

This item was submitted to [Loughborough's Research Repository](#) by the author.
Items in Figshare are protected by copyright, with all rights reserved, unless otherwise indicated.

Parallel scheduling of concurrent VLSI simulation modules onto a multiprocessor

PLEASE CITE THE PUBLISHED VERSION

PUBLISHER

© M.A. Rahin

PUBLISHER STATEMENT

This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at:
<http://creativecommons.org/licenses/by-nc-nd/2.5/>

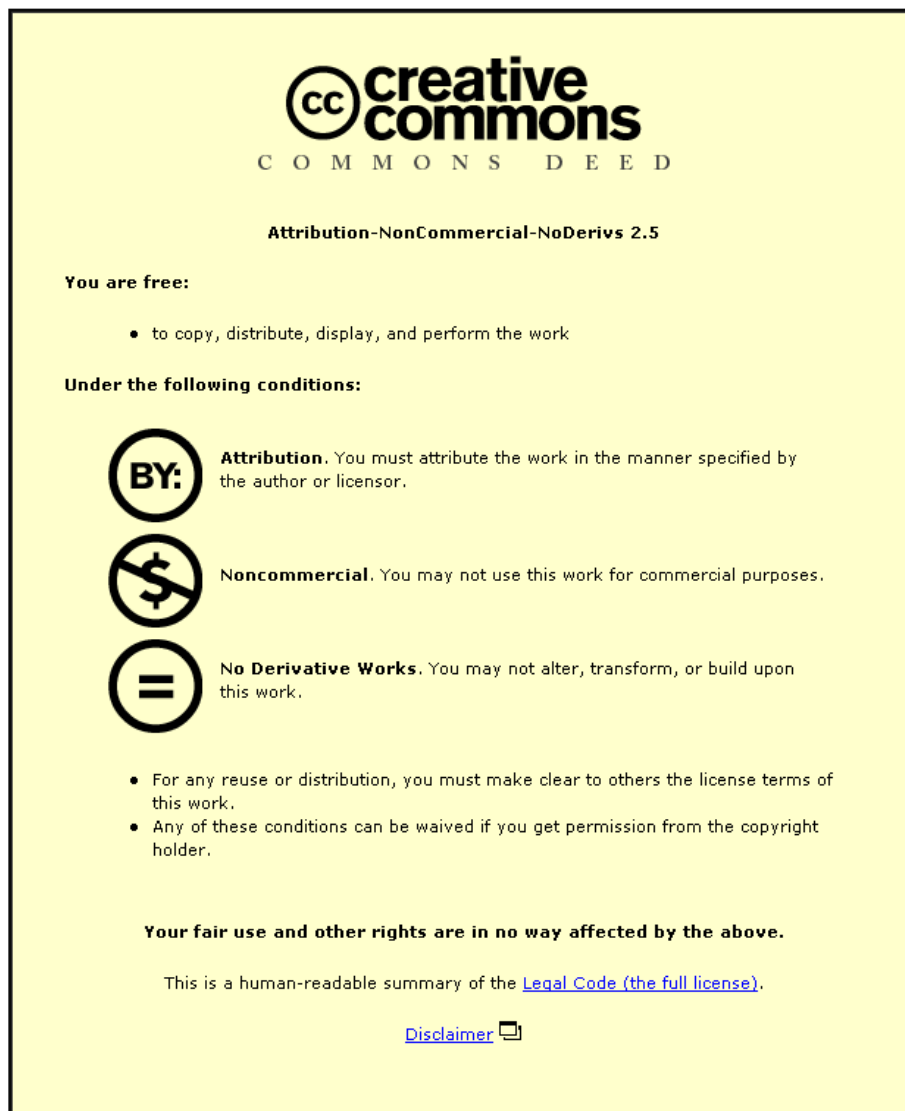
LICENCE

CC BY-NC-ND 2.5

REPOSITORY RECORD

Rahin, Mohammad A.. 2019. "Parallel Scheduling of Concurrent VLSI Simulation Modules onto a Multiprocessor". figshare. <https://hdl.handle.net/2134/27133>.

This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (<https://dspace.lboro.ac.uk/>) under the following Creative Commons Licence conditions.



For the full text of this licence, please go to:
<http://creativecommons.org/licenses/by-nc-nd/2.5/>

BLDSC no:- DX 98140

LOUGHBOROUGH
UNIVERSITY OF TECHNOLOGY
LIBRARY

AUTHOR/FILING TITLE

RAHIN, M A

ACCESSION/COPY NO.

036000250

VOL. NO.

CLASS MARK

15 FEB 1995

Loan copy

036000250 1



BADMINTON PRESS
18 THE HALECROFT
SYSTEM
LEICESTER LE7 8LD
ENGLAND
TEL 0533 602918

PARALLEL SCHEDULING OF CONCURRENT VLSI SIMULATION MODULES ONTO A MULTIPROCESSOR

by

MOHAMMAD A. RAHIN

*A Doctoral Thesis submitted in partial fulfilment of the
requirements for the award of Doctor of Philosophy
of
the Loughborough University of Technology*

July 1991

**Supervisor : Dr. J. Sheild
Department of Electronic & Electrical Engineering**

©by M.A. Rahin, 1991

Loughborough University of Technology Library	
Date	Mar 92
Class	
Acc No.	036000250

W9919443

University of Technology

LOUGHBOROUGH LEICESTERSHIRE LE11 3TU Telephone: 0509 263171 Ext. 4006 Telex: 94919

THE STUDENT OFFICE (HIGHER AWARDS)

CERTIFICATE OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements or in footnotes, and that neither the thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

Mohammad A. Rahin

25 July, 1991.

to my parents

ABSTRACT

This thesis reports on the research into multiprocessor based task scheduling algorithms as applied to the assignment of VLSI simulation modules onto a multiprocessor. Task scheduling falls into the category of combinatorial optimisation problems and is known to be NP-Hard. The goal of this research is to implement parallel heuristic scheduling algorithms for a general purpose multiprocessor system and to evaluate through simulation their relative performances.

The multiprocessor task scheduling problem in the context of VLSI simulation is presented first followed by a taxonomy of general scheduling algorithms. The factors determining the quality of a schedule are identified and an objective function that guides the heuristic algorithms is then formulated.

The first of the two algorithms examined is the Concurrent Recursive Binary Partitioning (CRBP). This heuristic is based on Kernighan-Lin's graph bi-partitioning algorithm. A l -way partition is achieved by applying binary partitioning recursively, the procedure taking the form of a binary tree. In its parallel implementation each node of this tree is executed independently by a group of available processors and only the best among the solutions obtained is accepted. This provides enhanced processor utilisation and also assures improved results. The factors affecting the performance of the l -way partitioning heuristic at different stages are examined and their optimum values are investigated.

The other parallel algorithm examined is the Concurrent Simulated Annealing (CSA). Simulated annealing is a powerful and robust tool for the solution of many difficult combinatorial optimisation problems. In spite of its ability to produce good quality solutions, it is beset by exceptionally long CPU time demand. The parallel implementation presented here is an attempt to speed-up its convergence time and works with an optimal number of non-interacting parallel moves thereby assuring minimal error due to interaction between parallel moves. Two simple but effective temperature schedules are used. These temperature schedules are in a way dependent on the problem instance and as such adapt themselves to the varying needs of the input problem instances.

Both the parallel heuristics are subjected to synthetic as well as some actual VLSI simulation data instances. It is found that CRBP has an overall edge in terms of speed of execution and performs well for small number of processors. However, its solution quality deteriorates considerably with the increase of processors. CSA on the other hand, performs uniformly throughout and favours well for larger systems. For larger systems, CRBP has the potential to be used as a pre-processor for a combined CRBP-CSA heuristic.

ACKNOWLEDGEMENTS

The author wishes to express his sincere thanks and gratitude to his supervisor Dr. J. Sheild for his continuous guidance and unending encouragement throughout the course of the research work. The author is also indebted to Dr. S. Datta for his moral and material support especially during the preparation of this thesis.

The financial support of the Association of Commonwealth Universities under whose Commonwealth Scholarships Programme the author was sponsored is gratefully acknowledged. The helpful and sympathetic administration of the above scholarship by The British Council is greatly appreciated.

Last, but not the least, a special heartfelt thanks goes to all the friends and acquaintances whose help, direct or indirect and company certainly made the author's time spent in Loughborough an enjoyable one.

CONTENTS

	page
ABSTRACT	iii
ACKNOWLEDGEMENTS	v
CONTENTS	vi
CHAPTER 1: Introduction	1
1.1 Multiprocessors & Multicomputers	2
1.1.1 Speed-up in a Multiprocessor	6
1.1.2 Factors Determining the Performance of a Multiprocessor ..	8
1.2 Application of Multiprocessors in VLSI Design	10
1.2.1 VLSI Design Processes	10
1.2.2 Acceleration of VLSI Design Process	12
1.2.3 VLSI Circuit Simulation	14
1.3 Multiprocessor Task Scheduling	17

1.4 Outline of the Dissertation	18
References	20
CHAPTER 2: The Task Scheduling Problem	22
2.1 Complexity of Task Scheduling Problem	23
2.1.1 Parallel Algorithms	25
2.2 Classification of Task Scheduling Algorithms	26
2.2.1 Static vs. Dynamic Scheduling	28
2.2.2 Optimal vs. Sub-optimal scheduling	29
2.2.3 Approximate vs. Heuristic Solutions	29
2.2.4 Load Balancing	30
2.3 Optimal and Sub-optimal Approximate Techniques	31
2.3.1 Solution Space Enumeration and Search	31
2.3.2 Graph Theoretic	31
2.3.3 Mathematical Programming	31
2.3.4 Queing Theoretic	32
2.4 Heuristic Technique	32
2.4.1 Constructive Method	32
2.4.2 Iterative Improvement Method	33
2.5 Local Minima and Optimal Solution	35
2.6 Contribution of this Dissertation	37
References	40
CHAPTER 3: The Graph Model	43
3.1 Graphical Representation: Background	44
3.2 Concurrent VLSI Timing Simulation	46
3.2.1 VLSI Circuit Partitioning Algorithm	50
3.3 The Graph Model	50
3.4 Communication and Computation	53
3.5 Formulation of the Cost Function	55
3.6 Graph Data Storage & Cost Calculation	59
3.6.1 Graph Data Storage	62
3.6.2 Cost Calculation	65
3.6.3 Cost Calculation in Iterative Improvement Environment ...	65
References	69

CHAPTER 4: Graph Partitioning	70
4.1 Graph Bi-Partitioning	71
4.1.1 The Modified KL Bi-Partitioning Heuristic	73
4.1.2 The Bi-Partitioning Algorithm in Action	85
4.2 Multiple-way Partitioning	86
4.2.1 Recursive Binary Partitioning	89
4.2.2 Concurrent Recursive Binary Partitioning	95
4.2.3 Performance of Recursive Binary Partitioning	99
References	101
CHAPTER 5: The Simulated Annealing Algorithm	102
5.1 Combinatorial Optimisation and Simulated Annealing	103
5.1.1 The Simulated Annealing Algorithm	103
5.2 Markov Chain Model of SA	108
5.3 Cooling Schedule	113
5.3.1 A Simple Cooling Schedule	115
5.3.2 A Polynomial-Time Cooling Schedule	116
5.4 Implementation of the SA Algorithm	122
5.4.1 Concise Problem Representation	122
5.4.2 Transition Mechanism	122
5.4.3 The Cooling Schedule	123
5.4.4 Performance Analysis of the SA Algorithm	129
References	145
CHAPTER 6: Concurrent Simulated Annealing	147
6.1 Speeding-up the SA Algorithm	148
6.1.1 Fast Sequential Algorithm	148
6.1.2 Hardware Accelerators	149
6.1.3 Design of Parallel Algorithms	149
6.2 Parallel Annealing Algorithms	151
6.3 Concurrent Simulated Annealing	157
6.3.1 Parallel Moves and Move Interactions	157
6.3.2 CSA Parallel Move Algorithm	159
6.3.3 Parallel Accept/Reject Decisions	168
6.3.4 CSA Implementation Models	171
6.4 Simulation Results	177

CONTENTS

References	179
CHAPTER 7: Conclusions & Discussion	181
7.1 Review of Results	182
7.2 Recursive Binary Partitioning & Simulated Annealing	190
7.3 Conclusions	192
7.4 Discussion	195
References	199
APPENDIX A	200
APPENDIX B	211

CHAPTER 1

Introduction

Ever increasing processing demands as encountered in different fields such as image processing, artificial intelligence, finite element analysis problems, weather forecasting, wind tunnel simulations, nuclear system, particle physics etc. have led the researchers to the development of several conventional high performance computers as well as conceptually new architectures namely non von Neumann systems and most importantly multiprocessor systems.

It has been identified quite long ago that the conventional uniprocessor computers are unable to meet the performance requirements of many computing intensive applications. Since, the first electronic digital computer (ENIAC) was built in 1945, the advances in uniprocessor computers can be attributed primarily to development of logic technology. Switching speed fell from one tenth of a second to nano seconds as the logic technology moved from electro-mechanical relays to vacuum tubes to transistors and

then to small, medium and large scale integrated circuits. In the last few years, it has become more difficult to achieve order of magnitude of speed-ups in computers by solely upgrading the logic technology, as the physical laws have been found to be the primary limiting factor. Radically different computer architectures namely *Data-Flow Machines* [1] have been proposed to overcome the bottleneck associated with the conventional *von Neumann* computers, but failed to attract wide spread acceptance as initially anticipated. The most obvious solution to the problem to date seems to lie in the exploitation of parallelism in the applications and executing the mutually independent task modules concurrently on a multiprocessor system. MIMD multiprocessors with multiple instruction-streams and multiple data-streams promise to be the general purpose computers of the future. Several commercial MIMD computers have already arrived in the market, e.g. those manufactured by Alliant, BBN, Cray, ELXSI, Encore, IBM, Intel, NCUBE, Sequent etc.

1.1 Multiprocessors and Multicomputers

Modern computer architectures can be classified in many different ways. One of the most popular higher level architecture wise classification is due to Flynn [2]. Flynn based his taxonomy of computer architectures on the concepts of instruction stream and data stream. An instruction stream is a sequence of instructions performed by a computer; a data stream is a sequence of data used to execute on instruction stream. Flynn categorised an architecture by the multiplicity of hardware used to manipulate instruction and data streams. Given the possible multiplicity of instruction and data streams, four classes of computers result.

1. SISD (*Single Instruction stream, Single Data stream*) : Almost all single processor computers fall into this category. Although instruction execution may be pipelined, computers in this category can decode only a single instruction in unit time. A SISD computer may also have multiple functional units (e.g. CDC 6600) being governed under the direction of a single control unit.

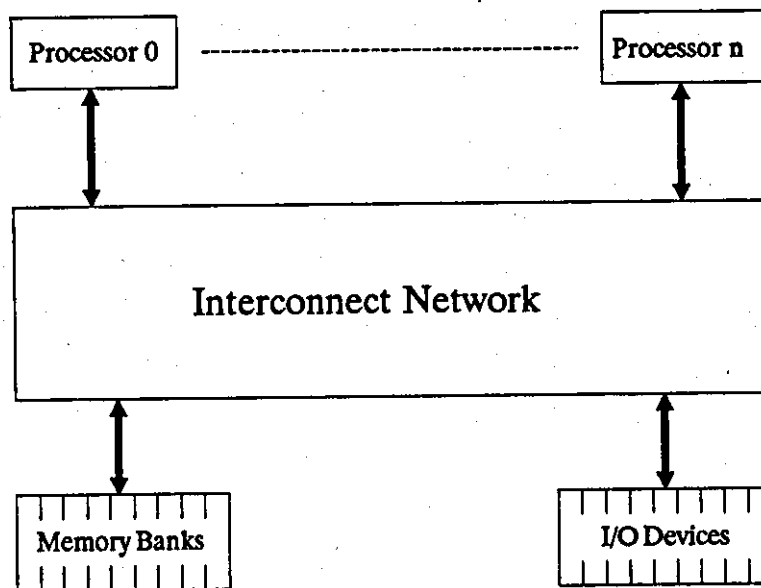


Fig. 1.1 A shared memory multiprocessor.

2. SIMD (*Single Instruction stream, Multiple Data stream*) : Processor arrays fall into this category. A processor array executes a single stream of instruction, but contains a number of arithmetic processing units, each capable of fetching and manipulating its own data. Hence, in any time unit, a single operation is in the same state of execution on multiple processing units, each manipulating different data.
3. MISD (*Multiple Instruction stream, Single Data stream*) : No computers fall into this category.
4. MIMD (*Multiple Instruction stream, Multiple Data stream*) : This is by far the most popular architecture for multiprocessors and well suited for future general purpose computing needs. MIMD computers are composed of a collection of full featured processing elements often with their own local or private memory and are capable of executing their own instructions independent of each other. The term MIMD is generally reserved for multiple CPU computers designed for parallel processing; that is, computers designed to allow interaction among their CPUs.

Flynn's classification scheme has been found to be too vague to allow a clear cut labeling of modern high performance computers. Kuck [3] enhanced Flynn's classification scheme into a more detailed form. Händler's [4] classification however accounts for the organisation of the main functional units of computers and uses some notations for expressing the pipelining and parallelism. MIMD designs can be further classified into two major groups [5]:

1. *Shared Memory Multiprocessors* : Multiprocessors are characterised by a shared memory. Shared memory multiprocessors can be further classified into two, (a) *Tightly Coupled* and (b) *Loosely Coupled*. In the case of the tightly coupled multiprocessors, the simplest processor inter-communication pattern assumes that all the processors work through a central switching mechanism to reach a shared global memory (Fig.1.1). There are a variety of ways to implement this switching mechanism, including a common bus to global memory, a crossbar switch and a packet-switched network. Examples include Carnegie-Mellon's C.mmp, Denelcor's HEP, Sequent's Balance and Symmetry.

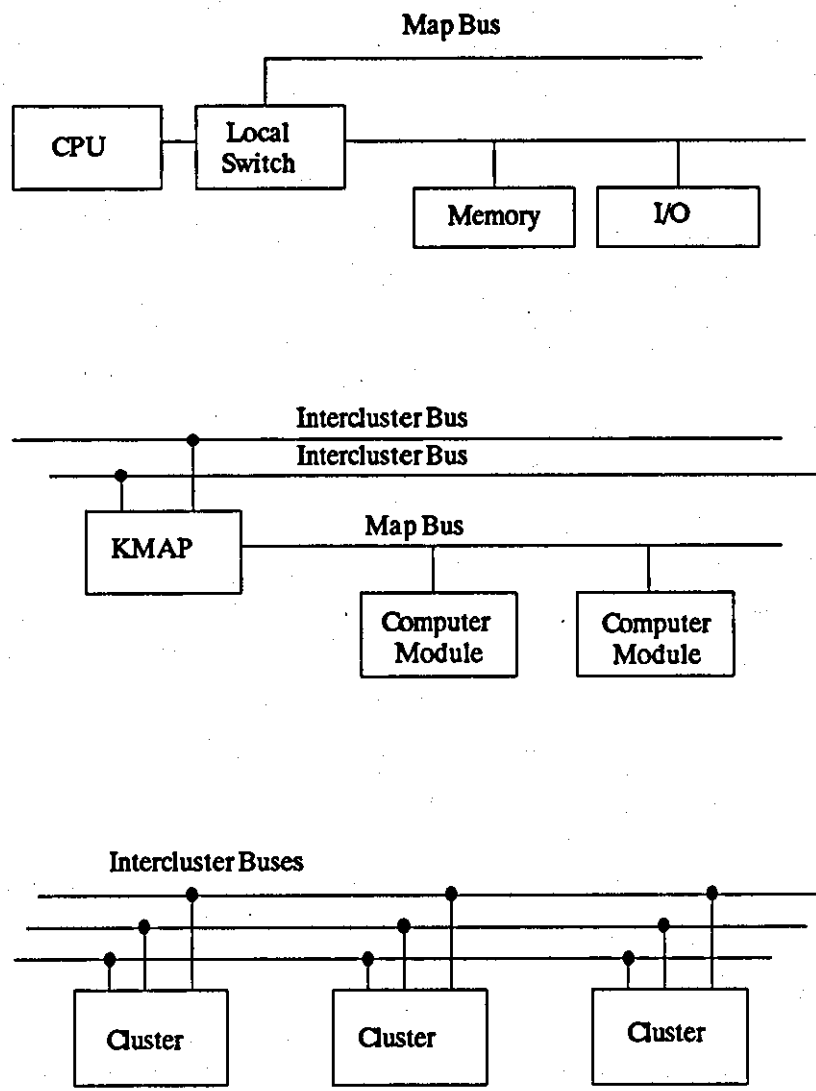


Fig. 1.2 The architecture of CM*, a shared memeory loosely coupled multiprocessor. Computer module (top), cluster (middle) and network of clusters (bottom).

Like tightly coupled multiprocessors, loosely coupled multiprocessors are also characterised by a shared address space. Unlike tightly coupled multiprocessors, the shared space on a loosely coupled multiprocessor is formed by combining the local memories of the CPUs. Hence, the time needed to access a particular memory on a loosely coupled multiprocessor depends on whether that location is local to the processor. Examples include Carnegie-Mellon's Cm*, BBN Butterfly.

2. *Message Passing Multicomputers*: Figure 1.3 shows the general structure of a message passing multicomputer. Each processor has its own local memory, and process cooperation occurs either through sending or receiving messages. The performance and scalability of the multicomputer is primarily determined by the interconnection network. The simplest interconnection network for a multicomputer is a bus capable of handling interprocessor messages (i.e. a local area network). Distributed systems have this structure and can indeed be used as a multicomputer for applications with sufficiently large granularity. Intel's iPSC, NCUBE's NCUBE/10, Ametek's S/14 are commercial multicomputers.

1.1.1 Speed-up in a Multiprocessor

The speed-up that can be achieved by a multiprocessor with n identical processors working concurrently on a single problem is at most n times faster than a single processor running the same program. In practice, the speed-up is much less, since some processors are idle at a given time because of conflicts over memory access or communication paths, inefficient algorithm that fails to exploit the natural concurrency in the problem etc. Fig.1.4 shows the various estimates of the actual speed-up, where speed-up is defined as the ratio between the time taken by that parallel computer executing the fastest serial algorithm and the time taken by the same parallel computer executing the parallel algorithm using n processors.

The lower bound in Fig.1.4 is known as the *Minsky's Conjecture*, giving a rather pessimistic view. The upper bound $\left(\frac{n}{\log n}\right)$ can be derived using simple assumptions and approximations [6].

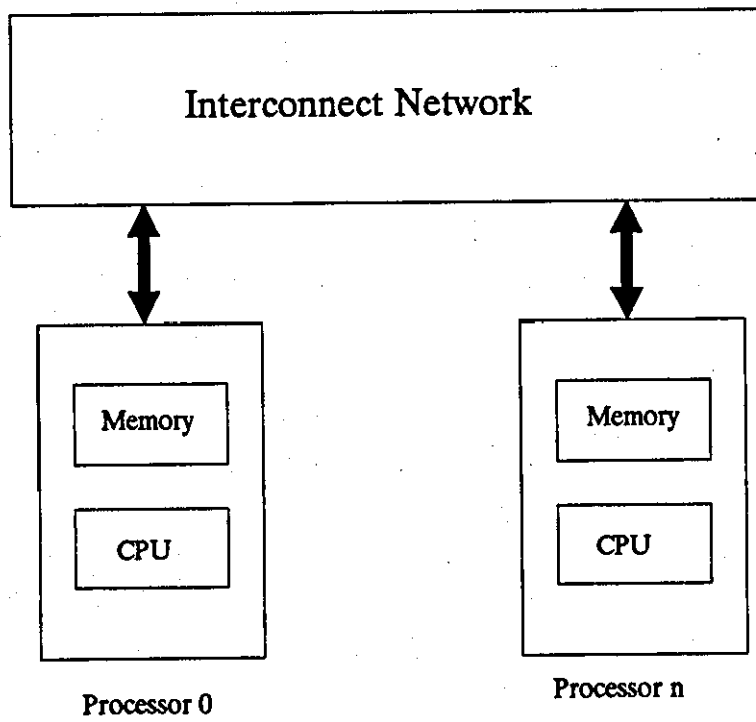


Fig. 1.3 A message passing multicomputer.

1.1.2 Factors Determining the Performance of a Multiprocessor

The followings are the factors considered to be associated with the performance capability of a multiprocessor system :

1. **Granularity** : An important issue in multiprocessor performance is the granularity of program execution. The granularity of a parallel program can be defined as the average size of a sequential unit of computation in the program, with no interprocessor synchronisation or communication (e.g. the average task size). For a given multiprocessor, there is a minimum program granularity value below which performance degrades significantly. This threshold value can be termed granularity of the multiprocessor. It is desirable for a multiprocessor to have a small granularity, so that it can efficiently support a wide range of programs. It is also desirable for a parallel program to have a large granularity, so that it can be executed efficiently on a wide range of multiprocessors.
2. **Scalability** : Scalability is another important property of multiprocessors. Scalability is the ability of a multiprocessor to provide a linear speed-up with an increase in the number of processors, assuming that the program being executed has sufficient parallelism and a large enough granularity. A multiprocessor architecture is usually designed to be scalable up to some specific number of processors. There is a fine balance between granularity and scalability in a multiprocessor. Increased scalability is typically achieved at the cost of large granularity. On a general note, tightly coupled systems usually have a smaller granularity and scalability than loosely coupled systems.
3. **Computation vs. Communication** : The introduction of parallelism has led to problems of communication which did not exist in uniprocessor systems. In the multiprocessor environment, one would expect to split the computation and assign the task modules to different processors. Consequently, the processors need to communicate the results of their computations. The more complex the task division, the more complex will be the communication pattern. One has to be more careful not to degrade the system performance by overwhelming the communication.

