

- Move file: Move a randomly selected file on a randomly selected processor to another processor. The new solution should not belong to the "Tabu List of Solutions". After moving, the processor should satisfy the storage constraint.

4.1.4 Tabu lists

There are four Tabu lists: "Tabu List of Solutions", "Tabu List of Added Programs", "Tabu List of Added Files" and "Tabu List of Program and File Set".

- "Tabu List of Solutions" records the recent solutions.

- "Tabu List of Added Programs" records the recent programs which were added to processors.

- "Tabu List of Added files" records the recent files which were added to processors.

- "Tabu List of Programs and File Set" records the recent program and file set.

4.1.5 Intensification strategies

Intensification strategies are designed to encourage solution features historically found to be good. They may also initiate a return to an attractive region to search it more thoroughly. In this research, two types of intensification strategies are implemented which may be viewed as a form of back-tracking.

Type 1 intensification strategy: After the predefined number (I_1) of moves from the solution x, if a better solution than the solution x cannot be found, then return to the solution x.

Type 2 intensification strategy: During the searching of the set of programs and files S, a best solution was found. After searching the predefined number (I_2) of different program and file sets, a better solution than the best solution found in the set S cannot be found, then search the program and file set S again.

4.1.6 Diversification strategies

TS diversification strategies are designed to lead the search into new regions and by this means can increase the effectiveness of exploring the solution space. The main purpose of diversification is to prevent searching processes from cycling. In addition, diversification can impart robustness to the search.

In this chapter, one diversification strategy, namely restarting, is implemented. Thus within the program and file set, we search the solution space for a predefined time before changing to a different program and file set.

4.1.7 The procedures of TS

The procedures of TS used in the chapter are the following:

Notations:

DSR_{best} :	DSR of x_{best} ;
$DSR_{tempbest}$:	DSR of $x_{tempbest}$;
S :	current program and file set;
S_{best} :	program and file set where x_{best} was found;
TL_N :	Tabu List of Program and File Set;
x :	current solution;
x_{best} :	the best solution found until now;
$x_{tempbest}$:	the temporary best solution;

The DSR of solution x can be computed by $DSR = f(x)$.

```
1.  Generate an initial feasible solution $x$; $TL_N = \Phi$
2.  $x_{best} = x$; $DSR_{best} = f(x_{best})$
3.  while s<s-max do
4.      while ($x$ satisfies cost constraint and completion time constrain) do
5.          Add program and update the "Tabu List of Added Programs"
6.      end while
7.      while ($x$ satisfies storage constraints) do
8.          Add file and update the "Tabu List of Added Files"
9.      end while
10.     if $S \notin TL_N$ then
11.         Update the $TL_N$
12.         $x_{best} = x$; $DSR_{tempbest} = f(x_{tempbest})$
13.         while t < t-max do
14.             Implement the type 1 intensification strategy
```

```
15.                 Sequentially implement one of "Exchange programs",
                    "Exchange files", "Move program", "Move file" and update
                    "Tabu List of Solutions"
16.                 $DSR = f(x)$
17.                 **if** $DSR > DSR_{tempbest}$ **then**
18.                     $x_{tempbest} = x$; $DSR_{tempbest} = DSR$
19.                 **end if**
20.                 t = t + 1
21.             **end while**
22.             Implement the type 2 intensification strategy
23              **if** $DSR_{tempbest} > DSR_{best}$
24.                 $x_{best} = x_{tempbest}$; $DSR_{best} = DSR_{tempbest}$
25.             **end if**
26.     **end if**
27.     **while** ( $x$ satisfy the basic constrain) **do**
28.             Reduce program
29.             Reduce file
30.     **end while**
31.     s = s + 1
32. **end while**
33. Return the best solution $x_{best}$ and $DSR_{best}$
```

4.2 Numerical examples

In this section, two numerical examples are illustrated. The first one is a four-node system and the results of the TS and GA are compared with that of an exhaustive search algorithm. The second example is a 10-node system and the result of the TS is compared with that of a GA. All algorithms have been implemented in VC++ 6.0 on a Pentium III 500 MHZ processor with 128 M of RAM.

4.2.1 A four-node distributed computing system

An exhaustive search algorithm was used to search the optimal assignment for the same four-node DCS used in section 3.3.1 and two optimum solutions were found. The maximized reliability of the four-node DCS is 0.8745 and the computing duration is 42 seconds.

A TS and GA were also implemented to solve the same problem. The parameters of TS are given in Table 4.1 and those of GA in Table 4.2. TS and GA were run 1,000 times and the resulting statistics are shown in Table 4.3.

Table 4.1: The parameters of TS for 4 node DCS

Length of "Tabu List of Solutions"	10
Length of "Tabu List of Added Programs"	1
Length of "Tabu List of Added Files"	1
Length of “Tabu List of Program and File Sets"	3
l_1	10
l_2	20
s-max	25
t-max	50

Table 4.2: The parameters of GA for 4 node DCS

Population size	50
Mutation probability	1.0%
Crossover probability	70%
Generation	40

Table 4.3: The result statistics of the TS and GA for four node DCS

Maximized DSR	0.8745		
DSR	0.8745	0.864	0.8115
DSR/DSR_{max}	100%	98.80%	92.80%
Frequency (TS)	86.3%	13.7%	0
Frequency (GA)	72.7%	15.6%	11.7%
	TS (s)	GA (s)	
Mean Computing duration	2.372	3.86	
Maximal Computing duration	4	5	
Minimal Computing duration	2	3	

The exhaustive search algorithm could find two optimal solutions and the maximized DSR was 0.8745 in 42 seconds. As shown in Table 4.3, the TS cannot guarantee optimal solutions but most results are either optimal or near optimal. In this case the probability for the TS to find the optimum solution is 86.3%, which means that the TS can get the optimum solution most often when the state space is small. The average computing duration of the TS is about 2.372 seconds which is far less than the duration time of exhaustive search algorithm. It is evident that TS can obtain the optimal or good enough solutions in far less time than that required by the exhaustive search algorithm. The performance of GA is very similar to that of TS, i.e., GA can also obtain the optimal or good enough solutions in far less time than that required by the exhaustive search algorithm. Comparing the results of TS with those of GA shows however that TS outperforms the GA with shorter computing time and better solution quality.

4.2.2 A ten-node distributed computing system

For the same 10-node distributed system as used in section 3.3.2, incorporating 10 programs and 12 files, the exhaustive algorithm cannot be run to completion. Hence only TS and GA were used to solve this example problem. The parameters of TS are given in Table 4.4 while those of the GA are shown in Table 4.5.

Table 4.4: The parameters of TS for 10 node DSC

Length of "Tabu List of Solutions"	50
Length of "Tabu List of Added Programs"	5
Length of "Tabu List of Added Files"	5
Length of “Tabu List of Program and File Sets"	10
l_1	10
l_2	20
s-max	25
t-max	50

Table 4.5: The parameters of GA for 10 node DSC

Population size	50
Mutation probability	1.0%
Crossover probability	70%
Generation	50

TS and GA were run 100 times, respectively, to get an accurate representation of the actual mean value. The statistic results are shown in Table 4.6. In the Table, mean DSR is the average of the solution values obtained from the 100 runs. It represents the expected DSR obtained using the algorithms. Mean duration is the average of the computing times and represents the expected duration.

Table 4.6: The result statistics of the TS and GA for ten node DSC

	TS	GA
Mean DSR	0.927	0.923
Min DSR	0.913	0.901
Max DSR	0.946	0.941
Standard Deviation of DSR	0.006	0.004
Mean Duration	118.8	268.4
Min Duration	103	200
Max Duration	139	295

Table 4.6 shows that in this case TS obtained a mean DSR of 0.927. This is higher than the mean DSR of 0.923 obtained by GA . The maximum DSR that TS obtained was 0.946 which is higher than the maximum DSR of 0.941 obtained by GA. Also, the

minimum DSR TS obtained was 0.913 which is higher than and the minimum DSR of 0.901 obtained by GA.

To make it easy to read the results, we present them as a histogram in Figure 4.1. The histogram clearly shows that TS produces solutions of better quality than those produced by GA.

The mean computing duration of TS was 118.8 seconds which is far less than the mean computing duration of GA at 268.4 seconds. Hence we conclude that in this case TS outperforms GA with shorter computing time and better solution quality.

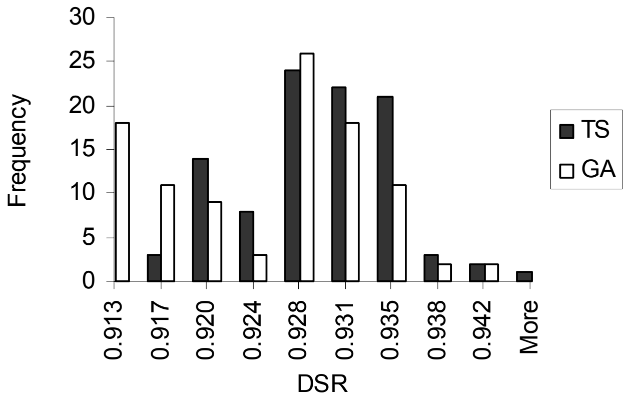


Figure 4.1: Histogram of the results of TS and GA for 10 node DCS

4.3 A Parallel Tabu Search

For this reliability oriented program and file allocation problem, TS outperforms GA. However, in some practical settings where scheduling must be produced within a short time interval, TS may not finish execution within the time limit. To reduce the execution time, a possible method is to parallelize the algorithm. Hence, a Parallel Tabu Search (PTS) is proposed to further improve the performance of the TS.

The PTS has almost the same procedures as the sequential TS. However, the initial solution generation is different from that of the sequential one. The sequential one uses a greedy algorithm to generate one initial solution, but the parallel one needs multiple initial solutions, which should all be different in order to search the solution space along different trajectories. Initial solutions are generated randomly, but all of them should be in *saturated state.*

There are three types of parallelization strategies that are often used in combination optimization:

1) Parallelization of operations within an iteration of the solution method,
2) Decomposition of problem domain or search space,
3) Multi-search threads with various degrees of synchronization and cooperation.

For this problem, the solution space can be partitioned into subsets, e.g. different program and file sets, so multiple search paths are maintained in parallel to search different subsets and accelerate the TS. The implementation runs on a network of workstations and follows a master-slave scheme. Each slave performs a partial TS, e.g. from step 12 to step 22 in the sequential TS, and the master co-ordinates the work and feeds the slaves with a new program and file set in the saturated state.

4.4 Computation results of PTS

The parallel TS was coded in C++ and process communications were handled by the *message passing interface* (MPI) software. The algorithm was implemented on a cluster of PCs and each PC is with a Pentium processor IV 2.4 GHZ processor.

The ten-node DCS described in Chapter 3 is again used to measure the performance of the proposed PTS. The results confirm that the PTS can find near-optimal solutions within acceptable execution times. It is worth noting that the PTS only searches the program and file sets that are in the saturated state that the number of copies of the programs and files is as large as possible. This is a kind of branch and bound method to make the search space far smaller than the original one. Consequently, the run time is reduced significantly.

The PTS also uses the random subset candidate list strategy to reduce the search space because even within one program and file set the search space is quite large. In addition, the PTS searches the regions more thoroughly by using two intensification strategies, and in these regions the probability is high that better solutions can be found. Furthermore, the PTS also uses a diversification strategy to overcome the local optimality and to increase the effectiveness and the robustness in exploring the solution space.

The results obtained support the idea that TS is a general-purpose heuristic technique and that a well-designed TS can obtain near-optimal solutions in acceptable time, especially when TS combines with other optimization techniques.

The speedup was used to measure the performance of the PTS. This is defined as the ratio between the sequential and parallel running time: $speedup(i) = T_1 / T_i$, where T_1 is the running time of the sequential program and T_i is the running time of the parallel program on *i* processors.

The results presented in Figure 4.2 indicate that the speedup of the PTS basically grows linearly when the number of processor is not very large. A possible reason for this is that the solution space is partitioned into subsets; every processor searches a subset; and only small amount of communication is needed. In this situation, the speedup would be expected to be roughly linear.

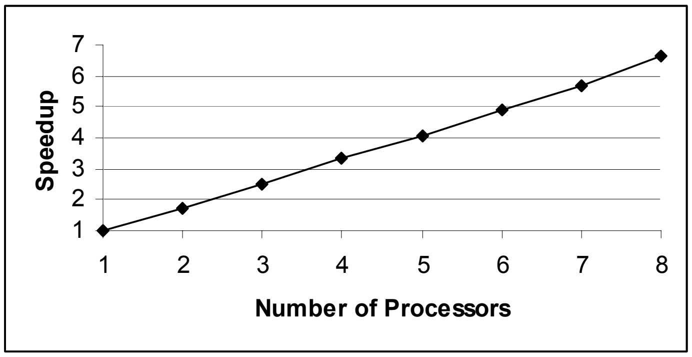


Figure 4.2: Speedup of the PTS

In addition, the mean of the solutions was computed for each case when processor number changes. This is shown in Table 4.7 where it can be seen that increasing the number of processors has negligible effect on the solution quality.

Table 4.7: The result statistics of the PTS when the processor number changes

Processor Number	Mean	Standard Deviation
1	0.94448	0.00175
2	0.94496	0.00189
3	0.94468	0.00172
4	0.94430	0.00154
5	0.94452	0.00157
6	0.94451	0.00171
7	0.94523	0.00190
8	0.94438	0.00166

4.5 Conclusions

In this chapter a TS was developed to solve the same program and file assignment problem as that presented in Chapter 3 with the objective of maximizing the DSR. The results of the TS were compared with those of GA. The results of the two numerical problems considered showed that TS outperformed GA with shorter computing time and better solution quality. However, it was evident that the design of a good TS requires far more insight into the problem and that much more effort is needed compared with the requirements for implementing a good GA for the same problem. Hence if we do have good knowledge of the state space, TS should be used, otherwise, GA may be a better choice.

To further improve the performance of the sequential TS, a PTS was proposed. The numerical results showed that the speedup of the PTS basically grew linearly when the number of processor was not very large. The simulation result also showed that the solution quality was not obviously affected by the number of processors used.

Chapter 5

A Completion Time Oriented Iterative List Scheduling for Distributed Computing Systems

In Chapters 3 and 4, a system reliability oriented allocation model was presented and several algorithms were proposed to solve the problem. Beside system reliability, the completion time is another very important requirement for distributed computing, so the scheduling of parallel applications to minimize the overall completion time (schedule length) is highly critical. A popular representation of the parallel application is the directed acyclic graph (DAG) in which the nodes represent application tasks and the directed arcs or edges represent inter-task dependencies. As the problem of finding the optimal schedule is NP-hard (Gary & Johnson 1979) in the general case, extensive heuristic algorithms have been proposed. These algorithms may be broadly classified into the following four categories:

- Task-Duplication-Based (TDB) scheduling (Papadimitriou & Yannakakis 1990, Colin *et al.* 1991, Palis *et al.* 1996, Ahmad & Kwok 1998, Darbha & Agrawal 1998, Park & Choe 2002, Kang & Agrawal 2003)

- Bound number of processors (BNP) scheduling (Hwang *et al.* 1989, McCreary & Gill 1989, Wu & Gajski 1990, Maheswaran & Siegel 1998, Park & Kim

2002, Radulescu & Gemund 2002, Topcuoglu *et al.* 2002, Dhodhi *et al.* 2002, Diaz *et al.* 2003, Oguz *et al.* 2003)

- Unbounded number of clusters (UNC) scheduling (Sarkar 1989, Wu & Gajski 1990, Kim & Yi 1994, Yang & Gerasoulis 1994, Kwok & Ahmad 1996, Srinivasan & Jha 1999)
- Arbitrary network topology (ANP) scheduling (El-Rewini & Lewis 1990, Sih & Lee 1993)

In TDB scheduling, the basic idea is to reduce the communication overhead by redundantly allocating some tasks to multiple processors. Non-TDB algorithms which assume arbitrary task graphs with arbitrary time on nodes and edges can be divided into two categories: one category assumes that the processors are fully connected to each other meaning that there is no communication contention; the other category assumes that the processors are linked by an arbitrary network topology (ANP) meaning that the scheduling process must consider the communication contention. The former category can be divided into further two categories: unbounded number of clusters (UNC) scheduling algorithms and bound number of processors (BNP) scheduling.

The algorithm presented in this chapter belongs to the last category, which is the most common case in the real world. More detailed descriptions and classifications of various scheduling strategies can be found in Kwok & Ahmad (Kwok & Ahmad 1999b).

List scheduling is a very popular method for BNP scheduling. The basic idea of list scheduling is to assign priorities to the tasks of the DAG and place the tasks in a list arranged in descending order of priorities. A task with a higher priority is scheduled before a task with a lower priority and ties are broken using some method. To compute the priorities of the tasks, the DAG must be labeled with the computation time of the tasks and the communication times of the edges. We differentiate the computation time label of a task in a DAG from the actual computation times of a task on all the processors, and refer to the former as the *time-weight* of the task. Similarly we refer to the communication time label of an edge on the DAG as the *time-weight* of the edge.

In a homogeneous distributed computing system, the computation times of a task on different processors are the same. Hence the time-weight of a node is its computation time on any processor. Similarly, in a homogeneous DCS, the communication times between two tasks on any link are same. Hence the time-weight of an edge is the communication time between the corresponding two tasks on any link.

In a heterogeneous DCS, on the other hand, the computation times of a task on different processors may be different, and so is the communication times between two tasks on different links. Hence, the time-weight of every node and the time-weight of every edge, which are labeled on the DAG, have to be computed during the scheduling process.

Several variant list schedulings have been proposed to deal with the heterogeneous environment, for example, Mapping heuristic (MH) (El-Rewini & Lewis 1990), Dynamic-Level Scheduling (DLS) algorithm (Sih & Lee 1993), Levelized-Min Time (LMT) algorithm (Iverson *et al.* 1995), and Heterogeneous Earliest-Finish-Time (HEFT) algorithm (Topcuoglu *et al.* 2002).

The HEFT algorithm significantly outperforms the DLS algorithm, MH and LMT algorithm in terms of average schedule length ratio, which is normalizing the schedule length to a lower bounder (Topcuoglu *et al.* 2002), speedup etc. The HEFT algorithm selects the task with the so-called highest upward rank value at each step and assigns the selected task to the processor, which minimizes its earliest finish time with an insert-based policy. When computing the priorities, the algorithm uses the task's mean computation time on all processors and the mean communication rates on all links. We believe that use of the mean is inadequate for task scheduling.

In this chapter, we propose an iterative algorithm that uses list scheduling for task allocation in heterogeneous computing systems. The algorithm generates an initial solution with moderate quality and then improves the solution iteratively. The priority for constructing the scheduling list and the processor selection policy are selected according to the conclusions of Kwok and Ahmad (Kwok Ahmad 1999a).

In each iteration step, the time-weights of the nodes and edges of the DAG are updated using results from the previous iteration. The initial solution is obtained by the mean computation times of all tasks on all processors as the time-weight of the corresponding node and the mean communication time of all communication links as the time-weight of the corresponding edge. During the iterative steps, the results of the previous iteration are used to compute and update the time-weights of the nodes and edges in order to construct a new list. The algorithm keeps the best solution found during the iterations and returns it on termination. The initial step happens to be the same as the HEFT algorithm (Topcuoglu *et al.* 2002). However, with subsequent schedule improvements, it can potentially find better schedules than the other algorithms mentioned above.

The algorithm has been tested on a large number of randomly generated problems of different sizes and two real applications. It was found that in the majority of the cases, there were significant improvements made to the initial schedules, which means that the proposed algorithm outperforms HEFT, DLS, MH, and LMT algorithms in terms of the average schedule length. In particular, the algorithm performs better when the tasks to processors ratio is large.

This chapter is organized as follows. Section 5.1 provides a formal description of the task scheduling problem. Section 5.2 introduces our scheduling algorithm. Section 5.3 gives a numerical example. Section 5.4 investigates the performance of our algorithm in various heterogeneous computing systems. Finally Section 5.5 draws some conclusions.

5.1 Task-scheduling problem

Notations:

$c_{i,j,k,l}$:	communication time from task v_i to task v_j when task v_i was assigned to processor p_k and task v_j was assigned to processor p_l;
$c_{i,j}^{s}$:	time-weight of the directed edge from task v_i to task v_j during the s-th iteration which is used to compute the priorities of the tasks;
$d_{i,j}$:	data transfer size (in bytes) from task v_i to task v_j;
$e_{i,j}$:	directed link from i-th task to j-th task;
$EST(v_i, p_j)$:	earliest computation start time of task v_i on processor p_j;
$EFT(v_i, p_j)$:	earliest computation finish time of task v_i on processor p_j;
p:	number of processors available in the system;

p_i:	i-th processor in the system;
$r_{i,j}$:	communication rate (in bytes/second) between processor p_i and processor p_j;
v:	number of tasks in the application;
v_i:	i-th task in the application;
$w_{i,j}$:	computation time to complete task v_i on processor p_j;
w_i^s:	time-weight of task v_i during the s-th iteration, which is used to compute the priorities of the tasks.

An application is represented by a directed acyclic graph **G**=(***V***, ***E***), where ***V*** is the set of v tasks that can be executed on any of the available processors; $\boldsymbol{E} \subseteq \boldsymbol{V} \times \boldsymbol{V}$ is the set of e directed arcs or edges between the tasks representing the dependency between the tasks.

For example, if $e_{i,j} \in E$, then task v_j cannot start before task v_i completes its execution. A task may have one or more inputs. When all its inputs are available, the task is triggered to execute. After its execution, it generates its outputs. A task with no parent node in the DAG is called an *entry task* and a task with no child node in the DAG is called an *exit task.*

Without lost of generality, we assume that the DAG has exactly one entry task v_{entry} and one exit task v_{exit}. If multiple exit tasks or entry tasks exist, they may be connected with zero time-weight edges to a single pseudo exit task or a single entry task that has zero time-weight. In addition, the system includes a set of p processors which are assumed to be fully linked. Hence there is no communication contention (Darbha & Agrawal 1998).

We assume that there is just one task executing on a processor at any one time and each processor is equipped with a queue to hold tasks waiting to execute on the processor. When all inputs of a task are available, the task is triggered to execute. After its execution, the outputs are generated. Once the execution of a task is completed, the processor is assumed to be immediately available for the execution of the next task scheduled on that processor. However, it is possible for a task to receive data from predecessors while another task is been executed, and likewise, it is possible for a task to send data to successor tasks. Furthermore, the output data items produced by the completed task are assumed to be available for all successor tasks to be executed on that processor. If multiple output data items produced by a task are to be transferred to successor tasks scheduled on other processors, then these data items are assumed to be transferred to their respective destination concurrently.