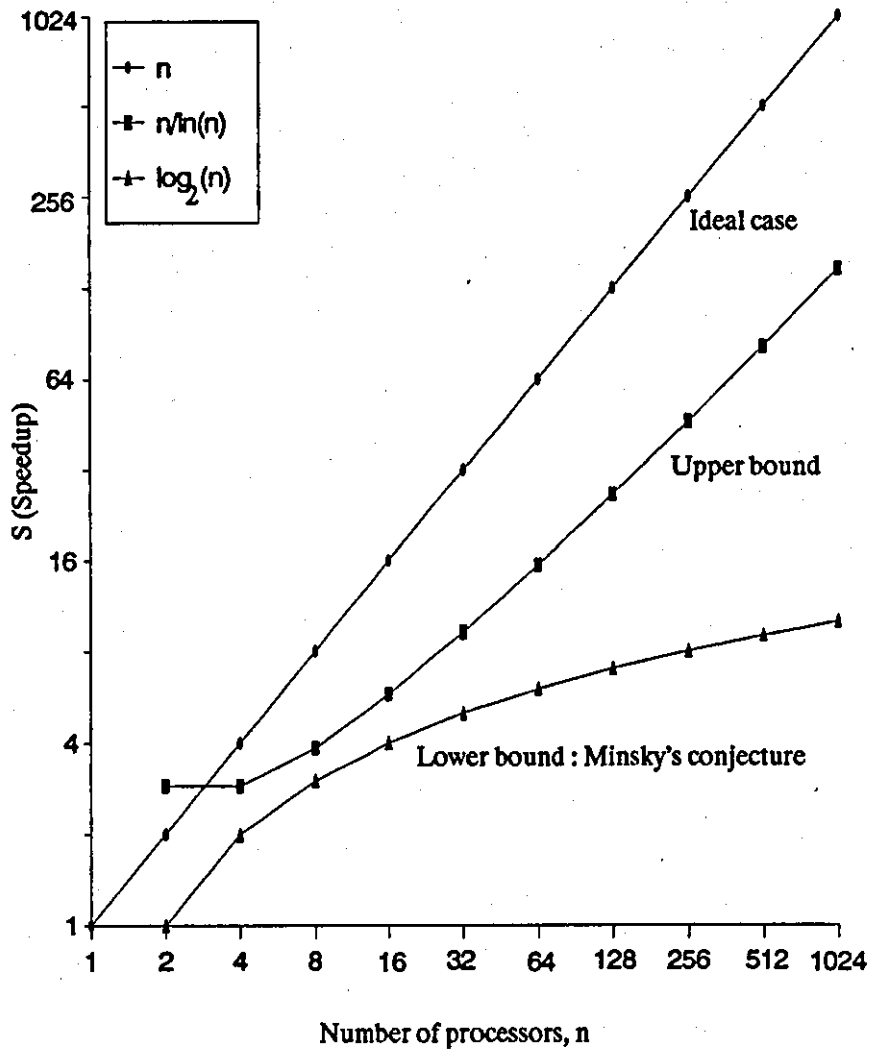


Fig. 1.4 Various estimates of the speedup of an n -processor system over a single processor.

There is a trade-off between communication and computation in parallel systems. Ideally, one would like to reduce the communication overhead and increase the parallelism by further splitting of the computation. It is well established that the communication time in a multiprocessor environment can not be neglected. Agerwala [7] has studied the relationship between communication and computation and concluded that the reason for decreased performance of present multiprocessor systems is insufficient emphasis on the role of communication.

1.2 Application of Multiprocessors in VLSI Design

The rapid development of multiprocessors as well as the fall of their prices has opened up many new engineering application areas which have been suffering from heavy computing demand. One of these new application areas is VLSI engineering. In recent years, due to the availability of advanced semiconductor process technologies as well as computer aided design (CAD) systems coupled with ever increasing drive for higher integration, a tremendous growth in chip density is being observed. Already VLSI chips containing more than 1 million transistors has been announced [8]. This very high chip density is forcing an unacceptably long design turn around time. The size and complexity of the design are the main factors responsible for this. Different phases of VLSI design is bestowed with rich inherent parallelism. These can be effectively exploited and their multiprocessor implementation would then provide a much faster design and verification turn around time.

1.2.1 VLSI Design Processes

Design methodology in the context of VLSI circuits can be defined as a set of codified techniques that is applicable to the VLSI design process. Design functions of interest in the VLSI design methodology can be categorised as follows [9] :

1. Chip specification and partitioning;
2. Chip design planning and initial implementation;
3. Subcircuit and module synthesis;
4. Simulation at different levels;

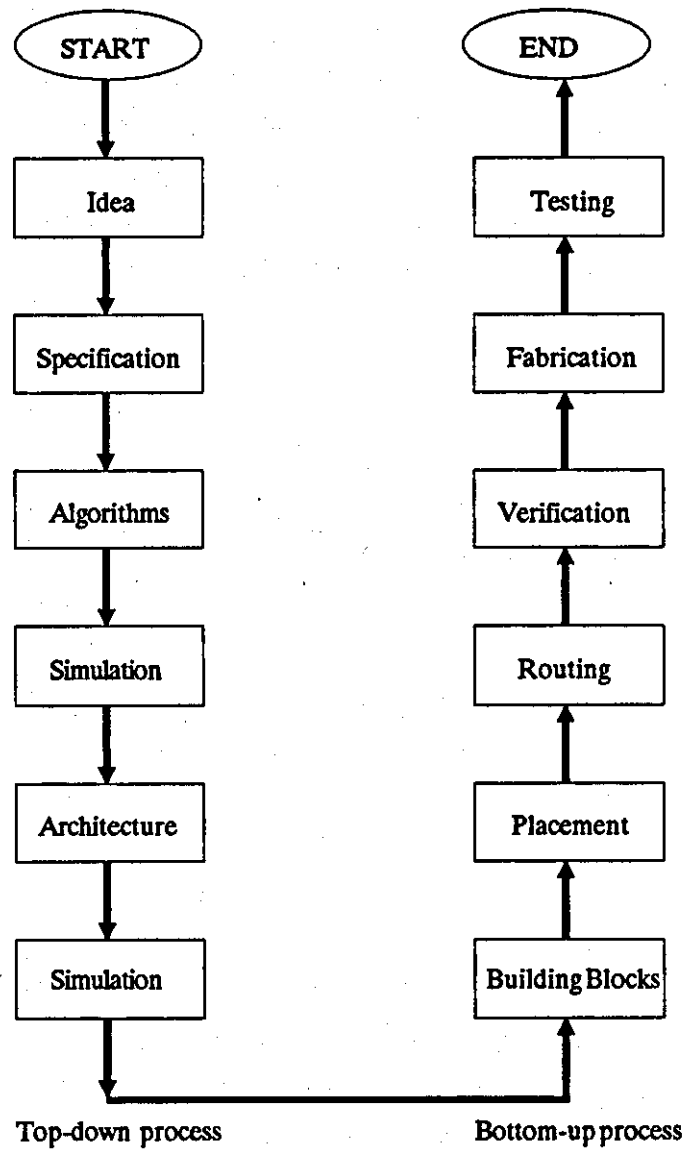


Fig. 1.5 The general design processes of a VLSI circuit.

5. IC mask layout;
6. Design verification;
7. Testability in design and product;
8. Test sequence generation;
9. Database management;
10. Design documentation.

The ultimate objective of studying design methodology is to facilitate the creation of better designs in less time. The prevalent design methodology today, and in the foreseeable future, is hierarchical in nature [10], where a set of universal circuits are designed, optimised, and stored in the library of a CAD system and the library circuits can be repeatedly accessed, modified, and used as building blocks to construct the desired system.

Figure 1.5 presents the overview of a typical VLSI design process. A top-down design flow is normally used to decompose the circuit under design into a network of smaller and simpler functional modules. Once a function implementation strategy has been established, a bottom-up flow is used to complete the physical design of the chip.

The design process begins by converting an idea for a VLSI circuit into more concrete circuit specifications. An algorithm is developed to perform the required operations and a suitable architecture is designed to carry out the chosen algorithm. Simulations are used to verify the correctness and to estimate the performance of both the algorithm and architecture. After the architecture of the circuit has been established, the design process enters the physical layout phase. Copies of the needed building blocks are fetched from the library, placed in some optimal manner, and interconnected as they will be on the chip. The layout is then simulated to verify its operation and performance with respect to the desired specifications. Finally layout mask is sent to a silicon foundry for fabrication.

1.2.2 Acceleration of VLSI Design Process

The simulation and verification phases of the design process of Fig.1.5 are composed of logic, timing, electrical and fault simulation and also *Design Rule Checking* (DRC) encompassing physical, thermal and connectivity

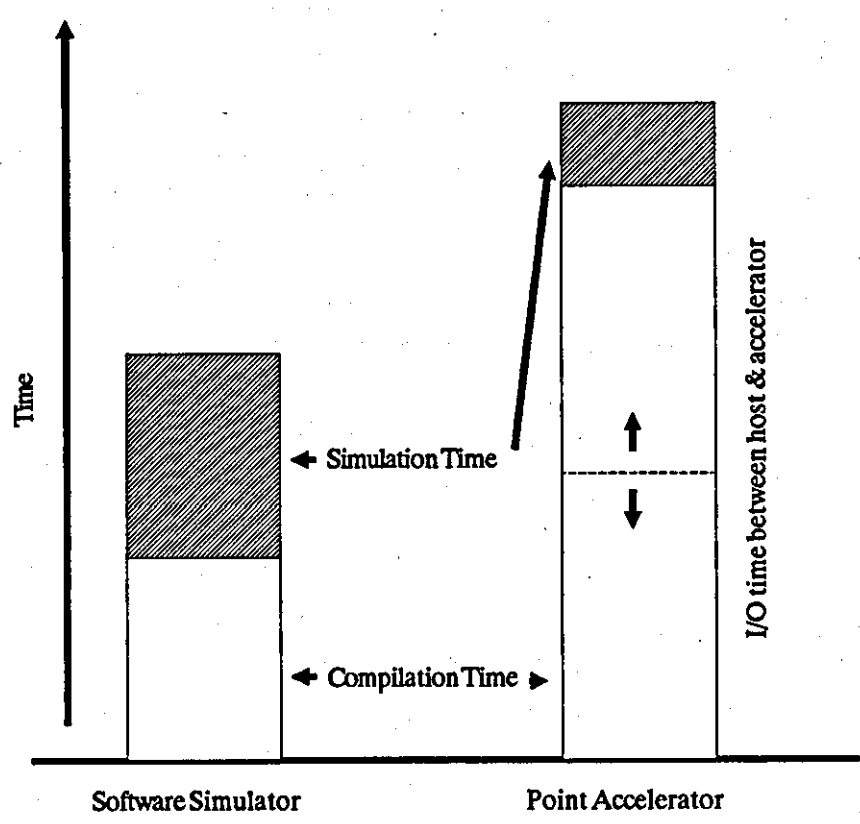


Fig. 1.6 I/O overhead in a typical point accelerator. Adapted from [12].

analysis. In league with the *Placement* and *Routing* phases, simulation is one of the most time consuming phases of a VLSI design. Placement and routing are difficult combinatorial optimisation problems and are known to be NP-Hard [11]. The simulation phase also have voracious appetite for CPU time often taking $O(n^2)$ to $O(n^3)$ time.

Several attempts have been made to speed-up the VLSI design process [11–19]. Two different approaches can be observed. The first approach is based on the use of dedicated special-purpose hardware device known as *Point Accelerators* [13–17]. These are usually single processor based special-purpose unit where the algorithm is realised in hardware and is attached to a design workstation. They provide very impressive speed-up factors at low cost some times upto a factor of 225 for relatively larger circuits [16]. Their main drawback however is that for smaller circuits the I/O bandwidth with the host workstation is too large to give any significant speed improvement (Fig.1.6). Also, as the algorithm is tied to the hardware, modification and upgrading becomes very difficult and expensive. Furthermore, they fare very poorly in integrating with the other phases of the VLSI design process.

The other approach is based on the exploitation of rich inherent parallelism present in the design phases and employing general-purpose multi-processor systems [11,12,18,19]. This approach is becoming more attractive with the advancement of computer technology as well as with their falling prices. The design algorithm is almost fully software based. Modification and upgrading can thus be done very easily and inexpensively. Also, the same hardware can be used for the rest phases of the design.

1.2.3 VLSI Circuit Simulation

Circuit simulation is one of the most important phases of a VLSI design process. A thorough and detailed analysis of the tentative design analysis is absolutely necessary before a very expensive fabrication begins. Design verification of a custom VLSI circuit takes place at a number of different levels of simulation as the design progresses. These range from *logic simulation* (using register-transfer level, gate level, or switch level simulation) to *electrical simulation*.

Electrical simulation usually provides a detailed and accurate analysis of a circuit. But is often too slow for simulation of large circuits. Circuit simulation using direct solutions, as in SPICE [20], is typically a factor of 10^4 to 10^5 slower than gate level logic simulation [21]. Several techniques for improving the speed of simulation have been proposed [17]. These include the exploitation of circuit latency by integrating only those circuit nodes which are changing voltages, using tearing techniques for matrix manipulations, deriving relaxation-based simulators that eliminate matrix decomposition. These improvements can give a speed-up factor of upto 10^2 , but is still inadequate for large circuits.

Switch level simulation, exemplified by RSim [22] and MOSSIM [23], uses event driven techniques and a logic level approximation of the network state. This results a very fast simulation compared to direct electrical simulation. Although the approximations are valid for many circuits, they often fail to produce correct results for many others. Another disadvantage of switch level simulation is that it typically provides only unit-delay timing information, or timing information that can have large errors for common circuits.

Timing simulation is a compromise between direct electrical simulation and switch level simulation. Compared to direct simulation this provides much higher performance at the cost of slightly lower accuracy. Also compared to switch level simulation, timing simulation provides better accuracy with limited analog effects such as rise and fall times, charge and current sharing and feedback. EMU [24] and MOTIS [25] identifies the regular use of driver-load gate structures and uses an approximate series-parallel formulae. Speed improvement techniques like discretisation of voltage into a small number of levels, and the calculation of time step for each node. Timing simulation claims improvements of upto 10^3 compared to direct electrical circuit simulation.

The accuracy of timing simulators can be improved further by decreasing the size of the time step. But, unfortunately, this increases the simulation time and for relatively larger circuits this may become very long. Table 1.1 shows the simulation time for four representative circuits ranging from

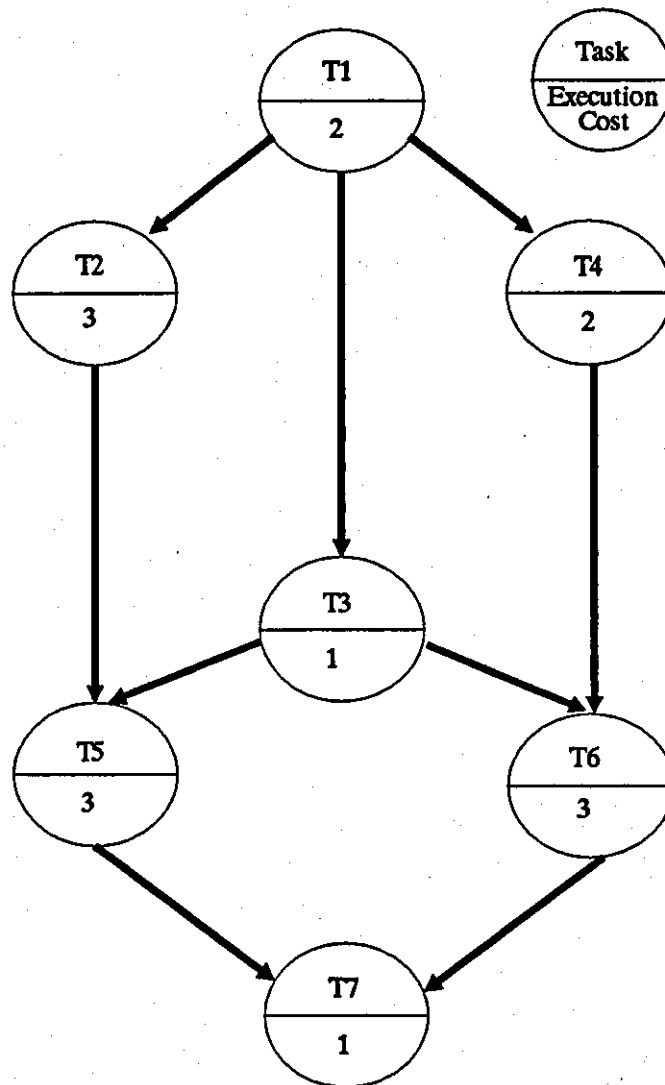


Fig. 1.7 A Computational Flow Graph (CFG), each node represents a sub-task to be performed.

small to medium sizes for different time step when simulated with EMU on a conventional uniprocessor system.

Table 1.1 Simulation time with EMU for different time steps.

Time Steps in pico Seconds	4x4 Multiplier (168 Nodes)	Phased Locked Loop (167 Nodes)	16x16 Multiplier (2577 Nodes)	Vector Coder (1746 Nodes)
100	11.00 Mins.	18.78 Mins.	10.12 Hrs.	10.96 Hrs.
200	8.10 Mins.	15.78 Mins.	7.32 Hrs.	10.00 Hrs.
500	6.93 Mins.	12.77 Mins.	5.84 Hrs.	10.11 Hrs.
1000	6.70 Mins.	12.62 Mins.	5.94 Hrs.	10.13 Hrs.

Further improvements in simulation time can be achieved by exploiting the inherent parallelism of a circuit and then running the simulation on a multiprocessor. CEMU [24] is a step in that direction. In CEMU the circuit under investigation is partitioned into a number of regions so that each region can be simulated independent of each other. Each such region which we shall call *simulation modules*, is composed of a number of voltage controlled current sources (e.g. a transistor) connected together to a capacitive node. The partitioned circuit can then be run on a multiprocessor by assigning each region or module to each of the processors. However, the performance of the multiprocessor implementation of the timing simulation is heavily dependent on the quality of region or module assignment onto the multiprocessor. An optimal assignment or scheduling guarantees higher throughput. However, the process of optimal scheduling or assignment is itself a difficult combinatorial optimisation problem for which no polynomial time optimal algorithm is known to exist [26]. In this dissertation the main theme will be the study of heuristic parallel multiprocessor scheduling algorithms in the context of VLSI timing simulation.

1.3 Multiprocessor Task Scheduling

The multiprocessor task scheduling problem can be defined as the process of allocating task modules to processors. The goal is to minimise the

parallel execution time. The parallel execution time depends on processor utilisation and on the overhead of interprocessor communication.

List scheduling algorithms [27] have been found to be quite successful for general task scheduling problems when communication overhead is ignored. They have linear time characteristics with a constant performance bound of 2 [28], meaning that the maximum execution time for a particular schedule generated by the list scheduling algorithm is twice that of the optimal parallel execution time.

Unfortunately, the scheduling problems become more difficult when communication overhead with arbitrary data sizes are considered. Even quite simple instances of the scheduling problems are often intractable. For example, let us consider the *Computation Flow Graph* (CFG) as illustrated in Fig.1.7. We are given a set of seven task modules with the direction of flow of information highlighted by the edges of the graph. Furthermore, the execution time needed by each task module is fixed and known in advance. Three processors are available. Scheduling the task modules onto the processors to minimise the time needed to complete all the task modules is an NP-Hard problem, meaning that it is unlikely that a polynomial time algorithm exists that can always find an optimal schedule, given an arbitrary CFG. Therefore, in general, we must resort to a approximate scheduling algorithm that gives a near optimal schedule in acceptable polynomial time.

The term task allocation is often used to describe task scheduling problems. These are almost synonymous terms with the former being posed in terms of resource allocation (from the resources' i.e. processors, memory etc. view point) and the latter from the consumer's view point. *Task assignment* is also used as an alternate term for task scheduling.

1.4 Outline of the Dissertation

The rest of the dissertation is organised as follows :

Chapter 2 starts with a proof of the NP-Completeness of the multi-processor task scheduling problem. An overview of the scheduling problems and different approaches taken in solving these problems are then presented.

Section 2.4 briefly describes the available heuristic techniques and finally a summary of the contributions made by this dissertation are presented.

Chapter 3 describes *The Graph Model*, used to represent the concurrent VLSI simulation system and its scheduling onto a multiprocessor. It also presents a *cost assignment* for the task system considered — communication and computation costs are discussed in this respect. A generic multiprocessor system is considered. This chapter also contains detailed data structure and various primitives used throughout.

Chapter 4 describes the modified Kernighan-Lin graph partitioning algorithm and its parallel implementation. A hierarchical partitioning strategy is adopted. Various aspects of its implementation are also discussed.

The Simulated Annealing (SA) algorithm is introduced in chapter 5. An implementation suitable for graph partitioning is used. Two different temperature schedules are used and their performances are compared.

Chapter 6 starts with a review of parallel SA algorithms in existence and describes the problems relating to their implementation. The *Concurrent Simulated Annealing* (CSA) algorithm is proposed here and some implementations are suggested. Simulation results are also presented and an overall performance comparison is made.

Chapter 7 wraps up with some general discussion and concluding remarks. Some suggestions for future work are also made.

Appendix A contains the data and statistics for the various data flow graphs used throughout.

Appendix B contains proof of a corollary used in chapter 5.

References :

1. Gurd, J.R., Kirkham, C.C. and Watson, I., *The Manchester Proto-type Data Flow Computer*, Comm. of ACM, 28(1), Jan 1985.
2. Flynn, M.J., *Very High-Speed Computing Systems*, Proc. IEEE, Vol. 54. No. 12, Dec 1966, pp. 1901-1909.
3. Kuck, D.J., *The Structure of Computers and Computations*, John Wiley, 1978.
4. Händler, W., *The Impact of Classification Schemes on Computer Architecture*, Proc. 1977 Intl. Conf. Parallel Processing, IEEE, August 1977, pp. 7-15.
5. Quinn, M.J., *Designing Efficient Algorithms for Parallel Processing*, McGraw-Hill Book Co., 1987.
6. Hwang, K. and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., 1985.
7. Agerwala, T., *Communication, Computation and Computer architecture*, in Proc. Intl. Comm. Conf., June 1977, pp. 209-215.
8. Perry, T.S., *Intel's Secret is Out*, IEEE Spectrum, Vol. 26, No. 4, April 1989, pp. 22-28.
9. David, M.E. and Gwyn, C.W., *CAD Systems for IC Design*, IEEE Trans. CAD, Vol. CAD-1, No. 1, Jan 1982, pp. 2-12.
10. Wallich, P., *The One-Month Chip Design*, IEEE Spectrum, Vol. 21, No. 9, Sept. 1984, pp. 30-34.
11. Kravitz, S.A., *Multiprocessor Based Placement by Simulated Annealing*, M.Sc. Thesis, Carnegie-Mellon Univ., USA, 1986.
12. Ambler, A.P., *An Overview of CAD Acceleration*, IEE Colloquium : Hardware Accelerators for VLSI CAD - A Tutorial, Sept. 1988.
13. Schmid, R. and Batinger, U., *A Hardware Accelerator to Support Effective Chip Floorplanning*, Intl Workshop on Hardware Accelerators, Oxford, England, 1987.
14. Bayer, J., *Application of a Pipelined Processor for Fast and Economic Design Rule Checking and Circuit Routing*, Intl. Workshop on Hardware Accelerators, Oxford, England, 1987.
15. Sieler, S.D., *A Hardware Assisted Methodology for VLSI Design Rule Checking*, Ph.D. Thesis, MIT, USA, 1985.