The communication time $c_{i,j,k,l}$ from task v_i to task v_j, when task v_i was assigned to processor p_k and task v_j was assigned to processor p_l is

$$c_{i,j,k,l} = d_{i,j} / r_{k,l} \tag{5.1}$$

The earliest execution start time on processor p_j of entry task v_{entry} is

$$EST(v_{entry}, p_j) = 0 \tag{5.2}$$

To compute the earliest execution start time of other tasks, the assignment of the immediate predecessor tasks must be known. Let us assume that v_m is one of the immediate predecessor tasks of v_i and v_m was assigned to the processor p_n. The earliest execution start time of task v_i on processor p_k is

$$EST(v_i, p_k) = \max\{Available(v_i, p_k), \max_{v_m \in pred(v_i)} (EFT(v_m, p_n) + c_{m,i,n,k})\} \quad (5.3)$$

where $Available(v_i, p_k)$ is the earliest time when processor p_k is available for task v_i execution; $pred(v_i) = \{ v_m \in \boldsymbol{V} \mid e_{mi} \in \boldsymbol{E}\}$ is the set of immediate predecessors of task v_i; $c_{m,i,n,k}$ is the communication time between task v_m and task v_i given that task v_m was assigned to processor p_n and task v_i was assigned to processor p_k. The inner maximization block in Equation (5.3) returns the *ready-time*, i.e., the time when all data needed by task v_i have arrived at processor p_k.

The earliest execution finish time on processor p_e of entry task v_{entry} is

$$EFT(v_{entry}, p_e) = w_{entry,e} \quad (5.4)$$

For other tasks, the earliest execution finish time of task v_i on processor p_k is

$$EFT(v_i, p_k) = w_{i,k} + EST(v_i, p_k) \quad (5.5)$$

After all tasks in the DAG have been scheduled to satisfy all precedence constraints, the schedule length *SL* is the earliest finish time of the exit task v_{exit}. That is

$$SL = EFT(v_{exit}, p_x) \quad (5.6)$$

where exit task v_{exit} has been assigned to processor p_x.

The primary objective of the scheduling problem is to minimize the schedule length *SL* by determining the assignment of tasks to processors subject to the tasks dependency constraints.

5.2 Iterative list scheduling algorithm

5.2.1 Graph attributes used by our algorithm

The time length of a directed path from tasks v_i to v_j is defined as the sum of all the time-weights of the tasks (including v_i and v_j) and time-weight of the edges along the path between v_i and v_j.

A critical path (CP) of a DAG is a path from the entry task to exit task, whose time length is the maximum. The bottom-level (*b-level*) (Kwok & Ahmad 1999a) of task v_i is the longest time-length from task v_i to the exit task and is bounded by the time-length of the critical path of the graph. The *b-level* of a task is a dynamic attribute because the time-weight of an edge may be zeroed when the two incident tasks are scheduled to the same processor.

5.2.2 The priority selection

Kwok and Ahmad (1999a) compared several list scheduling algorithms on a common homogeneous platform and concluded that the Modified Critical-Path (MCP) (Wu & Gajski 1990) algorithm performs better than others in terms of schedule length and running time of algorithms. The MCP algorithm uses the as-late-as-possible (ALAP) time of a task as the priority.

The ALAP time of a task is computed by first computing the time length of CP and then subtracting the b-level of the task from it. First, the MCP algorithm computes the ALAP times of all the tasks and then constructs a list of tasks in ascending order of ALAP times. Ties are broken by considering the ALAP times of the children of a task. The tasks on the list are then scheduled using the insertion approach, one by one to a processor that allows the earliest possible start time.

Because the length of the CP is a constant, our algorithm uses the *b-level* of a task as the priority. Our algorithm first computes the *b-levels* of all tasks and then constructs a list of tasks in descending order of *b-level* values. Ties in *b-levels* are recursively broken using the *b-levels* of the tasks' children.

5.2.3 Scheduling list construction

To construct the scheduling list for the initial solution, the time-weight of every task must be known. The initial time-weight of task v_i is assigned the mean value of the computation time of task v_i on all processors. That is

$$w_i^0 = \frac{\sum_{j=1}^{p} w_{i,j}}{p} \tag{5.7}$$

Similarly, the initial time-weight of the edge from tasks v_i to v_j based on the mean value across all the fully connecting links is

$$c_{i,j}^0 = \frac{d_{i,j}}{\sum_{k=1}^{p-1} \sum_{l=k+1}^{p} r_{k,l} \Big/ ((p^2 - p)/2)} \tag{5.8}$$

At the s-th iteration, suppose task v_i was allocated to processor p_k and task v_j was allocated to processor p_l at the previous iteration, then the time-weight of task v_i is:

$$w_i^s = (\alpha w_{i,k} + \sum_{m=1, m \neq k}^{m=p} w_{i,m}) \Big/ (p + \alpha - 1) \tag{5.9}$$

where α is a non-negative constant. The parameter α is referred to as the weighting factor, which has to be determined either heuristically or empirically. w_i^s is a weighted mean of the computation time $w_{i,j}$ of task v_i on all processors.

If $\alpha > 1$, then more weight is put on $w_{i,k}$. Because processor p_k is the processor to which task v_i was allocated during the (s-1)-th iteration, when we compute the time-weight of task v_i for s-th iteration, i.e., w_i^s, we put a weight on the computation time of task v_i on processor p_k, i.e., $w_{i,k}$. Hence, the time-weight of a task for s-th iteration depends on the assignments of the previous iteration, i.e., the (s-1)-th iteration.

The time-weight of the edge from task v_i to task v_j at the s-th iteration is:

$$c_{i,j}^s = \frac{d_{i,j}}{(\sum_{m}^{p-1} \sum_{n=m+1}^{p} r_{m,n} + (\alpha - 1) * r_{k,l}) \Big/ ((p^2 - p)/2 + \alpha - 1)} \tag{5.10}$$

The *b-level* of task v_i at the s-th iteration is defined by:

$$b^s(v_i) = w_i^s + \underset{v_j \in succ(v_i)}{Max} (c_{i,j}^s + b^s(v_j)) \tag{5.11}$$

where $succ(v_i) = \{ v_j \in \boldsymbol{V} \mid e_{ij} \in \boldsymbol{E} \}$ is the set of immediate successors of task v_i.

For the exit task v_{exit}, since it has no successor, its *b-level* is

$$b^s(v_{exit}) = w_{exit}^s \tag{5.12}$$

Based on the time-weights of the tasks and the time-weights of the edges, the scheduling list is constructed with respect to the *b-level.*

5.2.4 Processor selection step

Kwok and Ahmad (1999a) compared several list scheduling algorithms on a common homogeneous platform and concluded that insertion-based policy is better than the non-insertion-based policy during the processor selection step. Insertion-based policy permits the insertion of a task into an earliest idle time slot between two tasks that are already scheduled on the same processor. Hence, our algorithm assigns the selected task to the processor which minimizes its earliest finish time with an insert-based policy. The time slot must be larger than the computation time of the task being scheduled. In addition, the precedence constraint should be preserved. The procedure for looking for an idle time slot on one processor p_k for task v_i is as following:

```
1.  Compute the inner maximization block in Equation (5.3) as $ready\_time(v_i, p_k)$,
    i.e., the time when all data needed by task $v_i$ has arrived at processor $p_k$.
2.  $Avaiable(v_i, p_k) =$ the finish time of the last task in the task list of processor $p_k$
3.  **While** the task list of processor $p_k$ is not empty && $ready\_time(v_i, p_k)$ < the start
    time of the last task in the task list of processor $p_k$ **do**
4.      **if** (the finish time of the second last task >= $ready\_time(v_i, p_k)$) && (the start
            time of the last task – the finish time of the second last task)>= $w_{i,k}$ **then**
5.          $Avaiable(v_i, p_k) =$ the finish time of the second last task
6.      **else if** (the finish time of the second last task < $ready\_time(v_i, p_k)$) && (the
            start time of the last task – $ready\_time(v_i, p_k)$)>= $w_{i,k}$ **then**
7.          $Avaiable(v_i, p_k) = ready\_time(v_i, p_k)$
8.      **end if**
9.      Delete the last task from the task list of processor $p_k$
10. **end while**
```

11. $EST(v_i, p_k) = \max(Avaiable(v_i, p_k),\ \ ready\_time(v_i, p_k))$

where the task list of processor p_k consists of the tasks which have been assigned to the processor p_k and sorted by ascending finish time.

5.2.5 The procedure of the algorithm

The procedure for the iterative scheduling algorithm is described as following:

```
1.  $s = 0$.
2.  Compute the time-weights of the tasks with Equation (5.7)
3.  Compute the time-weights of the edges with Equation (5.8)
4.  BestSL = a very large number
5.  while $s \le s_{max}$ do
6.      Compute the b-levels for all tasks by traversing graph from the exit task
7.      Sort the tasks into a scheduling list by non-increasing order of b-level
8.      while the scheduling list is not empty do
9.          Remove the first task $v_i$ from the scheduling list
10.         for each processor $p_k$ do
11.             Compute $EFT(v_i, p_k)$ using the insertion-based scheduling policy
12.         end for
13.         Assign task $v_i$ to the processor that minimizes EFT of $v_i$
14.     end while
15.     ScheduleLength = $EFT(v_{exit}, P_{v_{exit}})$
16.     If ScheduleLength < BestSL then
17.         BestSL = ScheduleLength, and the current schedule is the best schedule
18.     end if
```

```
19.         Compute the time-weights of the tasks with Equation (5.9)
20.         Compute the time-weights of the edges with Equation (5.10)
21.     $s = s + 1$
22. **end while**
23. Return the best schedule
```

The initial step is the same as that of the Heterogeneous Earliest-Finish-Time (HEFT) algorithm (Topcuoglu *et al.* 2002) which significantly outperforms Dynamic-Level Scheduling (DLS) algorithm (Sih & Lee 1993), Mapping Heuristic (MH) (El-Rewini & Lewis 1990) and Levelized-Min Time (LMT) algorithm (Iverson *et al.* 1995) in terms of average schedule length ratio, speedup, etc. The improvement step of our algorithm has the potential to produce shorter schedule length than those of the HEFT, DLS, MH and LMT algorithms.

5.2.6 The time-complexity analysis

The time-complexity of scheduling algorithms for DAG is usually expressed in terms of the number of nodes *v*, the number of edges *e*, and the number of processors *p*. The time-complexity analysis for one iteration of our algorithm is as follows:

Computing the time-weights of the tasks and the edges can be done in time $O(vp)$. Computing the *b-levels* can be done in time $O(e+v)$. Sorting the tasks can be done in time $O(v \log v)$. The processor selection for all tasks can be done in time $O((ep + v^2/2) + vp)$, i.e., in time $O(ep + v^2)$. Hence, the time complexity for one iteration is:

$$O(vp + (e+v) + v\log v + ep + v^2) = O(ep + v^2)$$

If s_{max} denotes the maximum number of iterations which is normally small, then the time complexity of the whole algorithm is $O(s_{max}(ep+v^2))$ in the worst case.

For a dense graph when the number of edges is proportional to $O(v^2)$, the time complexity becomes $O(s_{\max}(v^2 p))$.

5.3 Numerical example

Figure 5.1 shows a DAG with 8 tasks and 11 edges. There are two processors available in the heterogeneous computing system. Table 5.1 shows the computation time of each task on every processor. For simplicity, we assume homogeneous communication and the communication times are as labeled on the edges in Figure 5.1.

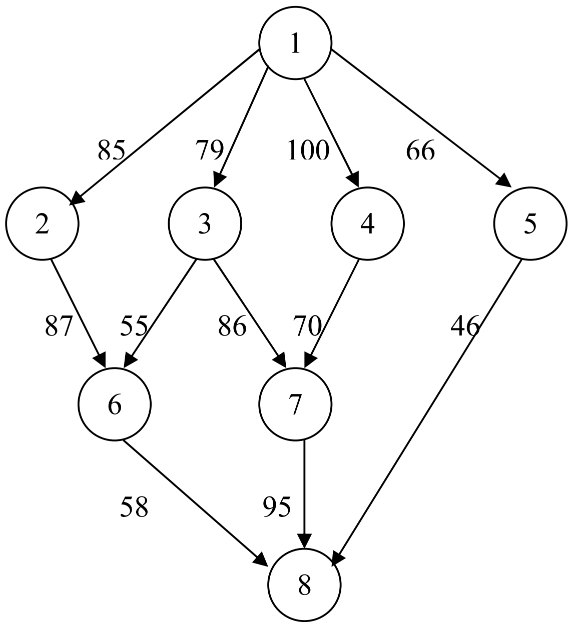


Figure 5.1: A sample directed acyclic graph with 8 tasks

Table 5.1: Computation times of every task on every processor

Task	P1	P2
1	70	84
2	68	49
3	78	96
4	89	26
5	30	88
6	66	86
7	25	21
8	94	36

The time-weights of the tasks are computed by using Equation (5.7) as follows:

$$w_1^0 = (70 + 84)/2 = 77$$

……

Similarly, the time-weights of other tasks can be obtained, as shown in Table 5.2.

The *b-levels* of the tasks are computed as follows:

$$b^0(v_8) = w_8^0 = 65$$

$$b^0(v_7) = w_7^0 + (c_{7,8}^0 + b^0(v_8)) = 23 + (95 + 65) = 183$$

……

$$\begin{aligned} b^0(v_1) &= w_1^0 + \max\{(c_{1,2}^0 + b^0(v_2)), (c_{1,3}^0 + b^0(v_3)), (c_{1,4}^0 + b^0(v_4)), (c_{1,5}^0 + b^0(v_5))\} \\ &= 77 + \max\{(85 + 344.5), (79 + 356), (100 + 310.5), (66 + 170)\} \\ &= 512 \end{aligned}$$

Table 5.2 shows the initial time-weights and *b-levels* of the tasks. The initial scheduling list of the tasks is $\{v_1, v_3, v_2, v_4, v_6, v_7, v_5, v_8\}$.

Table 5.2: Time-weights and *b-levels* of the tasks during initial step

Task	w_i^0	$b^0(v_i)$
1	77	512
2	58.5	344.5
3	87	356
4	57.5	310.5
5	59	170
6	76	199
7	23	183
8	65	65

The processor selection procedure is as follows:

$EST(v_1, p_1) = 0$

$EST(v_1, p_2) = 0$

$EFT(v_1, p_1) = 70$

$EFT(v_1, p_2) = 84$

$EFT(v_1, p_1) < EFT(v_1, p_2)$, so task v_1 is assigned to processor p_1.

$EST(v_3, p_1) = \max\{70, \{EFT(v_1, p_1) + c_{1,3,1,1}\} = \max\{70, (70+0)\} = 70$

$EST(v_3, p_2) = \max\{0, \{EFT(v_1, p_1) + c_{1,3,1,2}\} = \max\{0, (70+79)\} = 149$

$EFT(v_3, p_1) = w_{3,1} + EST(v_3, p_1) = 78 + 70 = 148$

$EFT(v_3, p_2) = w_{3,2} + EST(v_3, p_2) = 96 + 149 = 245$

$EFT(v_3, p_1) < EFT(v_3, p_2)$, so v_3 is assigned to processor p_1.

……

There is a special case when assigning task v_5. There is a time slot on processor p_1 between task v_3 and task v_7 ,which has already been assigned on processor p_1; and the time slot is larger than the computation time of task v_5 on processor p_1. Hence, the earliest time that processor p_1 is available for task v_5 execution is the time just after task v_3 finishes execution – 148, not the time just after task v_7 finishes execution – 325

$$EST(v_5, p_1) = \max\{148, \max(EFT(v_1, p_1) + c_{1,5,1,1}\} = \max\{148, 70\} = 148$$

$$EST(v_5, p_2) = \max\{316, \max(EFT(v_1, p_1) + c_{1,5,1,2}\} = \max\{316, (70 + 66)\} = 316$$

$$EFT(v_5, p_1) = w_{5,1} + EST(v_5, p_1) = 30 + 148 = 178$$

$$EFT(v_5, p_2) = w_{5,2} + EST(v_5, p_2) = 88 + 316 = 404$$

$EFT(v_5, p_1) < EFT(v_5, p_2)$, so task v_5 is assigned to processor p_1.

$$\begin{aligned} EST(v_8, p_1) &= \max\{325, \max\{(EFT(v_5, p_1) + c_{5,8,1,1}), (EFT(v_6, p_2) + c_{6,8,2,1}), \\ &\qquad (EFT(v_7, p_1) + c_{7,8,1,1}\}\} \\ &= \max\{325, \max\{(178 + 0), (316 + 58), (325 + 0)\}\} \\ &= 374 \end{aligned}$$

$$\begin{aligned} EST(v_8, p_2) &= \max\{316, \max\{(EFT(v_5, p_1) + c_{5,8,1,2}), (EFT(v_6, p_2) + c_{6,8,2,2}), \\ &\qquad (EFT(v_7, p_1) + c_{7,8,1,2}\}\} \\ &= \max\{316, \max\{(178 + 46), 316, (325 + 95)\}\} \\ &= 420 \end{aligned}$$

$$EFT(v_8, p_1) = w_{8,1} + EST(v_8, p_1) = 94 + 374 = 468$$

$$EFT(v_8, p_2) = w_{8,2} + EST(v_8, p_2) = 36 + 420 = 456$$

$EFT(v_8, p_1) > EFT(v_8, p_2)$, so task v_8 is assigned to processor p_2.

Hence, with the above insertion policy we obtain the task schedule which is illustrated by Figure 5.2.

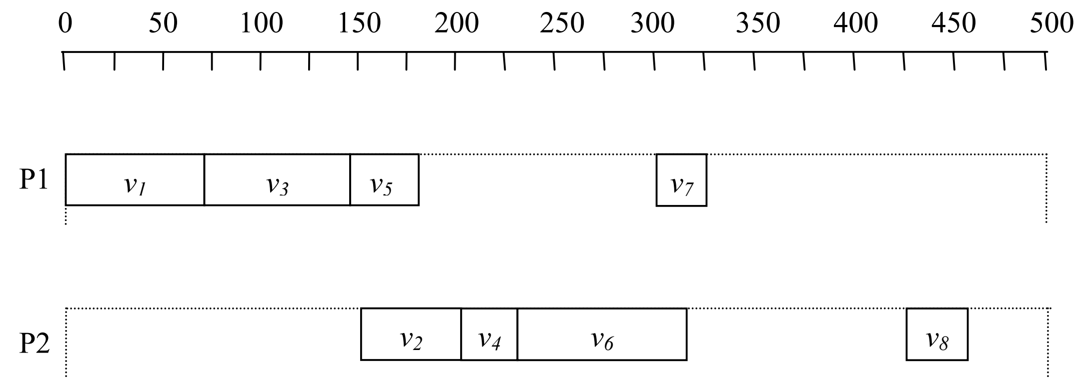


Figure 5.2: Scheduling of task graph during initial step

Table 5.3 shows the start time and finish time of all the tasks. It can be seen from Table 5.3 that current schedule length is 456.

Table 5.3: Start time and finish time of every task during initial step

Task	P1	P2
1	0-70	
3	70-148	
2		155-204
4		204-230
6		230-316
7	300-325	
5	148-178	
8		420-456

For the first iteration, we select 4 as the weighting factor. Then the time-weights of the tasks are computed as follows:

$$w_1^1 = {(4*70+84)} \big/ {(4+2-1)} = 72.8$$

……

Because task v_1 was assigned on processor p_1 during the initial scheduling, the processor p_1 is put more weight when the time-weight of the task v_1 is computed during the first iteration. We believe that the weighted mean of the computation time of the task on every processor can represent the time-weight of the task better than the mean of the computation time of the task on every processor. Table 5.4 shows the updated time-weights and *b-levels* of the tasks. The new scheduling list of the tasks is $\{v_1, v_4, v_3, v_2, v_7, v_5, v_6, v_8\}$. Figure 5.3 illustrates the new schedule obtained in the first iteration.

Table 5.4: Time-weights of the tasks and *b-levels* during first iteration

Task	w_i^1	$b^1(v_i)$
1	72.8	448.2
2	52.8	182.4
3	81.6	266.2
4	38.6	275.4
5	41.6	135.2
6	82	129.6
7	24.2	166.8
8	47.6	47.6

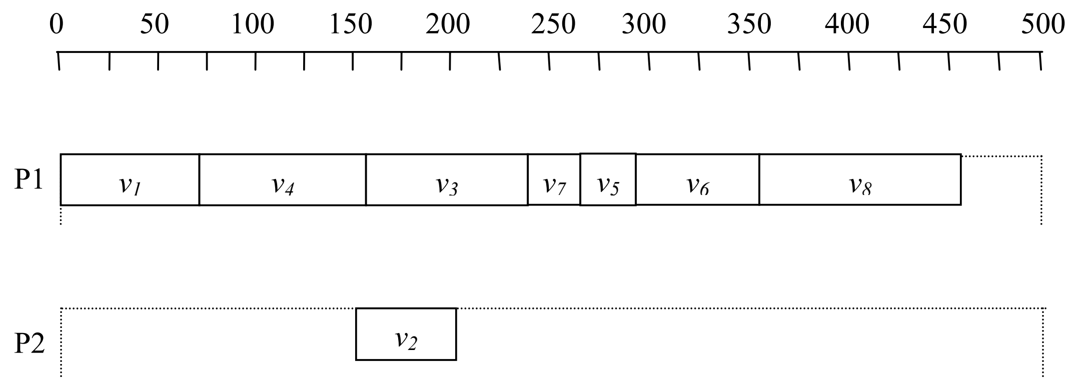


Figure 5.3: Scheduling of task graph during first iteration

Table 5.5 shows the start time and finish time of every task. It can be seen from Table 5.5 that the new schedule length is 452, which is less than 456 from the initial schedule. One possible reason for this is that we have used the weighted mean of the computation time of the task on every processor to represent the time-weight of the task, and this has placed more weight on the processor to which the corresponding task was assigned during the immediately previous iteration.

Table 5.5: Start time and finish time of every task during first iteration

Task	P1	P2
1	0-70	
4	70-159	
3	159-237	
2		155-204
7	237-262	
5	262-292	
6	292-358	
8	358-452	

For the second iteration, the time-weights of the tasks and the *b-levels* are as shown in Table 5.6. The new scheduling list of the tasks is $\{v_1, v_2, v_3, v_4, v_6, v_5, v_7, v_8\}$. Figure 5.4 illustrates the schedule obtained in the second iteration.