16. Blank, T., *A Survey of Hardware Accelerators Used in Computer Aided Design*, IEEE Design & Test, Vol. 1, No. 3, Aug 1984.
17. Lewis, D.L., *Hardware Accelerators for Timing Simulation of VLSI Digital Circuits*, IEEE Trans. CAD, Vol. CAD-7, No. 11, Nov 1986, pp. 1134-1149.
18. Deutsch, J.T. and Newton, A.R., *A Multiprocessor Implementation of Relaxation Based Electrical Circuit Simulation*, ACM-IEEE Design Automation Conf., USA, June 1984.
19. Coleman, N. and Ambler, A.P., *A Multiprocessor for General VLSI design Acceleration*, Intl. Workshop on Hardware Accelerators, Oxford, England, 1987.
20. Vladimirescu, A. and Lui, S., *The Simulation of MOS Integrated Circuits Under SPICE2*, Elec. Research Lab., Univ. of Calif. Berkley, USA, ERL Memo M80/7, 1980.
21. Kleckner, J.E., Saleh, R.A. and Newton, A.R., *Electrical Consistency in Schematic Simulation*, in Proc. IEEE Conf. on Computer Aided Design, Nov 1984.
22. Terman, C.J., *Simulation Tools for Digital LSI Design*, Lab. for Computer Science, MIT, USA, Tech. Rept. MIT/LCS/304.
23. Bryant, R.A., *A Switch-Level Model and Simulation for MOS Digital Systems*, Calif. Inst. of Tech., USA, tech. Rept. 5065:TR:93, 1983.
24. Ackland, B.D., Ahuja, S.R., Linstrom, T.L. and Romero, D.J., *CEMU — A Concurrent Timing Simulator*, in Proc. IEEE Intl. Conf. Computer Aided Design, 1985.
25. Chawla, B.R., Gummel, H.K. and Kozak, P., *MOTIS — A MOS Timing Simulator*, IEEE Trans. Circuits & Syst., Vol. 22, No. 12, Dec 1975, pp. 751-756.
26. Garey, M.R. and Johnson, D.S., *Computers and Intractability : A Guide to the Theory Of NP-Completeness*, Freeman, San Fransisco, USA, 1979.
27. Helmbold, D.P., *Parallel Algorithms for Scheduling and Related Problems*, Ph.D. Thesis, Stanford Univ., USA, 1987.
28. Graham, R.L., *Bounds on Multiprocessor Timing Anomalies*, SIAM J. of App. Math., 17(2), March 1969.

CHAPTER 2

The Task Scheduling Problem

Scheduling problems are combinatorial optimisation problems. Each instance consists of task modules to be scheduled on a certain number of processors. The solution is a *schedule* indicating when and where each task module is to be executed. A schedule of task modules to processors can formally be described by a function from the set of task modules to the set of processors, $f : T \rightarrow n$. In a system of T task modules and n processors there are n^T possible schedules of tasks to processors. The difficulty, is therefore, to pick out an optimal schedule from the exponentially many different possibilities. A performance criterion in the form of a function is thus used to compare the n^T possible schedules and to associate a cost with each schedule. An optimal schedule is one which minimises the cost function.

2.1 Complexity of Task Scheduling Algorithms

The complexity of an algorithm can be given by a function $f: \mathbb{N} \rightarrow \mathbb{N}$ (where \mathbb{N} is the set of all natural numbers), which characterises the execution time of the algorithm in terms of the size of its input. Algorithmic characterisation of $f(\cdot)$ (i.e., constant, logarithmic, linear, polynomial, exponential etc.) are used to identify the complexity of the associated algorithm. Let, P be the set of all problems π for which there exists a deterministic polynomial time algorithm to solve the problem π . Likewise, let NP be the set of all problems which have non-deterministic polynomial time algorithmic solutions. The term, NP-Complete is used to describe problems that are the hardest ones in NP . A problem π is said to be NP-Complete if i) $\pi \in NP$, and furthermore ii) $\pi \in P$ implying $P = NP$. Since, it is widely believed that $P \neq NP$, a proof that a problem π is NP-Complete is equivalent to showing that π can not be solved efficiently and that probably the best deterministic algorithm to solve π is at least of exponential time complexity. NP-Complete problems are posed in the form of a decision problem and the corresponding optimisation problem is known to be in the class of NP-Hard problems.

Vairavan and DeMillo [1] using a synchronous parallel computation model showed that any algorithm designed to generate an optimal n -processor (n fixed) schedule of a loop free computer program, when such a schedule exists, would demand exponential time. We here also present a proof of the NP-Completeness of the multiprocessor task scheduling problem using the *reducibility* property of NP-Complete problems. This is a two part proof, where in Part I we show that the problem is in NP . In the second part, the scheduling problem is shown to be polynomially transformable to a *quadratic assignment problem* which is NP-Complete.

Theorem (2.1). *The above mentioned multiprocessor task scheduling problem is NP-Complete.*

Proof :

Part I :

We consider a *non-deterministic Turing machine* (NDTM). The NDTM would make a first guess at picking a *processing element* (PE)

out of n such PEs and assign it to the first node of the task graph. In the next step another PE would be selected for allocation. This process would continue until all the nodes of the task graph are allocated. A guess should be such that the communication overhead of the resulting schedule is minimised and computational loads are evenly balanced among the PEs. This would take $O(n)$ time on the NDTM, which proves that the problem is in NP.

Part II :

We define an assignment matrix $X[T, n]$ with components,

$$x_{iq} = \begin{cases} 1 & ; \quad \text{if task } i \text{ is assigned to PE } q, \\ 0 & ; \quad \text{Otherwise.} \end{cases} \quad (2.1)$$

Then,

$$\sum_{q=1}^n x_{iq} = 1 \quad ; \quad 1 \leq i \leq T, \quad (2.2)$$

since each task must be assigned to exactly one processor. Also, for cases $T \geq n$, each processor can have more than one task assigned to it.

Therefore,

$$\sum_{i=1}^T x_{iq} \geq 0 \quad ; \quad 1 \leq q \leq n. \quad (2.3)$$

We now define,

e_{ij} = Amount of data transfer from node i to node j of the task graph.

w_{iq} = computational load of node i of the task graph on processor q .

It can be easily seen from the above that the total communication cost may now be expressed as,

$$C_c = \sum_{\substack{i,j=1 \\ i \neq j}}^T \sum_{\substack{q,r=1 \\ q \neq r}}^n e_{ij} x_{iq} x_{jr}, \quad (2.4)$$

and the total execution cost,

$$C_e = \max_{q=1 \dots n} \left\{ \sum_{i=1}^T w_{iq} x_{iq} \right\}. \quad (2.5)$$

Therefore, the scheduling problem is transformed to the following optimisation problem,

$$\text{minimise } \{\alpha|C_c| + \beta|C_e|\} \quad (2.6)$$

subject to,

$$\sum_{q=1}^n x_{iq} = 1 \quad ; \quad 1 \leq i \leq T, \quad (2.7)$$

and

$$\sum_{i=1}^T x_{iq} \geq 0 \quad ; \quad 1 \leq q \leq n. \quad (2.8)$$

This is equivalent to a quadratic assignment problem. Since, the quadratic assignment problem is NP-Complete[2,3], the present multiprocessor task scheduling problem is also NP-Complete.

Q.E.D.

2.1.1 Parallel Algorithms

The major driving force behind the research and development of parallel computers is to speed-up the solution process of difficult algorithmic problems. But, surely an undecidable problem can not be solved by a parallel computer no matter how large or complex it is. The reason being that every parallel computer can be simulated by a sequential processor, running around and doing every processor's work in an appropriate order. In this sense, the Church/Turing thesis also applies to parallel models of computation too: *the class of solvable problems is insensitive even to the addition of parallelism.*

But, can parallelism turn intractable problems into tractable ones? In other words, is it possible to have a polynomial time algorithm for a problem having exponential time sequential solution? To answer this, we recall that all problems in NP have reasonable (polynomial) solutions that are non-deterministic. If a correct guess is made out of a large possibilities, it would lead to a positive solution. Now, if we have an unlimited number of processors, we can employ each processor to explore each possibility. If one of the processors finds a correct solution then indeed a polynomial time

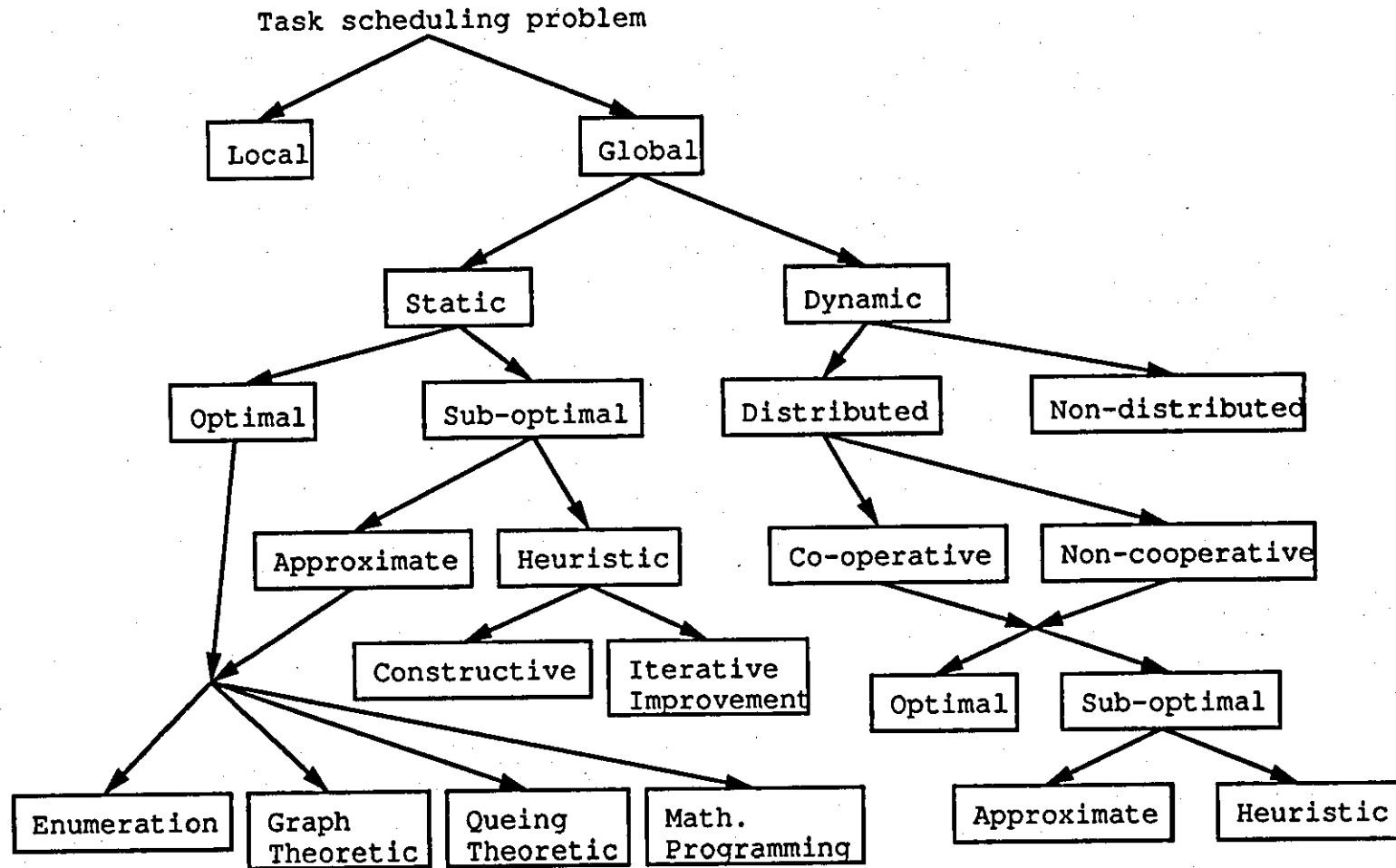
parallel solution exists. On the other hand, if none of the processors find a solution then the problem can be thought of as truly intractable.

It is, therefore, seen that reasonable parallel solutions for intractable problems are theoretically feasible. But, in reality many practical problems exist. NP-Complete problems are not known to be intractable — they are merely conjectured to be so. Thus, the possibility of simulating NP-Complete problems in parallel in polynomial time does not imply that parallelism can get rid of a problem of its inherent intractability, since we do not know whether or not NP-Complete problems are actually intractable. Furthermore, the number of processors required to solve an NP-Complete problem in reasonable time is itself exponential, requiring billions or trillions of processors in some cases. Even, if it is possible to have that many processors in a multiprocessor system with the advent of technology, the communication overhead would be too large to cope with. The algorithm to control the communication links and channels would also be of exponential complexity. With all this overhead, a super-polynomial number of processors would require a super-polynomial amount of real time to carry out even a polynomial number of instructions. In practical terms, therefore, it is not possible to have a reasonable parallel counterpart for an unreasonable (exponential) sequential algorithmic solution.

Nonetheless, a practical multiprocessor system, both coarse and fine grained, are of immense importance for many real world applications. In many cases an almost linear speed-up is possible. Again, for the solution of difficult NP-Complete problems, a heuristic algorithm can be chosen over an exact algorithm, which when properly implemented on a parallel machine, would give reasonably good solution within a time of practical value.

2.2 Classification of Task Scheduling Algorithms

The multiprocessor task scheduling problem bears a close resemblance to the classical job sequencing problems as encountered in production management. These types of problems have been described a number of times and in a number of different ways in the literature [1,4-6]. In this classification, however, we take a slightly different view and concentrate on the mechanism or policy of efficient and effective management of the access to and on the



Adaptive vs. Non-adaptive
 Load Balancing vs. Minimum Interprocessor Communication (IMC)
 Policy Mechanism vs. Bidding

Fig. 2.1 Classification of different task scheduling approaches.

use of various resources by its various consumers. Obviously, in this scenario, the processors form part of the resources and the program tasks constitute the consumers waiting to be executed on the processors.

In this short classification, the goal here is to familiarise with a commonly accepted set of terms and also to present a means to compare past works in the area of multiprocessor task (or distributed process) scheduling in a qualitative way. A hierarchical classification proceeds first, to be followed by a flat classification scheme which is felt necessary as some choices of characteristics may be made independent of previous design choices and thus require different attention. The classification tree is presented in Fig. 2.1. This classification however does not take into consideration the different possible strategies for the parallelisation of the scheduling algorithms itself. In the following sections we present some of the selected and pertinent categories of the classification tree. The selected classification presented here is based on Casavant and Kuhl's taxonomy [7] which can be consulted for detailed study.

2.2.1 Static vs. Dynamic Scheduling

In the case of *static scheduling* also known as *deterministic scheduling* [4], the task modules are preassigned to the processors before the execution actually begins. Hence, each executable task in a system has a static assignment to a particular processor, and each time that task is submitted for execution, it is assigned to that same processor. Static scheduling can be successful only when *a priori* knowledge of the execution behaviour of the tasks is available. This includes the input-output profile as well as the computational load of each task. Static scheduling is attractive because it eliminates scheduling overhead entirely at run time. Further, there is a greater opportunity to optimise the interprocessor communication. The disadvantage is that the execution behaviour estimate may be inaccurate leading to inefficient schedule and that also static schedule is tied to a particular hardware configuration and a new schedule is thus necessary every time there is a change in hardware architecture or topology.

On the other hand, in the case of *dynamic scheduling* the assignment of tasks to processors is left till the run time. The assignment procedure

takes place dynamically along with the actual processing of the tasks. This leads to a heavy run time scheduling problem. Nevertheless, this approach is favoured when execution behaviour of the tasks is not available or difficult to ascertain. Dynamic scheduling can again be grouped into *distributed* and *non-distributed* (or *centralised*) scheduling. As the names suggest, in the case of non-distributed scheduling the task of global dynamic scheduling should reside in a single processor and in the other case it is physically distributed among the processors.

2.2.2 Optimal vs. Sub-optimal Scheduling

In some cases of multiprocessor task scheduling, where all the information regarding the state of the system as well as the execution behaviour of the tasks are known, an *optimal* assignment is feasible [6, 8–9]. These optimal assignments are based on some simple criterion functions and results of their appropriate optimisations. Examples include minimising total process completion time, maximising utilisation of resources in the system, or maximising system throughput. In the event that a more robust and accurate criterion function is used resulting the solution computationally infeasible, *sub-optimal* solutions may be more desired [10–11]. As shown in section 2.5 an optimal solution for the multiprocessor task scheduling problem is very difficult to achieve and as such a sub-optimal solution is more realistic and desirable.

2.2.3 Approximate vs. Heuristic Solutions

In the approximate method, use of the same computational model for the algorithm is used. However, instead of searching the whole solution space which is deemed very time expensive, search is stopped when a good solution is found. This is taken as the *sub-optimal approximate* solution. The difficulty, however, arises in the determination of a good solution. In the cases where a metric is available for evaluating a solution, this technique can be used to decrease the time taken to find an acceptable solution.

Heuristic methods are favoured for the solution of many combinatorial optimisation problems due to its ability to provide *near-optimal* solutions

in reasonable time. Different heuristic scheduling algorithms have been proposed [5, 10–14]. This method is best suited when a good and realistic assumption about *a priori* knowledge concerning the execution behaviour of the tasks can be made. Heuristic schedules often use an indirect rather than direct approach to monitor the system performance and this indirect approach is much simpler to implement and calculate. For example, clustering of tasks [5] can be employed so that heavily communicating tasks are grouped together and assigned to the same processor and also physically separating the tasks which would benefit from parallelism. This directly decreases the overhead involved in passing information between processors while reducing the interference among tasks which may run without synchronisation with one another.

2.2.4 Load Balancing

Load balancing has received a great deal of attention [14–17]. This is more of a design choice than a separate algorithmic approach and as such placed under the flat classification in the classification tree of Fig. 2.1. This brings fairness to the hardware resource utilisation. The basic idea is to attempt to balance (in some sense) the load on all processors in such a way as to allow progress by all tasks on all processing elements to proceed at approximately the same rate. This approach is best suited for homogeneous multiprocessor system since this allows all processors to know a great deal about the structure of the other processors.

Incorporation of load balancing criterion in the cost function of a heuristic algorithm is very important and brings higher processor utilisation. A heuristic algorithm which minimises the interprocessor communication in a schedule totally ignoring load balancing, would assign all the tasks to a single processor as the communication overhead between tasks assigned to the same processor is considered negligibly small. As a result the schedule though an optimal one for the criterion considered would be taken as highly inefficient.

2.3 Optimal and Sub-optimal Approximate Techniques

For both optimal and sub-optimal approximate solutions of static multiprocessor scheduling there are four basic categories of algorithms that can be used. These are described in the following sub-sections :

2.3.1 Solution Space Enumeration and Search

A complete enumeration and search for the optimal solution can be very time expensive and of exponential time complexity. However, the problem can be transformed into a simpler one and state-space search can then be used to arrive at an optimal or acceptable sub-optimal solution. Shen and Tsai [9] restated the problem as weak homomorphic graph matching problem and used the A* algorithm after collecting relevant heuristic informations for an optimal solution.

2.3.2 Graph Theoretic

In graph theoretic approach the scheduling problem is modelled as a network with undirected edges and an attempt is then made to find the maximum flow across a cut, resulting in an optimal solution. Stone [18] used Ford-Fulkerson's *Max Flow Min Cut* [19] algorithm to find an optimal schedule. However, his algorithm is applicable for 2-processor systems only. Stone and Bokhari [20] used this idea for n -processor homo/heterogeneous systems. However, this works only when the intertask communication pattern is constrained to be tree structured. Lo [10] also proposed a three stage algorithm composed of graph theoretic and heuristic method for a near optimal n -processor schedule. The two main drawbacks of graph theoretic approach is its lack of mechanism to accommodate load balancing and to incorporate various resource constraints into the model.

2.3.3 Mathematical Programming

Various mathematical programming techniques like *branch-and-bound*, *backtracking*, *0-1 integer programming* etc. can be successfully applied to solve multiprocessor scheduling problems. Chu [21] used 0-1 integer

programming technique for optimal file allocation in a multiprocessor system which bears close resemblance to our problem in hand. Ma et. al [6] used branch-and-bound method to minimise the interprocessor communication and also for balanced processor utilisation. Kasahara and Narita [22] used a combination of branch-and-bound and critical path methods. Though mathematical programming techniques are flexible enough to incorporate many system constraints, they are still of exponential time complexity. These techniques can be used in some cases, but their generalisation is reduced by their demand for large time and space.

2.3.4 Queing Theoretic

Queing theory can also be applied for the solution of multiprocessor scheduling problems [15, 19]. Klinrock and Nilsson [23] considered a M/G/1 queing system model. Their cost function is based on task waiting time and their required service time. The problem is posed as an optimisation problem and mathematical programming technique is used to optimise the total cost. However, the solution is found to be sub-optimal. Generalisation for a n -processor model is also found difficult.

2.4 Heuristic Technique

Heuristic algorithms by their very nature can adopt many different possible approaches. However, for the multiprocessor task scheduling and also for many other similar combinatorial optimisation problems a simple classification can be attempted. Two different basic approaches can be thought of. These are:

1. Constructive Method
2. Iterative Improvement Method.

2.4.1 Constructive Method

The constructive scheduling algorithm begins with the assignment of one or a few *seed* task modules and then gradually builds up the total schedule by assigning a new free task module in succession each to a processor, always taking the best momentary decision for any particular assignment. The

approach is however *greedy* in nature as the decision at each instant is based on the current effect and not on the global effect. As a result the resulting solution often turns out to be rather inferior.

Still, there are many combinatorial optimisation problems where greedy constructive algorithms can be used to produce good solutions with high probability. For many such problems an exhaustive search is far too impractical and in the absence of other fast heuristic algorithms, greedy constructive algorithms are the only *real and wise* choice. One classical example is for the solution of *Travelling Salesman Problem* (TSP), where greedy constructive algorithms have been found very effective. For the multiprocessor task scheduling problem a good use of the constructive algorithm can be made whereby heavily connected task modules are grouped together and assigned to the same processor. This would help to minimise the communication cost component of the objective function.

2.4.2 Iterative Improvement Method

The iterative improvement method is of special interest in this dissertation. As mentioned earlier, the task scheduling problem falls into the class of combinatorial optimisation problems involving large solution space. Though, theoretically it is possible to find the best solution by generating and evaluating all possible solutions, in practical terms it turns out to be an impossible task due to the exponential growth of complete enumeration algorithms.

Heuristic algorithms provide much promise to find a good solution in reasonable time. Due to the greedy nature, the constructive heuristic algorithms often fail to keep up to this promise. The iterative improvement method is a viable alternative which can be thought of made up of two phases.

For the solution of task scheduling problem, an initial schedule is generated in the first phase either by a constructive algorithm or by random method. This initial schedule is then iteratively improved in the second phase. A new schedule is generated at each step by introducing some

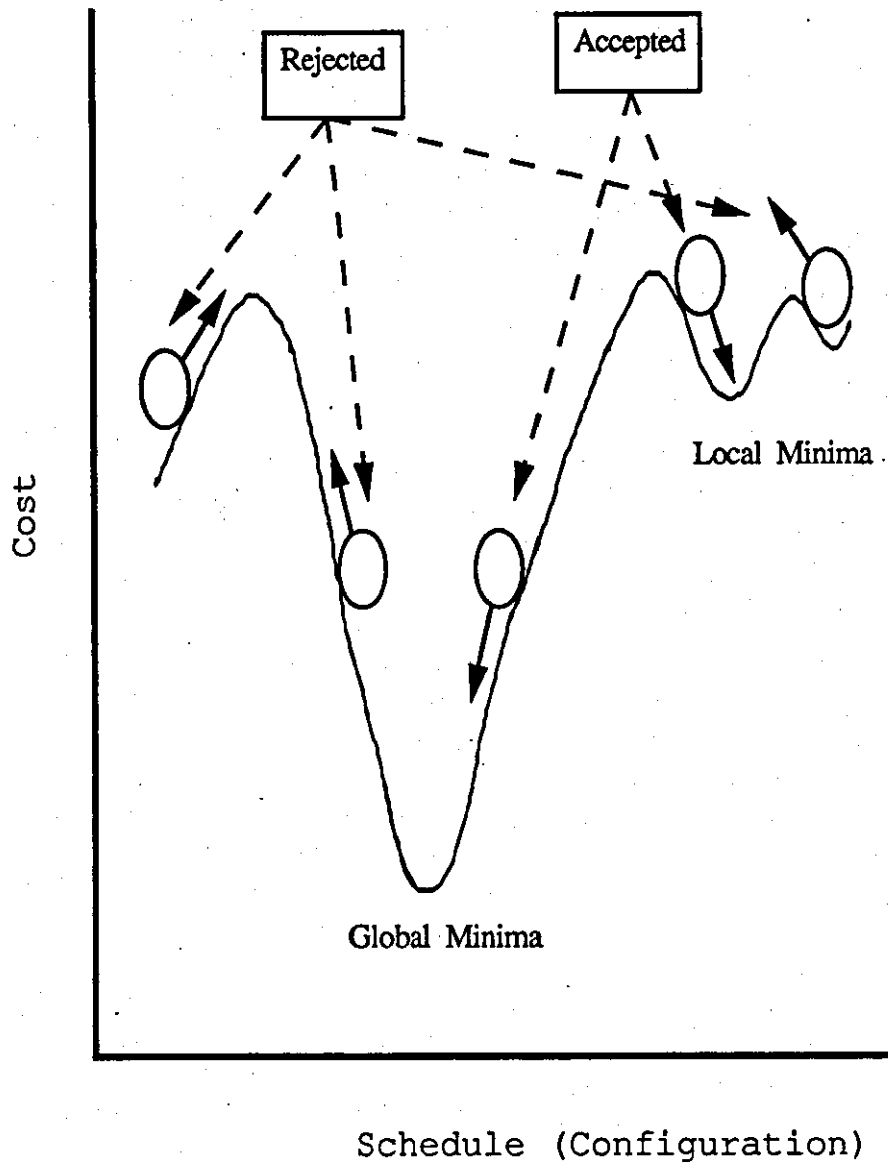


Fig. 2.2 The local minima traps of a cost function.

modifications to the present schedule (configuration). The modifications that can be made are :

1. Changing the assignment of a task module to a new processor.
2. Swapping the assignments of two task modules.

A cost (*objective*) function is defined to guide the heuristic search in a direction that will improve the schedule. If the cost of a new schedule is lower than that of the previous one, the new one is accepted and further modifications are inflicted upon it. Otherwise, if the cost is increased in the new schedule, it is rejected and the previous one is retained for further modifications. The iterative improvement process is continued until no further improvement can be obtained or any other predefined stopping criterion is satisfied.