Table 5.6: Time-weights of the tasks and *b-levels* during second iteration

Task	w_i^2	$b^2(v_i)$
1	72.8	450
2	52.8	292.2
3	81.6	234
4	76.4	183
5	41.6	124
6	70	152.4
7	24.2	106.6
8	82.4	82.4

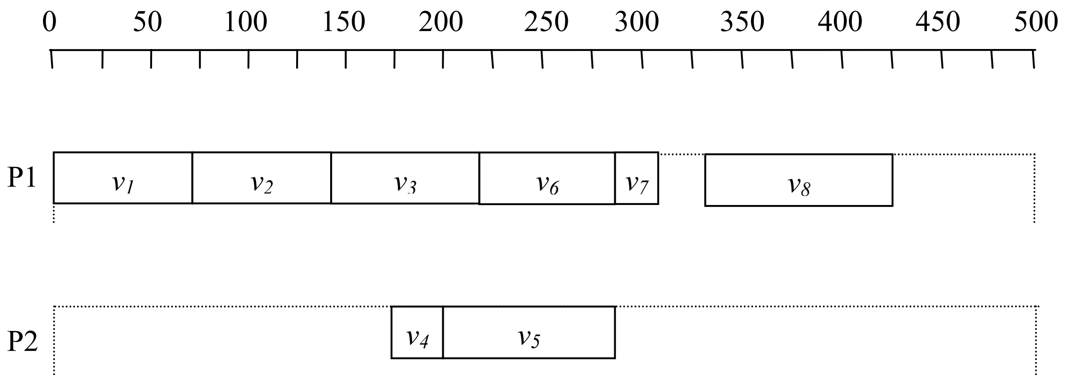


Figure 5.4: Scheduling of task graph during second iteration

Table 5.7 shows the start time and finish time of every task. We note from Table 5.7 that the current schedule length is 424, which is less than the 452 obtained during first iteration.

Table 5.7: Start time and finish time of every task during second iteration

Task	P1	P2
1	0-70	
2	70-138	
3	138-216	
4		170-196
6	216-282	
5		196-284
7	282-307	
8	330-424	

By exhaustive search algorithm, we obtain the optimal schedule of 364 for this case. To compare the result of the iterative algorithm with the optimal solution, we use the *degradation from the best* (Kwok & Ahmad 1999a), which is defined as: *(the result - the best)/ the best*.

The *degradation from the best* for this case is (424-364)/364 or 16.67%. This compares with the *degradation from the best* of (456-364)/364 or 25.27% obtained from the HEFT. The iterative algorithm has therefore improved the scheduling after two iterations in this case.

5.4 Performance analysis based on randomly generated application graphs

In order to analyze the performance of our algorithm, we randomly generate some application graphs. Our objective is to study the amount of improvement to the initial schedule length that can be achieved by our iterative algorithm.

5.4.1 Generation of random application graphs

With selected input, the random graph generator outputs the weighted directed acyclic graph, the computation times of every task at every processor, the communication rate of every link, and the data transfer size between tasks. The input of the random graph generator is as follows:

- Number of tasks (v)
- Height of the DAG (h): The v tasks are randomly partitioned into h levels.

- The link density (β): The probability ($P_{l(i,j)}$) that there is a directed link from the tasks of level i to the tasks of level j is:

$$P_{l(i,j)} = \frac{\beta}{(j-i)} \tag{5.13}$$

  where $j > i, \quad i, j \in (1, h)$.

- Number of processors (p)

- The maximum computation time ($C_{\max}$) and the minimum computation time ($C_{\min}$): The computation time of every task on every processor is a uniform random variable on the interval $(C_{\min}, C_{\max})$.

- The maximum communication rate ($R_{\max}$) and the minimum communication rate ($R_{\min}$): The communication rate $r_{i,j}$ between processor p_i and processor p_j is a uniform random variable on the interval $(R_{\min}, R_{\max})$.

- Communication to computation time ratio (CCR): It is the ratio of the average communication time to the average computation time. The average communication times between two tasks on every link is a uniform random variable on the interval $(CCR * C_{\min}, CCR * C_{\max})$. Then the data transfer size between tasks can be obtained.

5.4.2 Comparison with optimal solutions

When the solution space is not very large, we can obtain the optimal solutions by the exhaustive search algorithm. With the parameters in Table 5.8, the link density is varied from 0 to 1 with increments of 0.1; the CCR varied through the values 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5 and 10. The algorithm was run 1,000 times and the "*degradation from the best"* value computed for each case. The results show that the average *degradation from the best* is 7.44%, which shows that the results of the proposed algorithm are near optimal solutions.

Table 5.8: The Parameters the base example

Number of Tasks	8
Number of Processors	2
DAG Height	4
Minimum Computation Time	20
Maximum Computation time	100
Minimum Communication Rate	1
Maximum Communication Rate	4
Weighting factor	4
Number of Iteration	5

5.4.3 Simulation results

An investigation was carried out to determine how the various parameters of the algorithm impact the degree to which the initial schedules are improved through the iterative steps. The *schedule length improvement ratio* is defined as:

$$r_i = \frac{l_i - l_f}{l_i} \tag{5.14}$$

where l_i is the initial schedule length and l_f is the finial schedule length.

With the parameters shown in Table 5.9, the weighting factor is varied from 0 to 10 with increments of 1, and then varied as 20, 100, 1000; the link density is varied from 0 to 1 with increments of 0.1; and the CCR varied through the values 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5 and 10. The simulation was run 1,000 times for each case, resulting in a total of 10,780,000 runs. The results showed that in 71.30% of the cases the schedule was improved and that the average improvement ratio was 5.4%, Given its initial step is the same as that of the HEFT (Topcuoglu *et al.* 2002), the proposed algorithm can produce shorter schedule length than previous algorithms in most cases.

Table 5.9: The parameters for DAG and scheduling

Number of Tasks	40
Number of Processors	3
DAG Height	10
Minimum Computation Time	20
Maximum Computation time	100
Minimum Communication Rate	1
Maximum Communication Rate	4
Number of Iteration	5

5.4.4 Sensitivity analysis of link density, weighting factor and CCR

To investigate how the link density impacts the results, we compute the percentage of improved cases and the average improvement ratios with various link density levels. The results are shown in Figures 5.5 and 5.6, respectively.

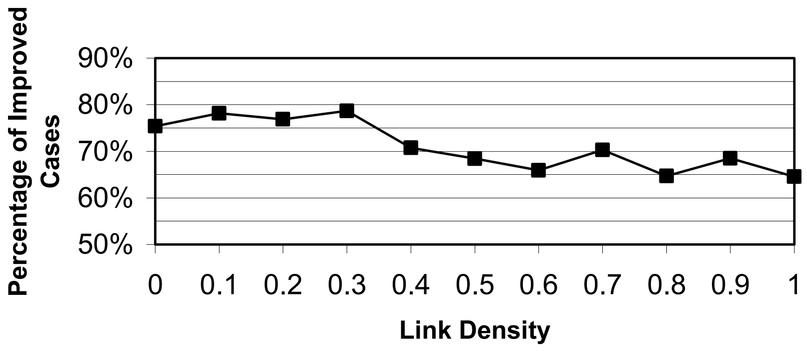


Figure 5.5: Percentage of improved cases varies with the link density

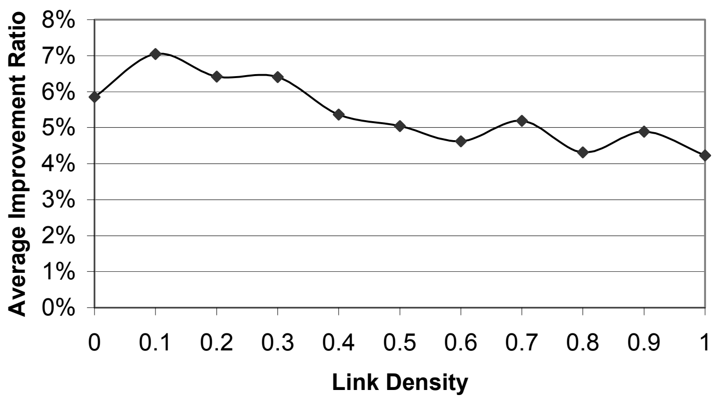


Figure 5.6: Average improvement ratio varies with the link density

Figures 5.5 and 5.6 show that when the link density is varied from 0 to 0.1, the percentage of improved cases and the average improvement ratio first increase and then gradually decrease with increase of link density. When the link density is 0, all tasks are independent, which means that computing the time-weights of the edges in the iterative steps have hardly any impact on the schedule. Hence the percentage of improved cases and the average improvement ratio, when the link density is 0, are both lower than those when the link density is 0.1.

With the link density increasing, however, the task dependencies have more and more impact on the *b-levels*. On the other hand, computing the time-weights of the edges in the iterative steps has less impact on the *b-levels.* Therefore, the percentage of improved cases and the average improvement ratio will gradually decrease when the link density is increased.

To investigate how the weighting factor impacts the result, we compute the percentage of improved cases and the average improvement ratio for a series of weighting factors. The results are shown in Figures 5.7 and 5.8, respectively.

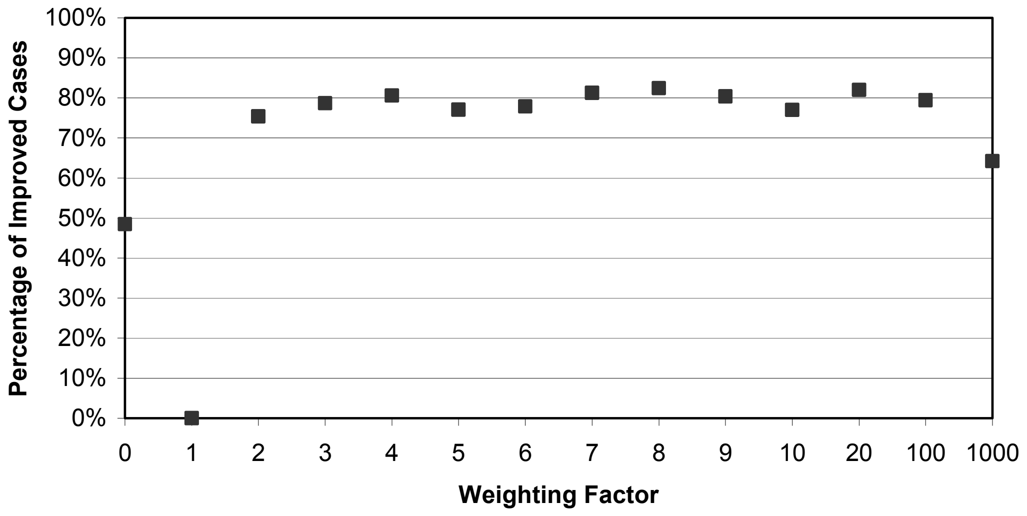


Figure 5.7: Percentage of improved cases varies with the weighting factor

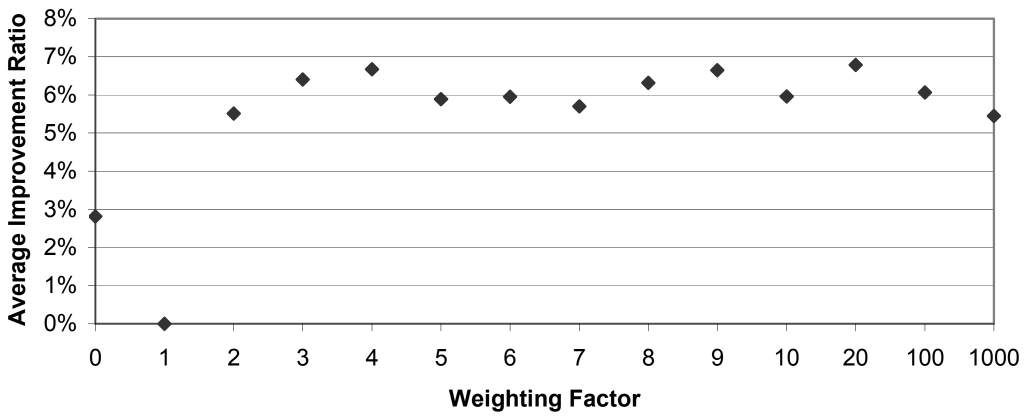


Figure 5.8: Average improvement ratio varies with the weighting factor

Figures 5.7 and 5.8 show that when the weighting factor is 0, both the percentage of improved cases and the average improvement ratio are lowest apart from when the weighting factor is 1. When the weighting factor equals 0, the equations for computing the time-weight of task v_i and the time-weight of the edge from task v_i to task v_j during s-th iteration reduce to the following two equations:

$$w_i^s = \sum_{m=1, m\neq k}^{m=p} w_{i,m} \Big/ (p-1) \tag{5.15}$$

$$c_{i,j}^s = \frac{d_{i,j}}{(\sum_{m}^{p-1} \sum_{n=m+1}^{p} r_{m,n} - r_{k,l}) \Big/ ((p^2 - p)/2 - 1)} \tag{5.16}$$

A weighting factor of 0 value means that, during *s*-th iteration, the time-weights of the tasks are computed by ignoring the processor to which the corresponding task was assigned in the preceding iteration. The time-weights of the edges are computed by ignoring the link between the two processors to which the corresponding tasks are assigned in the preceding iteration.

When the weighting factor is equal to 1, the equations for computing the time-weights of the tasks and the time-weights of the edges during iterations are the same as those during initial step. Therefore, the final schedule is the same as the initial one.

When the weighting factor is increased from 2 to 20, the percentage of improved cases and the average improvement ratio have trivial difference. This means that the final schedule is not sensitive to the weighting factor.

When the weighting factor is equal to 100 or higher, the percentage of improved cases and the average improvement ratio have a decreasing trend. When the number of processors is far less than the weighting factor, the equation for computing the time-weight of task v_i and the time-weight of the edge between task v_i and task v_j during iterations reduce to the following two equations:

$$w_i^s = w_{i,k} \tag{5.17}$$

$$c_{i,j}^s = d_{i,j} / r_{k,l} \tag{5.18}$$

This means that the time-weight of task v_i is the computation time of task v_i on the processor to which the task was assigned during the preceding iteration; the time-weight of the edge from task v_i to task v_j is the communication time of corresponding tasks on the processors to which the corresponding tasks were assigned in the preceding iteration.

The following discusses how the CCR impacts the results. We compute the percentage of improved cases and the average improvement ratio with different CCR levels. The results are as shown in Figures 5.9 and 5.10, respectively.

Figure 5.10 shows that the average improvement ratio gradually increases when CCR is increased, but Figure 5.9 shows that the percentage of improved cases gradually decreases when CCR is increased. Therefore, when CCR is large we take a higher risk that the iterative steps will not improve the final schedule length, but we will obtain higher average improvement ratio if the final schedule length is indeed less than the initial one.

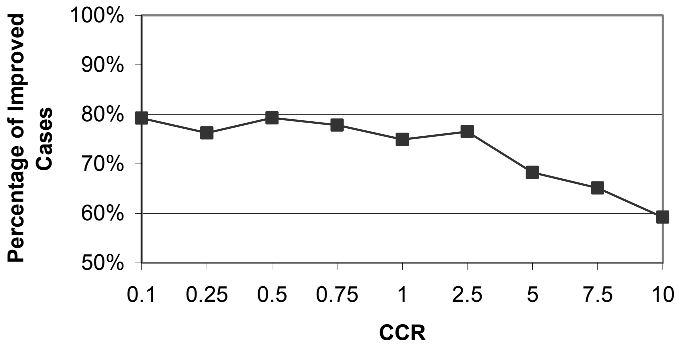


Figure 5.9: Percentage of improved cases varies with the CCR

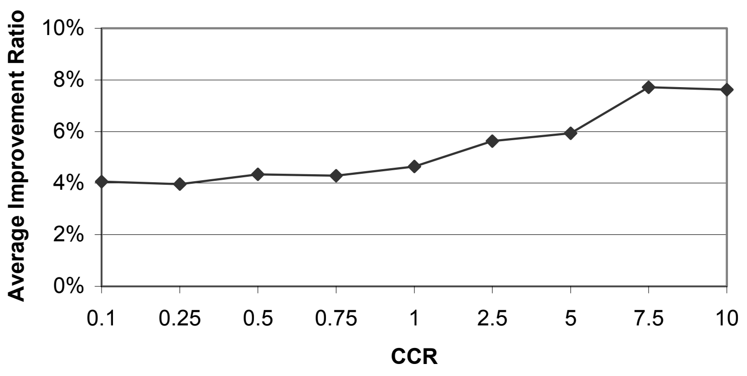


Figure 5.10: Average improvement ratio varies with the CCR

5.4.5 Sensitivity analysis of the task number and the processor number

Based on the parameters in Table 5.10, the weighting factor is varied from 0 to 10 with increments of 1 and then varied as 20, 100, 1000; the link density is varied from 0 to 1 with increments of 0.1; the CCR is varied as 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5 and 10; the task number is varied as 3, 6, 10, 20, 40, 60, 80, 100. The simulation is run 100 times under each case. We compute the percentage of improved cases and the average improvement ratio with task number 3, 6, 10, 20, 40, 60, 80 and 100. The results are shown in Figures 5.11 and 5.12, respectively.

Table 5.10: The Parameters for DAG and scheduling

Number of Processors	3
DAG Height	10
Minimum Computation Time	20
Maximum Computation time	100
Minimum Communication Rate	1
Maximum Communication Rate	4
Number of Iteration	5

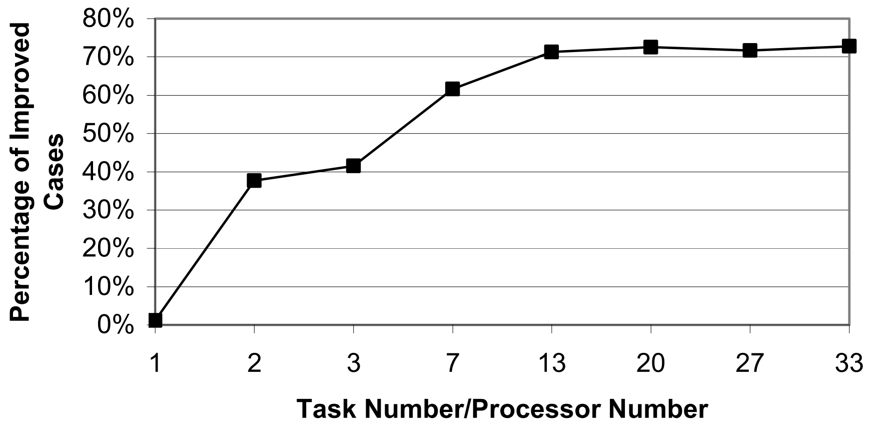


Figure 5.11: Percentage of improved cases varies with task number/processor number

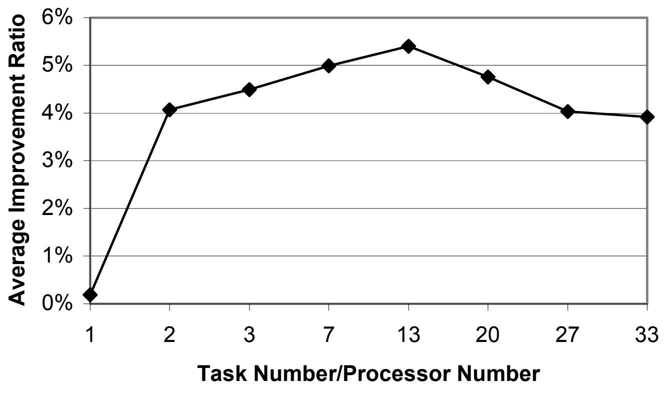


Figure 5.12: Average improvement ratio varies with task number/processor number

Figure 5.11 shows that when the ratio of task number to processor number is small, the percentage of improved cases is very small, i.e., the iterative steps make hardly any improvement in the initial schedule. However, when the ratio is increased, the percentage of improved cases is increased. When the ratio is 13 or greater, the percentage of improved cases reaches the maximum and stops increasing.

Figure 5.12 shows that average improvement ratio has the same trend as the percentage of improved cases when the task number to processor number ratio is less than or equal to 13. The improvement ratio, however, begins to decrease when the ratio of task number to processor number is greater.

We repeated the above simulation with the task number fixed at 100 and the processor number varying as 3, 6, 10, 20, 40, 60, 80, and 100. The results are shown in Figures 5.13 and 5.14.

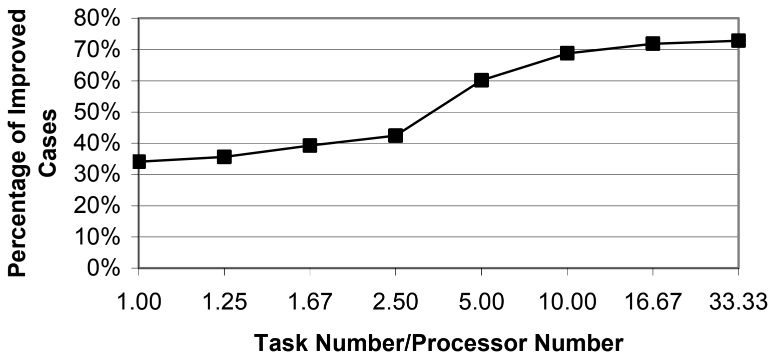


Figure 5.13: Percentage of improved cases varies with task number/processor number

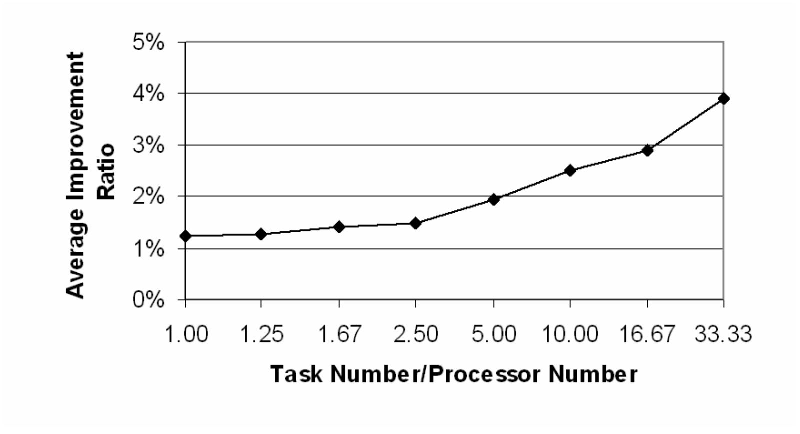


Figure 5.14: Average improvement ratio varies with task number/processor number

Figure 5.13 shows that the percentage of improved cases increases when the ratio of task number to processor number is increased. When this ratio exceeds a certain value, the percentage of improved cases reaches the maximum and levels out. The trend is similar to that in Figure 5.11 and means that the iterative algorithm is more effective when the ratio of task number to processor number is large.