2.5 Local Minima and Optimal Solution

Most heuristic algorithms search for a solution only in the direction that improve the cost functions. One inherent drawback of this type of heuristic search is that it can be easily trapped into a local minima of the cost function. The example presented in Fig. 2.2 is used to demonstrate this problem.

The curve in Fig. 2.2 may be considered as the cost function of an iterative improvement process and the circles can be used to indicate the costs of certain schedules. Since a new schedule is generated by introducing small modifications to the present schedule, its corresponding location on the curve is most likely to be somewhere near that of the present schedule. The traditional iterative improvement algorithm only accepts schedules that have reduced the cost. This criterion of schedule acceptance implies that the process can only go downhill into a *local minima* and any uphill movement is forbidden. Thus, the search process can not climb over the peak of the curve to reach the *global minima*.

Different algorithmic solutions have so far been proposed [24–26] that gives a certain degree of hill climbing capability to the basic iterative improvement heuristics. This ensures a good near optimal solution for NP-Hard problems. As an illustration we here present a brief introduction of the

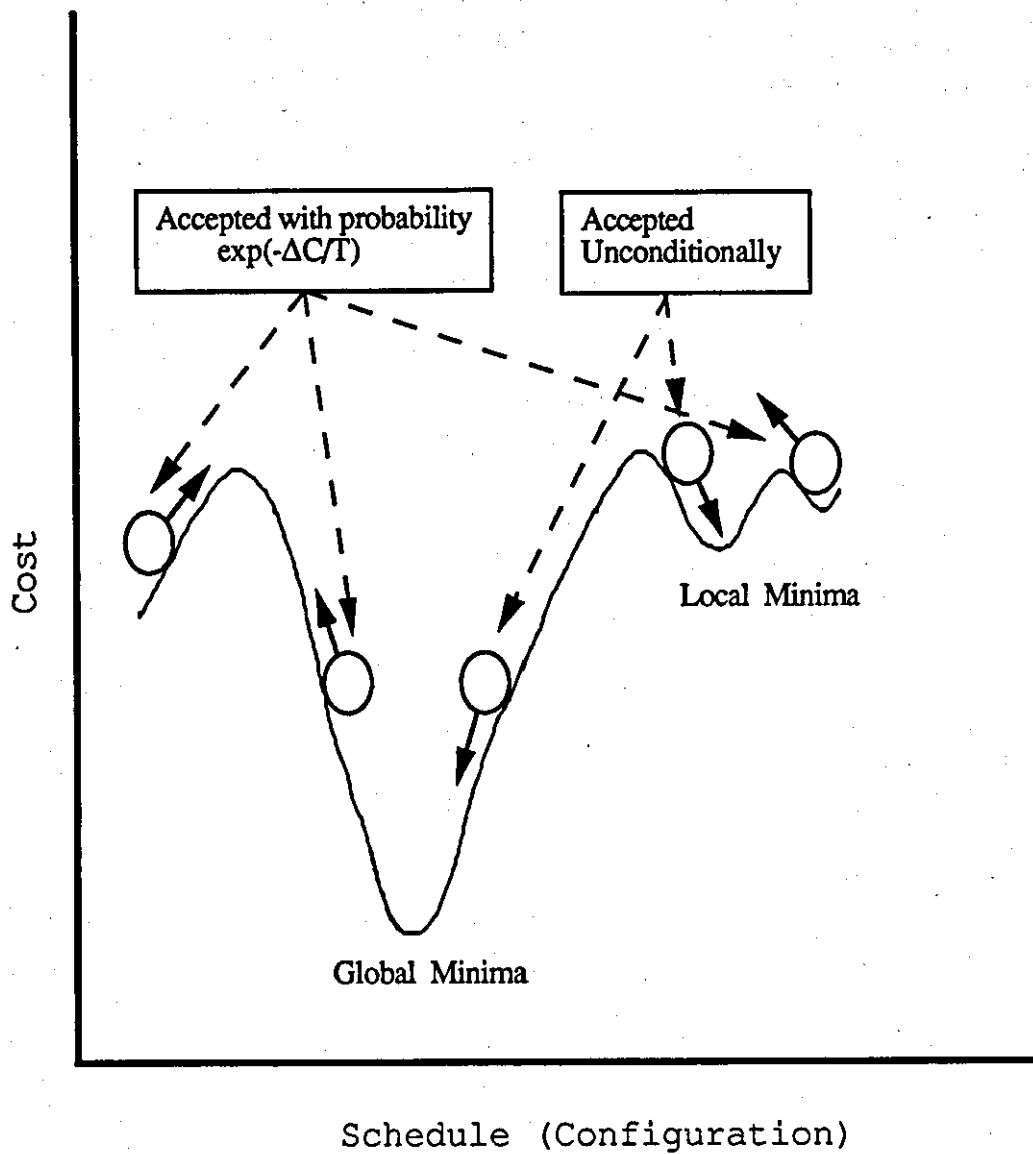


Fig. 2.3 The hill climbing capability of simulated annealing algorithm.

Simulated Annealing [24] algorithm which is recently being successfully used for the solution of many difficult problems including network partitioning, VLSI cell placement, numerical problems using Monte-Carlo method as well as task scheduling problem. An analogy is made with the statistical mechanics which deals with the behaviour of systems with many degrees of freedom in thermal equilibrium at a finite temperature, to combinatorial optimisation which finds the minimum of a given function depending on many parameters. Simulated annealing brings minor but, vital modification to the traditional iterative improvement method. Instead of rejecting outright a new schedule (or configuration in general case) which results in an increase in the cost function (ΔC), the modified algorithm accepts this new configuration with a certain probability,

$$P(S_i) = e^{(-\Delta C(S_i))/t}, \quad (2.9)$$

where S_i represents a certain configuration at any instant i .

This conditional probability is dictated by the Maxwell-Boltzmann statistics of statistical physics. The parameter t , an analog of temperature in the physical process is a very important control parameter. This control parameter t , also known as temperature, is slowly reduced from an initial high value to a final very low value, where the solution is thought to be frozen (converged). A configuration has a high probability of being accepted at high temperature for the same cost increase. On the other hand, when the temperature is lowered, the probability of accepting a cost increasing configuration is smaller. This is equivalent to high hill climbing capability at the initial high temperature which is necessary to explore the overall solution space as much as possible without being trapped into a local minima. A slow reduction to a very low temperature over a good number of steps, ensures the configuration to settle down either to the global minima or somewhere very near to it. Fig. 2.3 shows the effect of simulated annealing algorithm.

2.6 Contribution of this Dissertation

Static scheduling is favoured over dynamic scheduling. The main reason being that a detailed *a priori* knowledge about the execution and input-output behaviour of the VLSI simulation modules can be easily obtained.

Apart from this, in a typical VLSI design, the same circuit may have to be simulated a few times, each time with a new set of parameters to test all the possible eventualities. The cumulative scheduling overhead of dynamic scheduling would be much too high to render this almost impractical.

The item that is conspicuously absent from the classification tree of Fig. 2.1 is the distributed static scheduling algorithm and its siblings. A parallel or distributed static scheduling algorithm is much desired for practical applications. Apart from the obvious speed-up advantage, a parallel scheduling algorithm would definitely improve the machine utilisation of the available hardware system. In the VLSI design environment the same hardware can then be used for a variety of purposes including electrical/logic/timing simulation, floor plan design, wire routing, fault analysis etc.

Unfortunately not much research has been addressed to the problem of parallel static scheduling algorithm, although some research into parallel heuristic algorithm for the VLSI cell placement problem has been reported [27-29]. The VLSI cell placement problem bears a close similarity with the multiprocessor task scheduling algorithm. In this dissertation, the main theme is concentrated on the design and performance study of various parallel heuristic task scheduling algorithms. In addressing the general problem of parallel task scheduling this dissertation makes the following specific contributions :

1. The definition of a problem and system independent graph representation of the simulation task. With appropriate modification this can be used to represent parallel programs waiting to be executed on a multiprocessor.
2. The definition of a cost model for the task assignment. This model is general and flexible enough to represent a wide range of VLSI simulation problems and also different multiprocessor architecture.
3. The scheme for the incorporation of both execution and communication load on the target system.

4. Use of simple hierarchical partitioning method which can be easily implemented in a parallel environment. This, coupled with Kernighan-Lin's 2-way partitioning algorithm [25] has been found to be favourable for smaller multiprocessor system.
5. A parallel implementation of the simulated annealing heuristic with optimal number of parallel moves. this is thought to achieve faster convergence than other similar approach. .

References :

1. Vairavan, K. and DeMillo, R.A., *On the Computational Complexity of Generalised Scheduling Problem*, IEEE Trans. Comp., Vol. 2-25, No. 11, Nov, 1976, pp. 1067-1073.
2. Garey, M.R. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman Press, 1979.
3. Sahni, S. and Gonzalez, T., *P-Complete Approximation Problems*, J. ACM, Vol. 23, March 1976, pp. 555-565.
4. Gonzalez, *Deterministic Processor Scheduling*, ACM Computing Surveys, Vol. 9, No. 3, Sept. 1979, pp. 173-204.
5. Chu, W.W., Holloway, L.T., Lan, M. and Efe, K., *Task Scheduling in Distributed Data Processing*, Computer, Vol. 13, No. 11, Nov. 1980, pp. 57-67.
6. Ma, P.R., Lee, E.Y.S. and Tsuchiya, M. , *A Task Allocation Model for Distributed Computing Systems*, IEEE Trans. Comp., Vol. C-31, No. 1, Jan. 1982, pp.41-47.
7. Casavant, T.L. and Kuhl, J.G., *A Taxonomy of Scheduling in General Purpose Distributed Computing Systems*, Tech. Report, Dept. Elect. & Comp. Engg., Univ. of Iowa, 1986.
8. Bokhari, S.H., *A Shortest Tree Algorithm for Optimum Assignment Across Space and Time in a Distributed Processor Systems*, IEEE Trans. Soft. Engg., Vol. SE-7, No. 6, Nov. 1983, pp. 335-341.
9. Shen, C. and Tsai, W., *A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a MinMax Criterion*, IEEE Trans. Comp., Vol. C-34, No. 3, March 1985, pp. 197-203.
10. Lo, V., *Task Assignment in Distributed Systems*, Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 1985.
11. Efe, K., *Heuristic Models for Task Scheduling in Distributed Systems*, Computer, Vol. 15, June 1982, pp. 50-56.
12. Ward, M.O. and Romeo, D.J., *Assigning Parallel Executable Interconnecting Subtasks to Processors*, 1984 Intl. Conf. Parallel Proc. Aug. 1984, pp. 392-394.

13. Ibrira, O.H. and Kim, C.E., *Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors*, J. ACM, Vol. 24, No. 2, April 1979, pp. 280-289.
14. Sheild, J., *Partitioning Concurrent VLSI Simulation Programs onto a Multiprocessor by Simulated Annealing*, IEE Proc., Vol. 134, Pt. E, No. 1, Jan 1987, pp. 24-30.
15. Chow, T.C.K. and Abraham, T.A., *Load Balancing in Distributed Systems*, IEEE Trans. Soft. Engg., Vol. SE-8, No. 4, July 1982, pp. 401-417.
16. Chow, Y.C. and Kohler, W.H., *Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System*, IEEE Trans. Comp., Vol. C-28, No. 5, May 1979, pp. 354-361.
17. Ni, L.M. and Hwang, K., *Optimal Load Balancing in a Multiple Processor System with Many Job classes*, IEEE Trans. Soft. Engg., Vol. SE-11, No. 5, May 1985, pp. 491-496.
18. Stone, H., *Multiprocessor Scheduling With the Aid of Network Flow Algorithms*, IEEE Trans. Soft. Engg., Vol. SE-3, Jan 1977, pp. 85-93.
19. Ford, L.R. and Fulkerson, D.R., *Flows in Networks*, Princeton Univ. Press, N.J., USA, 1962.
20. Stone, H. and Bokhari, S.H., *Control of Distributed Processes*, Computer, Vol. 11, July 1978, pp. 97-106.
21. Chu, W.W., *Optimal File Allocation in Multiple Computing Systems*, IEEE Trans. Comp., Vol. C-18, No. 10, Oct. 1969, pp. 885-889.
22. Kasahara, H. and Narita, S., *Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing*, IEEE Trans. Comp., Vol. C-33, No. 11, Nov. 1984, pp. 1025-1029.
23. Klinrock, L. and Nilsson, A., *On Optimal Scheduling Algorithms for Time-shared Systems*, J. ACM, Vol. 28, No. 3, July 1981, pp. 477-486.
24. Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P., *Optimization by Simulated Annealing*, Science, Vol. 220, No. 4598, May 1983, pp. 671-680.
25. Kernighan, B.W. and Lin, S., *A heuristic procedure for partitioning graphs*, Bell Syst, Tech. J., 1970, pp. 291-307.
26. Holland, J.H., *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor, Mi, USA, 1975.

27. Casotto, A., Romeo, F. and Sangiovanni-Vincentelli, A.A., *A Parallel Annealing Algorithm for the Placement of Macro Cells*, IEEE Trans. CAD, Vol. CAD-6, No. 5, Sept. 1987, pp. 838-847.
28. Darema, D., Kirkpatrick, S. and Norton, *Parallel Algorithms for Chip Placement by Simulated Annealing*, IBM J Res., Vol. 31, No. 3, May 1987, pp. 391-402.
29. Kravitz, S.A., *Multiprocessor Based Placement by Simulated Annealing*, M.Sc. Thesis, Carnegie-Mellon Univ., 1986.

CHAPTER 3

The Graph Model

This chapter describes the graphical model used to represent the concurrent VLSI time simulation systems. Also described are the formulations of the cost function for the scheduling of these simulation systems onto a multiprocessor. The method of calculating the communication and load imbalance costs of a particular schedule is also discussed.

The scheduling techniques described later are based on estimates of the VLSI simulation system's performance characteristics, such as parallelism and a measure for the costs for execution time and communication overhead. The graph model as used expresses these performance characteristics and leaves the remaining aspects (i.e., circuit simulation principles) unspecified. This facilitates the graph model to be flexible over many different circumstances, as in the scheduling of parallel programs onto multiprocessors.

The scheduling of the graph model assumes a general purpose *Multiple Instruction stream, Multiple Data stream* (MIMD) computer organisations.

A multiprocessor is a collection of communicating processing elements. The performance characteristics which are of prime interest are the processor execution times and scheduling and communication overheads.

3.1 Graphical Representation: Background

Graphs are a popular data structure in many different applications including *performance evaluation techniques* like PERT, CPM and RAMPS [1], computer program representation, control flow analysis etc. *Dags*, *flow graphs* and *data flow graphs* have common use in program representation.

To represent program expressions containing common sub-expressions, expression dags (*directed acyclic graph*) are used [2]. An internal node represents an operator and its children represent its operands. An *edge* in an expression dag thus represent the *data dependence*. Likewise, a basic block dag being similar to an expression dag, represents an entire basic block. It however requires extra precedence edges to represent operators with side effects correctly (e.g. array and pointer assignment, function calls). The flow graph [2] of a program is another kind of graph used to represent computer programs. Its *nodes* represent computation and edges represent flow of control. A path in the flow graph represents a possible execution sequence in the program.

The data flow model of computation is also based on a graphical representation of programs. A data flow graph is the executable machine code for a data flow machine [3]. As in a dag, nodes represent operators and edges represent operands. All the data are represented by *tokens* which flow along the edges of the data flow graph. A node which has tokens on all of its input edges is ready to *fire*. It executes by consuming all its input tokens and producing a token on each of its output edges. Data flow's parallelism lies in the node's ability to fire concurrently and in the pipelining due to token streams. Data flow graphs are in fact networks which implement programs rather than just program representation.

Process flow graphs introduced by Shaw [4] are used to describe a system of processes with precedence constraints. The process flow graph is a dual of the dag representation in the sense that it inverts the use of nodes and

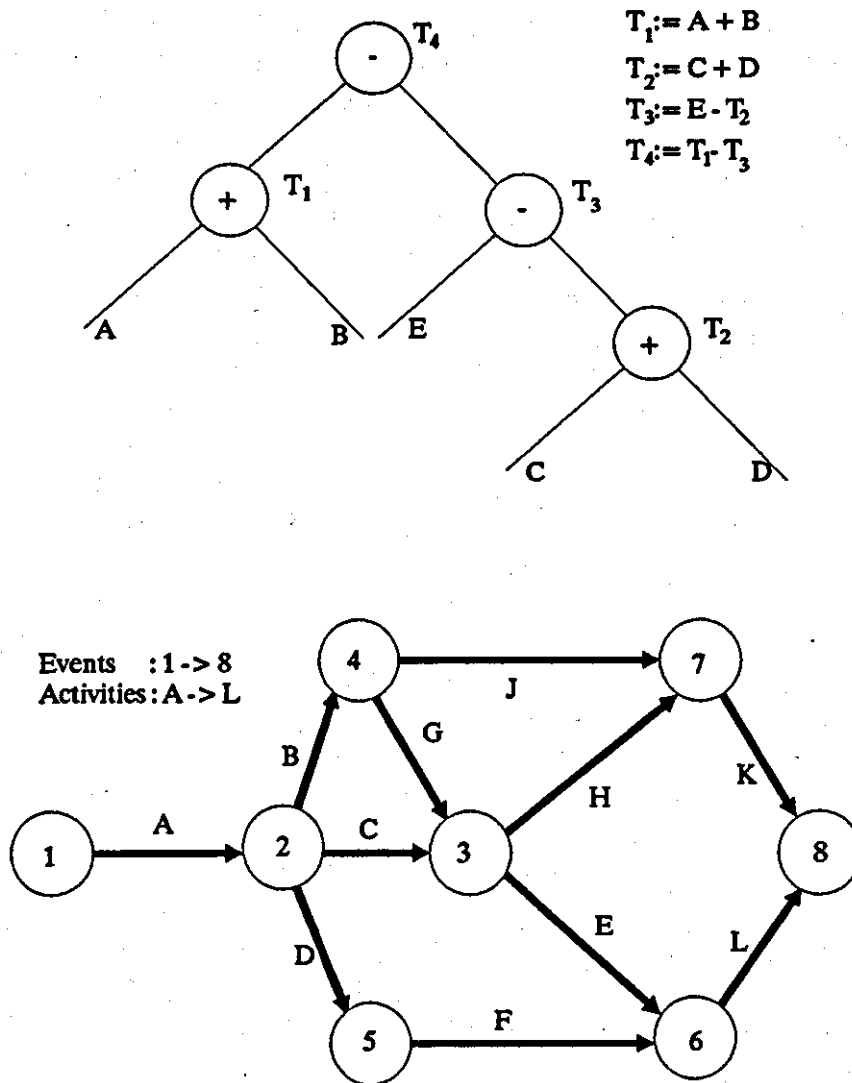


Fig 3.1 Examples of graphs. An expression dag (top) and a process flow graph (bottom).

edges. Edges in a process flow graph represent processes and nodes enforce precedence constraints by serving as synchronisation points.

Process flow graphs are a form of *Activity On Edge* (AOE) networks, where the activity occurs on edges and the nodes define the precedence constraints. PERT, CPM and RAMPS are of this type. Dags are a form of *Activity On Vertex* (AOV) networks, where the activity occurs in the vertices (nodes) and the edges define the precedence constraints. Fig. 3.1 shows two examples of AOV and AOE graphs.

3.2 Concurrent VLSI Timing Simulation

A VLSI timing simulator like EMU [5] models an MOS circuit as a set of capacitive nodes interconnected by various voltage controlled current sources as shown in Fig. 3.2. A single transistor (whose channel current depends on gate, drain and source voltages) or combinational logic gates (whose output current depends on the various gate input voltages) can be thought of as such current sources. Two such unidirectional sources can be used to represent a bidirectional circuit element, e.g. a pass transistor. Decomposition of all combinatorial gates to their transistors yields greater accuracy, but in such cases the simulator takes longer time to complete.

In the circuit the voltage V_i at any node i can be obtained by adding the currents generated by all current sources driving that node and integrating their charging effect on the node capacitance C_i . For a sufficiently small time step $t - t_0$, the voltage on node i can thus be calculated as,

$$v_i(t) = v_i(t_0) + \frac{I_{tot}(t - t_0)}{C_i} \quad (3.1)$$

The choice of time step above dictates the accuracy and numerical stability of this forward integration scheme.

EMU has an automatic means to control the time step. It starts with a maximum allowable time step t_m and during the simulation it sub-divides this time step dynamically according to circuit activity. During simulation, EMU tries to maintain the change in voltage at a node during a time step close to a preset threshold value and thus adjusts the size of the time step accordingly.

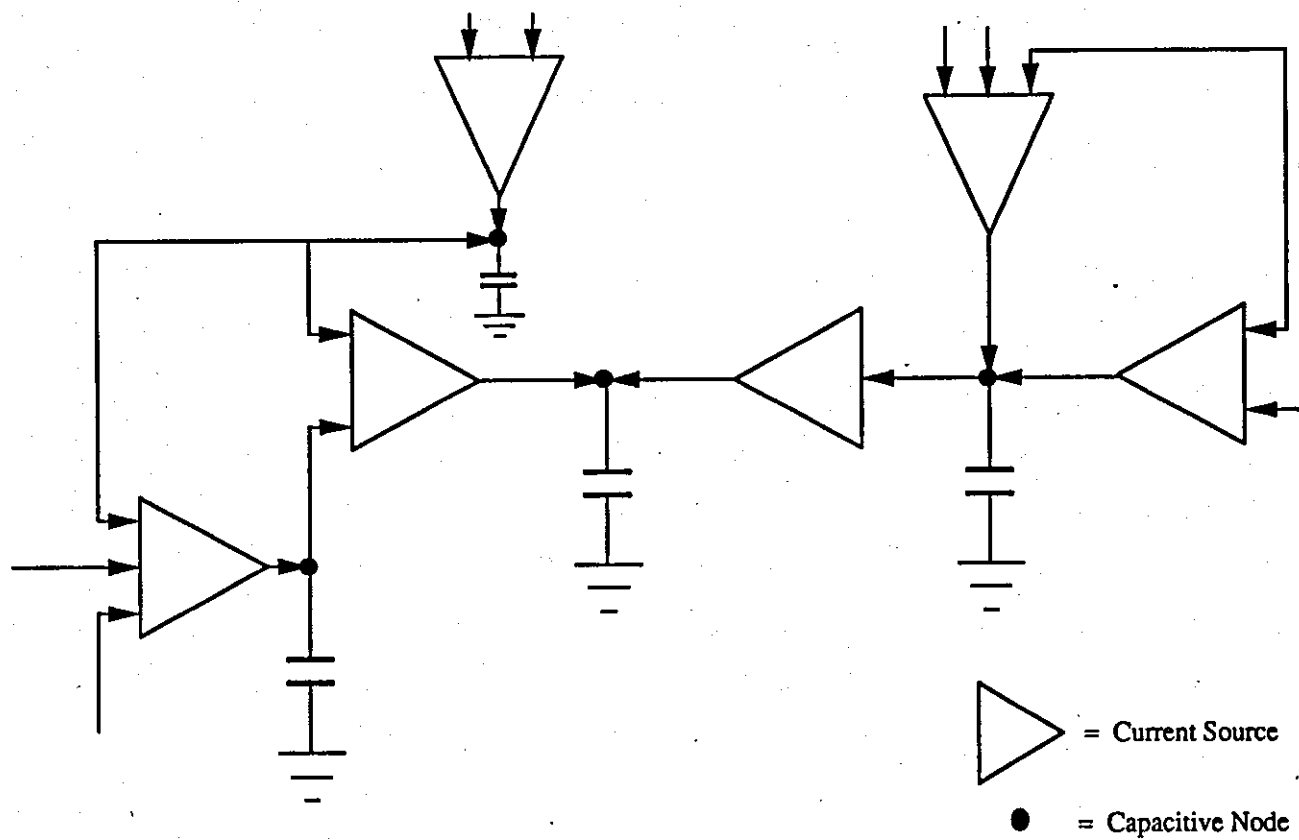


Fig. 3.2 An MOS circuit model. Adapted from [5].

EMU has two different operating modes. In *circuit* mode, the time step for the whole circuit is adjusted if the activity of any node exceeds the threshold. This guarantees accuracy but means the whole circuit has to be simulated in minute detail even if only a few nodes are active. In *node* mode, each node's own activity determines the size of its simulation time step. This method provides certain speed-up but is prone to numerical instability especially when the behaviour of one node is closely linked to that of another.

Circuit mode simulation is favoured by VLSI designers because of its improved stability. But, in a multiprocessing environment circuit mode simulation is amenable to a number of problems. It requires a global knowledge of the current value of the time step. This further requires that all processors be exactly synchronised in terms of simulation time. Also, it requires large volume of interprocessor data transfer to update node voltages during high circuit activity. Dividing the circuit into some loosely coupled regions would help to alleviate these problems. Time step sub-division can thus be performed locally within regions without sacrificing accuracy or succumbing to numerical stability.

For the purpose of circuit sub-division, two circuit nodes are considered tightly connected when they are joined by a bidirectional component. In other words, in a transistor only model, two circuit nodes joined by a pass transistor can be placed in the same region. The power supply nodes V_{SS} and V_{DD} can not be affected by any other node and as such they belong to no region.

An example of region sub-division is shown in Fig. 3.3. Region boundaries are only crossed by nodes driving transistor gate inputs. A sub-divided MOS circuit can be modelled as a directed acyclic graph (dag) where the nodes (vertices) of the graph can be used to represent the circuit regions and edges of the graph can represent the electrical connection between the regions of the circuit. A node (vertex) weight of the dag can be assigned proportional to the simulated electrical activity and an edge weight can also be assigned proportional to the voltage value transferred. The resulting dag is thus an activity on vertex (AOV) graph.

3.2.1 VLSI Circuit Partitioning Algorithm

The algorithm [5] used to sub-divide an MOS circuit into regions where each region can be simulated independently on a processor can be described as follows:

Regions are identified by constructing directed tree graphs linking all the nodes in a region. For each node i , a pointer P_i is defined and then if possible P_i is made to point to another node in the same region whose node number is less than that of i . From each region a root is identified whose pointer is set to NULL. The node with the lowest node number in the region is the root node.

The algorithm starts with allocating exactly one node, the root node, to each region and holding all the nodes of circuit disjoint. For each transistor the source and drain nodes s and d are identified respectively. These nodes are placed in regions R_s and R_d respectively. The root nodes of these regions r_s and r_d are then determined by simply following the linked pointers through to the end of the list. Then whichever root node has the higher node number, its pointer is set to point to the other root node.

Once all transistors have been processed, each node belongs to a region characterised by its root node number. This root node number can be easily determined for each node by following the linked pointers through to the end of the list.

3.3 The Graph Model

In this section we define a *graph model* designed to support automatic scheduling for the concurrent VLSI timing simulation modules onto a multiprocessor. As outlined in the preceding section, each module of the simulation graph can be thought of as a separate computing *task*. However, it is to be borne in mind that the partitioning of the MOS VLSI circuit into regions which can be simulated concurrently is more of a *data decomposition* than the decomposition of the simulation program itself. Also, when these regions (nodes of the concurrent simulation graph) are scheduled onto a multiprocessor and submitted to run, essentially the same timing simulation

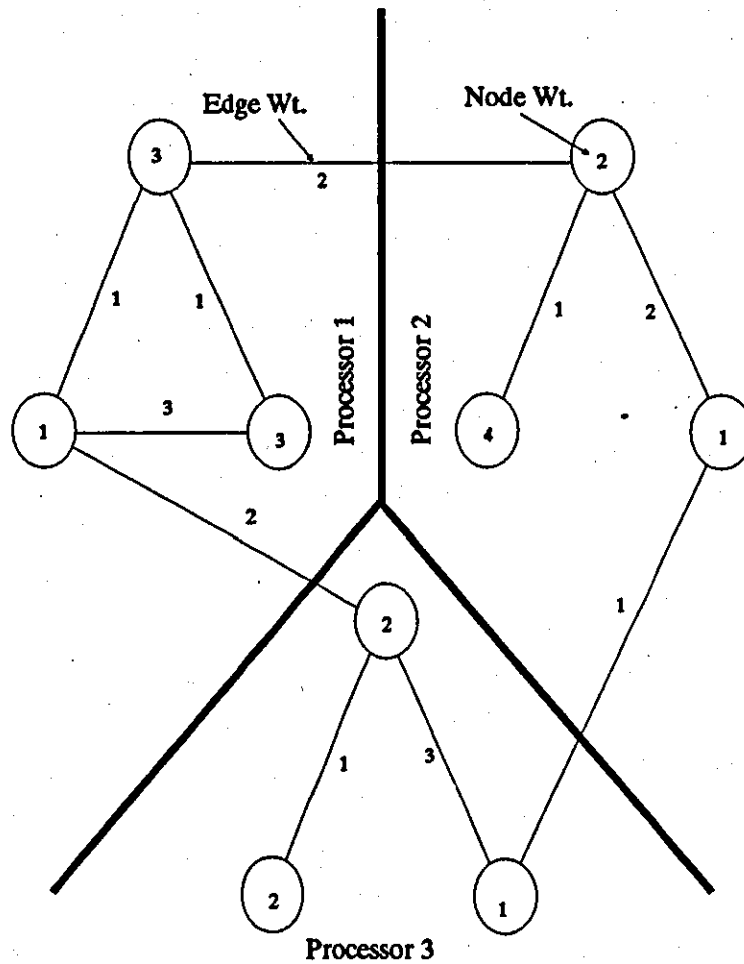


Fig. 3.4 Graph representation of a 9-node, 3-processor task system.

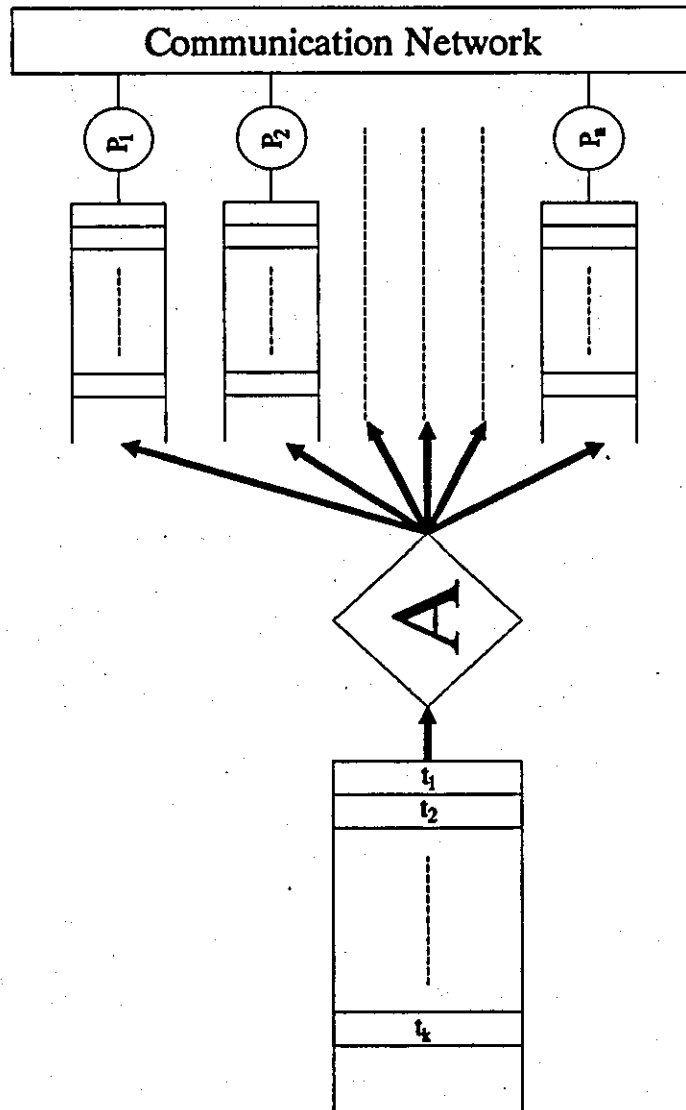


Fig. 3.5 A generic multiprocessor system.

program runs on each processor albeit asynchronously depending on the nature of the circuit regions. Nevertheless, for the scheduling purpose it is quite safe to treat each node of the simulation graph as a separate computing task.

In the task scheduling problem, we consider the *task system* as a set of k task modules, $T = \{t_1, t_2, \dots, t_k\}$ and a set of n processors, $P = \{p_1, p_2, \dots, p_n\}$ with execution costs x_{iq} , $1 \leq i \leq k$, $1 \leq q \leq n$ and communication costs c_{ij} , $1 \leq i, j \leq k$. The cost of a schedule can be defined as the total sum of the *execution* and *interprocessor communication* (IPC) costs as incurred by the schedule. An optimal schedule results in the minimum cost and there may be more than one optimal schedule for a particular task system.

For the scheduling problem a weighted and directed network flow graph is a suitable problem representation. We consider such a graph $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_k\}$ are the weighted nodes of the graph representing the task modules of the simulation graph and $E = \{e_{ij}, 1 \leq i, j \leq k\}$ are the weighted edges between nodes representing the *intermodule communication* (IMC) costs. Comparing this model with task system $T = \{t_1, t_2, \dots, t_k\}$, $P = \{p_1, p_2, \dots, p_n\}$ we find that except for the execution costs x_{iq} (cost of executing task module i on processor q) there exists a straight mapping of the task system onto the graph model $G = (V, E)$. With a homogeneous multiprocessor system with identical processors which lends itself to present day technology, the execution cost x_{iq} of any task module i on any processor q is the same for every processor in the ensemble. In such circumstances, the weight of the nodes of the graph $G = (V, E)$ can represent the execution costs or the execution complexity of the task modules. Fig. 3.4 shows a 9-node task graph scheduled onto a 3-processor system.

3.4 Communication and Computation

Multiprocessing enhances system performance by employing several processors to handle the processing load. A representation of a generic multiprocessing system is shown in Fig. 3.5. The main elements are a set of task modules $\{t_1, t_2, \dots, t_k\}$ and a module allocation mechanism A , which

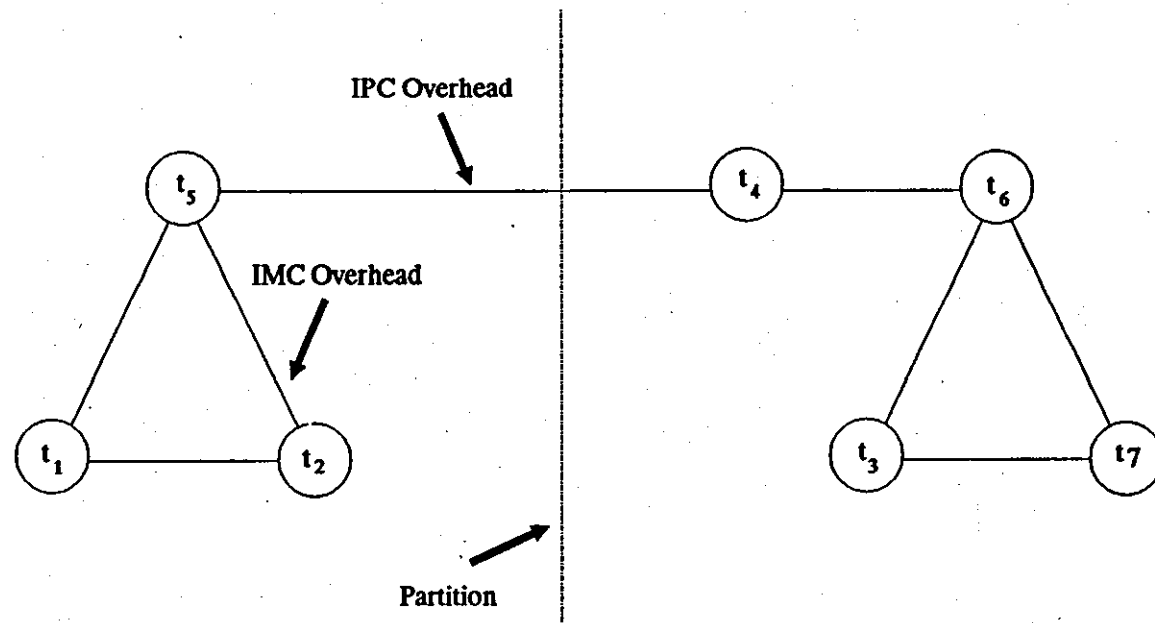


Fig.3.6 Intermodule communication cost (IMC) across a partition giving rise to interprocessor communication cost (IPC) overhead.

assigns each of the k modules to one of the n processors $\{p_1, p_2, \dots, p_n\}$. Here we consider each module as a sub-task of a single processing job. These modules may need to transfer data among themselves giving rise to intermodule communication (IMC) costs. Modules may be assigned to different processors. When modules may have data to communicate to one another, the processors to which they are assigned must then communicate with each other. This constitutes the *interprocessor communication* (IPC) overhead. IPC, is therefore, a function of IMC and the module assignment. Clearly $IPC = IMC$ where the communicating modules are not coresident.

Assuming independent processing modules, the most intuitive maximum throughput allocation strategy is to assign modules to processors so that all processors in the system are evenly balanced. Such a balanced load assignment strategy is shown in Fig. 3.7. We consider six non-communicating modules to be assigned to a set of three identical processors and also that the processing demands for all the six modules are identical.

We now consider introducing the IMC profile information in the above system. If we then attempt to maximise the system throughput by minimising IPC without considering load balancing, all modules will be assigned to the same processor. The resulting scheduling would provide minimum IPC overhead but the processing time for the job would increase by a factor of three. This situation is shown in Fig. 3.8.

It is clear from the above that the two conflicting factors, namely IPC and load balancing, influence the design of an optimal schedule. The task scheduling problem is to assign modules to processors for maximum system performance by balancing these two conflicting factors.

3.5 Formulation of the Cost Function

Continuing with the graph model it is easy to visualise that the process of scheduling a set of VLSI simulation modules onto a n -processor MIMD system is simply the partitioning of the simulation graph itself into n non-empty disjoint sub-graphs and then to allocate each of these sub-graphs to

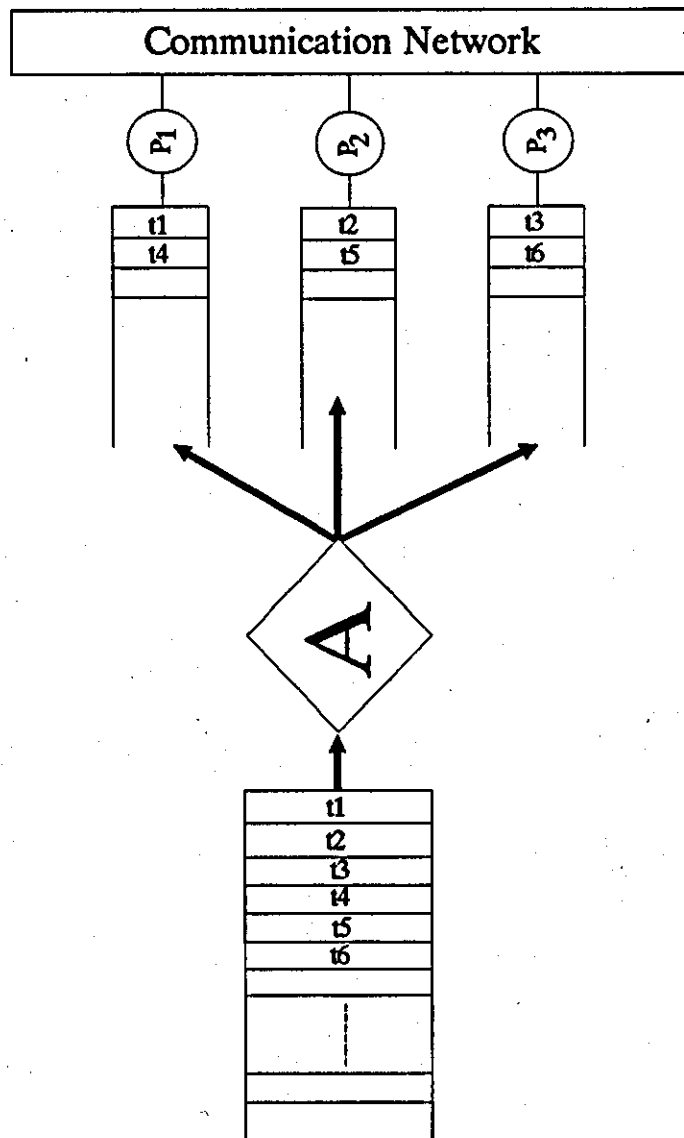


Fig. 3.7 A load balanced assignment strategy.

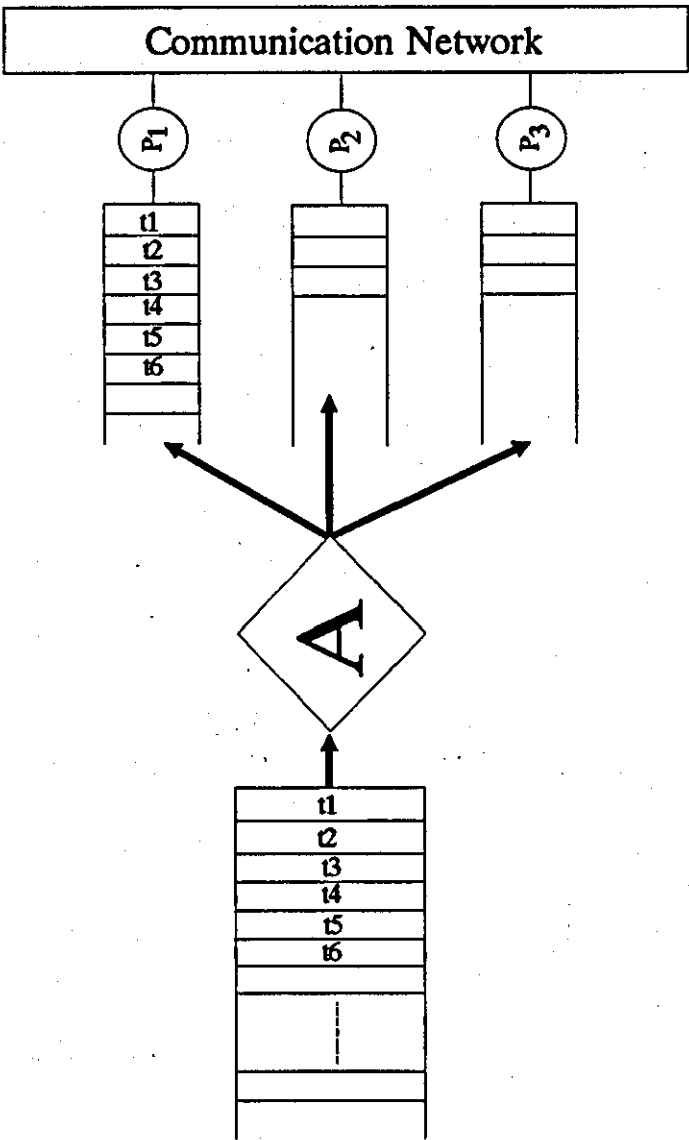


Fig.3.8 A minimum IPCoverhead schedule.

a separate processor. So for n -way partitioning of the graph $G = (V, E)$, we have

$$\bigcup_{i=1}^n g_i = G \quad (3.2)$$

where g_1, g_2, \dots, g_n are the resulting non-empty disjoint sub-graphs.

The *cutset* of the partition is the sum of all the weighted edges with nodes in more than one sub-graph and accounts for the *communication cost* in the final schedule. The *load imbalance* is the maximum difference between the total weights of any two sub-graphs, thus accounting for the *completion time* of the final schedule.

To formulate an equation for the total cost of a partition (i.e., that of the schedule) we identify the following two as the contributing factors :

- a. Interprocessor Communication (IPC) Cost
- b. Cost due to load imbalance

and then proceed as follows.

Let, there be k nodes in the simulation graph and n identical processors in the multiprocessor system, where as we take a coarse grain model $n \ll k$. if the execution cost of node i is $w_i (= |v_i|)$, then we calculate the average load on each processor as,

$$Load_{av} = \frac{1}{n} \sum_{i=1}^n w_i \quad (3.3)$$

Clearly this is the expected load on each processor for a perfectly load balanced schedule.

When a partition x is allocated to processor q , then total execution load on that processor is,

$$\begin{aligned} Load_q &= \sum_{v_i \in g_x} |v_i| \quad ; \quad \forall v_i \in G \\ &= \sum_{v_i \in g_x} w_i \end{aligned} \quad (3.4)$$

Borrowing the definition of load imbalance from the graph partition analogy, we find the load imbalance cost of the schedule,

$$\begin{aligned} C_b &= \max_{s=1 \dots k} \left(\sum_{v_i \in g_s} |v_i| \right) - \min_{s=1 \dots k} \left(\sum_{v_i \in g_s} |v_i| \right) \quad ; \quad \forall v_i \in G \\ &= \max_{s=1 \dots k} \left(\sum_{v_i \in g_s} w_i \right) - \min_{s=1 \dots k} \left(\sum_{v_i \in g_s} w_i \right) \end{aligned} \quad (3.5)$$

To calculate the total communication cost of the schedule we note that only those edges which are in the cutset of a partition, in other words only those edges between nodes assigned to different processors contribute to the communication cost. If, e_{ij} represents the intermodule communication cost, a measure of data transmission from node i to node j when $i \neq j$, then for C_c , the total communication cost we have,

$$C_c = \sum_{i,j=1}^k e_{ij} \quad v_i \in g_p; \quad v_j \in g_q; \quad p \neq q. \quad (3.6)$$

An overall cost function can then be formulated by combining the two components

$$C_t = C_b + z.C_c, \quad (3.7)$$

where z is a relative weight factor between costs due to communication between nodes of the graph across a partition and the load imbalance in the schedule. The value of z is thought to be dependent on the simulation graph itself and also on the hardware interconnection topology of the multiprocessor system. A large value of z may make the multiprocessor implementation of VLSI simulation very difficult and a small value of z may demand for algorithms utilising fine grain parallelism as a means for improved load balance in the system [6].

3.6 Graph Data Storage & Cost Calculation

In this section we introduce three sub-sections. In sub-section 3.6.1 we discuss the data storage method used to represent the graph information in the computer programs. Sub-section 3.6.2 describes the methods used to calculate the communication and load-imbalance costs. In sub-section 3.6.3

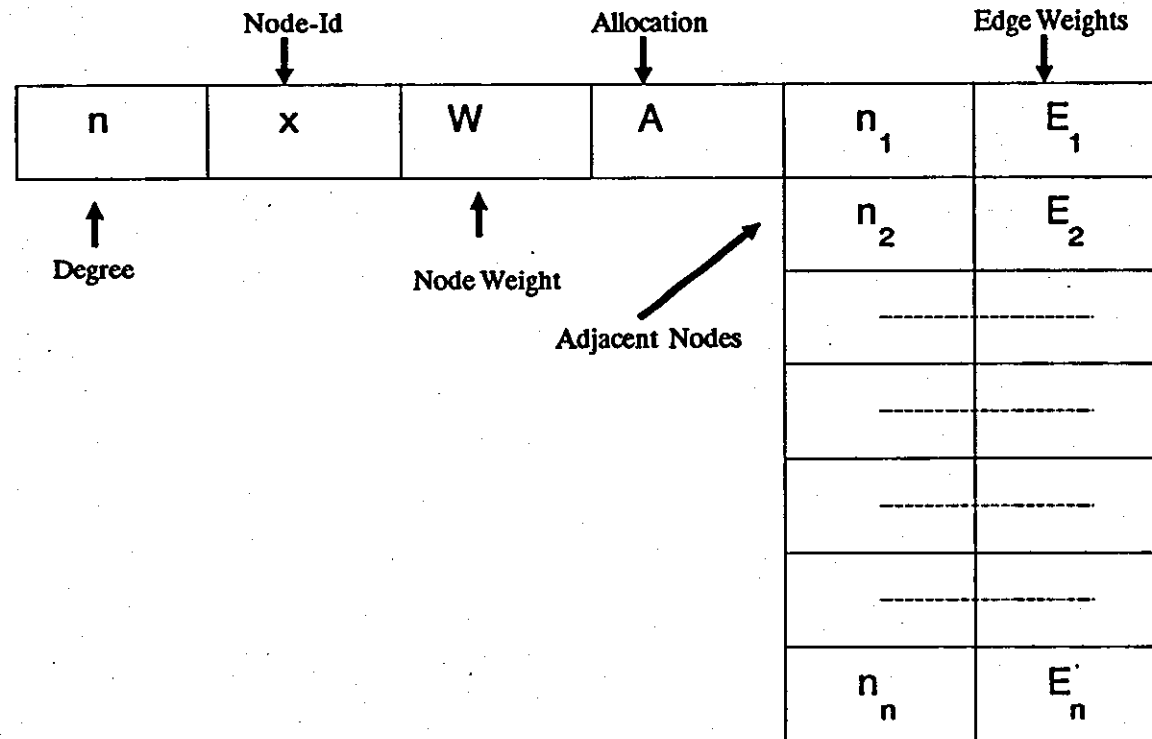
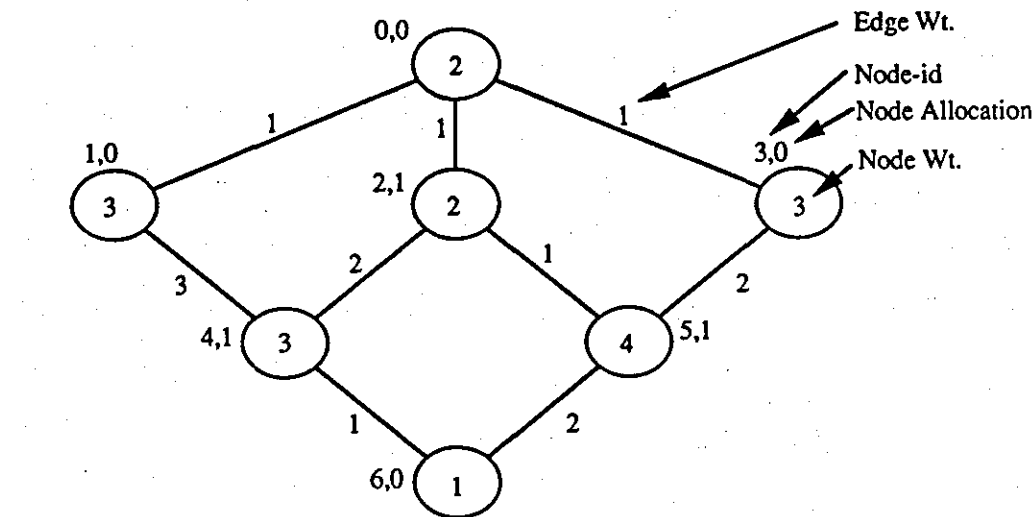


Fig. 3.9 Structure of a single record used for graph data storage.