Figure 5.14 shows that, unlike the trend shown in Figure 5.12, the average improvement ratio always increases when the ratio of task number to processor number is increased. In Figure 5.12, the processor number is fixed as 3 and, in Figure 5.14, the task number is fixed as 100, which cause the different trend. However, in both figures, when the ratio of task number to processor number is large, the proposed algorithm perform better than it does when the ratio is very small.

5.5 Performance analysis on application graphs of real world problems

The use of real applications is common for testing the performance of algorithms (Srinivasan & Jha 1999, Topcuoglu *et al.* 2002, Woodside & Monforton 1993, Wu &

Gajski 1990). Therefore, in addition to running the iterative algorithm on randomly generated DAGs, we also ran it on two real world problems: a digital signal processing (DSP) example (Woodside & Monforton 1993) and a Gaussian elimination (Wu & Gajski 1990).

5.5.1 DSP

We selected a DSP example to test the iterative algorithm because the computation time and the communication data can be estimated very accurately. There are 119 tasks in the DSP task graph. The task graph of the DSP and the parameters for the DSP can be found in Woodside & Monforton (1993).

In this case, just the CCR value and the processor number are changed. The CCR is varied as 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5 and 10 and the processor number is varied as 2, 4, 6, 8, 10, 20, 30, 40, 50, 60, 70, and 80. An appropriate weighting factor was selected and the algorithm was run 1,000 times in each case. The results are shown in the Figures 5.15 and 5.16.

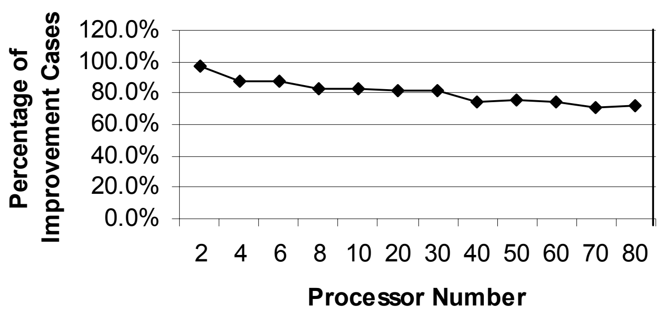


Figure 5.15: Percentage of improved cases varies with processor number

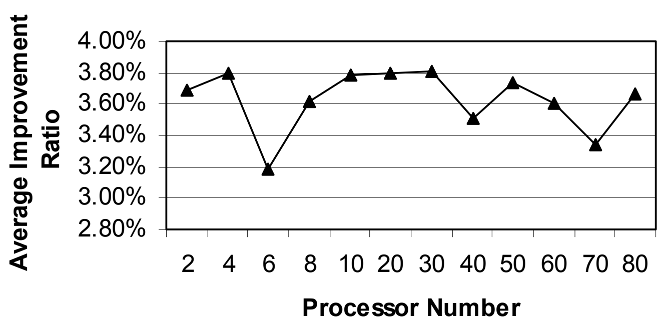


Figure 5.16: Average improvement ratio varies with processor number

Figure 5.15 shows that when the processor number is small, the percentage of improved cases is very close to 100, indicating that in the majority of the cases there is improvement in the initial schedule. However, the percentage of improved cases decreases slightly as the processor number is increased. The trend is the same as that obtained for the randomly generated DAGs.

The results therefore confirm again that this iterative algorithm is more effective when the ratio of task number to processor number is large.

Figure 5.16 shows that, unlike the findings for the randomly generated DAGs, there is no clear relation in this case between the average improvement ratio and the task number. It is possible that the special data structure of DSP cause this difference.

5.5.2 Gaussian elimination

The task graph of the Gaussian elimination, with a matrix size of 5, can be found in Wu & Gajski (1990). The total number of the tasks for this case is equal to $\frac{m^2+3m}{2}-2$, where m is the matrix size. We use a Gaussian elimination with matrix size of 50 and therefore the total number of the tasks is 1323. The CCR varied as 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5 and 10 and the processor number from 2 to 40 with increments of 2. We selected an appropriate weighting factor and ran the algorithm 1000 times in each case. The results are shown in the Figures 5.17 and 5.18.

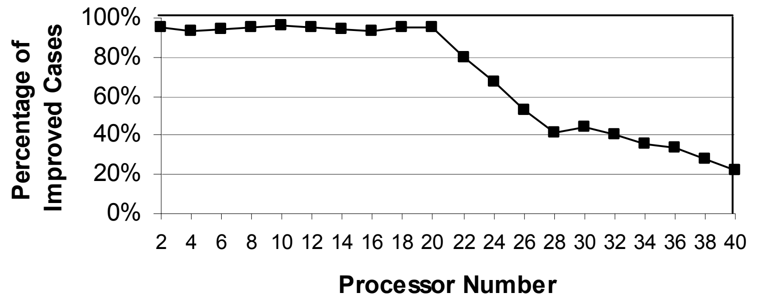


Figure 5.17: Percentage of improved cases varies with processor number

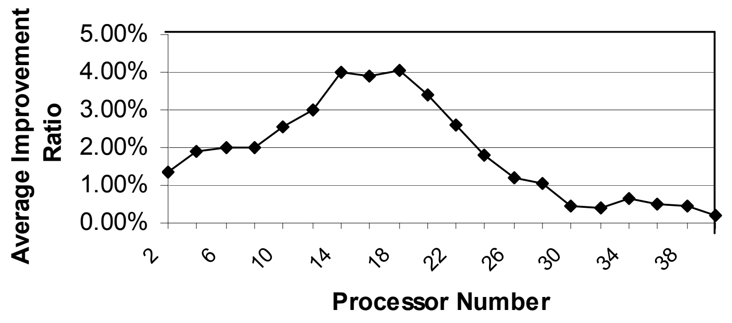


Figure 5.18: Average improvement ratio varies with processor number

Figure 5.17 shows that the percentage of improved cases is virtually constant and very close to 100 over the range of processor number from 2 to 20, but decreases sharply when the processor number exceeds 20. Again, the results support the earlier conclusion that the iterative algorithm is more effective when the ratio of task number to processor number is large.

Figure 5.18 shows that average improvement ratio increases with the percentage of improved cases when the processor number is less than 20. The improvement ratio, however, begins to decrease when the processor number is greater than 20. This shows again that when the ratio of task number to processor number is large, the proposed algorithm perform better than it does when the ratio is very small. However, when the ratio of task number to processor number is very large, the performance of the proposed algorithm may become worse.

5.6 Conclusions

In this chapter, an iterative list scheduling algorithm for the heterogeneous DCSs is proposed and studied. Bottom-level (*b-level)* was selected as priority in constructing the scheduling list. The *b-levels* were computed with the mean of the computation times of a task on every processor and the mean of the communication times of an edge on every link during the initial step, and with the weighted mean during subsequent iterations. The processor selection step uses the insertion-based policy that considers the possible insertion of a task to an idle time slot between two already-scheduled tasks. The initial step of our algorithm is the same as that of the HEFT (Topcuoglu *et al.* 2002). However, the iterative algorithm can produce shorter schedule lengths through subsequent iterations than those obtained by the HEFT (Topcuoglu *et al.* 2002), DLS (Sih & Lee 1993), MH (El-Rewini & Lewis 1990) and LMT (Iverson *et al.* 1995) algorithms.

We determined the percentage of cases that resulted in an improved final schedule and the average improvement ratio with randomly generated task graphs under various parameters and two real applications. It is observed that when the ratio of task number to processor number is small, the iterative algorithm does not perform well but when the ratio is greater than a certain value, an improvement in the final schedule is obtained in most of cases that were simulated. The possible reason is when the task number is much larger than the processor number, there are much more choice for the processor selection. The probability that the initial solution is far from the optimal solution is big, i.e. there are more space to improve the initial solution.

A sensitivity analysis carried out showed that the percentage of cases in which the final schedule length is less than the initial one and the average improvement ratio are both insensitive to the weighting factor used for computing the mean during an iteration, which make it easy to select the weighting factor.

Chapter 6

Reliability and Completion Time Oriented Tabu Search for Distributed Computing Systems

In Chapter 5, a completion time oriented task scheduling problem was studied, in which the tasks could be dependent and the data dependencies were represented by directed acyclic graphs (DAG). The general DAG scheduling problem has been shown to be NP-complete (Garey & Johnson 1979) and this has stimulated researchers to propose a myriad of heuristic algorithms. Most proposed scheduling algorithms are based on minimizing the completion time (schedule length) without considering the possible failure of the processors or relevant network resources. However, in reality, processor and network failures are possible and these can have an adverse effect on applications being executed on the system, especially in a large network of processors. Large, long-running applications are particularly sensitive to failures. In a failure-prone system, assigning tasks to processors without considering possible failures may result in a significant increase in the average completion time of the application when failures occur. This chapter addresses scheduling methods which simultaneously minimize schedule length and maximize system reliability.

Conventional multi-objective optimization methods often combine multiple objectives to form a single composite one by using, for example, the weighted-positive-sum approach. However, in most cases this combination is very difficult or even impossible.

Alternatively, only one objective is optimized while others are treated as constraints (Erschler *et al.* 1976, Fox 1987). In this case however, the algorithm may not find a feasible solution because the problem is over-constrained. Fonseca (1995) employed the concept of Pareto's optimality using an evolutionary algorithm to obtain a set of solutions at multiple points along the tradeoff surface of multiple objectives simultaneously. Evolutionary algorithms have been extensively adopted to solve the multi-objective optimization problems because they deal simultaneously with a set of possible solutions (population). This enables a set of Pareto-optimal solutions to be obtained in a single iteration of the algorithm, while traditional methods have to perform a series of runs to obtain a set of solutions. Nevertheless it is very difficult to recombine two solutions to generate new solutions due to the data dependency.

Hou *et al.* (1994) proposed a crossover schema for DAG scheduling to minimize the schedule length. However, in some cases feasible solutions cannot be generated by using this method. Ahmad & Dhodhi (1995) and Kwok & Ahmad (1997) used genetic algorithms to solve DAG scheduling problem, but in their methods a chromosome is a scheduling list. Hence, post-processing is needed to obtain the final schedule; i.e. the ordered tasks have to be assigned to the appropriate processor to minimize the schedule length. Hence, it is not easy to adopt this method for considering the system reliability. Oh & Wu (2004) proposed a genetic algorithm for task scheduling in multiprocessor systems to minimize the total tardiness of tasks and the number of homogeneous processors, and it is very difficult to utilize the algorithm to address the proposed problem too. In this chapter, we propose a TS algorithm to solve the multi-objective problem.

Tan *et al.* (2003) presented an exploratory multi-objective evolutionary algorithm (EMOEA) and proposed "lateral interference" for population diversity. Experimental results have shown that EMOEA performs well in searching and distributing non-dominated solutions uniformly along the trade-offs. To apply “lateral interference”, it is necessary to compute the distance metric among the solutions. However, this is not a straightforward process when the objectives are non-commensurable. This chapter proposes two schemes to compute the distance among the solutions.

This chapter is organized as follows. Section 6.1 describes the scheduling problem for the heterogeneous DCS. Section 6.2 introduces the multi-objective optimization problem and proposes two definitions of metric length. Section 6.3 presents a Tabu Search algorithm to solve the proposed problem. Section 6.4 describes some simulations to compare the results with the two definitions of metric length. Finally, Section 6.5 presents the conclusions of this chapter.

6.1 Modelling

Notations:

$c_{i,j,k,l}$:	communication time from task v_i to task v_j when task v_i was assigned to processor p_k and task v_j was assigned to processor p_l;
$d_{i,j}$:	data transfer size (in bytes) from task v_i to task v_j;
e:	number of directed links among the tasks
$e_{i,j}$:	directed link from i-th task to j-th task;
$EST(v_i, p_j)$:	earliest start time of task v_i on processor p_j;
$EFT(v_i, p_j)$:	earliest finish time of task v_i on processor p_j;
$l_{i,j}$:	direct link between processor p_i and processor p_j;
p:	number of processors available in the system;
p_i:	i-th processor in the system;

$r_{i,j}$:	communication rate (in bytes/second) between processor p_i and processor p_j ;
R :	distributed computing system reliability;
R_i :	reliability of the processor p_i , which is the probability that processor p_i is functional;
$R_{i,j}$:	reliability of the directly link $l_{i,j}$ between processor p_i and processor p_j ;
SL :	schedule length;
UR :	distributed computing system unreliability.
v:	number of tasks in the application;
v_i :	i-th task in the application;
$w_{i,j}$:	computation time of task v_i on processor p_j ;
λ_i :	failure rate of processor p_i ;
$\lambda_{i,j}$:	failure rate of link $l_{i,j}$ between processor p_i and processor p_j .

An application is represented by a DAG. An example of a small DAG is shown in Figure 6.1. The details of the application modeling and the computation of the completion time have been described in Chapter 5. The following will discuss the reliability analysis of the system.

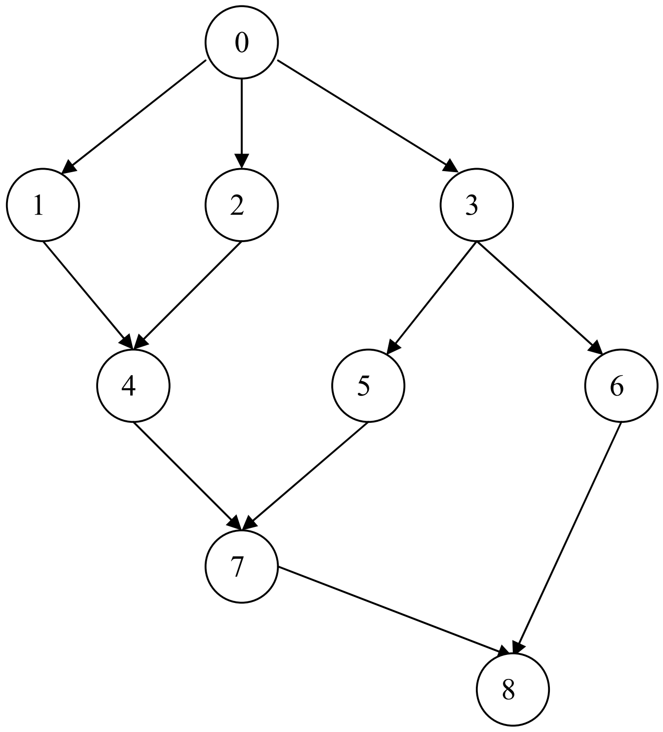


Figure 6.1: A DAG example

The failures of processors and links in the system are assumed to follow a Poisson process and to be statistically independent. Furthermore, once a processor or link has failed, it is assumed that it remains in the failed state for the remainder of the execution of the application. Similar assumptions have been used by Shatz & Wang (1989), Shatz *et al.* (1992), Kartik & Murthy (1997), Iverson (1999). To successfully execute the application, each processor should be functional during the time that its assigned tasks are executing and each relevant link should be functional during the time that corresponding inter-task communication are executing.

Since the failure of a processor follows a Poisson process, at time *t* the reliability of a processor p_i is $e^{-\lambda_i t}$. The reliability requirement of processor p_i is therefore:

$$R_i = e^{-\lambda_i \max(EFT(v_j, p_i))} \tag{6.1}$$

The *max* term in Equation (6.7) gives the finish time of the last task executed on processor p_i.

Similarly, the reliability requirement of link $l_{i,j}$ is:

$$R_{i,j} = e^{-\lambda_{i,j} f_{i,j}} \tag{6.2}$$

where $f_{i,j}$ is the finish time of the last communication between processor p_i and processor p_j, i.e., the time that link $l_{i,j}$ is required to be functional for the inter-task communication during the execution of the application.

Recall that the failures of processors and links in the system are assumed to be statistically independent. Therefore, the reliability of the systems *R* is:

$$R = \prod_{i=1}^{v} R_i \cdot \prod_{i,j=1}^{v} R_{i,j} \tag{6.3}$$

The objective of this problem is to maximize the reliability of the system and minimize the schedule length. These two objectives are non-commensurable and may be competing. The problem usually has no unique, perfect solution, but a set of non-dominated and possible solutions, known as the Pareto-optimal set (Ben-Tal 1980). For convenience, we convert the objective of maximizing distributed system reliability (DSR) to minimizing system unreliability (*UR*), where $UR = 1 - DSR$. Hence, the objectives of the proposed problem are:

$$\textbf{Minimize: } SL \text{ and } UR \tag{6.4}$$

6.2 Multi-objective optimization

A general multi-objective minimization optimization problem simultaneously minimizes n objective functions $f_k, k = 1,2,\cdots,n$. It is possible that the objective functions are a nonlinear vector function F of a general decision variable s in a whole solution space S , where $F(s) = (f_1(s), f_2(s),\cdots, f_n(s))$. To facilitate the description, we first give the follows definitions (Tan *et al.* 2002).

Definition 6.1 (Pareto dominance): A given vector $u = (u_1, u_2,\cdots,u_n)$ is said to dominate another vector $v = (v_1, v_2,\cdots, v_n)$ *iff*

$$\forall i \in \{1,2,\cdots,n\},\ u_i \le v_i \quad \wedge \quad \exists i \in \{1,2,\cdots,n\},\ u_i < v_i$$

Definition 6.2 (Pareto-optimal): Given a set of solution $S = \{s_1, s_2,\cdots s_m\}$, a solution $s_i \in S$ is said to be Pareto-optimal *iff* no solution $s_j \in S$ dominates solution s_i .

Pareto-optimal solutions are also called non-dominated or efficient solutions. The corresponding objective vectors are referred to as non-dominated. The set of all non-dominated vectors is called the non-dominated set, or tradeoff surface, of the problem.

Definition 3 (Pareto front): Given a multi-objective optimization function $F(s)$ and a set of Pareto-optimal solution Ω , the Pareto-front is:

$$\{\bar{u} = F(s) = (f_1(s),\ldots, f_n(s)) \mid s \in \Omega)$$

Tan *et al.* (2003) proposed a Tabu-based exploratory multi-objective evolutionary algorithm (EMOEA), which uses the Tabu list to prevent the search from becoming trapped in local optima and to promote concurrently the evolution towards the global trade-offs. Tan *et al.* (2003) also presented a new lateral interference to distribute non-dominated solutions along the discovered Pareto-front uniformly. The lateral interference can be applied in either the parameter or objective domain. The basic concepts of the lateral interference based population distribution method are as follows.

First the population is ordered according to the Pareto ranking scheme proposed by Fonseca & Fleming (1995). The ranking scheme assigns all non-dominated individuals as rank 1. The rank of a dominated solution is equal to the number of solutions dominating it plus one. An example for a minimization problem of two objectives f_1 and f_2 is shown in Figure 6.2, in which the numbers are the ranks of the individuals.

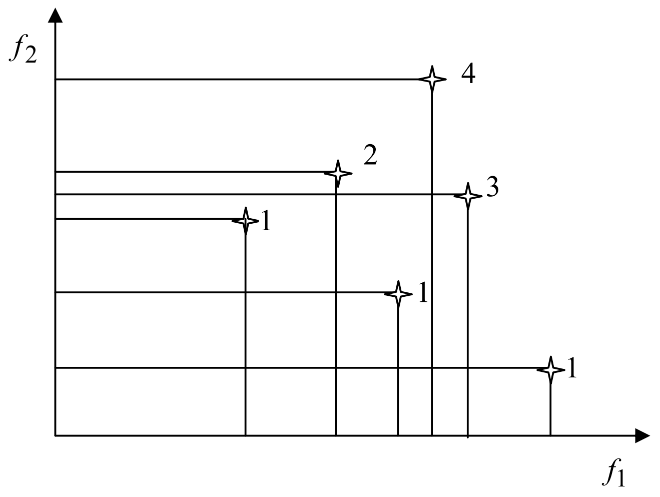


Figure 6.2: Pareto ranking scheme for multi-objective optimization

The lateral interference takes place between the individuals which have the same rank. Given a sub-population P' consisting of N' m-dimensional individuals of the same rank, the metric distance between any two individuals s_i and s_j is defined by

$$d(s_i, s_j) = \| s_i - s_j \|_2 \tag{6.5}$$

where $\| . \|_2$ implies the 2-norm.

If T_i denotes the territory of the individual s_i, and $s_i^{nearest}$ the nearest individual to individual s_i, then $s_j \in T_i$ *iff* $d(s_j, s_i^{nearest}) \le d(s_i, s_i^{nearest})$.

If an individual s_j belongs to the territory of individual s_i, individual s_j can be subjected to interference from individual s_i. The more individuals subjected to interference, the less chance of survival. Tan *et al.* (2003) use the *interference severity* to denote the number of the territories to which an individual belongs.

In this case, where the preferences of the two objectives are not known, the ranking scheme based on the Pareto optimality is an appropriate approach to compute the fitness of each individual in an evolutionary algorithm (Srinivas and Deb 1994; Fonseca 1995). The solutions with the same rank according to the Pareto–ranking schema can be differentiated by the *interference severity.* However, one challenge is how to define the metric distance between two solutions, as this is a 2-dimensional vector, i.e., system unreliability and schedule length which are non-commensurable.

Given a set of solutions $X^{'}$ consisting of N_x 2-dimensional solutions, which have the same rank, we propose two metric lengths $d(s_i, s_j)$ and $d^{'}(s_i, s_j)$ between two solutions and compare the solutions by using these two.

$$d(s_i, s_j) = \sqrt{\left(\frac{UR(s_i)}{\overline{UR}} - \frac{UR(s_j)}{\overline{UR}}\right)^2 + \left(\frac{SL(s_i)}{\overline{SL}} - \frac{SL(s_j)}{\overline{SL}}\right)^2} \tag{6.6}$$

where $UR(s_i)$ is the value of system unreliability of solution s_i, and $SL(s_i)$ is the value of schedule length of solution s_i; $\overline{UR}$ is the mean of system unreliability of all solutions in $X^{'}$, and $\overline{SL}$ is the mean of schedule length of all solutions in $X^{'}$.

$$d^{'}(s_i, s_j) = \sqrt{(\frac{UR(s_i)}{\sigma(UR)} - \frac{UR(s_j)}{\sigma(UR)})^2 + (\frac{SL(s_i)}{\sigma(SL)} - \frac{SL(s_j)}{\sigma(SL)})^2} \tag{6.7}$$

where $\sigma(UR)$ is the standard deviation of system unreliability of all solutions in $X^{'}$, and $\sigma(SL)$ is the standard deviation of schedule length of all solutions in $X^{'}$.

6.3 A Tabu Search for the multi-objective scheduling

Tabu Search (TS) is a competing meta-heuristic method for many of the complex combinatorial optimization problems (Glover & Laguna 1997). Unlike evolutionary algorithms, TS is not population-based but successively moves from solution to solution and terminates with either an optimal or a near-optimal solution. However the global optimum for multi-objective optimization is a set of Pareto-optimal solutions, instead of a single optimum. To overcome this problem, the proposed algorithm involves two different lists: a Pareto optimal solution list and Tabu list. The Pareto optimal solution list stores the current Pareto optimal solutions while the Tabu list records the recently visited solutions and is used to avoid revisiting a state in the short term. For limiting the computational effort required, limits are placed on the length of these two lists

The procedure of the algorithm is follows:

```
1. Randomly generate a feasible solution $x$, and put the solution into the Pareto
   optimal solution list $PL$ and the Tabu List $TL$. // $x$: Current solution.
2. while $s < s - \max$ do
3.       Generate a new solution $x$ by one step "move" and $x \notin TL$
4.       Update $TL$.
```

```
5.          if $x$ dominates some solutions in $PL$ then
6.              $x$ replaces the dominated solutions in $PL$.
7.          else if $x$ does not dominate any solution and is not dominated by any
                    solution in $PL$ then
8.                  Add $x$ into $PL$
9.          end if
10.         if $|PL| > L_p$ then   //$|PL|$: Length of $PL$; $L_p$: Predefined value of the
    length of $PL$.
11.             All the solutions are ranked according to the interference severity
                increasingly
12.             The last one is deleted from $PL$
13.         end if
14.         Implement intensification strategy
15.         Implement diversification strategy
16. end while
17. Return the Pareto optimal solution list
```

Encoding:

In this case the solution representation should satisfy two conditions:

1. Every task is present and appears only once;

2. The data dependency should be satisfied.

On this basis, the solution is coded as follows:

$$\begin{gathered} P_1 : v_i, \cdots, v_j \\ \vdots \\ P_p : v_k, \cdots, v_l \end{gathered}$$

Each solution consists of p substrings, each of which corresponds to a processor.

Each list corresponds to the tasks executed on a processor, and the order of the tasks in the list indicates the order of execution. The dependency between the tasks on different processors is considered when the schedule length is computed.

Move:

During the "move" operator, the dependencies among the task must be satisfied. For example, if there is a directed link from task v_i to task v_j and a directed link from task v_j to task v_k, then task v_k can not be executed before task v_i.

To express the direct dependency relation between the tasks, we use the adjacency matrix $A = (a_{ij})$, where

$$a_{ij} = \begin{cases} 1, & \textit{if there exist a direct link from } \text{task } v_i \textit{ to } \text{task } v_j \\ 0, & \textit{otherwise} \end{cases} \tag{6.8}$$

To obtain the indirect dependency between the tasks, we first compute the attainability matrix, $t = (t_{ij})$, where

$$t_{ij} = \begin{cases} 1, & \textit{if there exist a path from } \text{task } v_i \textit{ to } \text{task } v_j \\ 0, & \textit{otherwise} \end{cases} \tag{6.9}$$

The attainability matrix can be derived from adjacency matrix using Warshall's algorithm. Task v_j cannot start before task v_i completes its execution if $t_{ij} = 1$.

There are two styles of moves:

- Exchange the order of two selected tasks:
    - Randomly select a processor where there are at least two tasks.
    - Randomly select a task v_j from the selected processor, and given there is a task list on the selected processor $(..., v_i, v_j, v_k, ...)$.
        - If $t_{jk} = 0$, then exchange the order of task v_j and task v_k.
        - If $t_{jk} > 0$ and $t_{ij} = 0$, then exchange the order of task v_i and task v_j.
        - If $t_{ij} > 0$ and $t_{jk} > 0$, then give up.
- Move a selected task to another processor:
    - Randomly select a task on a randomly selected processor.
    - Move the selected task to end of task list of another randomly selected processor.
    - Move the task forward until it cannot be moved further.

After the move, a deadlock may happen as shown in the DAG given in the Figure 6.1 where the following scheduling is obtained by the move.

P0: T2, T4, T3, T6

P1: T0, T5, T1, T7, T8

In this case, task T4 cannot be executed before task T1 finishes, and task T5 cannot be executed before task T3 finishes. When deadlock happens, if the task T_i is not ready to be executed, we check the successor task T_j of T_i. If task T_j is ready to be executed, we exchange the order of task T_i and task T_j.

Intensification strategy:

If after the predefined number of “moves”, we still cannot find a Pareto optimal solution, then the last found Pareto optimal solution is used as the current solution. This strategy can initiate a return to the attractive regions for a more thorough search. However, to avoid a repeated search for the same solution, the length of Tabu list should be larger than the predefined number of “moves” for intensification strategy.

Diversification strategy:

After the predefined number of “move”, a solution in Pareto optimal solution list is randomly selected as the current solution. This selected solution cannot be the last found Pareto optimal solution as this strategy would restrict the search to the neighbors of one Pareto optimal solution.

6.4 Simulation study

In this section two metrics are used to validate and compare the performance of the proposed TS with the two distance computation schemes. For this problem, the actual trade-off surface cannot be obtained through the method of deterministic enumeration as it can in some other cases. Hence we propose the “comparison of non-dominated solution number” method to measure the performance of the proposed algorithm. This method comprises the following:

> Given two sets of non-dominated solutions A and B, set $C = A \cup B$ and delete the dominated solutions in set C. Then compare $|A \cap C|$ and $|B \cap C|$.

In addition, we also adopt the metric – "uniform distribution (UD) of non-dominated solutions" (Tan *et al.* 2002) to measure the performance of the proposed algorithm as follows:

Given a set of non-dominated solutions $X^{'}$,

$$UD(X^{'}) = \frac{1}{1 + S_{is}} \tag{6.10}$$

where S_{is} is the standard deviation of the *interference severity* of all the solutions in set $X^{'}$, and is formulated as,

$$S_{is} = \sqrt{\frac{\sum_{i=1}^{N_x} (is(x_i) - \overline{is}(X^{'}))^2}{N_x - 1}} \tag{6.11}$$

where N_x is the size of the set $X^{'}$; $is(x_i)$ is the *interference severity* of the solution x_i; $\overline{is}(X^{'})$ is the mean value of *interference severities* of all solutions in set $X^{'}$.

6.4.1 Performance analysis on randomly generated DAGs

We randomly generate a set of DAGs by using the random graph generator, which has been described in Chapter 5. The random graph generator outputs the weighted directed acyclic graph, the computation times of every task at every processor, the communication rate of every link, and the data transfer size between the tasks.

In this case, there is a set of computers which are interconnected by a switch-based network. The computers and the network can both be heterogeneous. The failure rate of the processors and the links are assumed to be uniformly distributed between 1×10^{-3} and 1×10^{-4} (Dogan & Ozguner 2002, Plank & Elwasif 1997).

Using the parameters shown in Table 6.1, the link density is varied from 0.1 to 1 with increments of 0.1 and the CCR varied through the values 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5 and 10.

The parameters of the TS are listed in Table 6.2. The length of Tabu list cannot be less than the predefined number for intensification strategy, as this would cause repeat searching for the same individuals in accordance with the intensification strategy which repeatedly returns the search to the last Pareto optimal solution if another Pareto optimal solution cannot be found within the predefined number of “moves”.

On the other hand the length of the Tabu list cannot be too large, as this may cause rejection of too many “moves”.

The simulation was run 100 times for each case. In each case the algorithm was run in three schemes:

- without considering the lateral interference,
- considering lateral interference according Equation (6.6),
- considering lateral interference according Equation (6.7).

Equation (6.6) computes the distance between two solutions by using the mean of all solutions in the current solution set, whereas Equation (6.7) computes the distance between two solutions by using the standard deviation of all solutions in the current solution set. For ease of notation, we refer to these three schemes as SW, SA, and SD, respectively. Based on “comparison of non-dominated solution number”, there is no difference among these three schemes in 96.8% of the cases. However, comparison of the three schemes based on "uniform distribution of non-dominated solutions", presented in Table 6.3, shows that SA and SD are better than SW in most cases, whatever SA or SD is used to compute UD and the results based on UD of SA and that of SD are almost the same. The results show that "lateral interference" can benefit to the uniformly distributing the Pareto-optimal solutions along the trade-offs, i.e. population diversity, whatever the schema is used to compute the distance among solutions. The results also are also not sensitive to the schema used to compute the distance among solutions in this case.

Table 6.1: The parameters for DAG

Number of Tasks	100
Number of Processors	10
DAG Height	10
Minimum Computation Time	20
Maximum Computation Time	100
Minimum Communication Rate	1
Maximum Communication Rate	10

Table 6.2: The parameter of TS for random DAG

Length of Pareto Optimal Solution List	10
Length of Tabu List	50
Predefined Number for Intensification	20
Predefined Number for Diversification	500
s-max	5000

Table 6.3: Comparison of three schemes based on UD for random DAG

Scheme A	Scheme B	Percentage of B is better than A	Scheme used to compute UD
SW	SA	83.70%	SA
	SD	80.60%	
	SA	83.70%	SD
	SD	80.60%	
SD	SA	10.40%	SA
SA	SD	9.80%	SD

6.4.2 Performance analysis on a real-world problem

The performance of algorithms is commonly tested using real applications (Srinivasan & Jha 1999, Topcuoglu *et al.* 2002, Woodside & Monforton 1993, Wu & Gajski 1990). Therefore, in addition to the randomly generated DAGs, we also ran the proposed algorithm on a real-world problem: Gaussian Elimination (Wu & Gajski 1990). The task graph of the Gaussian Elimination, whose matrix size is five, can be found in Wu & Gajski (Wu & Gajski 1990).

The total number of the tasks is equal to $\frac{m^2+3m}{2}-2$, where m is the matrix size. A Gaussian Elimination whose matrix size is 50 is used giving a total number of the tasks of 1,323. In this case, the number of processors is 10. CCR is varied through the values 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5 and 10. The parameters of TS are listed in Table 6.4. The simulation was run 100 times for each case and, as before, in each case the algorithm was run using the three schemes: SW, SA and SD.

It was found that, based on "comparison of non-dominated solution number", there is no difference between these three schemes in 95.9% of the cases. However, based on "uniform distribution of non-dominated solutions", a comparison of the results for the three schemes as detailed in Table 6.5 provide conclusions similar to those obtained with the Random DAG problem shown in Table 6.3. Thus SA and SD are shown to be better than the scheme SW in most cases, whatever SA or SD is used to compute UD and the results of SA and that of SD are almost the same.

Table 6.4: The parameter of TS for Gaussian Elimination

Length of Pareto Optimal Solution List	20
Length of Tabu List	50
Predefined Number for Intensification	20
Predefined Number for Diversification	500
s-max	5000

Table 6.5: Comparison of three schemes based on UD for Gaussian Elimination

Scheme A	Scheme B	Percentage of B is better than A	Scheme used to compute UD
SW	SA	93.90%	SA
	SD	90.60%	
	SA	93.90%	SD
	SD	90.60%	
SD	SA	5.40%	SA
SA	SD	6.30%	SD

6.5 Conclusions

In the scheduling of DCSs, maximizing system reliability and minimizing schedule length should be considered simultaneously rather than separately as done by most researchers. In this chapter, we presented a multi-objective optimization problem by maximizing the DCS reliability (minimizing system unreliability) and minimizing the schedule length simultaneously and proposed a Tabu search algorithm to solve this problem. At the same time we adopted the "lateral interference" to investigate two schemes to distribute the Pareto optimal solutions along the Pareto-front uniformly. Randomly generated DAGs and a real application task graph – Parallel Gaussian Elimination were used to evaluate the performance of the proposed algorithm.

The "non-dominated solution number" and the "uniform distribution of non-dominated solutions" were the two performance measures used to compare the two schemes considering the "lateral interference" and the one without considering the "lateral interference". For "non-dominated solution number" it was found that there is basically no difference among three schemes. For “uniform distribution of non-dominated solutions, the two schemes considering the “lateral interference” are much better than the one without considering it and there is basically no difference between the first two schemes. Hence, “lateral interference” can improve the “uniform distribution of non-dominated solutions” and is not sensitive to the different computation schemes of distances between the solutions.

Chapter 7

Modelling and Analysis of Service Reliability for Distributed Computing Systems

Distributed systems have been increasingly applied in many safety-critical systems (Levitin 2002, Leger *et al.* 1999), such as the banking systems, military systems, power plants and so forth. System reliability is very important to these types of systems because failures may cause much loss in monetary term or lives.

Since distributed systems are developed to provide services with specific objectives such as running a computer program, controlling a production process or completing other tasks, the service reliability of the distributed system is a key criterion of QoS (Quality of Service). A definition of distributed service reliability can be the probability to successfully provide the service in a distributed environment and this is the definition adopted in this chapter.

Most of the distributed systems can be modeled as centralized heterogeneous distributed systems. This type of system consists of a number of subsystems managed by a control center. For example, for the Client/Server (Browser/Server) systems, every Client/Browser in the sub-distributed systems is managed by a control center of Servers. For the IP telephone systems, the control center manages the computers in sub-distributed systems to provide telephone services. The service reliability in a centralized heterogeneous distributed system is determined not only by the system availability of the control center, but also by distributed program reliabilities of the subsystems.

The system availability of the control center is of major concern because an unavailable control center will sometimes cause critical problems to a service (Pham *et al.* 1997, Sols & Nachlas 1995). Srinivasan and Jha (1999) described a method to determine an allocation that introduces safety into a heterogeneous distributed system and at the same time attempts to maximize its availability. On the other hand, the reliability of each program in the system is also important to a service. The system availability and distributed program reliability have been discussed in the Chapter 2.

However, most of the earlier research on system availability or distributed program/system reliability cannot be simply implemented to analyze the service reliability of centralized heterogeneous distributed systems because reliability is affected by many factors including system availability and distributed program/system reliability. This chapter studies the properties of centralized heterogeneous distributed systems and develops a general model for the analysis. Based on the model, algorithms are developed to obtain the service reliability of the system.

This chapter is organized as follows. Section 7.1 presents a model for a centralized heterogeneous distributed system (CHDS), and develops a solution algorithm for the distributed service reliability in CHDS. The implementation of system availability function of the virtual machine (VM) is also studied. Section 7.2 gives an application example to illustrate the procedure and the feasibility of the algorithm. Then in Section 7.3 we analyze the performance and sensitivity of the system availability function, both of which are important issues in the application of this type of model.

7.1 Centralized heterogeneous distributed system (CHDS) and analysis

Most distributed service systems can be modeled by a CHDS. This type of distributed system incorporates heterogeneous subsystems with various operation platforms on different computers in diverse topological networks managed by a control center. The structure of CHDS is depicted by Figure 7.1.

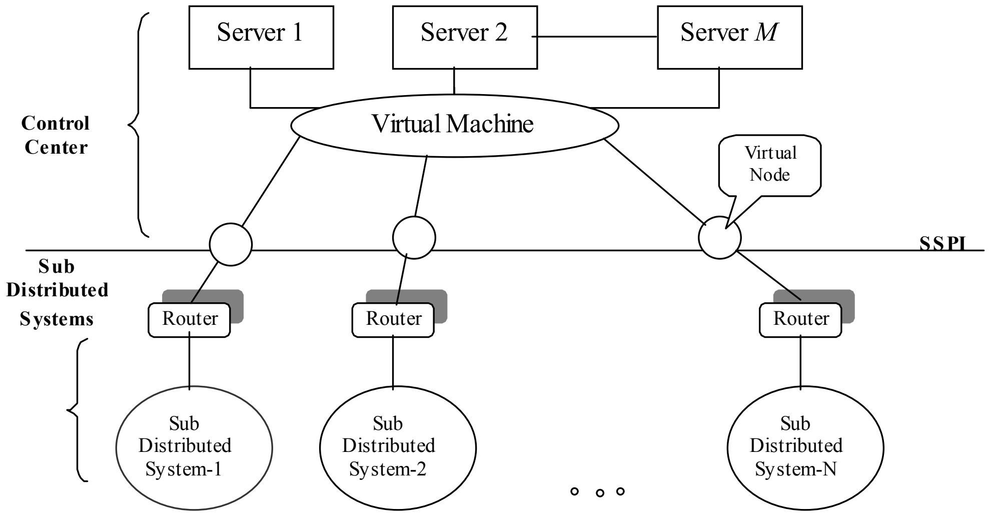


Figure 7.1: Structure of the centralized heterogeneous distributed service system

The control center consists of M servers ($M \geq 1$). These servers support a VM. The VM can manage and control programs and data from heterogeneous subsystems through virtual nodes. The virtual nodes can mask the differences among various platforms. They are a type of virtual executing element that only includes a basic unit for executing data, i.e. CPU and Memory. The entities of VM and virtual nodes are supported by the software and hardware in the control center.

The heterogeneous sub-distributed systems are composed of different types of computers with various operating systems connected by diverse topologies of networks.

These subsystems exchange data with the VM through SSPI (System Service Provider Interface). They are connected with virtual nodes by routers which enable them to cooperate to achieve a distributed service under the management of a VM such as the wide-area computing or grid technology.

Most service systems can be categorized as CHDSs as shown in Figure 7.1. For example, in Client/Server (Browser/Server) systems, the control center can be viewed as Servers and every Client (Browser) can be viewed as a node in the sub-distributed systems. In IP telephone systems, every terminal is a telephone and the computers in sub-distributed systems and the control center provide the services such as connecting two distant telephones and calculating the fee. This structure has also been applied into some other areas such as banks, hospitals, companies and libraries.

The centralized heterogeneous distributed service system in Figure 7.1 can also be reduced to some other systems. First, if the subsystems use identical operation platform and computers, the heterogeneous subsystems can be reduced to homogeneous subsystems. Second, if a system has only one distributed system to complete a service without a control center, we can omit the control center and retain only one of the subsystems. Under this condition, the distributed service reliability becomes the same as the distributed program reliability (Kumar *et al.* 1986, Lin *et al.* 1999a). Finally, if a service system is equipped with only a control center, we can ignore the sub-distributed systems. In this condition, the distributed service reliability becomes the same as system availability (Lai *et al.* 2002).

The whole process for a service in a distributed system may be repeated frequently so the reliability analysis of a distributed service is crucial for a distributed system.

7.1.1 Service reliability analysis of CHDS

Notations:

$A(t)$:	availability function of VM at time *t*,
DSR$_i$:	distributed system reliability for *i*-th sub-distributed system,
$P_0(t)$:	probability that the VM is in working state at time *t*,
$P_1(t)$:	probability that the VM is in malfunctioning state at time *t*,
$R_s(t_b)$:	distributed service reliability function of t_b,
t_b:	initial time for the service,
T_{bf}^j:	time point at which the *j-th* program need the files prepared in the VM
T_{bp}^k:	beginning time when the *k-th* program runs in VM,
T_{ex}^k:	execution time period for those programs in VM,
VM*i*:	VM used in subsystem *i*.

In this chapter, distributed program (system) reliability is defined as the probability of successful execution of a program (all the programs) running on multiple processing elements that need to retrieve data files from other processing elements (Kumar *et al.* 1986). The system availability of the control center or VM is the probability for it to be available.

For a distributed system, the distributed service reliability is defined as the probability to successfully achieve the service in a distributed system. This will depend on both system availability to provide the service and the system reliability in providing the service.

7.1.2 General model of distributed service reliability

In a distributed service system, a service includes different distributed programs completed on different computers. Some later programs might require several precedent programs to be completed. Every program requires a certain execution time. The execution of some programs might require certain input files that are saved or generated on different computers of the distributed systems (Kumar & Agrawal 1993). The overall distributed service reliability depends on the availability of a program for the service, the availability of input files to the program and the service reliability of the subsystem.

The reliability of a service is determined by the reliability of distributed programs in each subsystem and the availability of the control center. If a service can be achieved successfully, the programs running in every subsystem must be successful. The VM should be available at the moment any program needs a certain input file prepared in VM. It also has to be available during the period when the programs are being executed in VM.

The critical path method (Hillier and Lieberman 1995) can be used to determine the time point at which the programs require the files prepared in the VM (T_{bf}^{j}) (*j*=1,2,…*J*). We can also obtain the starting time when the programs runs in the VM (T_{bp}^{k}) and the corresponding execution time period for those programs (T_{ex}^{k}) (*k*=1,2,…,*K*).

If $A(t)$ is the availability of VM at time *t* and we assume that the programs require input files at the beginning time, T_{bf}^{j}, the availability of the input files can be calculated as

$$P_f(j) = A(T_{bf}^{j}), j=1,2,\ldots J. \tag{7.1}$$

It is assumed that the VM is available from the beginning to the end when a program runs on it otherwise, the program fails. Thus, the average availability of the programs, which start at time T_{bp}^{k} with the execution time period T_{ex}^{k}, can be calculated as

$$P_{pr}(k) = \int_{T_{bp}^{k}}^{T_{bp}^{k}+T_{ex}^{k}} A(t)dt \,/\, T_{ex}^{k}, k=1,2,\ldots,K. \tag{7.2}$$

Let *N* be the number of subsystems in the CHDS. The DSR for the *i*-th subsystem is denoted by DSR$_i$ (*i*=1,2,…,*N*). Let the VM initially be a perfect node in every subsystem and compute DSR$_i$ (*i*=1,2,…,*N*) for every subsystem.

In order to calculate distributed service reliability, some additional assumptions on statistical independence are needed:

1) DSR$_i$ (*i*=1,2,…,*N*) is assumed to be mutually independent;
2) The files prepared in the VM are also mutually independent;
3) The programs running in the VM are mutually independent. Although the independence assumption may not always be true, it is acceptable as a first order approximation. In fact, when service requests are independent, the failures of the VMs in providing the service will also be independent.

The distributed service reliability function to the initial time, t_b, can be calculated by

$$R_s(t_b) = \prod_{i=1}^{N} DSR_i \cdot \prod_{j=1}^{J} P_f(j) \cdot \prod_{k=1}^{K} P_{pr}(k) \tag{7.3}$$

We view the VM as a perfect node in calculating DSR_i without considering the availability of prepared files and executed programs in it. Thus, the service reliability is the whole DSR $\prod_{i=1}^{N} DSR_i$ multiplied by the availability of files and programs in VM.

The availability of files and programs in VM can be expressed as the product of $\prod_{j=1}^{J} P_f(j)$ and $\prod_{k=1}^{K} P_{pr}(k)$. Hence, the overall distributed service reliability function can be expressed as Equation (7.3). Note that the model is a general one and any specific reliability and availability functions can be used.

7.1.3 Solution algorithm

In applying the general approach, we need the structure of CHDS and can then use the model in Section 3.1. The algorithm for the calculation of the distributed service reliability can be presented as the following six steps:

Step 1: Identify the structure of CHDS and relationship between programs and files;

Step2: Obtain the availability function of the VM with any existing models;

Step 3: Let the VM be a perfect node in every subsystem and calculate DSR$_i$ (i=1,2,…,*N*);

Step 4: Using the critical path method to determine T_{bf}^{j} (j=1,2,…*J*) and T_{bp}^{k} , T_{ex}^{k} (*k*=1,2,…,*K*);

Step 5: Calculate $P_f(j)$ and $P_{pr}(k)$ as shown in Equation (7.1-7.2).