Degrec	Node-id	Node Wt	Allocation	Adjacent Node 1		Adjacent Node 2		Adjacent Node 3	
				Node-id	Edge-wt	Node-id	Edge-wt	Node-id	Edge-wt
3	0	2	0	1	1	2	1	3	1
2	1	3	0	0	1	4	3	-	-
3	2	2	1	0	1	4	2	5	1
2	3	3	0	0	1	5	2	-	-
3	4	3	1	1	3	2	2	6	1
3	5	4	1	2	1	3	2	6	2
2	6	1	0	4	1	5	2	-	-

Fig. 3.10 A typical graph and the corresponding data storage table.

we describe an improved method of cost calculations based on change in the configuration of a task system in an iterative improvement environment.

3.6.1 Graph Data Storage

Since graphs are selected as the chosen data representation of the VLSI simulation system, a suitable and efficient data storage method is necessary for the fast simulation of their scheduling algorithms. There are many ways to do this, and the subject is by no means trivial, since certain methods of graph data storage are particularly efficient for certain kinds of calculations. The three most common graph data storage methods are,

- a. The Branch-List Method
- b. The Adjacency-List Method
- c. The Adjacency-Matrix Method.

For the simulation of the task scheduling algorithms to be discussed in later chapters, we have selected the *adjacency-list* method of data storage. This method allows faster search than the *branch-list* method. Another consideration is that it takes much less storage than the *adjacency-matrix* method.

A node i in a graph is defined as *adjacent* to another node j in the same graph if there exists an edge between nodes i and j . An adjacency-list method of graph data storage is essentially a one dimensional array of a *record* each describing a separate node of the graph. The first *field* of such a record describes the degree of the node which is defined as the total number of incoming and outgoing edges at the node and the other field is a list of all its adjacent nodes. For our purpose, we however extend the above rudimentary data storage scheme by introducing some more additional fields to the record which gives the node identification number, node weight, node allocation and in the list of connected nodes an additional field to each of the entry which gives the weight of the edge concerned. Fig. 3.9 shows the structure of such an enhanced record. In Fig. 3.10 we show a graph with an arbitrary allocation and the corresponding data storage table.

```

Function Imbalance_Cost : Integer;
    Initialise the processor load list;
    For i := 0 To Max_Node Do
        Processor_No := Find_Allocation(Graph[i].node_id);
        Load[Processor_No] := Load[Processor_No] +
            Graph[i].node_id;
    EndFor;
    Find the processors with the heaviest (Heavy) & lightest (Light)
    loads;
    Imbalance_Cost := Heavy - Light;
EndFunction;

Function Communication_Cost : Integer;
    C := 0;
    For i := 0 To Max_Node Do
        Present_Node := Graph[i].node_id;
        For j := 1 To Degree(Present_Node) Do
            Adjacent_Node := Graph[Present_Node].adjacent[j];
            Are Present_Node & Adjacent_Node co-resident ?;
            If Not co-resident Then
                C := C + Graph[Present_Node].Edge_Wt[j];
            EndIf;
        EndFor;
    EndFor;
    Communication_Cost := C Div 2;
EndFunction;

```

Fig. 3.11 Pseudo-Pascal description of the two cost evaluating functions.

```
Procedure Iterative_Improvement;  
  Get starting configuration;  
  Present_Configuration := Start_Configuration;  
  Repeat  
    New_Configuration := Move(Present_Configuration);  
    Determine Diff_Cost;  
    If Diff_Cost < 0 Then  
      Present_Configuration := New_Configuration;  
    EndIf;  
  Until Solution Is Frozen;  
EndProcedure;
```

Fig. 3.12 Pseudo-Pascal description of an iterative improvement algorithm.

3.6.2 Cost Calculation

In an algorithm that tries to minimise an objective or cost function the evaluation of the function itself at different moments during the running of the algorithm is a regular phenomenon. If the evaluation procedure itself is too mathematically involved, a far too long time would then be spent for function evaluation. A fast evaluation is thus always favoured.

Fortunately in the model used for the task scheduling, the mathematical expressions for the cost function is a simple one. Equations 3.5 and 3.6 give the algebraic expression for the two components of the cost function given in eq. 3.7. A simple computer program translation of eqs. 3.5 and 3.6 is easily possible. The data storage method used also helps in fast evaluation of these equations. Figs. 3.11a and 3.11b show the Pseudo-Pascal representation of the functions used to evaluate eqs. 3.5 and 3.6 respectively. For the evaluation of the communication cost, a simple scanning through each and every node's adjacency list and also obtaining a cumulative sum of the edge weights for non-coresident nodes is all that is necessary. This sum however needs to be halved as each edge is traversed exactly twice. Similarly, for the process of evaluating imbalance cost we need to produce a list of processor loads and then to scan through it to find the most heavily and lightly loaded processors. Their difference would then give the imbalance cost, a measure of the completion time of the execution for that particular schedule.

3.6.3 Cost Calculation in Iterative Improvement Environment

Both the components of the cost function can be evaluated in linear time and thus appear to be very promising. However, a heuristic algorithm that employs an iterative improvement, technique even a linear time cost calculation may appear to be too high. An iterative improvement type of algorithm usually starts with a random initial allocation (or any other initial allocation provided by a pre-processor) and then iteratively improves it by bringing in small local changes at each iteration to the current configuration. In the most simplest form this is achieved either by changing the allocation of any chosen node to another processor or by swapping the allocations of any two chosen nodes. This simple task can be thought of as a unit task and we

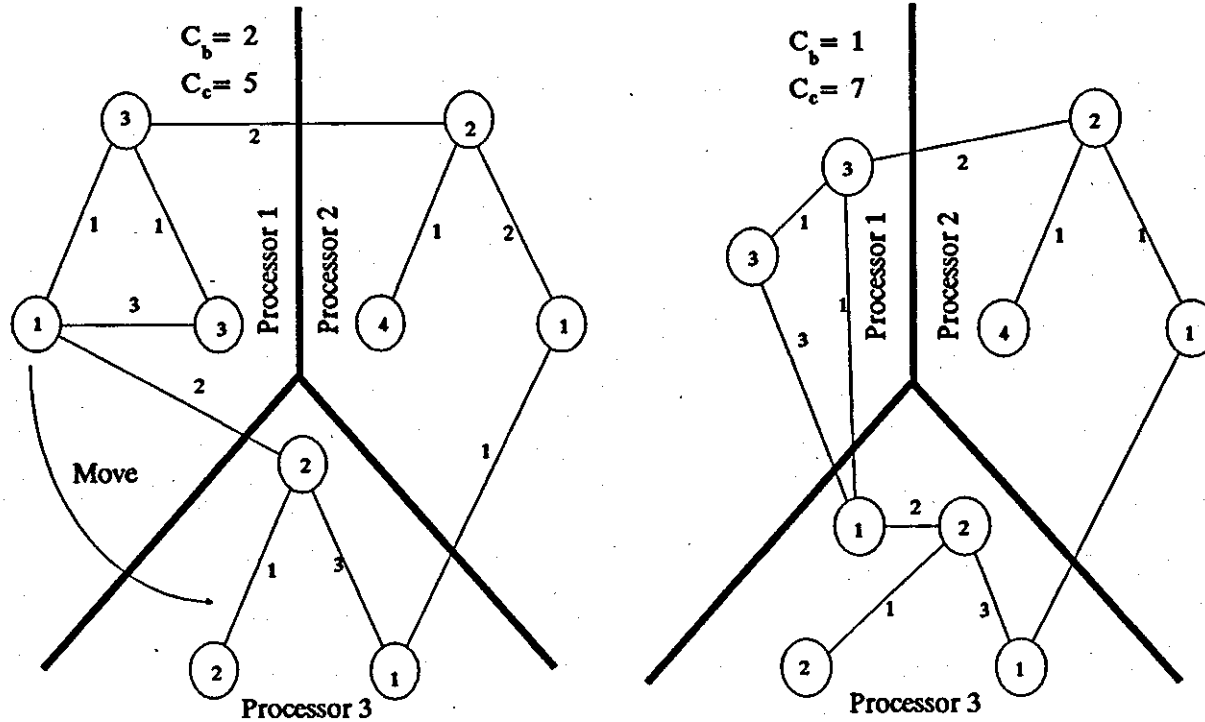


Fig. 3.13 A 'move' during an iterative improvement procedure. Configurations before (left) and after (right) the move.

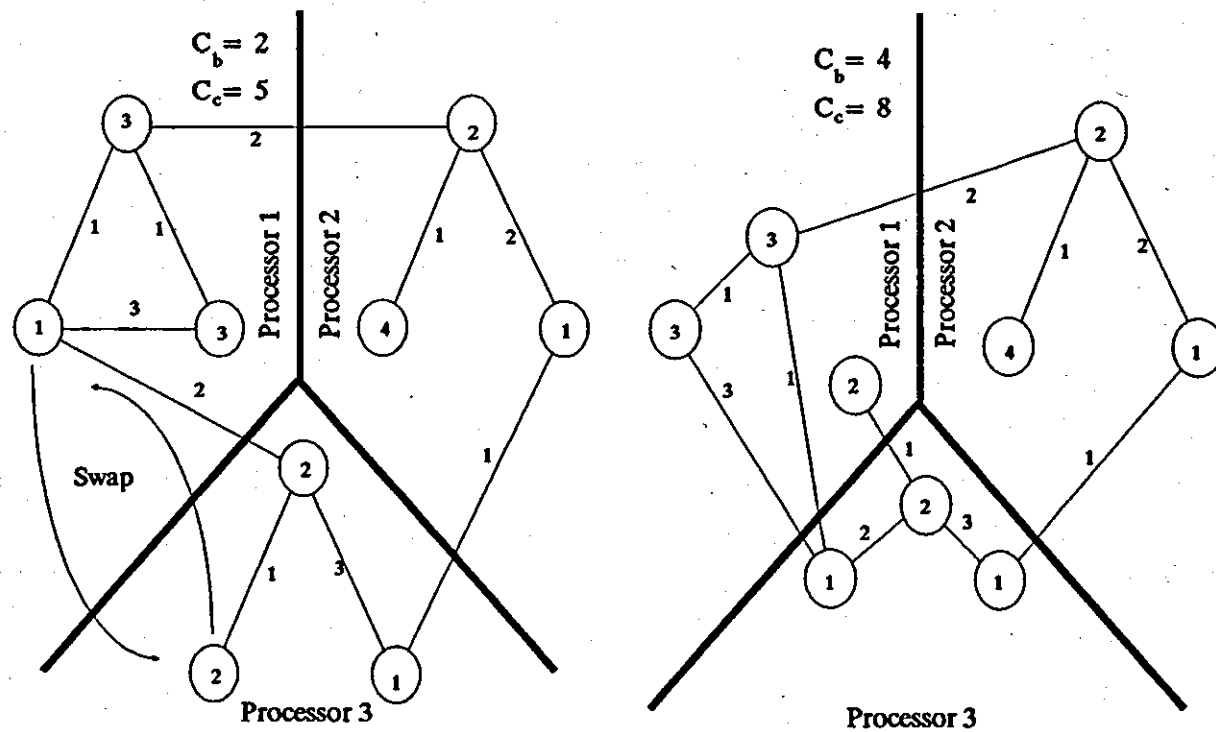


Fig. 3.14 A 'swap' during an iterative improvement procedure. Configurations before (left) and after (right) the swap.

could call this a *move* or *swap*. A Pseudo-Pascal description of an iterative improvement algorithm is shown in Fig. 3.12. A graphical representation of a single move and also a swap and their resulting consequence is shown in Figs. 3.13 and 3.14 respectively. Such moves are repeatedly attempted until the solution is *frozen*. The criterion for the determination of the freezing of the solution can be formulated in many different ways. But, whatever the criterion is used, a large number of iterations is usually needed. This large number of iterations compounded with the linear time demand of the cost evaluation at each iteration may become a serious bottleneck especially for larger graphs.

Looking back into the Pseudo-Pascal description of the iterative improvement method, we find that the algorithm itself is more concerned with the difference in cost after a move is made than the cost of each new and changed configuration to be evaluated afresh. In each move, whether it is a single move or a swap, only the nodes and the processors to which they are allocated are involved in any change in the current configuration. This fact can be usefully utilised to evaluate the change in cost in a constant time. In all subsequent discussions the differential cost calculation is always used.

References :

1. Horowitz, E. and Shani, S., *Fundamentals of Data Structures*, Computer Science Press Inc., 1976.
2. Aho, A.V., Sethi, R. and Ullman, J.D., *Compiler Principles, Techniques and Tools*, Addison Wesley, 1978.
3. Gurd, J.R., Kirkham, C.C. and Watson, J., *The Manchester Prototype Data Flow Computer*, Comm. of the ACM, 28 (1), Jan 1985.
4. Shaw, A.C., *The Logical Design of Operating Systems*, Prentice-Hall Inc., N.J., USA, 1974.
5. Ackland, B., Ahuja, S. and Romero, D.J., *A Partitioning Algorithm for Concurrent Timing Simulation*, AT&T Bell Lab. Tech. Report, 1984.
6. Sheild, J., *Partitioning concurrent VLSI simulation programs onto a multiprocessor by simulated annealing*, IEE Proc. Vol. 134, Pt. E, No. 1, Jan 1987.

CHAPTER 4

Graph Partitioning

This chapter describes the partitioning of a graph with costs on its edges and nodes into a given number of disjoint non-empty sub-graphs so as to minimise the sum of the costs on all edges cut and also to balance the sum of the costs of nodes in each sub-graph. This problem arises in parallel processing applications where it is required to assign a large number of processing jobs to a fixed (relatively smaller) number of processors, and also in VLSI design applications such as component layout. The multi-way partitioning algorithm presented here is adapted from the graph *bi-partitioning* procedure due to Kernighan and Lin [1]. The adapted algorithm is hierarchical in nature and lends itself for an easy implementation on a hypercube multiprocessor system.

It is known that graph and network partitioning problems are NP- Hard [2]. Therefore, attempts to solve these problems have concentrated on finding heuristics which will yield approximate solutions in polynomial time. Several

different approaches have been taken to devise approximate algorithms for the graph partitioning problems [1,3,4]. The heuristic proposed by Kernighan and Lin [1] is easy to implement for general purpose partitioning problems and provides good quality near-optimal solutions in a relatively faster time of $O(k^2 \log k)$ for k nodes in the graph. It has also become the basis for most of the iterative improvement partitioning algorithms generally used.

4.1 Graph Bi-Partitioning

Graph partitioning arises naturally in scheduling concurrently executable task modules onto a multiprocessor. For example, in Stone's [5] *max-flow, min-cut* assignment algorithm which provides an optimal assignment of modules to processors. The graph representing the task system is cut in such a way that the number of edges cut is minimal.

The 2-way graph partitioning problem can be described as follows : Given an arbitrary graph G with k weighted nodes the graph must be partitioned into two disjoint and non-empty sub-graphs or blocks. The objective behind this partitioning is to minimise the number of edges cut across the partition and also to maintain a rough balance on the total weights of each block. Let the number of nodes in the two blocks are k_1 and k_2 , such that $k = k_1 + k_2$.

For a task system with k task modules and two processors, the total number of possible schedules is given by,

$$Total = \frac{k!}{2 k_1! k_2!} \quad 4.1$$

For large k the expression above would result in a very large number indeed and there is no computationally efficient algorithm to arrive at an optimal schedule. Kernighan-Lin's iterative procedure however produces near-optimal solution in reasonable polynomial time.

The Kernighan-Lin (KL) algorithm starts with an arbitrary partition with two equal sized blocks A and B and then repeatedly improves the partition to obtain a near-optimal solution. The essential idea behind the KL heuristic is the assumption that there are some nodes in block A and an equal number of nodes in block B that are *out of place* in the sense that if

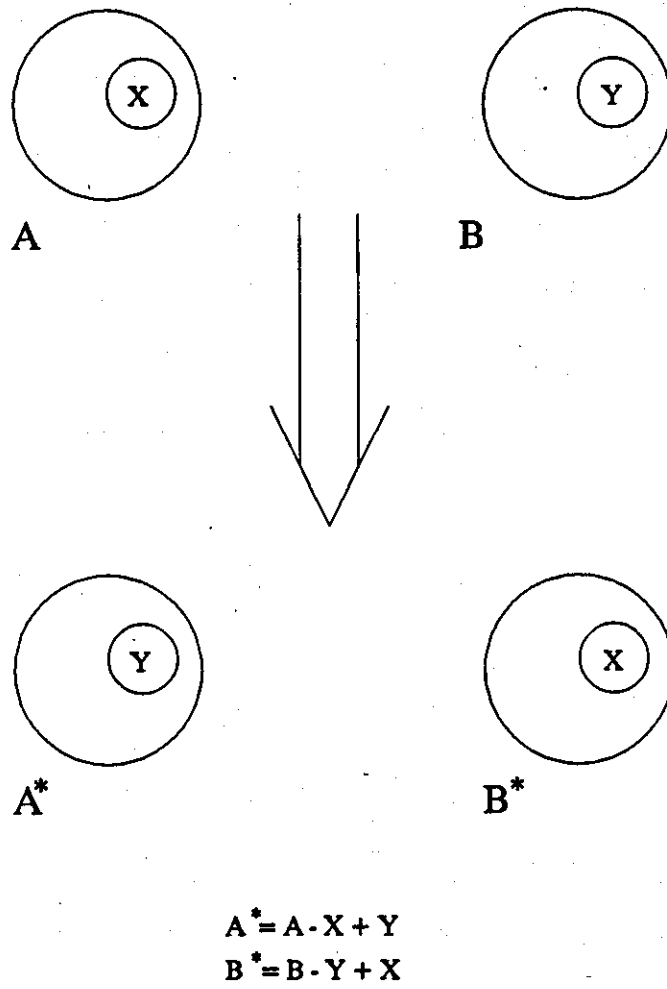


Fig. 4.1 Generating an optimal 2-way partition from an arbitrary 2-way partition.

these are interchanged, the resulting partition would improve. The algorithm spends bulk of its time finding these out of place nodes. The KL algorithm in its original form works with graphs with unity weighted even number of nodes such that the bi-partition splits the graph evenly minimising only the number of edges cut. However, the graph representing the task system as used in the multiprocessor task scheduling problem have nodes with different weight values and also the partitioning criterion in such cases place equal emphasis on minimising the number of edges cut and maintaining a balance on the total weights of the two resulting blocks (sub-graphs). This essentially does not result in equal sized blocks. Slight modification to the original KL algorithm is thus in order. In the following sub-section the modified KL heuristic is described and associated formulae are derived.

4.1.1 The Modified KL Bi-Partitioning Heuristic

We consider a graph $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_k\}$ are the k weighted nodes with the weights represented by a weight matrix $W = (w_i = |v_i|)$, $i = 1, \dots, k$. The edges are represented by $E = \{e_{ij}, 1 \leq i, j \leq k\}$. We assume that the edge weights are non-negative and also we rule out any loop at any node, i.e., $e_{ii} = 0$. We wish to partition the graph V into two non-empty disjoint blocks A and B such that the sum of weights of the edges across the partition is minimal and also that the total weights of the two blocks A and B are evenly matched. We thus have,

$$V = A \cup B. \quad 4.2$$

The partition cost due to mismatch of total weights of blocks A and B ,

$$C_b = \sum_{v_i \in A} w_i - \sum_{v_i \in B} w_i \quad 4.3$$

and that due to the edges cut in the partition,

$$C_c = \sum_{i,j=1}^k e_{ij} \quad ; \quad v_i \in A, v_j \in B \quad 4.4$$

The 2-way graph partitioning problem can thus be transformed to the following optimisation problem,

$$\text{minimise } \{\alpha|C_c + \beta|C_b\}$$

subject to

$$C_b = \sum_{v_i \in A} w_i - \sum_{v_i \in B} w_i$$

and

$$C_c = \sum_{i,j=1}^k e_{ij} \quad ; \quad v_i \in A, v_j \in B.$$

Central to any iterative improvement heuristic the Kernighan-Lin (KL) heuristic starts with an arbitrary 2-way partition A and B so that,

$$A \cap B = \{\}. \quad 4.5$$

Kernighan and Lin suggested that there exists two sub-sets $X \subset A$ and $Y \subset B$ with $|X| \leq |A|$ and $|Y| \leq |B|$, which if interchanged would produce the minimum cost 2-way partition as shown in Fig. 4.1. These two sub-sets are built by repeatedly choosing two nodes, one each from A and B , so that their interchange would produce best gain in the cost and then separating them from A and B for not to be used again in that iteration until all the nodes in the graph are similarly used. A single such iteration constitutes a *pass* and in each pass a pair of lists is maintained one for each sub-set (block) in decreasing order of the gain values associated with each node chosen for an interchange. At the end of a pass an equal number of nodes from A and B which constitute X and Y respectively are actually interchanged such that maximum gain in partition cost is achieved. The resulting A^* and B^* are then renamed to A and B respectively and the whole procedure is repeated once again until no further gain in partition cost can be obtained. The final partition is then taken as the optimal (near optimal to be precise) solution.

One drawback of the above procedure is that it fails to address the essential load-balance criterion of a partition. The modification presented here is similar to that proposed by Fiducia and Matheyses [6] and works with moving only one node at a time from one block to the other instead

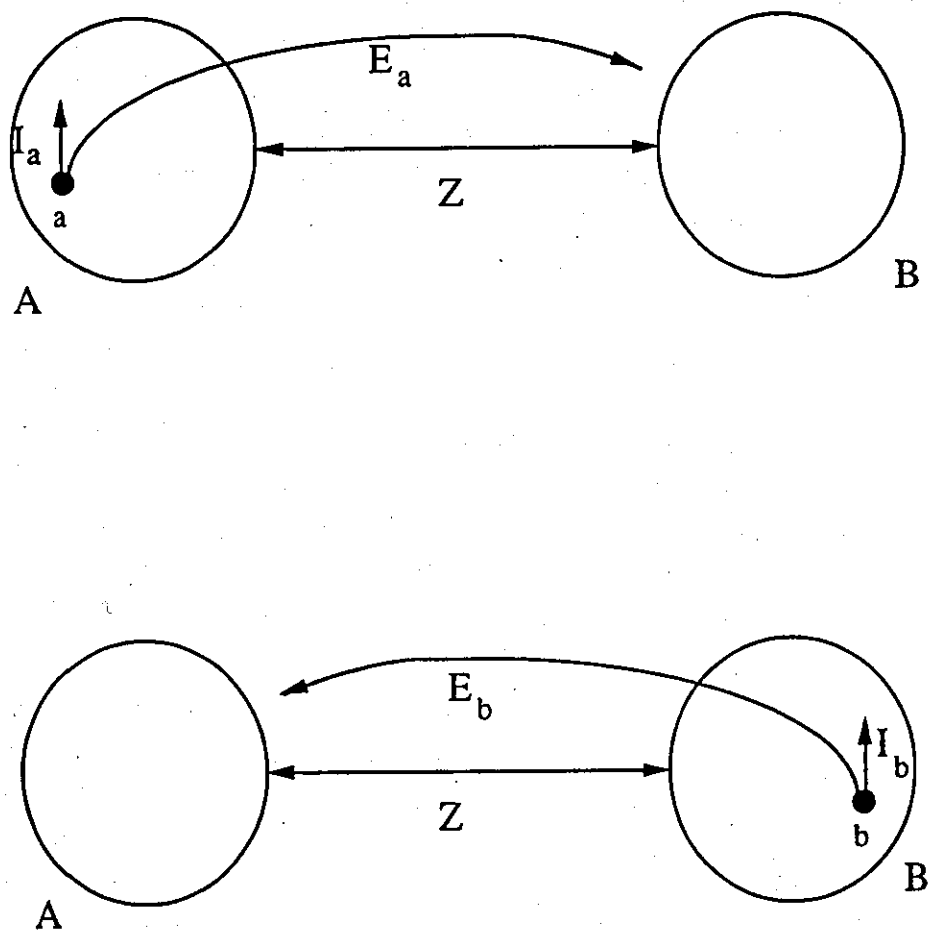


Fig. 4.2 The internal and external cost components of node $a \in A$ (top) and node $b \in B$ (bottom).

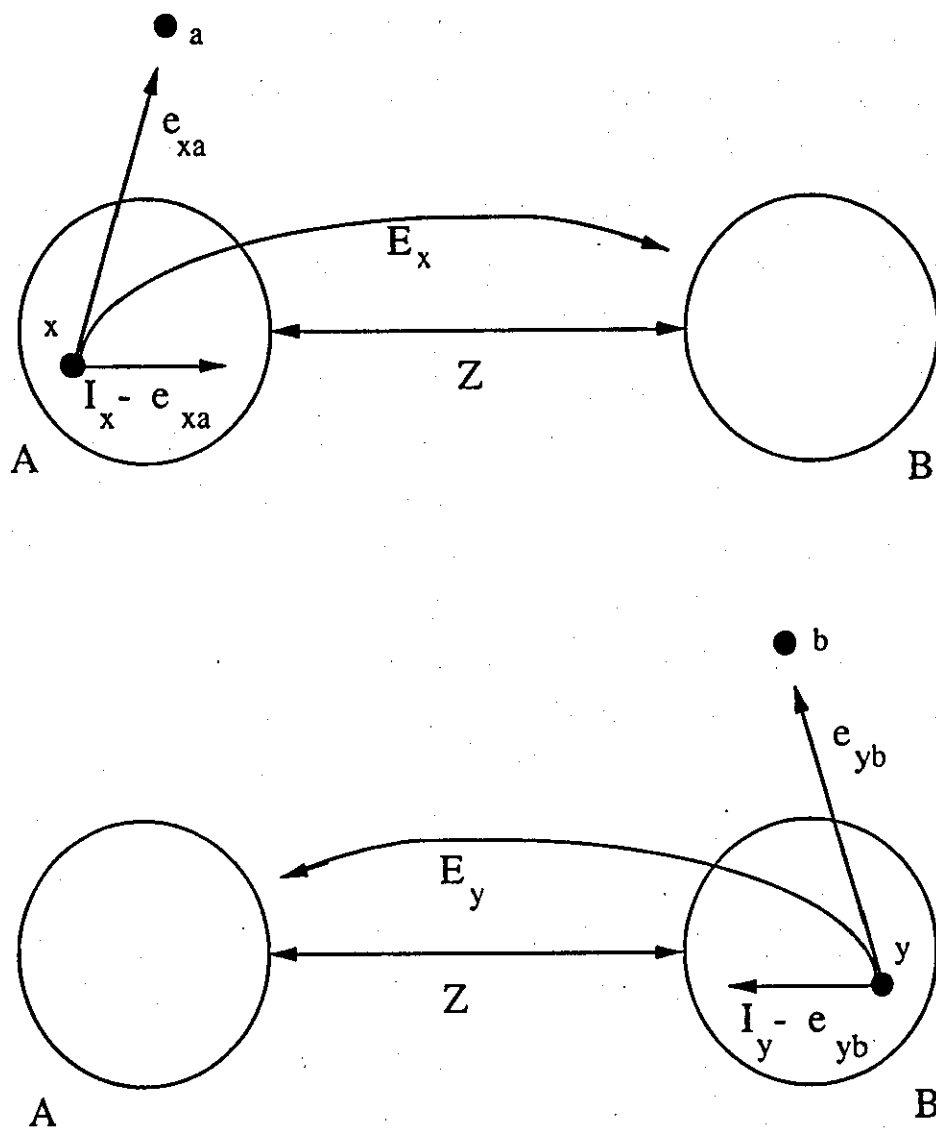


Fig. 4.3 The internal and external cost components of node $x \in A - \{a\}$, after node a is removed (top) and node $y \in B - \{b\}$, after node b is removed (bottom).

of an interchange of two nodes. This provides far more flexibility in size of the blocks and can effectively incorporate the load-balance criterion. To derive the formulae for computing and updating the gain values we proceed as follows :

Let us define for each node $a \in A$ external and internal costs E_a and I_a respectively as follows,

$$E_a = \sum_{y \in B} e_{ay} \quad 4.6a$$

and

$$I_a = \sum_{x \in A} e_{ax} \quad 4.6b$$

and similarly for each node $b \in B$,

$$E_b = \sum_{x \in A} e_{bx} \quad 4.6c$$

and

$$I_b = \sum_{y \in B} e_{by} \quad 4.6d$$

The internal cost of a node signify how strongly it is connected to all other nodes in the same block and the external cost signifies its contribution to the communication component of the partition cost.

We now let,

T = total cost due to all external connections between the blocks of a 2-way partition.

Z = total cost due to all connections between A and B that do not involve $a \in A$.

Therefore,

$$T = Z + E_a \quad 4.7$$

When node $a \in A$ is moved from A to B , the value of T is changed to T' .

$$T' = Z + I_a \quad 4.8$$

```

Procedure 2-Way-Partition ( $A, B$ );
  Repeat
     $X := A; \quad Y := B;$ 
    Compute the  $D$  values for all  $a \in A$  and for all  $b \in B$ ;
    For  $i := 1$  To  $n$  Do
      Find a node  $Z_i$  from either  $X$  or  $Y$  that maximises  $g$ ;
      Move  $Z_i$  from its current block to the other;
      Remove  $Z_i$  from further consideration in this pass;
      Update  $D$  values for all nodes in  $X - \{Z_i\}$  or  $Y - \{Z_i\}$ ;
    EndFor;
    Find  $K$  that maximises  $g_{max} = \sum_{i=1}^K g_i$ ;
    If  $g_{max} > 0$  Then
      Move all  $Z_1, Z_2, \dots, Z_K$  from  $A$  or  $B$  to the other
      block;
    EndIf;
  Until  $g_{max} = 0$ ;
EndProcedure;
  
```

Fig. 4.4 The modified KL 2-way partitioning algorithm.

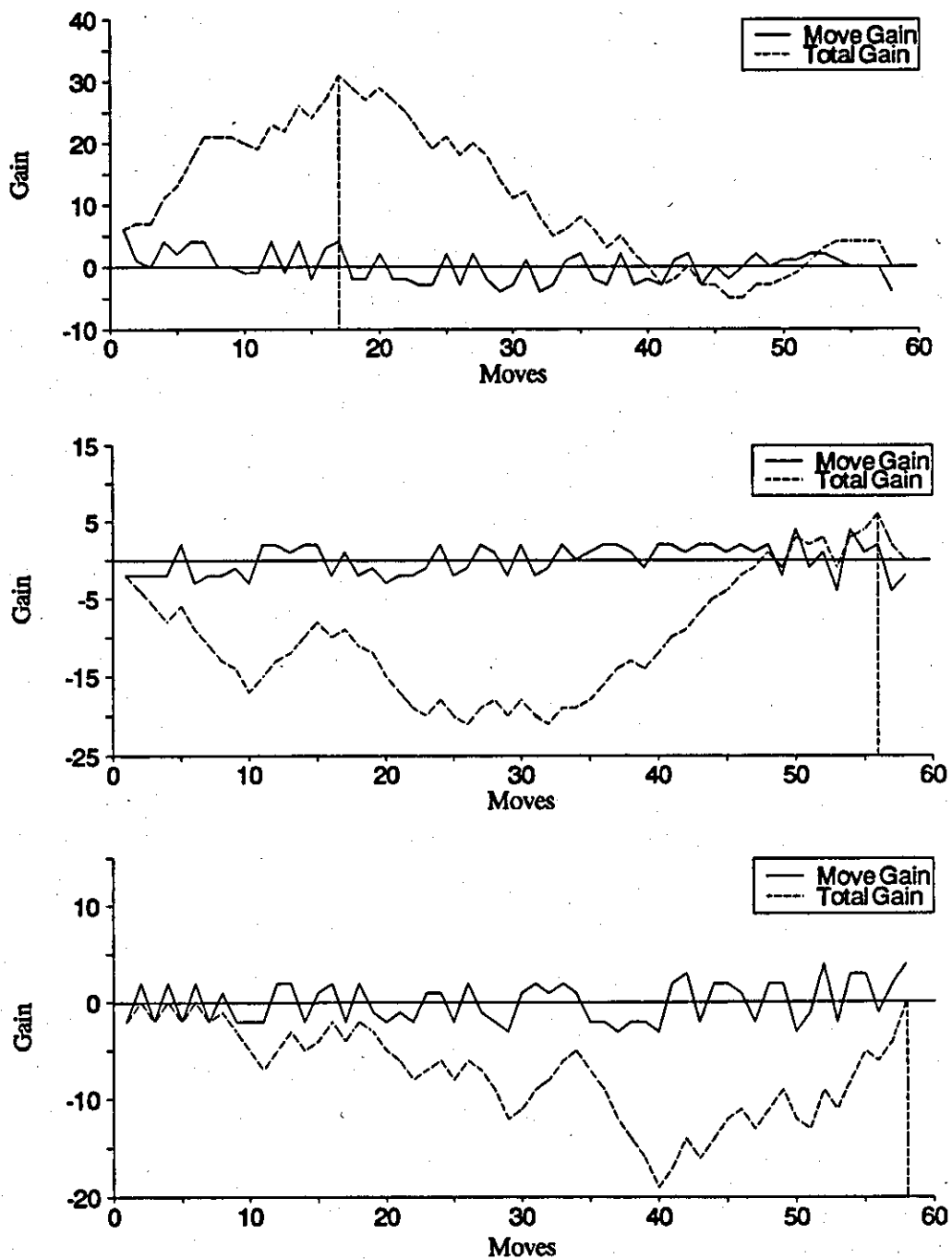


Fig. 4.5 Move-Gain profile of three passes in Kernighan-Lin's 2-way graph partitioning procedure. A representative graph with 58 nodes is used as the data instance.

Therefore, the gain in partition cost when node a is moved from A to B is,

$$\begin{aligned}
 \text{Gain} &= \text{OldCost} - \text{NewCost} \\
 &= T - T' \\
 &= E_a - I_a \\
 G_a &= D_a
 \end{aligned} \tag{4.9a}$$

Similarly, for moving any node $b \in B$ from B to A , we get

$$\begin{aligned}
 G_b &= E_b - I_b \\
 &= D_b
 \end{aligned} \tag{4.9b}$$

Once a node $a \in A$ (or $b \in B$) is selected for a move from A to B (or from B to A) it is isolated from rest of the nodes and the internal and external gain values of all other nodes which are affected because of the move, are updated.

When a node $a \in A$ is selected for a move and kept aside from rest of the nodes, the D values of all free nodes (those which are not yet selected for a move in the pass) in A (nodes in B are unaffected at this moment) are changed. These can be calculated as follows,

For any free node $x \in A - \{a\}$,

$$\begin{aligned}
 D'_x &= E'_x - I'_x \\
 D'_x &= (E_x + e_{xa}) - (I_x - e_{xa}) \\
 D'_x &= E_x - I_x + 2e_{xa} \\
 D'_x &= D_x + 2e_{xa}
 \end{aligned} \tag{4.10a}$$

Similarly, when a node $b \in B$ is selected for a move and kept aside from rest of the free nodes, for any free node $y \in B - \{b\}$ we can have,

$$D'_y = D_y + 2e_{by} \tag{4.10b}$$

Table 4.1 Table showing the performance of the three rules considered in the modified KL 2-way partitioning heuristic.

Graph	Number of Nodes	Average Degree	Average Initial Cost	Rule A Final Cost			Rule B Final Cost			Rule C Final; Cost		
				Max.	Min.	Ave.	Max.	Min.	Ave.	Max.	Min.	Ave.
6N2C	6	3.00	6.36	94.27	47.13	61.27	92.10	46.05	50.65	47.21	47.21	47.21
4x4 Multiplier	58	2.83	46.88	81.07	27.73	53.03	32.47	19.48	23.85	30.23	19.44	22.72
Frequency Locked Loop	68	4.09	89.26	85.14	12.32	53.88	37.18	6.56	16.38	31.78	6.81	15.16
16x16 Multiplier	415	5.31	667.23	74.49	59.35	67.19	44.09	23.90	32.57	41.44	24.48	31.04
Vector Coder	899	4.53	1104.53	77.77	62.02	69.93	24.16	6.61	15.93	28.16	11.04	19.61

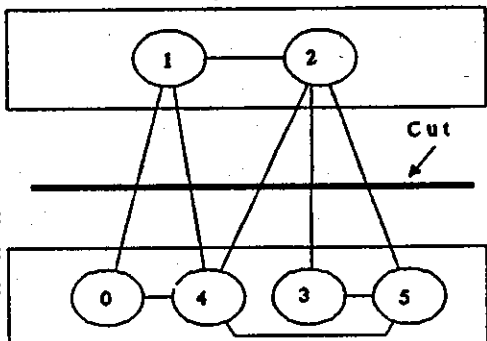
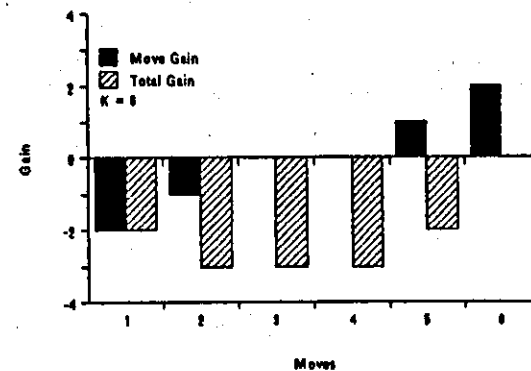
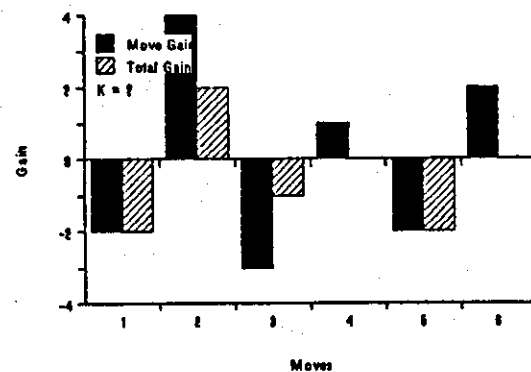
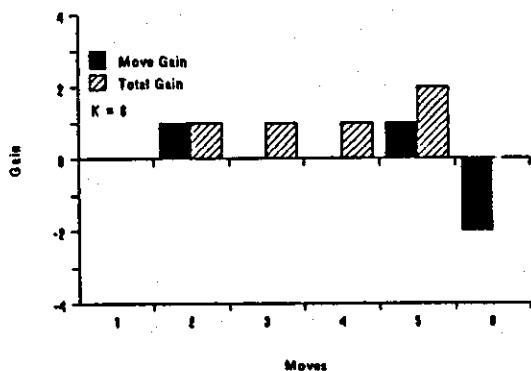
The essential steps of the modified KL bi-partitioning procedure is shown in Fig. 4.4 in pseudo-Pascal form. The outer Repeat-Until loop represents the passes while the inner For-End loop shows the activities in each pass.

The basic 2-way partitioning procedure used here is similar in spirit to the Fiduccia-Matheyases variant of the KL heuristic. An initial 2-way partition from a pre-processor or randomly generated one is taken first. A sequence of maximally improving node transfers from one block of the partition to the other are then attempted. This iterative improvement heuristic is otherwise very similar to the Kernighan and Lin's min-cut heuristic except for the use of one-way node movements instead of node exchanges. The use of node transfers in this fashion ensures acceptable level of load-balance even when the node weights vary by a wide margin.

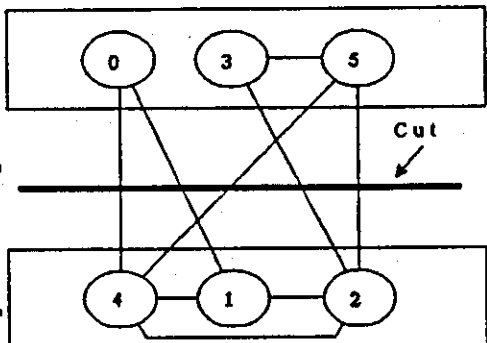
The success of the graph bi-partitioning heuristic depends on the choice of a candidate node for a transfer. Consequently, the procedure of Fig. 4.4 spends most of the time in selecting the best node for a transfer. In the modified KL heuristic where a single node is nominated for a transfer in each iteration of a pass, the choice of the block as a supply pool of the nominated node also needs to be explored. The following three different rules were examined.

- A : Two candidate nodes one each from blocks *A* and *B* are selected so that each has the highest gain value in it's respective block. These are then nominated for transfers in sequence. This rule is very similar to the KL move exchange with the notable exception that the interaction (edge) between the two nodes is ignored as the two node transfer procedures are actually carried out separately and also that a total of odd number of node transfers are allowed.
- B : Of the two blocks *A* and *B*, any one is selected at random as the source of the node in the current iteration. The node with the highest gain value in the selected block is then nominated for transfer. This rule adds Monte-Carlo flavour to the heuristic.
- C : At each step, two nodes one each from blocks *A* and *B* with the highest gain value in its own block are considered. The node with the higher

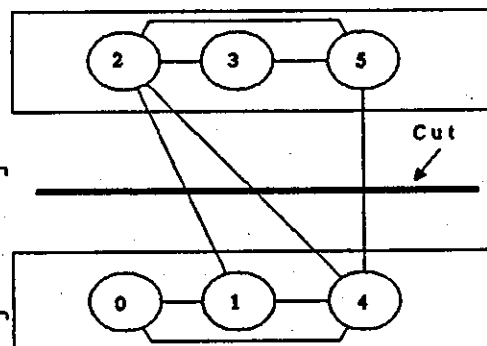
Fig. 4.6 Bi-partitioning of a 6-node graph in three passes. The bar graphs on the left column show the gains associated with a single move and also the total gains upto that move in the pass corresponding to that of the figure to its right.



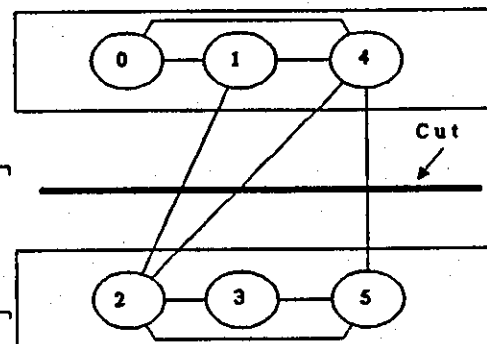
a. Initial arbitrary partition. Cost = 7.



b. Partition after pass 1. Cost = 5.



c. Partition after pass 2. Cost = 3.



d. Partition after pass 3. Cost = 3.

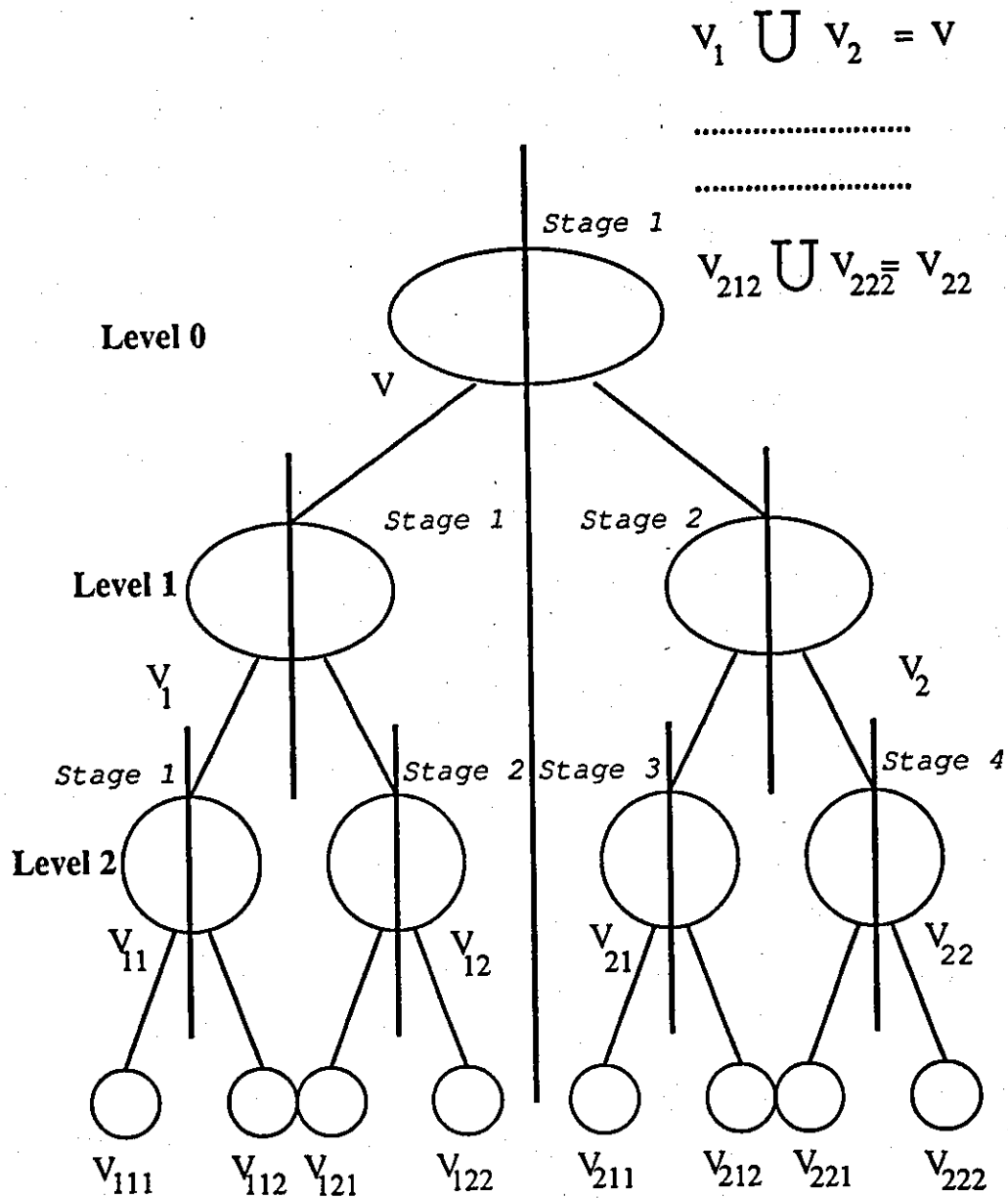


Fig.4.7 Recursive binary partitioning for a 8-processor task system.

gain value among these two is then actually taken as the nominated node.

In each case, in the event of any one of the two blocks becoming exhausted of free nodes, the other block is taken as the sole pool of free nodes. The gain value computation for each free node in all cases involves both the load-balance and communication aspects of the partition cost.

4.1.2 The Bi-partitioning Algorithm in Action

As expected the three different rules used for the nomination of a node for an eventual node transfer performed differently. Their relative performances are presented in Table 4.1.

Five different simulation graph instances are used to compare the performances of the three rules described earlier. Of these the last four graphs are taken from actual VLSI simulation systems and the first one is a simple artificially generated graph with a known optimal partition cost of 3. In each case a total of 50 runs are made and the maximum, minimum and average final partition cost values are tabulated. These are presented in the form of percentage of the average random partition cost which is calculated from 200 random partitions. As expected, Rule A which is a distorted replica of the KL move exchange mechanism, performed very badly throughout and proved to some extent the inability of the KL move exchange to handle graphs with unequal node weights in situations where load-balance is an important issue. Rules B and C performed very similarly with the performance of the latter marginally better in all cases. The only exception is with the largest graph (Vector Coder) where Rule B performed better than Rule C. The variation in the graph instances is thought to have an important effect on the apparent failure of Rule C in the above case.

Not all moves that are nominated in a single pass of the bi-partitioning procedure actually improve the partition, even though in each iteration of the pass, nodes with highest gain values are selected. During a single pass a running sum of the gain values associated with each node nominated for a move is maintained. At the end of a pass this running sum value is analysed and its maximum value is determined. A number of moves starting from the

move in iteration 1 to the move in iteration i , where the maximum value of the running sum is found are then actually carried out. This process is repeated only to be stopped when the maximum value of the running sum is zero, i.e., when further improvement is found infeasible. The moves that are carried out at the end of each pass may contain some moves that actually worsens the partition. This quality of the KL heuristic to accept moves that actually worsen the partition allows the heuristic to perform better than other comparable heuristics such as randomised iterative improvement method. Moves with a negative gain (i.e., those which worsen the partition) gives the KL heuristic some kind of hill climbing capability and helps it to come out from a local minima and to settle in somewhere very close to the global minima of the optimisation surface. However, there is no mechanism available to control this hill climbing capability and as a result a globally optimal solution can not be guaranteed at all times.