Step 6: Calculate the distributed service reliability function to the initial time, t_b, through Equation (7.3).

Note that we can implement different models and methods to calculate distributed service reliability. (1) For subsystems, there are two conditions to calculate DSR$_i$: a) Assume the nodes in sub-distributed system are perfect. The DSR$_i$ can be calculated through the algorithms (Kumar *et al.* 1986, Kumar *et al.* 1988, Chen & Huang 1992, Kumar & Agrawal 1993, Chen *et al.* 1997). b) Assume the nodes in sub-distributed system are imperfect. The DSR$_i$ can be calculated through the algorithms (Ke & Wang 1997, Lin *et al.* 1999b). (2) For the availability function of the VM $A(t)$, it can be calculated differently through the methods given in (Welke *et al.* 1995, Lai *et al.* 2002, Hariri & Mutlu 1995, Laprie & Kanoun 1992) and so on if the conditions match the assumptions in these articles.

7.2 An application example

An actual bank automatic payment system is investigated as a numerical example of the service analysis of a CHDS. In this system, there is a payment center and three sub-payment systems.

7.2.1 The structure of CHDS

The structure of this distributed service system is described in Figure 7.2.

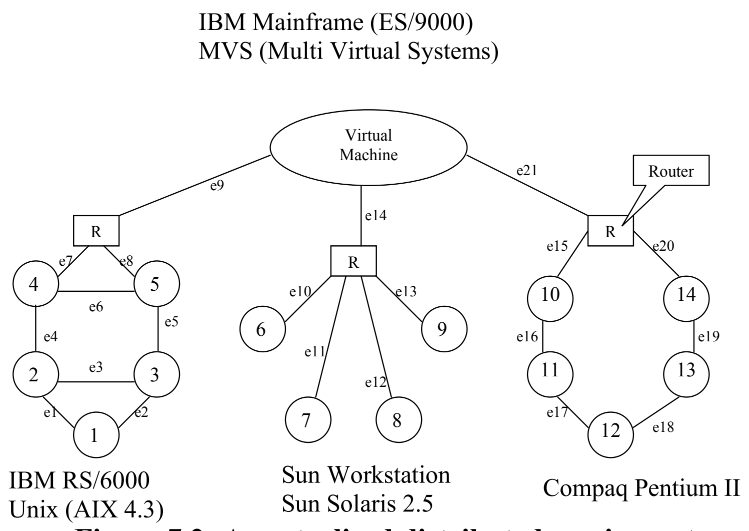


Figure 7.2: A centralized distributed service system

In Figure 7.2, there are three subsystems. The network topologies are common topologies, a star topology and a ring topology, in which "*R*" means router and $e_i, i \in [1, \ 21]$ is the links among the nodes.

Table 7.1 shows the programs and prepared files arranged in the distributed system. Table 7.2 shows the relationship between programs and their precedent programs. If there are no precedent programs for a program, it can run at initial time when input files are available. Table 7.2 also shows the input files and execution times for every program. If there are no input files required by a program, the program can run immediately after its precedent programs are completed.

Table 7.1: The programs and prepared files in different nodes

Node	Programs	Files
1	P1	F1, F5
2	P4	F1, F2
3	P2, P3	F2, F5
4	P2, P3	F2, F5
5	P4	F3, F6
6	P5, P7	F6
7	P6	F7, F8, F9
8	P7	F7, F8, F9
9	P5, P6	F6
10	P8, P11	F10, F11, F12
11	P9	F11
12	P10	F10
13	P9, P10	F12
14	P8, P11	F10, F11, F12
VM	SP1, SP2, SP3, SP4	F4, F13, F14

Table 7.2: Required files, precedent programs and execution time for programs

Programs	Required Files	Precedent Programs	Execution Time (T_{ex})
P1	F1,F2,F3	------	5
P2	F2,F4,F6	------	25
P3	F1,F3,F5	P1,P2	32
P4	F1,F2,F4,F6	SP1,SP2	33
SP1	F6	P3,P6	43
P5	------	------	17
P6	F6,F13,F9	P5	19
P7	F6,F8	SP2,SP3	21
SP2	F2,F11	P9,P10	16
P8	------	P1	45
P9	F11,F12	P5	121
P10	F11,F14	SP1	37
SP3	F3,F8	P8,P10	21
P11	F14,F10,F12	SP3	32
SP4	F5,F12	P4,P7,P11	20

"------" means no precedent programs or no input files.

7.2.2 The availability function

The VM is assumed to have a failure intensity function $\lambda(t)$. There are maintenance personnel to repair the failure of the VM and the repair time is exponentially distributed with parameter $\mu = 0.5$. For the failure intensity function of the VM, we use the GO model presented by Goel and Okumoto (1979) in which the failure intensity function is given by

$$\lambda(t) = ab\exp(-bt) \tag{7.4}$$

The values of *a* and *b* are assumed to be 10 and 0.01, respectively, in this example. We incorporate this model into the Markov process as a time-dependent Markov model. Note that any other model can be used but this model is selected here because it is the most widely used.

In our Markov model, we assume that there are two states, up (working state) and down (malfunctioning state). Let $P_0(t)$ be the probability for the VM to be working at time *t*, and $P_1(t)$ be the probability for it to be in a malfunctioning state at time *t*. The corresponding Kolomogorov's differential equations are

$$P_0^{'}(t) = \mu P_1\ (t) - \lambda(t) \cdot P_0\ (t) \tag{7.5}$$

and

$$P_1(t) = 1 - P_0\ (t) \tag{7.6}$$

With the initial conditions $P_0(0)=1$, $P_1(0)=0$, it can be shown that

$$P_0(t) = \left[\int_0^t \mu e^{(\mu x - ae^{-bx})} dx + 1/e^a \right] \cdot e^{(-\mu t + ae^{-bt})} \tag{7.7}$$

which is the availability function $A(t)$ in our case.

7.2.3 The distributed system reliability

The DSR from the left subsystem to the right subsystem in Figure 7.1 is denoted by DSR$_i$ (*i*=1,2,3). The three subsystems can be separated as shown in Figure 7.3.

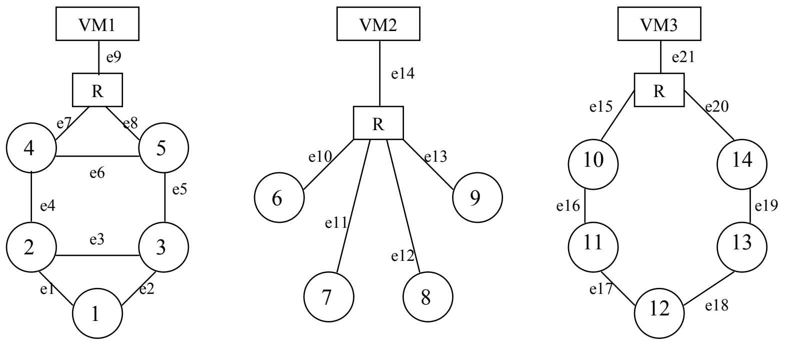


Figure 7.3: The separated subsystems from Figure 7.1.

In Figure 7.3, VM*i* (*i*=1,2,3) represents the VM used in subsystem *i*. Now we calculate DSR$_i$ (*i*=1,2,3) numerically with the assumptions that all the nodes are perfect and the probability for every communication edge to be available is 0.9. The graphs for DSR_1 can be reduced through the rules in FST-SPR algorithm presented in Chen and Huang (1992) as shown in Figure 7.4.

Hence, we can obtain the result of DSR_1=0.9496 through the GEAR algorithm presented by Kumar and Agrawal (1993). In the same way, we can get DSR_2=0.8817 and DSR_3=0.9068.

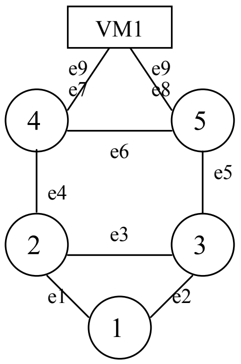


Figure 7.4 The reduced graph for subsystem 1.

7.2.4 The distributed service reliability function

The critical path graph for the example given in Table 7.2 is drawn in Figure 7.5. The value marked on the edge is the execution time, those on the node are the starting times and the dot-arrow lines represent the critical path.

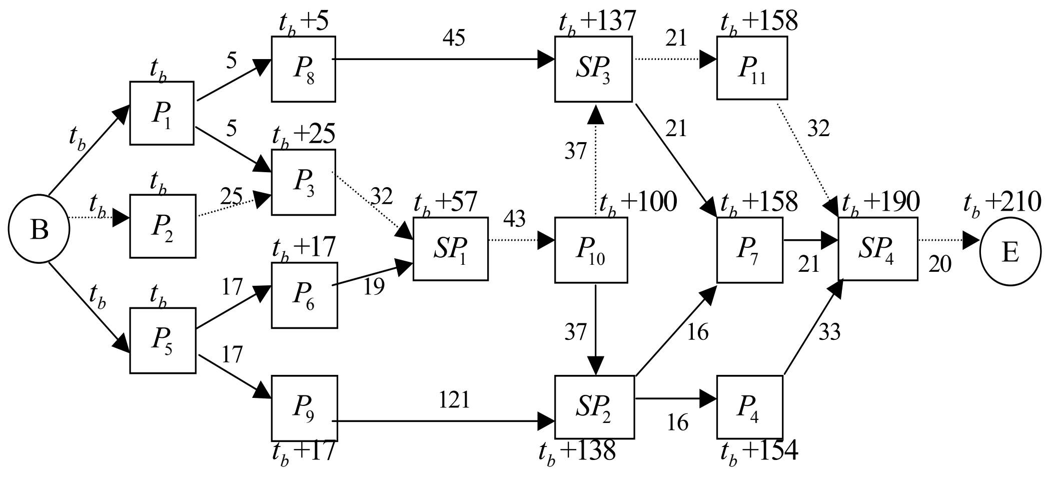


Figure 7.5: Critical path for Table 7.2

From the critical path shown in Figure 7.5 and Table 7.2, T_{bf}^{j} (j=1,2,…,5) can be shown to be {t_b, t_b+17, t_b+100, t_b+154, t_b+158} for the programs {P2, P6, P10, P4, P11} using the files prepared in the VM. We can also get T_{bp}^{k} (k=1,2,3,4) to be {t_b+57, t_b+138, t_b+137, t_b+190) and the corresponding execution time period T_{ex}^{k} to be {43, 21, 16, 20} for the programs {SP1, SP2, SP3, SP4} executed in the VM.

With Equations (7.1, 7.2, 7.7), we get

$$P_f(j) = A(T_{bf}^{j}),\, j=1,2,\ldots 5,$$

in which T_{bf}^{j} is {t_b, t_b+17, t_b+100, t_b+154, t_b+158} and

$$P_{pr}(k) = \int_{T_{bp}^{k}}^{T_{bp}^{k}+T_{ex}^{k}} A(t)dt / T_{ex}^{k},\; k=1,2,3,4,$$

in which T_{bp}^{k} is {t_b+57, t_b+138, t_b+137, t_b+190) and T_{ex}^{k} is {43, 16, 21, 20}.

Hence, using Equation (7.3), we can obtain the distributed service reliability function to service starting time t_b as

$$R_s(t_b) = \prod_{i=1}^{3} DSR_i \prod_{j=1}^{5} P_f(j) \prod_{k=1}^{4} P_{pr}(k) \tag{7.8}$$

This distributed service reliability function has the form displayed in Figure 7.6.

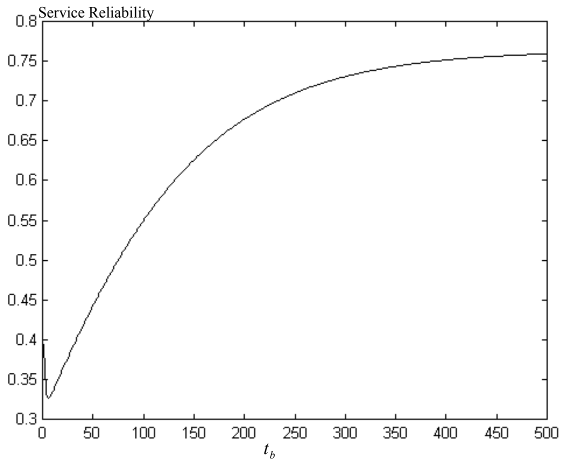


Figure 7.6: Typical distributed service reliability function to service starting time.

From Figure 7.6, it can be observed that the lowest service reliability is not at the initial time point when the software failure intensity of the VM is the highest as Equation (7.4). This is because we assumed that the initial state for the VM is up (working). When t_b is larger than the lowest point, the distributed service reliability increases. This is because identified bugs of the VM are fixed, resulting in a decrease in the failure intensity. Towards the end, the distributed service reliability approaches a steady state availability of 0.7592, which is obtained by

$$\prod_{i=1}^{3} DSR_i = 0.9496 \times 0.8817 \times 0.9068 = 0.7592.$$

When the availability of VM approaches 1, the distributed service reliability is approaching 0.7592.

7.3 Further analysis and application of the general model

With specific input parameters, the distributed service reliability can be computed. Via the modelling and further analysis, some general conclusions can be drawn. The VM in the control center is the heart of the CHDS, and hence, the system availability $A(t)$ of the VM is critical to the distributed service reliability. In order to achieve a high reliability of the service, the control center should be equipped with sufficient maintenance personnel to repair the failures of the VM. The availability function of the VM can help the decision maker to allocate maintenance personnel effectively at different stages and to decide the release time that provides certain pre-required system availability. In this section we discuss some related analysis that makes use of the general model and which could be of importance in practical applications.

7.3.1 A general approach

The system availability reaches the lowest point at an early stage. This is because a large number of faults are identified when system testing begins. The system availability starts recovering after the lowest point and approaches a steady value after an extended period of time. This state is reached when identified faults have been fixed. The time at which the system availability is at its minimum is important as around this time point t^*, a significant amount of effort needs to be put into fault fixing and system testing to help increase system availability of the VM quickly. When the faults are fixed, the system availability recovers and effort on fault fixing and testing can be reduced accordingly. Eventually, only a few faults will be left and at this stage, the manpower for the fault fixing and system testing of the VM can be moved to elsewhere. Hence, the minimum system availability time point t^* is an important indicator for the managers of the control center for enabling them to distribute the resources on the VM at different stages.

It is easy to calculate the time of minimum system availability if the availability function of the VM, $A(t)$, is known. By differentiating $A(t)$, and then solving $A^{'}(t)$ =0, we can get the solution that is the minimum time point t^{*}.

Furthermore, if the management wants to know the time when the VM system reaches certain availability level A_L, the system availability function $A(t)$ can be used by solving the equation of $A(t) = A_L$. Its solution can help the managers to decide the release time of the VM accordingly. For example, the customers may require the system availability to be at least A_L. Hence, we need to know the time point when the system availability reaches this required system availability level. At this point the testing can be stopped and the system can be released.

Another important issue in this type of analysis is the sensitivity studies. Usually the model parameters are assumed to be known. A deviation from the assumed value could lead to significant differences between the actual and the calculated values. To minimize these errors, effort should be made to obtain accurate estimates of the important parameters. Since a number of parameters are involved, it is useful to identify the ones that influence the results most. Sensitivity analysis of the parameters is therefore highly recommended. The results obtained from these analyses can help decision makers and analysts to better allocate the resources.

7.3.2 The application example revisited

To clearly address some of the issues raised in the previous section, we revisit the application example in Section 7.2 with some further analysis. This type of study is important in system studies and for the management to fully make use of the modelling and analysis.

7.3.2.1 Minimum system availability of the VM

The minimum availability point of Equation (7.7) can be obtained by taking the derivative and setting it to zero. That is

$$A'(t) = \mu e^{(\mu t - ae^{-bt})}\; e^{(-\mu t + ae^{-bt})} + \left[\int_0^t \mu e^{(\mu x - ae^{-bx})} dx + \frac{1}{e^a}\right] \cdot e^{(-\mu t + ae^{-bt})} \cdot (-\mu - abe^{-bt})$$

$$= \mu + \left[\int_0^t \mu e^{(\mu x - ae^{-bx})} dx + \frac{1}{e^a}\right] \cdot e^{(-\mu t + ae^{-bt})} \cdot (-\mu - abe^{-bt}) \qquad (7.9)$$

Let $A'(t)$ =0 and let t^* be the solution, i.e.,

$$\mu + \left[\int_0^{t^*} \mu e^{(\mu x - ae^{-bx})} dx + \frac{1}{e^a}\right] \cdot e^{(-\mu t^* + ae^{-bt^*})} \cdot (-\mu - abe^{-bt^*}) = 0 \qquad (7.10)$$

It is not difficult to obtain the value of t^* numerically by using *Maple* or *Mathematica*, or some other symbolic software.

For example, with parameters a=10, b=0.01 μ =0.5, Equation (7.10) can be solved by *Maple* to give t^*=8.88 and the minimum system availability $A(t^*)$=0.8453.

7.3.2.2 Time to achieve a required system availability

Suppose that the customers require the system availability to be at least A_L. From Equation (7.7), this can be obtained by solving the following equation

$$A(t) = \left[\int_0^t \mu e^{(\mu x - ae^{-bx})} dx + \frac{1}{e^a}\right] \cdot e^{(-\mu t + ae^{-bt})} = A_L \qquad (7.11)$$

Since there may be two solutions, we require that $t \geq t^*$ where t^* can be solved by Equation (7.10) first.

A simple approximation is presented here for solving Equation (7.11) and carrying out further analytical study. In a Markov Chain, there is a transition time from initial state to steady state. We assume that it takes more time between the initial time and the release time of the test than the transition time of the Markov process. Based on this assumption, from the equations for long-run Markov chain (Hillier & Lieberman 1995) we get

$$A(t) = P_0(t) = \frac{\mu}{a\ be^{-bt} + \mu} \tag{7.12}$$

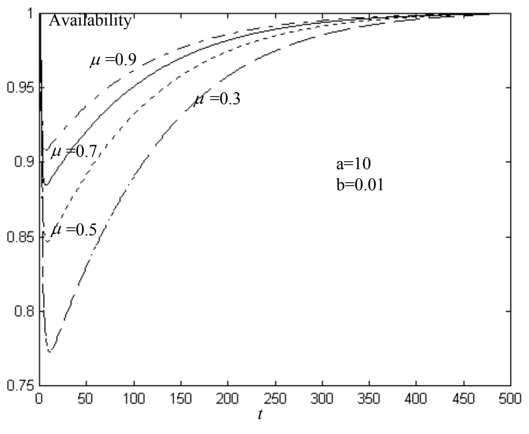
In order to calculate the time point that satisfies the customers' requirement A_L, let $A(t) = A_L$ and t can be obtained as

$$t = -\frac{1}{b} \cdot \ln\left((\frac{1}{A_L} - 1) \cdot \frac{\mu}{ab} \right) \tag{7.13}$$

In our example with parameters a=10, b=0.01 μ =0.5, the time point for A_L=0.98 can be calculated be 228.24.

7.3.2.3 Sensitivity analysis

There are three parameters in the availability function (7.7), a, b and μ. The sensitivity of different parameters is shown in Figure 7.7 and Figure 7.8.



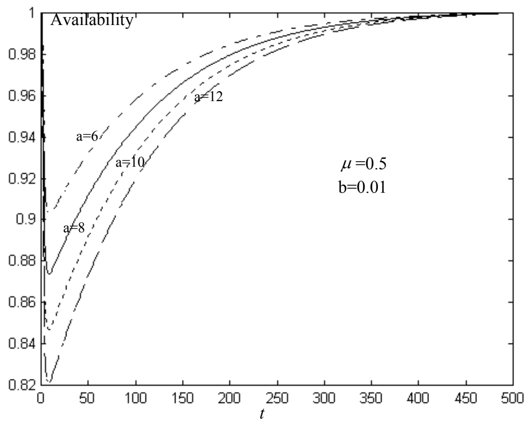


Figure 7.7: Sensitivity of μ (left) and *a* (right)

As expected, a greater repair rate implies higher system availability. Similarly, when *a* increases, the system availability decreases because the failure intensity function increases as Equation (7.4). However, in the case of parameter *b* the effect is not obvious, as shown in Figure 7.8

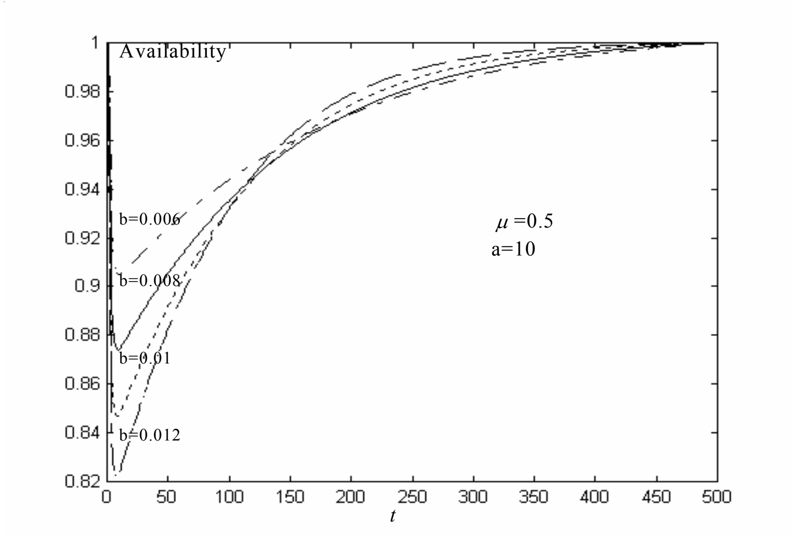


Figure 7.8: Sensitivity of *b*.

The curves in Figure 7.8 cross each other which means that when *b* increases, system availability decreases at the early stage and increases at the later stage. Equation (7.4) can be used to explain this. There are two parts in the failure intensity function of GO model, *ab* and exp(-*b*t). When *b* increases, the first part, *ab*, increases while the second part, exp(-*b*t), decreases. Thus, at the early stages, when the time *t* is small the influence of the second part is less than that of the first part so the failure intensity function increases and the system availability decreases. Conversely, at the later stage, the time *t* is large and the influence of the second part is more than that of the first part so the failure intensity function decreases and the system availability increases.

With Equation (7.10), we can calculate the time point of the minimum system availability and the time a certain availability is achieved. On the other hand, it would be useful to see the influence of the repair rate on these two quantities. We analyze the Markov model with the numerical example presented in Section 7.3.2. It is assumed that a=10 and b=0.01. Let μ change from 0.3 to 0.7 to calculate the minimum system availability point through Equation (7.10). The time of the minimum system availability t^* *vs.* the repair rate μ is shown in the left curve of Figure 7.9. The minimum system availability $A(t^*)$ *vs.* μ is depicted in the right curve of Figure 7.9.

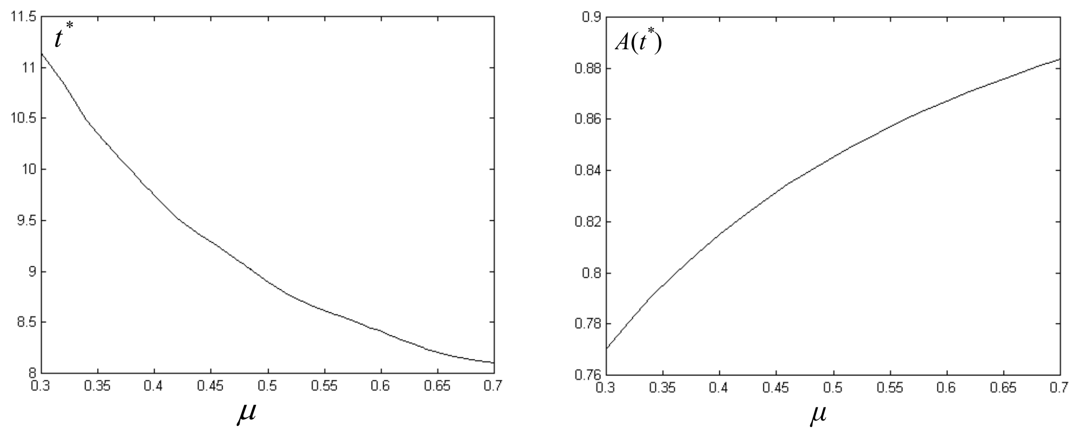


Figure 7.9: Sensitivity analysis of repair rate

From Figure 7.9, we can see the rate of decrease in t^* (rate of increase in $A(t^*)$) as the repair rate μ increases. We can also see that $A(t^*)$ is a convex function of μ. This means that adding $\Delta\mu$ on a small μ improves more availability than adding the same $\Delta\mu$ on a large μ. The curve of "t^* *vs.* μ" is concave, which means that adding $\Delta\mu$ on a small μ reduces more time of minimum availability than adding the same $\Delta\mu$ on a large μ. This type of study is useful for allocating the maintenance personnel optimally, which is another interesting problem for the further research.

7.4 Conclusions

In this chapter, a general model was presented for the widely CHDS. Based on this model, a solution algorithm was presented and the time for the VM to reach either its minimum system availability or a specifically required system availability was studied.

An application of the model on an actual bank automatic payment system was shown. In addition, sensitivity analysis were carried out to determine the effects of the intrinsic parameters on the system availability and the lowest availability point.

Since our approach is general and the CHDS has been applied in different areas, the algorithm for the distributed service reliability analysis can be used to estimate the reliability of the service in a distributed system during both the testing phase and the operational phase. During the testing phase, the service reliability function can help to allocate testing resources accordingly. For example, around the minimum service reliability time, more maintenance personnel and testing resources should be allocated to test and repair the system than at the later stage when the service reliability is high and the amount of testing resource can therefore be reduced.

Also, if given a requirement on the service reliability after release, the time for release can also be determined. Moreover, for projects with fixed deadlines, the model can help system managers to determine the testing intensity or manpower according to the estimated reliability performance given different levels of testing intensity. Furthermore, during the operational phase, the quality of service can also be assessed through the service reliability measure.

Chapter 8

Conclusions and Future Work

In this chapter, a summary of the merits and the limitations of the work conducted is offered and areas for future research are suggested to conclude this dissertation.

8.1 Conclusions

The system reliability and schedule length are two very important criteria for DCSs. Hence, this dissertation focuses on heuristics algorithms to maximize the system reliability and/or to minimize the completion time (schedule length).

8.1.1 Reliability oriented algorithms

When the topology of a DCS is fixed, the DSR depends mainly on the assignment of various resources such as the programs and files. Especially for systems with long mission times or with a large number of processors, an improved program allocation can increase the system reliability dramatically.

8.1.1.1 Modelling

There has been extensive work done on the development of program and file allocation algorithms designed to maximize system reliability. However in most of this work the program and file allocations have been considered separately whereas to achieve the highest level of system reliability these allocations should be considered simultaneously. Chapter 3 of this dissertation presents a reliability-oriented optimization model in which both program allocation and file allocation are considered together. In real world situations, there are a number of constraints on, for example, the storage, costs and completion times. To make the model more practical, therefore these constraints have also been taken into account making it more practical and more comprehensive than models previously developed by others, as can be seen from the following comparison.

Kumar *et al.* (1995a) developed a genetic algorithm (GA) to solve a file allocation scheme. In their scheme, the objective function was to maximize the distributed program reliability (DPR). From the system level viewpoint, the distributed systems reliability (DSR) describes the system better than the DPR. Hence, the objective of our optimization model was to maximize the DSR. When the number of programs is set to one, the objective to maximize DSR is the same as maximizing DPR, the objective function in Kumar *et al.* (1995a). When the program allocation is fixed, the models will degenerate to the file allocation problem discussed by Kumar *et al.* (1995a). In the optimization model of Kumar *et al.* (1995a) the different constraints, for example the total number of copies of each file and the memory constraint at each node, are discussed. In our model, some additional constraints such as the cost constraint and completion time constraints are considered. Although more constraints make the GA more difficult to implement, they make the optimization model more practical.

In the optimization model of Kartik & Murthy (1997) to solve the program allocation problems for maximizing the DSR, the network topology was assumed cycle-free. Our proposed optimization model does not limit the topology and permits redundancies, which makes it more generally applicable. Also, the program allocation and file allocation are both considered together by our proposed optimization model, making it more general and practical than that of Kartik & Murthy (1997).

A sensitivity analysis showed that the optimal assignment was robust to variations in program prices and that extended completion time might improve the DSR and cause more computers to become available for carrying out other services without sacrificing the DSR.

Program allocation and file allocation problems are NP-hard,. However, considering program and file allocation together and taking into account resource constraints makes the problem harder to solve. Hence, in chapter 3 a genetic algorithm is also proposed to aid solution of this problem.

8.1.1.2 Genetic algorithm

Genetic algorithms are easy to model and be implemented to solve various problems. However, the crossover and mutation operators may produce some infeasible solutions. To overcome this problem, adjustment (repair) operators were implemented to adjust an infeasible solution to a feasible one. The repair scheme is suitable for the case where infeasible solutions appear frequently but can be repaired without too much computational cost. In addition, the "fitness" which is a function of reliability is used instead of directly using the reliability. This function can enlarge the difference between the individuals to give more chance to the better individuals, so that the algorithm can converge rapidly.

Numerical simulations were run to evaluate the performance of the proposed GA. When the solution space is small, the results of GA were compared with that of an exhaustive search algorithm. The comparison showed that in most cases the GA could obtain the optimal solutions with much less computation time. On the other hand, when the solution space is very large, the exhaustive algorithm cannot finish in an acceptable time but the genetic algorithm can obtain some good solutions and hence is strongly recommended.

8.1.1.3 Tabu Search and comparison with the GA

For many combinatorial optimization problems, GA can provide excellent results. However, GA is a population-based search, and requires the evaluation of multiple prospective solutions over many generations. Hence, for some complex problems, GA's may need a significant amount of computational effort. In addition, when the problem has certain constraints, the crossover and mutation may produce some infeasible solutions. Some effort is needed to deal with these infeasible solutions. Unlike GA's, TS is not population-based but successively moves from solution to solution. This offers some potential for improved efficiency if it also provides the same quality of solutions in a shorter time or provides improved quality for the same time.

Comparative studies show that in some cases GA outperforms TS, but in others TS outperforms GA. Due to the widely different views on the efficiency, chapter 4 proposed a TS and then compared the performance of GA and TS to gauge their suitability for solving the program and file allocation problem.

The whole solution space is inherently partitioned into several subsets according to the number of the copies of the programs and files. The TS combined with the "branch-and-bound" technique was implemented and particular features such as "back-tracking" and "restarting" were incorporated.

The results from two numerical examples showed that TS outperforms GA with short computing time and better solution quality. However, the design of good TS requires far more insight into the problem and much more effort is needed compared to the requirements for implementing a good GA for the same problem. Hence if we do have good knowledge of the state space, TS should be used, otherwise, GA may be a better choice.

8.1.1.4 A parallel TS

In some practical situations, scheduling must be completed within a short time interval. To shorten the execution time of an algorithm without comprising the solution quality, a natural method is to parallelize the algorithm. Hence, in chapter 4 a Parallel Tabu Search (PTS) is proposed to solve the program and file allocation problem.

For this problem, the solution space can be inherently partitioned into a number of subsets, and multiple search paths used in parallel to search different subsets so as to accelerate the TS. The implementation of PTS followed a master-slave scheme. The simulation results showed that the speedup of the PTS basically grows linearly with number of processors when the number of processor was not very large. A possible reason for this is that when the solution space is partitioned into subsets, every processor searches a subset; and only a small amount of communication is needed, thus generating an approximately linear function. The simulation results showed the solution quality was virtually unaffected by the number of processors.

8.1.2 Completion time oriented algorithm

Completion time is another important parameter in DCSs. In a parallel application the data dependencies can be represented by a directed acyclic graph (DAG). Intensive research has been done on DAG scheduling to minimize the completion time (schedule length), which is a NP-hard problem. However, most of these algorithms assume that the DCSs were homogeneous, for example, list scheduling. Heterogeneous Earliest-Finish-Time (HEFT) algorithm (Topcuoglu *et al.* 2002) adopted list scheduling for heterogeneous systems, it significantly outperformed Mapping heuristic (MH) (El-Rewini & Lewis 1990), Dynamic-Level Scheduling (DLS) algorithm (Sih & Lee 1993), Levelized-Min Time (LMT) algorithm (Iverson *et al.* 1995) in terms of average schedule length ratio, speedup, etc. However, as it only uses the mean value to construct the scheduling list, the scheduling may be misdirected.

Chapter 5 proposed an iterative algorithm based on the idea of list scheduling for heterogeneous systems. The algorithm generates an initial solution with moderate quality and then improves the solution iteratively. During the iterative steps, the results of the previous iteration are used to construct a new list. The initial step happens to be same as HEFT. Consequently, if the final schedule length is less than the initial one, the iterative algorithm can produce shorter schedule length than those of the HEFT, DLS, MH, LMT.

To test the performance of the proposed algorithm, a random generator of direct acyclic graphs was designed. Simulations were run on a large number of randomly generated problems of different sizes and two real applications, and the results showed that in the majority of cases, there were significant improvements made to the initial schedules, which means that the proposed algorithm outperforms HEFT algorithm, DLS algorithm, MH, LMT algorithm in terms of the average schedule length. In particular, the algorithm performs better when the tasks to processors ratio is large.

Sensitivity analysis shows that neither the percentage that final schedule length is less than the initial one nor the average improvement ratio are sensitive to the weighting factor ,i.e., the weight when computing the mean during iteration.

8.1.3 Completion time and reliability oriented algorithm

Although intensive research has been done on DAG scheduling, most proposed scheduling algorithms are designed to minimize the schedule length without consideration of the possible failure of the computation machines or associated network resources. In a failure-prone system, assigning tasks to machines without considering possible failures may result in a significant increase in the average execution time of the application in the presence of failures. Chapter 6 described a multi-objective optimization problem to minimize schedule length and maximize the system reliability simultaneously.

Evolutionary algorithms have been extensively adopted to solve the multi-objective optimization problems, however for this particular problem, it is very hard to recombine two solutions to generate new solutions due to the data dependency. Hence, a TS was proposed to solve this multi-objective problem.

To distribute the Pareto-optimal solutions along the Pareto-front uniformly, “lateral interference” was adopted. To apply “lateral interference”, it is necessary to compute the distance metric between the solutions and two schemes have been proposed to do this.

Randomly generated DAGs and a real application were used to determine the performance of the proposed algorithm. The “non-dominated solution number” and the “uniform distribution of non-dominated solutions” are the two performance measures used to compare the two schemes, one considering the “lateral interference” and the the other without. For “non-dominated solution number” there was basically no difference between the three schemes. For “uniform distribution of non-dominated solutions, the two schemes considering the “lateral interference” were much better than the one without considering it and there was basically no difference between the first two schemes.

8.1.4 Reliability analysis and computation for DCS

Service reliability of a distributed system is a key criterion of QoS (Quality of Service). Most of the distributed systems can be modeled as CHDSs. The service reliability in a CHDS is determined not only by the system availability of the control center, but also by distributed program reliabilities of the subsystems. Most earlier research on system availability or reliability cannot be simply applied to analyze the service reliability of CHDSs.

Chapter 7 described a general model for a CHDS. Based on this model, a solution algorithm was presented and the time for the VM to reach its minimum system availability or required system availability was studied. An application of the model to an actual bank automatic payment system was presented. In addition, a sensitivity analysis was conducted to determine the effect on the system availability of certain intrinsic parameters and their effect on the lowest availability point.

8.2 Future work

For the program and file allocation problem proposed in chapter 3, when the solution space is very large, the exhaustive algorithm cannot finish execution in acceptable time, and therefore no optimal solutions are available to compare with the results of the proposed algorithms. The upper bound, if available, would be very useful for evaluating the performance of the algorithms proposed in this dissertation. Although 1 is the upper bound of the system reliability, for some cases, the system reliability of the optimal solution is far from 1. A method to obtain a realistic upper bound for the proposed problem would be valuable as the alternative value of 1 is in many cases not appropriate .

Most DAG scheduling to minimize the schedule length assumes that the processors of the system are fully linked, which means that there is no communication contention. The assumption is true when the number of the processors is not large, but when the number of the processors is very large, this assumption is obviously not true. When communication contention exists, communication scheduling and routing must be taken into account. Future research on DAG scheduling to maximize the schedule length should consider this communication contention.

References:

Aggarwal, K.K. and Rai, S. (1981), Reliability evaluation in computer-communication networks, IEEE Transactions on Reliability, R-30(1), pp. 32-35.

Ahmad, I., and Kwok, Y.-K. (1998), On exploiting task duplication in parallel program scheduling, IEEE Transactions on Parallel and Distributed Systems, 9 (9), pp. 872-892.

Augugliaro, A., Dusonchet, L., and Sanseverino, E.R. (1999), Genetic, Simulated Annealing and Tabu Search algorithms: Three heuristic methods for optimal reconfiguration and compensation of distribution networks, European Transactions on Electrical Power, 9 (1), pp. 35-41.

Balicki, J., and Kitowski, Z., (2001), Multicriteria evolutionary algorithm with tabu search for task assignment, Lecture Notes in Computer Science, 1993, pp. 373-384.

Bannister, J.A. and Trivedi, K.S. (1983), Task allocation in fault-tolerant distributed systems, Acta Informatica, 20 (3), pp. 261-281.

Baykasoglu, A. (2002), Linguistic-based meta-heuristic optimization model for flexible job shop scheduling, International Journal of Production Research, 40 (17), pp. 4523-4543.

Ben-Tal, A. (1980), Characterization of Pareto and lexicographic optimal solutions, In Multiple Criteria Decision Making Theory and Application, Fandel G., and Gal, T. editors, vol. 177 of Lecture Notes in Economics and Mathematical Systems. Berlin, Germany: Springer-Verlag, pp. 1–11.

Bokhari, S.H. (1979), Dual processor scheduling with dynamic reassignment, IEEE Transactions on Software Engineering, SE-5 (4), pp. 341-349.

Bokhari, S.H. (1981), A shortest tree algorithm for optimal assignments across space and time in a distributed processor system, IEEE Transactions on Software Engineering, 7 (6), pp. 583-589.

Braun, T.D., Siegel, H.J., Beck, N., Bölöni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B., Hensgen, D., and Freund, R.F. (2001), A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems, Journal of Parallel and Distributed Computing, 61 (6), pp. 810-837.

Budenske, J.R., Ramanujan, R.S. and Siegel, H.J. (1997), On-line use of off-line derived mappings for iterative automatic target recognition tasks and a particular class of hardware platforms, In Proceedings of the Sixth Heterogeneous Computing Workshop, pp. 96 –110.

Budenbender, K., Grunert, T., and Sebastian, H.J. (2000), A hybrid Tabu Search/Branch-and-Bound algorithm for the direct flight network design problem, Transportation Science, 34 (4), pp. 364-380.

Casavant, T.L. and Kuhl, J.G. (1998), A taxonomy of scheduling in general-purpose distributed computing systems, IEEE Transactions on Software Engineering, 14 (2), pp. 141-154.

Cerny, V. (1985), Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm, Journal of Optimization Theory and Application, 45 (1), pp. 41–51.

Chang, M.S., Chen, D.J., Lin, M.S. and Ku, K.L. (1999), Reliability analysis of distributed computing systems in ring networks, Journal of Communication and Networks,1 (1), pp.68-77.

Chang, M.S., Chen, D.J., Lin, M.S. and Ku K.L. (2000), The distributed program reliability analysis on star topologies, Computer & Operations Research, 27 (2), pp. 129-142.

Chang, P-Y., Chen, D-J and Kavi, K.M. (2001), File allocation algorithms to minimize data transmission time in distributed computing systems, Journal of Information Science and Engineering, 17 (4), pp. 633-646.

Chari, K. (1996), Resource allocation and capacity assignment in distributed systems, Computers & Operations Research, 23 (11), pp. 1025-1041.

Chen, D-J., Chen, R-S. and Huang, T-H. (1997), A heuristic approach to generating file spanning trees for reliability analysis of distributed computing systems, Computers & Mathematics with Applications, 34 (10), pp. 115-131.

Chen, D-J. and Huang, T-H. (1992), Reliability analysis of distributed systems based on a fast reliability algorithm, IEEE Transactions on Parallel and Distributed Systems, 3 (2), pp. 139 –154.

Chen, C.L., Lee, C.S.G., and Hou, E.S.H. (1988), Efficient scheduling algorithms for robot inverse dynamics computation on a multiprocessor system, IEEE Transitions on Systems, Man, Cybernetics, 18 (5), pp. 729-743.

Chen, D-J. and Lin, M-S. (1994), On distributed computing systems reliability analysis under program execution constraints, IEEE Transactions on Computers, 43 (1), pp. 87-97.

Chen, W-H. and Lin, C-S. (2000), A hybrid heuristic to solve a task allocation problem, Computers & Operations Research, 27 (3), pp. 287-303.

Chern, M.S., Chen, G.H. and Liu, P. (1989), An LC branch and bound algorithm for module assignment problem, Information Processing Letters, 32 (2), 61-71.

Chiu, C.C., Yeh, Y.S. and Chou, J.S. (2002), A fast algorithm for reliability-oriented task assignment in a distributed system, Computer Communications, 25 (17), pp. 1622-1630.

Chu, W.W., Holloway, L.J., Lan, M.T., and Efe, K. (1980), Task allocation in distributed data processing, Computer 13, pp. 57-69.

Coello, C.A.C. (1996), An Empirical Study of Evolutionary Techniques for Multiobjective Optimization in Engineering Design, Ph.D. thesis, Department of Computer Science, Tulane University, New Orleans, USA.

Coello, C.A.C. (1999). A comprehensive survey of evolutionary-based multiobjective optimization techniques, Knowledge and Information Systems, 1 (3), pp. 269–308.

Coffman E. (1976), Computer and Job-Shop Scheduling Theory, John Wiley & Sons.

Colin, J.Y., Chretienne, P. and C.P.M. (1991), Scheduling with small computation delays and task duplication, Operations Research, 39 (4), pp. 680-684.

Coulouris, G. and Dollimore, J. (2000), Distributed Systems: Concepts and Design, Addison-Wesley, 3rd Ed.

Cvetkovic, D., Parme, I. and Webb, E. (1998), Multi-objective optimization and preliminary airframe design, The Integration of evolutionary and adaptive Computing Technologies with Product/System Design and Realization, Parme, I., Ed. Springer-Verlag, New York, pp. 255-267.

Darbha, S. and Agrawal, D.P. (1998), Optimal scheduling algorithm for distributed-memory machines, IEEE Transactions on Parallel and Distributed Systems, 9 (1), pp. 87 –95.

Dhodhi, M.K., Ahmad, I., Yatama, A. and Ahmad, I. (2002), An integrated technique for task matching and scheduling onto distributed heterogeneous computing system, Journal of Parallel and Distributed Computing, 62 (9), pp. 1338-1361.

Diaz, A.R., Tchernykh, A. and Ecker, K.H. (2003), Algorithms for dynamic scheduling of unit execution time tasks, European Journal of Operational Research, 146 (2), pp. 403-416.

Dogan, A., and Ozguner, F. (2002), Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing**,** IEEE Transactions on Parallel and Distributed Systems 13 (3), 308 – 323.

Dowdy, L.W. and Foster, D.V. (1982), Comparative models of the files assignment problem, ACM Computing Survey, 14 (2), pp. 287-311.

El-Rewini, H. and Lewis, T.G. (1990), Scheduling parallel program tasks onto arbitrary target machines, Journal of Parallel and Distributed Computing, 9 (2), pp. 138-153.

Erschler, J., Roubellat, F., and Vernhes, J.P. 1976. Finding some essential characteristics of the feasible solutions for a scheduling problem, Operations Research 24, 774–783 .

Fonseca, C. M. (1995), Multi-objective Genetic Algorithms with Application to Control Engineering Problems, Ph.D. Thesis, Department of Automatic Control and Systems Engineering, University of Sheffield, Sheffield, UK.

Fonseca, C.M. and Fleming, P.J. (1995), An overview of evolutionary algorithms in multi-objective optimization, Evolutionary Computation, 3 (1), pp.1-16.

Fourman, M.P. (1985), Compaction of symbolic layout using genetic algorithms, in Genetic Algorithms and Their Applications: Proc. 1st Int. Conf. Genetic Algorithms, J. J. Grefenstette, Ed. Princeton, NJ: Lawrence Erlbaum, pp. 141–153.

Fox, M.S. (1987), Constraint-Directed Search: A Case Study of Job-Shop Scheduling, Pitman, London.

Fu, H.P. and Su, C.T. (2000), A comparison of search techniques for minimizing assembly time in printed wiring assembly, International Journal Production Economic, 63 (1), pp. 83-98.

Gary, M.R. and Johnson, D.S. (1979), Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Co.

Gen, M., Ida, K. and Li, Y. (1995), Solving bi-criteria solid transportation problem with fuzzy numbers by genetic algorithm, International Journal of Computers and Industrial Engineering, 29, pp. 537-543.

Gen, M. and Cheng, R. (1997), Genetic Algorithms and Engineering Design, John Wiley and Sons, Inc., New York.

Gerasoulis, A. and Yang, T. (1992), A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors, Journal of Parallel and Distributed Computing, 16 (4), pp. 276-780.

Ghafoor, A., and Yang, J. (1993), Distributed heterogeneous supercomputing management system, IEEE Computer, 26 (6), pp. 78–86.

Glover, F. (1989), Tabu search-Part I, ORSA Journal on Computing, 1 (3), pp.190-206.

Glover, F. (1990), Tabu search-Part II, ORSA Journal on Computing, 2 (1), pp. 4-32.

Glover, F., Laguna, M. (1997), Tabu Search, Kluwer Academic Publishers, Boston.

Goel, A.L. and Okumoto, K. (1979), Time dependent error-detection rate model for software reliability and other performance measures, IEEE Transactions on Reliability, R-28 (3), pp. 206-211.

Goel, A.L. and Soenjoto, J. (1981), Models for hardware-software system operational-performance evaluation, IEEE Transactions on Reliability, R-30 (3), pp. 232-239.

Goldberg, D.E. (1989), Genetic Algorithms in Search of Optimization and Machine Learning, Addison Wesley.

Hajela, P., and Lin, C. Y. (1992), Genetic Search Strategies in Multicriterion Optimal Design, Structural Optimization, 4, pp. 99-107.

Hariri, S. and Mutlu, H. (1995), Hierarchical modelling of availability in distributed systems, IEEE Transactions on Software Engineering, 21 (1), pp. 50-56.

Hillier, F.S. and Lieberman, G.J. (1995), Introduction to Operations Research, McGroaw-Hill, New York.

Hilliard, M. R., Liepins, G.E., Palmer, M. and Rangarajen, G. (1989), The computer as a partner in algorithmic design Automated discovery of parameters for a multiobjective scheduling heuristic, In Sharda, B., Golden, L., Wasil, E., Balci, O. and Stewart, W. editors, Impacts of Recent Computer Advances on Operations Research, North-Holland Publishing Company, New York.

Holland, J.H. (1975), Adaptation in Natural and Artificial Systems, Ann Arbor. MI: Univ. of Michigan Press.

Hou, E.S.H, Ansari, N., and Ren, H. (1994), A genetic algorithm for multiprocessor scheduling, IEEE Transactions on Parallel and Distributed Systems, 5 (2), pp. 113-120.

Hwang, G-J. and Tseng, S-S. (1993), A heuristic task assignment algorithm to maximize reliability of a distributed system, IEEE Transactions on Reliability, 42 (3), pp. 408-415.

Hwang, J.J., Chow, Y.C., Anger, F.D. and Lee, C.Y. (1989), Scheduling precedence graphs in systems with interprocessor communication times, SIAM Journal of Computing, 18 (2), pp. 244-257.

Ignatius, P.P. and Murthy, S.R.C. (1997), On task allocation in heterogeneous distributed computing systems, Computer Systems Science and Engineering, 12 (4), pp. 231-238.

Iverson, M., Ozuner, F. and Follen, G. (1995), Parallelizing existing applications in a distributed heterogeneous environment, In Proceedings of Heterogeneous Computing Workshop, pp. 93-100.

Iverson, M. A. (1999), Dynamic Mapping and Scheduling Algorithms for a Multi-User Heterogeneous Computing Environment, Ph.D. thesis, The Ohio State University, Columbus, Ohio.

Jozefowska, J., Mika, M., Rozycki, R., Waligora, G. and Weglarz, J. (1998), Local search metaheuristics for discrete-continuous scheduling problems, European Journal of Operation Research, 107 (2), pp. 354-370.

Jozefowska, J., Mika, M., Rozycki, R., Waligora, G. and Weglarz, J. (2002), A heuristic approach to allocating the continuous resource in discrete-continuous scheduling problems to minimize the makespan, Journal of Scheduling, 5 (6), pp. 487-499.

Kafil, M. and Ahmad, I. (1998), Optimal task assignment in heterogeneous distributed computing systems, IEEE Concurrency, 6 (3), pp. 42–51.

Kang, O.H. and Agrawal, D.P. (2003), Scalable scheduling for symmetric multiprocessors (SMP), Journal of parallel and distributed computing, 63 (3), pp. 273-257.

Kartik S., and Murthy C.S.R. (1995), Improved task-allocation algorithms to maximize reliability of redundant distributed computing systems, IEEE Transactions on Reliability, 44 (4), pp. 575-586.

Kartik, S. and Murthy, C.S.R. (1997), Task allocation algorithms for maximizing reliability of distributed computing systems, IEEE Transactions on Computers, 46 (6), pp. 719–724.

Ke, W.J. and Wang, S.D. (1997), Reliability evaluation for distributed computing networks with imperfect nodes, IEEE Transactions on Reliability, 46 (3), 342-349.

Kim, D.W., Kim, K.H., Jang, W. and Chen, F.F. (2002), Unrelated parallel machine scheduling with setup times using simulated annealing, Robotics and Computer-Integrated Manufacturing, 18 (3-4), pp. 223-231.

Kim, D. and Yi, B.G. (1994), A two-pass scheduling algorithm for parallel programs, Parallel Computing, 20 (6), pp. 869-885.

Kirkpatrick, Jr.S., Gelatt, C. and Vecchi, M. (1983), Optimization by simulated annealing, Science, 220 (4598), pp. 498–516.

Kumar, A. and Agrawal, D.P. (1993), A generalized algorithm for evaluating distributed-program reliability, IEEE Transactions on Reliability, 42 (3), pp. 416-426.

Kumar, A., Rai, S. and Agarwal, D.P. (1988), On computer communication network reliability under program execution constraints, IEEE Journal of Selected Areas in Communications, 6 (8), pp. 1393-1400.

Kumar, A., Pathak, R.M. and Gupta, Y.P. (1995a), Genetic algorithm based approach for file allocation on distributed systems, Computers & Operations Research, 22 (1), pp. 41-54.

Kumar, A., Pathak R.M., Gupta, Y.P. and Parsaei, H.R. (1995b), A genetic algorithm for distributed system topology design, Computers & Industrial Engineering, 28 (3), pp. 659-670.

Kumar, V.K.P, Hariri, S. and Raghavendra, C.S. (1986), Distributed program reliability analysis, IEEE Transactions on Software Engineering, SE-12 (1), pp. 42-50.

Kwok, Y.-K. and Ahmad, I. (1996), Dynamic critical-path scheduling: an effective technique for allocating task graphs onto multiprocessors, IEEE Transactions on Parallel and Distributed Systems, 7 (5), pp. 506-521.

Kwok, Y.-K., and Ahmad, I. (1997), Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm, Journal of Parallel and Distributed Computing, 47 (1), pp. 58-77.

Kwok, Y.-K. and Ahmad, I. (1999a), Benchmarking and comparison of the task graph scheduling algorithms, Journal of Parallel and Distributed Computing, 59 (3), pp. 381-422.

Kwok, Y.-K. and Ahmad, I. (1999b), Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Surveys, 31 (4), pp. 406-471.

Lai, C.D., Xie, M., Poh, K.L., Dai, Y.S. and Yang, P. (2002), A model for availability analysis of distributed software/hardware systems, Information and Software Technology, 44 (6), pp. 343-350.

Laprie, J.C. and Kanoun, K. (1992), X-ware reliability and availability modelling, IEEE Transactions on Software Engineering, SE-18 (2), pp.130-147.

Leger, J.B., Iung, B., Beca, A.F.D. and Pinoteau, J. (1999), An innovative approach for new distributed maintenance system: application to hydro power plants of the REMAFEX project, Computers in Industry, 38 (2), pp. 131-148.

Levitin, G. (2002), Asymmetric weighted voting systems, Reliability Engineering and System Safety, 76 (2), pp. 205-212.

Li, Y. A. and Antonio, J. K. (1997), Estimating the execution time distribution for a task graph in a heterogeneous computing system, Proceedings of the 1997 Workshop on Heterogeneous Processing, Geneva, Switzerland, pp. 172-184.

Liepins, G.E., Hilliard, M.R., Richardson, J. and Palmer, M. (1990), Genetic algorithms application to set covering and travelling salesman problems, In Brown, D. E. and White, C.C. editors, Operations research and Artificial Intelligence: The integration of problem-solving strategies, pp. 29-57. Kluwer Academic, Norwell, Massachusetts.

Lin, M.S. (2003), Linear-time algorithms for computing the reliability of bipartite and (# <= 2) star distributed computing systems, Computers and Operations Research, 30 (11), pp. 1697-1712.

Lin, M.S., Chen, D.J. and Horng, M.S. (1999a), The reliability analysis of distributed computing systems with imperfect nodes, The Computer Journal, 42 (2), 129-141.

Lin, M.S., Chang, M.S. and Chen, D.J. (1999b), Efficient algorithms for reliability analysis of distributed computing systems, Information Sciences, 117 (1-2), pp. 89-106.

Lis, J. and Eiben, A.E. (1997), A multi-sexual genetic algorithm for multiobjective optimization, In Proc. IEEE Int. Conf. Evolutionary Computation, Indianopolis, IN, pp. 59–64.

Lutfiyya, H.L., Bauer, M.A., Marshall, A.D. and Stokes, D.K. (2000), Fault management in distributed systems: a policy-driven approach, Journal of Network and Systems Management, 8 (4), pp. 499-525.

Maheswaran, M. and Siegel, H.J. (1998), A dynamic matching and scheduling algorithm for heterogeneous computing systems, In Proceedings of Heterogeneous Computing Workshop, pp. 57-69.

Mahfoud, S. W. (1995), Niching Methods for Genetic Algorithms, Ph.D. dissertation, University of Illinois, Urbana-Champaign, 1995.

Mahmood, A. (2001), Task allocation algorithms for maximizing reliability of heterogeneous distributed computing systems, Control and Cybernetics, 30 (1), pp. 115-130.

Martin, E.W. and Millo, R.A.D. (1986), Operational survivability in gracefully degrading distributed processing systems, IEEE Transactions on Software Engineering, SE-12 (6), 693-704.

Mayer, D.G., Belward, J.A. and Burrage, K. (1998), Tabu search not an optimal choice for models of agricultural systems, Agricultural Systems, 58 (2), pp.243-251.

McCreary, C. and Gill, H. (1989), Automatic determination of grain size for efficient parallel processing, Communications of ACM, 32 (9), pp. 1073-1078.

Merwin, R.E., and Mirhakak, M. (1980), Derivation and use of a survivability criterion for DDP systems, In Proceedings of the National Computer Conference, 139-146.

Montgomery, D.C., and Runger, G.C. (2002), Design and analysis of single-factor experiment: the analysis of variance. Applied Statistics and Probability for Engineers, John Wiley & Sons, Inc, Third Edition.

Murthy, I. and Ghosh, D. (1993), File allocation involving worst case response times and link capacities: Model and solution procedure, European Journal of Operational Research, 67(3), pp. 418-427.

Nakaniwa, A., Onishi, M., Ebara, H. and Okada, H. (2001), Sensitivity analysis in optimal design for distributed file allocation systems, IEICE Transactions on Communications, E84B (6), pp. 1655-1663.

Oguz, C., Ercan, M.F., Cheng, T.C.E. and Fung, Y.F. (2003), Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop, European Journal of Operational Research, 149 (2), pp. 390-403.

Oh, J., and Wu, C. (2004), Genetic-algorithm-based real-time task scheduling with multiple goals, Journal of Systems and Software, 71 (3), pp. 245-258.

Osyczka, A. (1978), An approach to multicriterion optimization problems for engineering design, Computer Methods in Applied Mechanics and Engineering, 33, pp. 309-333.

Osyczka, A. (1984), Multicriterion Optimization in Engineering with FORTRAN programs, Ellis Horwood Limited.

Palis, M.A., Lieu, J.-C. and Wei, D.S.L. (1996), Task clustering and scheduling for distributed memory parallel architectures, IEEE Transactions on Parallel and Distributed Systems, 7 (1), pp. 46-55.

Papadimitriou, C.H. and Yannakakis, M. (1990), Towards an architecture-independent analysis of parallel algorithms, SIAM Journal of Computing, 19 (2), pp. 322-328.

Park, C.-I. and Choe, T.-Y. (2002), An optimal scheduling algorithm based on task duplication, IEEE Transactions on Computers, 51 (4), pp. 444–448.

Park, H.J. and Kim, B.K. (2002), An optimal scheduling algorithm for minimizing the computing period of cyclic synchronous tasks on multiprocessors, 51 (4), pp. 444-448.

Pathak, R.M., Kumar, A. and Gupta, Y.P. (1991), Reliability oriented allocation of files on distributed systems, In Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, pp. 886 -893.

Pham, H., Suprasad, A., and Misra, R.B. (1997), Availability and mean life time prediction of multistage degraded system with partial repairs, Reliability Engineering and System Safety, 56(2), pp. 169-173.

Pierre, S. and Elgibaoui, A. (1997), A Tabu-Search approach for designing computer-network topologies with unreliable components, IEEE Transactions on Reliability, 46 (3), pp. 350-359.

Plank, J.S. and Elwasif, W.R. (1997), Experimental assessment of workstation failure and their impact on checkpointing systems, International Symp. Fault-Tolerat Computing, pp. 48-57.

Radulescu, A. and Gemund, A.J.C. van (2002), Low-cost task scheduling for distributed-memory machines, IEEE Transactions on Parallel and Distributed Systems, 13 (6), pp. 648 – 658.

Raghavendra, C.S., Kumar, V.K.P. and Hariri, S. (1988), Reliability analysis in distributed systems, IEEE Transactions on Computers, 37 (3), pp. 352 –358.

Rao, S. (1986), Game theory approach for multiobjective structural optimization, Computers and Structures, 25 (1), pp.119-127.

Reeves, C.R. (1993), Improving the efficiency of tabu search in machine sequencing problems, Journal of the Operational Research Society, 44, pp. 375–382.

Richardson, J.T., Palmer, M.R., Liepins, G. and Hilliard, M. (1989), Some guidelines for genetic algorithms with penalty functions, In Schaffer, J.D. editor, Proceedings of the Third International Conference on Genetic Algorithms, pp. 199-197, George Mason University, Morgan Kaufmann Publishers.

Ritzel, B.J., Eheart, W., and Ranjithan, S. (1994), Using genetic algorithm to a solve a multiple objective groundwater pollution containment problem, Water Resources Research, 30 (5), pp. 1589-1603.

Rosenberg, R.S. (1967), Simulation of Genetic Populations with Biochemical Properties. PhD thesis, University of Michigan, Ann Harbor, Michigan.

Sait, S.M., and Youssef, H., (1999), VLSI Physical Design Automation: Theory and Practice, McGraw-Hill Book Co., Europe (also copublished by IEEE Press, New York) 1995 (reprinted with corrections by World Scientific 1999).

Sarkar, V. (1989), Partitioning and Scheduling Parallel Programs for Multiprocessors, MIT Press, Cambridge, Massachusetts.

Satyanarayana, A. (1982), A unified formula for analysis of some network reliability problems, IEEE Transactions on Reliability, R-31 (1), pp. 23-32.

Schaffer, D. (1985), Multiple objective optimization with vector evaluated genetic algorithms, In Genetic Algorithms and their Applications Proceedings of the First International Conference on Genetic Algorithms, pp. 93-100, Lawrence Erlbaum.

Sena, G.A., Megherbi, D. and Isern, G. (2001), Implementation of a parallel Genetic Algorithm on a cluster of workstations: Traveling Salesman Problem, a case study, Future Generation Computing Systems, 17 (4), pp. 477-488.

Shatz, S. M. and Wang, J.-P. (1989), Models & algorithms for reliability-oriented task-allocation in redundant distributed-computer systems, IEEE Transactions on Reliability, 38 (1), pp.16-21.

Shatz, S.M., Wang, J-P. and Goto, M. (1992), Task allocation for maximizing reliability of distributed computing systems, IEEE Transactions on Computers, 41 (9), pp. 1156-1168.

Shen, C.C., and Tsai, W.H. (1985), A graph matching approach to optimal task assignment in distributed computing systems using a mini-max criterion, IEEE Transactions on Computers, 34 (3), pp. 197-203.

Sih, G.C. and Lee, E.A. (1993), A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Transactions on Parallel and Distributed Systems, 4 (2), pp. 75-87.

Sols, A., and Nachlas, J.A. (1995), Availability of multifunctional systems. Reliability Engineering and System Safety, 47 (2), pp. 69-74.

Solich, R. (1969), Zadanie programowania liniowego z wieloma funkcjami celu (linear programming problem with several objective functions), Przeglad Statystyczny, 16, pp. 24-30 (In Polish).

Srinivas, N. and Deb, K. (1994), Multi-objective optimization using non-dominated sorting in genetic algorithms, Evolutionary Computation, 2 (3), 221–248.

Srinivasan, S. and Jha, N.K. (1999), Safety and reliability driven task allocation in distributed systems, IEEE Transactions on Parallel and Distributed Systems, 10 (3), pp. 238-251.

Stone, H.S. (1977), Multiprocessor Scheduling with the aid of network flow diagrams, IEEE Transactions on Software Engineering, 3 (1), pp. 85-93.

Stone, H.S. (1978), Critical load factors in two-processor distributed systems, IEEE Transactions on Software Engineering, 4 (3), pp. 254-258.

Stone, H.S. and Bokhari, S.H. (1978), Control of distributed processor, Computer, 11, pp. 97-106.

Subrata, R. and Zomaya, A.Y. (2003), A comparison of three artificial life techniques for reporting cell planning in mobile computing, IEEE Transactions on Parallel and Distributed Systems, 14 (2), pp. 142-153.

Sumita, U. and Masuda, Y. (1986), Analysis of software availability/reliability under the influence of hardware failures, IEEE Transactions on Software Engineering, SE-12(1), pp. 32-41.

Surry, P.D., Radcliffe, N.J. and Boyd, I.D. (1995), A Multi-Objective Approach to Constrained Optimization of Gas Supply Networks: The COMOGA Method, In Fogarty, T.C. editor, Evolutionary Computing. AISB Workshop. Selected Papers. Lecture Notes in Computer Science, pp.166-180, Springer-Verlag, Sheffield, U.K.

Tan, K. C., Lee, T. H. and Khor, E. F. (2002), Evolutionary algorithms for multi-objective optimization: performance assessments and comparisons, Artificial Intelligence Review, 17 (4), 251–290.

Tan, K.C., Khor, E.F., Lee, T.H. and Yang, Y.J. (2003), A tabu-based exploratory evolutionary algorithm for multi-objective optimization, Artificial Intelligence Review, 19 (3), pp. 231—260.

Tom, P.A. and Murthy, C.S.R. (1998), Algorithms for reliability-oriented module allocation in distributed computing systems, Journal of Systems and Software, 40 (2), pp. 125-138.

Tom, P.A. and Murthy C.S.R. (1999), Optimal task allocation in distributed systems by graph matching and state space search, Journal of Systems and Software, 46 (1), pp. 59-75.

Topcuoglu, H., Hariri, S. and Wu, M.-Y. (2002), Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems, 13 (3), pp. 260 –274.

Tripathi, A.K., Vidyarthi, D.P. and Mantri, A.N. (1996), A genetic task allocation algorithm for distributed computing systems incorporating problem specific knowledge, International Journal of High Speed Computing, 8 (4), pp. 363-370

Tseng, C.H. and Lu, T.W. (1990), Mini-max multi-objective optimization in structural design, International Journal for Numerical Methods in Engineering, 30, pp. 1213-1228.

Vidyarthi, D.P. and Tripathi, A.K. (2001), Maximizing reliability of distributed computing system with task allocation using simple genetic algorithm, Journal of Systems Architecture, 47 (6), pp. 549-554.

Wang, L.L. and Tsai, W.H. (1988), Optimal assignment of task modules with precedence for distributed processing by graph matching and state space search, BIT, 28 (1), 54-68.

Welke, S.R., Johnson, B.W. and Aylor, J.H. (1995), Reliability modelling of hardware/software systems, IEEE Transactions on Reliability, 44 (3), pp. 413-418.

Wong, A.K.Y. and Dillon, T.S. (2000), A fault tolerant model to attain reliability and high performance for distributed computing on the Internet, Computer Communication, 23 (18), pp. 1747-1762.

Woodside, C.M. and Monforton, G.G. (1993), Fast allocation of processes in distributed and parallel systems, IEEE Transactions on Parallel and Distributed Systems, 4 (2), pp. 164-174.

Wu, M.-Y. and Gajski, D. D. (1990), Hypercool: a programming aid for message-passing systems, IEEE Transaction on Parallel and Distributed Systems, 1 (3), pp. 330-343.

Yang, T. and Gerasoulis, A. (1994), DSC: scheduling parallel tasks on an unbounded number of processors, IEEE Transactions on Parallel and Distributed Systems, 5 (9), pp. 951-967.

Yeh, Y.S. and Chiu, C.C. (2001), A reversing traversal algorithm to predict deleting node for the optimal k-node set reliability with capacity constraint of distributed systems, Computer Communication, 24 (3-4), pp. 422-433.

Youssef, H., Sait, S.M. and Adiche, H. (2001), Evolutionary algorithms, simulated annealing and tabu search: a comparative study, Engineering Applications of Artificial Intelligence, 14 (2), pp. 167-181.

Zomaya, A.Y., and Teh Y.H. (2001), Observations on using genetic algorithms for dynamic load-balancing, IEEE Transactions on Parallel and Distributed Systems, 12 (9), pp. 899 –911.

Publications:

Liu, G.Q., Poh, K.L., Xie, M. (2005), Iterative List Scheduling for Heterogeneous Computing, Journal of Parallel and Distributed Computing, 65(5), pp. 654-665.

Liu, G.Q., Poh, K.L., Xie, M. (2005), Schedule Length and Reliability Oriented Multi-objective Scheduling for Distributed Computing, To appear in Proceedings of the 10th Annual International Conference on Industrial Engineering Theory, Applications & Practice, Clearwater, Florida, USA.

Liu, G.Q., Xie, M., Dai, Y.S., Poh, K.L. (2004), On Program and File Assignment for Distributed Systems, Computer Systems Science & Engineering, 19 (1), pp. 39-48.

Liu, G.Q., Poh, K.L., Xie, M. (2003), A Parallel Tabu Search for Reliability Based Program and File Allocation, In Proceedings of the 8th Annual International Conference on Industrial Engineering Theory, Applications & Practice, Las Vegas, Nevada, USA.

Dai, Y.S., Xie, M., Poh, K.L., Liu, G.Q. (2003), A study of service reliability and availability for distributed systems, Reliability Engineering & System Safety, 79 (1), pp. 103 -112.