To illustrate the activities in a bi-partitioning procedure we present two figures 4.5 and 4.6. In Fig. 4.5 a graph with 58 nodes is partitioned into two blocks where a total of three passes are needed. The gain values associated with each nominated move (continuous line) and the corresponding running sum upto that move (broken line) are plotted. The vertical dotted line on the moves axis shows the point where the running sum is highest in that pass and the corresponding number on the moves axis is the number of moves that are actually carried out in that pass. Fig. 4.6 graphically describes the process of bi-partitioning of a 6 node graph with known optimal bi-partition cost of 3. The process starts with an arbitrary partition with a cost of 7. The partitions after the end of each pass are also shown along with the profiles of the moves made in that pass.

4.2 Multiple-way Partitioning

Multiple-way graph partitioning is a natural progression from a graph bi-partitioning process. There are several ways in which a 2-way iterative improvement graph partitioning algorithm can be adapted to multiple block partitioning. Three such methods can be immediately thought of. The first two of these are suggested in [1] and they both involve repeated use of 2-way partitioning.

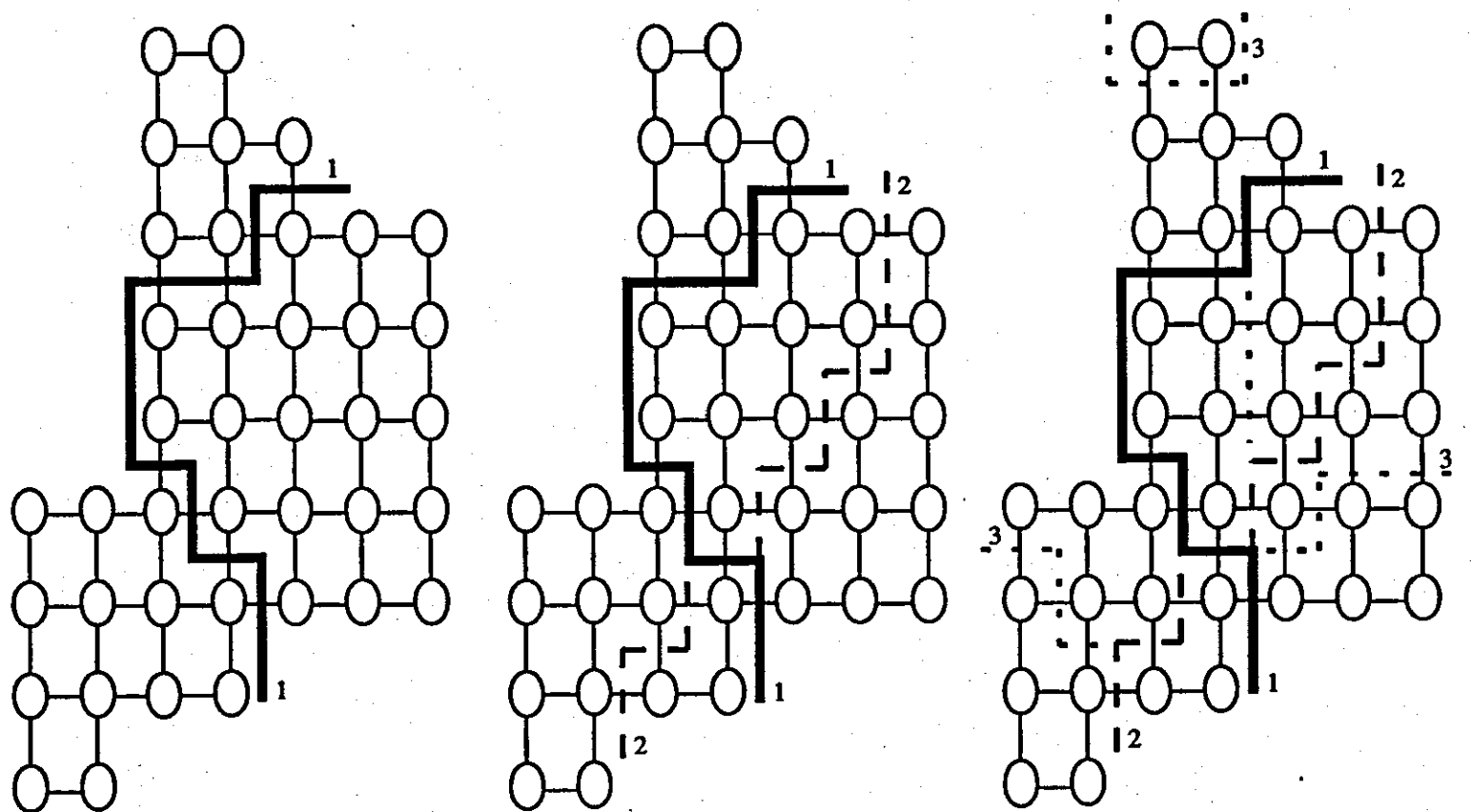


Fig. 4.8 An arbitrary graph partitioned into 8 blocks in three levels (left to right). Adapted from [8].

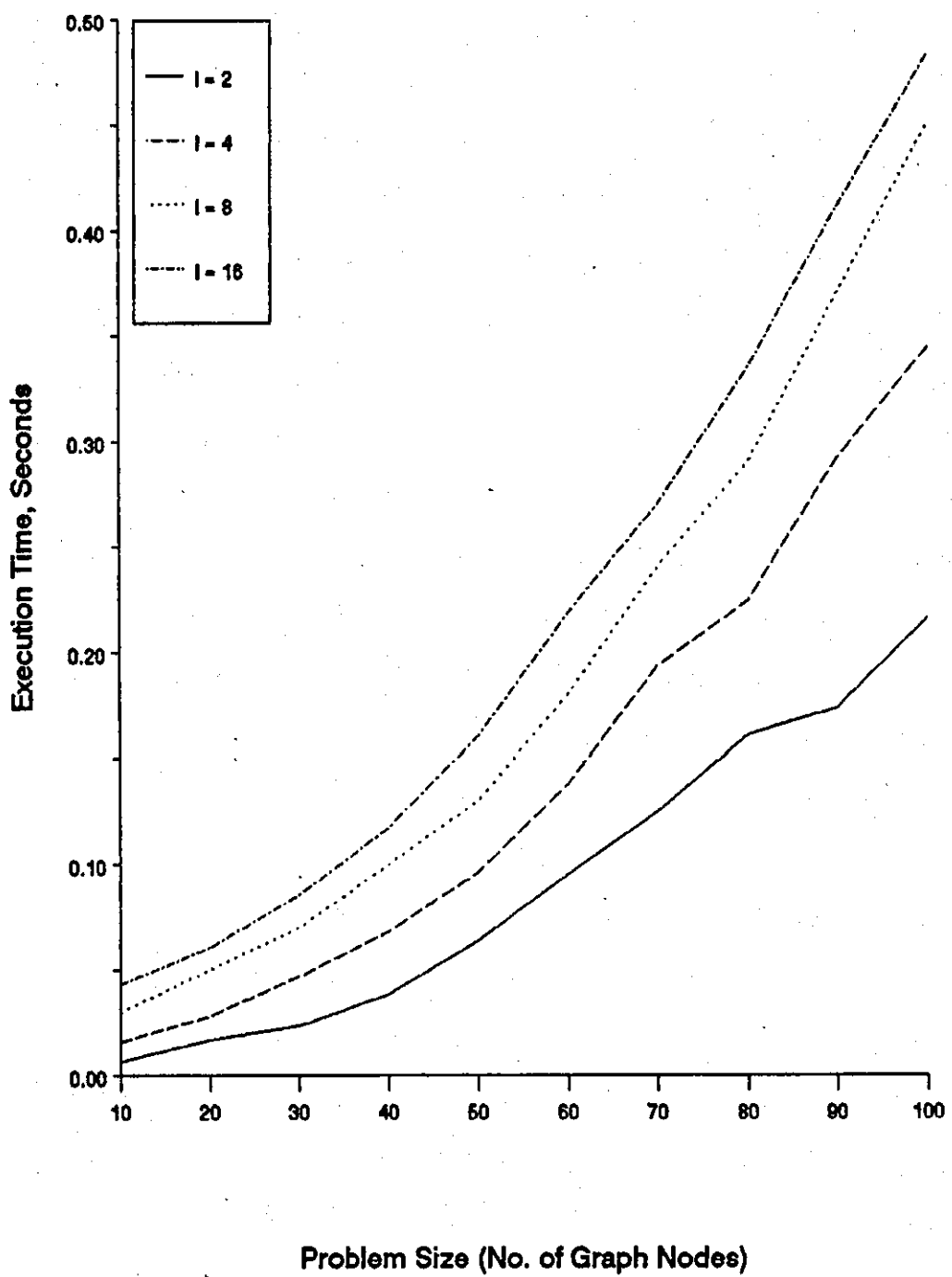


Fig. 4.9 Average execution time as function of the problem size for various number of partitions for the KL recursive binary partitioning algorithm.

The first consists of starting with an arbitrary l -way partition and successively choosing pairs of blocks and applying the 2-way partitioning algorithm to these pairs. Since, this method tries to maintain pairwise optimality explicitly, the final solution is often of not very good quality. The number of passes required grows with l , the number of total partitions sought and k , the total number of nodes in the graph. The result is also sensitive to the choice of initial partition and the way the pairs of blocks are chosen.

The second method consists of the hierarchical use of 2-way partitioning algorithm. For this method to work the number of partitions l , must be an integer power of 2. This method provides fast convergence and can be used to produce the starting partition for use in method described above or any other iterative improvement algorithm.

As an alternative to the repeated use of 2-way partitioning algorithm, the third method attempts to improve the partition uniformly at each step. In this scheme, at each iteration during a pass all possible moves of each free node from its home block to all other blocks are considered and only the best move is accepted. This method though appears to give good quality solutions, is computationally too involved and as such is not suitable for time critical applications.

Of the three different methods described above, we shall concentrate on the second method only because of its fast convergence ability and also because of the ease of exploiting macro level parallelism from the procedure. The following section describes the hierarchical partitioning procedure in detail.

4.2.1 Recursive Binary Partitioning

The *recursive binary partitioning* (RBP) method for the solution of multi-way graph partitioning problem addressed in this section is very efficient in terms of run time and also gives reasonable quality solution. Similar technique for the solution of multiprocessor scheduling of finite element modelling problems have been reported [7,8].

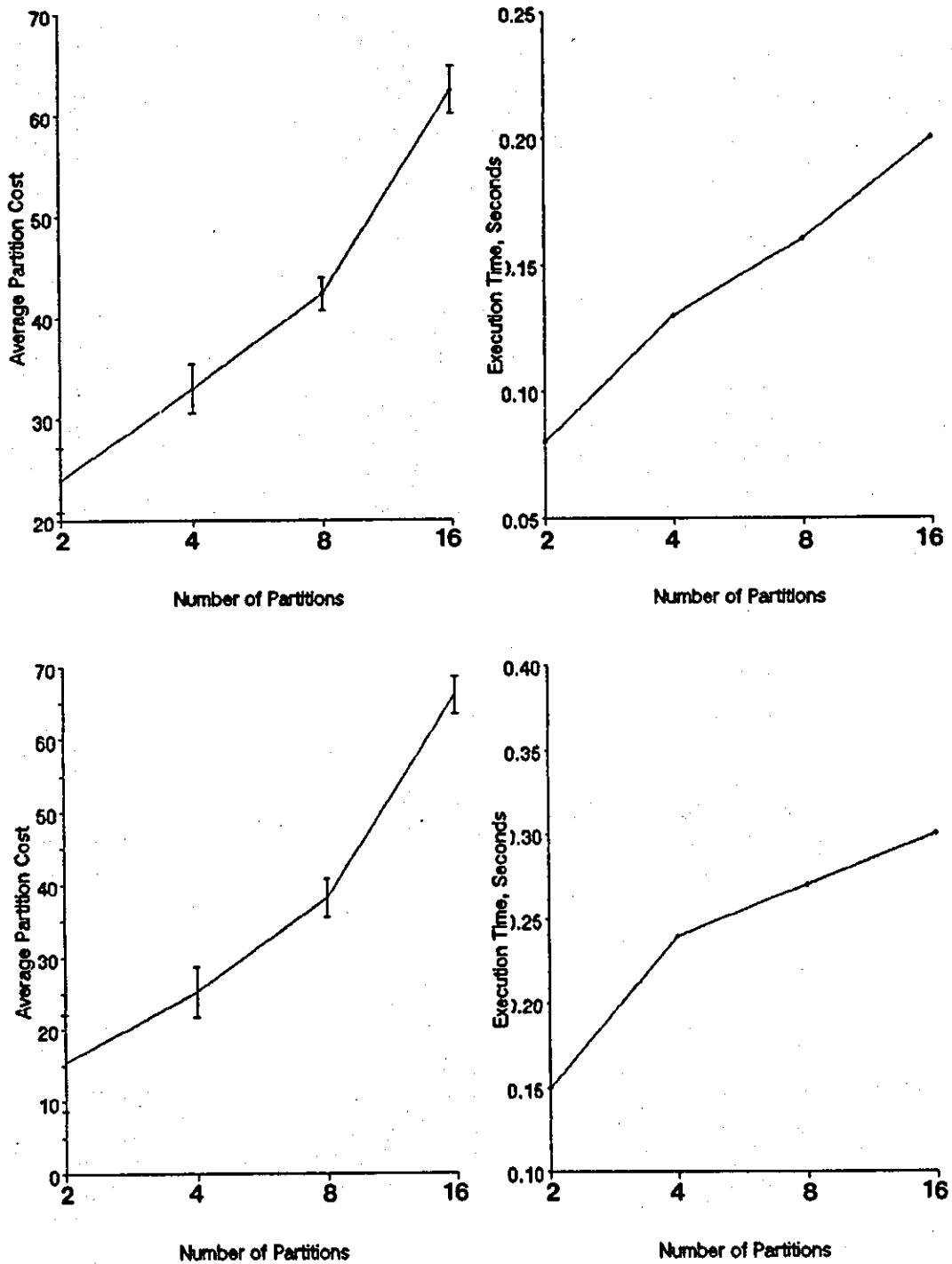


Fig. 4.10 Partition cost and the execution time as functions of the number of partitions for the KL recursive binary partition algorithm. Graph instances used are 4x4 multiplier ckt. (top) and frequency locked loop ckt. (bottom).

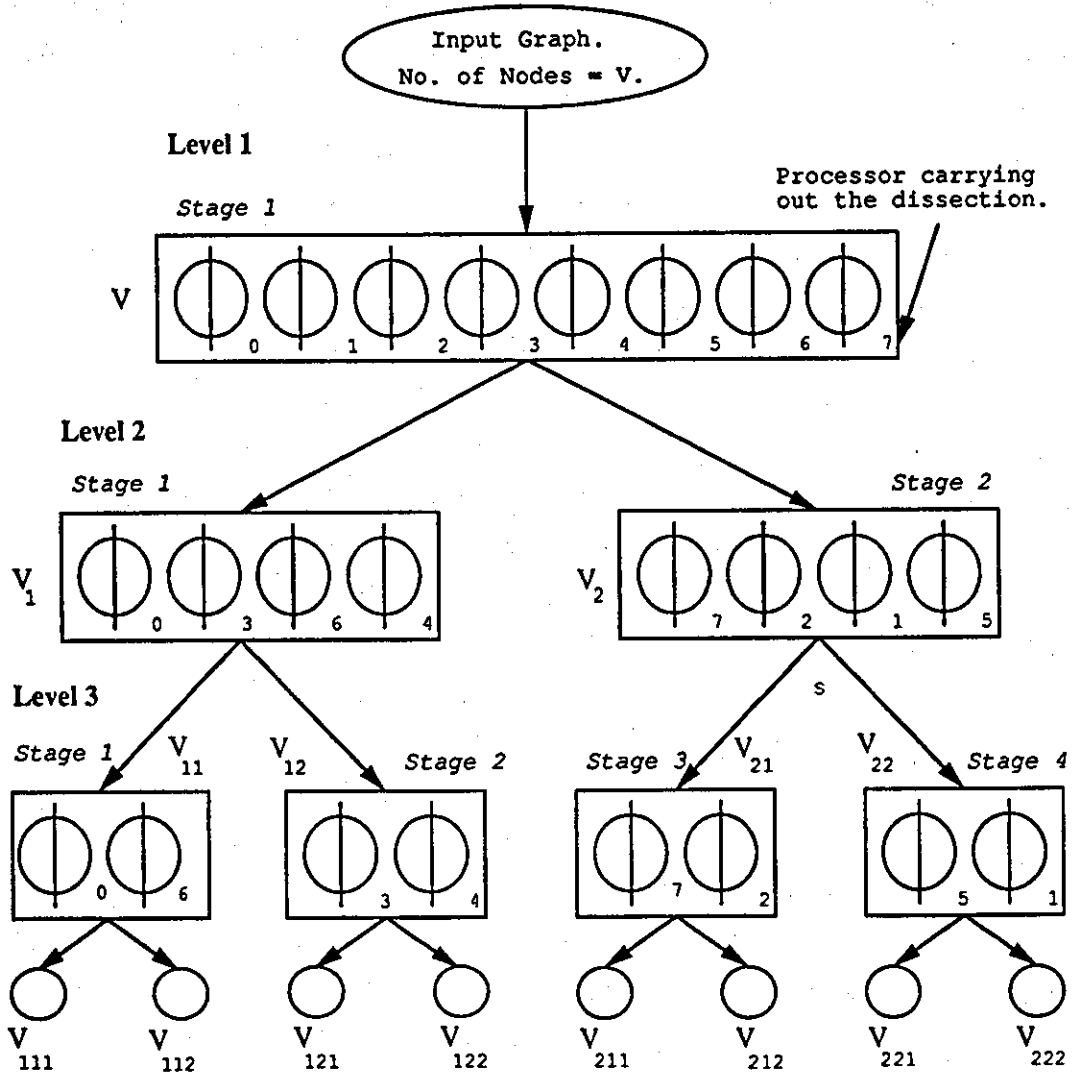


Fig. 4.11 Concurrent recursive binary partition process in action.

The procedure proceeds as follows. The given graph with k nodes is first split into two blocks such that the optimisation criterion established earlier (eqs. 4.3 & 4.4) is properly maintained. These two blocks are then recursively subdivided as many times as required. Clearly, the number of blocks the graph can be partitioned must be exactly 2^l , where l is an integer and represents the depth of partitioning. Thus, this procedure is useful for scheduling problems when the number of processors is a power of 2. The time required by recursive binary partitioning procedure can be estimated as follows. Each pass of Kernighan-Lin's 2-way partitioning algorithm requires $O(k^2 \log k)$ time. The number of passes required for the convergence is generally small and lies between 2-6 and as such is not strongly dependent on the size of the graph. Again by employing better search algorithm for selecting a candidate node for its transfer from its home block to the other, a lower bound run-time $O(k^2)$ can be easily obtained. For a l -way partitioning using the recursive binary partitioning procedure the total run-time requirement becomes $O(k^2 \log l)$.

Being a heuristic procedure, the recursive binary partitioning procedure does not guarantee an optimal solution. However, reasonably good quality solution is expected. Iqbal et. al [9] found the upper bound on the difference between the optimal solution and the solution yielded by the recursive binary partitioning procedure when load balance is the sole optimising criterion and the task system considered is a chain of structured parallel or pipelined program and a chain of processors. Their upper bound is controlled by the most heavily weighted module in the chain.

The overall partitioning strategy in a recursive binary partitioning takes the form of a binary tree. This is illustrated in Fig. 4.7. Fig. 4.8 shows a mesh structured arbitrary graph partitioned into 8 blocks in three levels of partitioning. The results obtained through simulation are presented in Figs. 4.9 & 4.10. The average execution time required by the Kernighan-Lin recursive binary partitioning (KL-RBP) algorithm is shown in Fig. 4.9 for different sizes of the problem graph. The problem size here is defined as the number of nodes of the graph representing a task system. In this investigation some synthetically generated graphs with nodes ranging from 10 to 100 with average degree ranging from 2.8 to 3.6 are used. For all the partition sizes

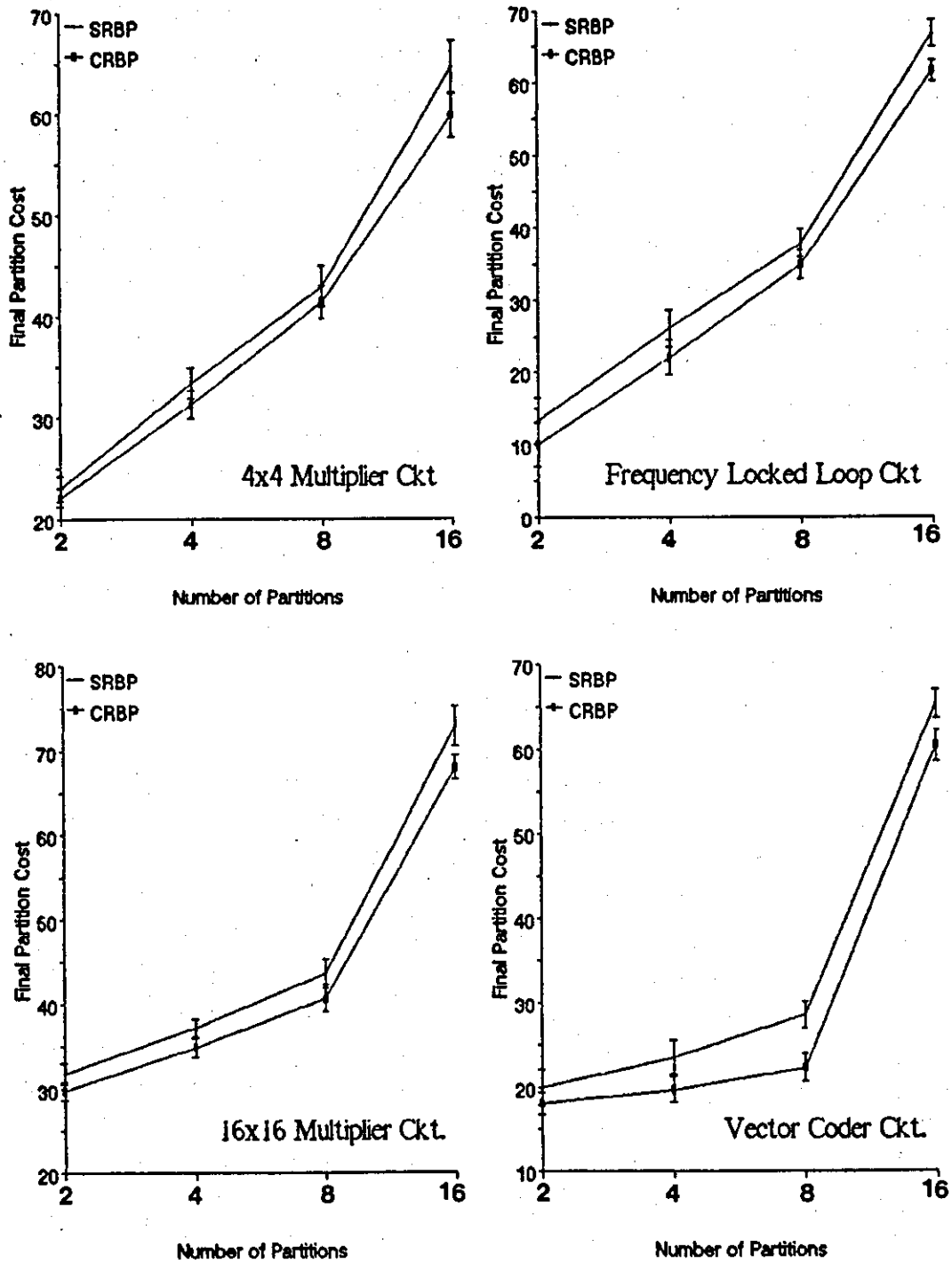



Fig. 4.12 Graphs showing the performance comparison between the serial (SRBP) and concurrent (CRBP) l -way recursive partitioning algorithms. Four representative graph data instances are used.

Table 4.2 : Table showing the relative performances of the serial and concurrent l-way (Recursive Binary Partitioning) KL algorithms.

Graphs 		4x4 Multiplier		Frequency Locked Loop		16x16 Multiplier		Vector Coder	
Number of Partitions		SRBP	CRBP	SRBP	CRBP	SRBP	CRBP	SRBP	CRBP
2	Maximum	32.12	28.48	29.45	29.39	37.04	35.93	28.59	26.87
	Average	23.00	22.12	13.32	10.04	31.77	29.68	19.94	18.08
	Minimum	19.27	19.72	6.80	6.78	26.16	24.35	11.15	10.15
4	Maximum	36.67	33.50	35.75	26.94	42.10	37.72	29.50	24.14
	Average	33.41	31.29	26.08	22.07	37.16	34.84	23.44	19.62
	Minimum	29.34	29.13	21.76	20.01	33.93	30.03	18.19	13.04
8	Maximum	46.85	43.35	42.10	38.69	47.08	44.43	34.18	25.19
	Average	42.95	41.39	37.81	34.85	43.62	40.66	28.59	22.28
	Minimum	39.26	38.25	33.27	31.90	39.79	37.11	24.43	19.00
16	Maximum	68.35	68.33	71.40	65.75	77.93	71.33	70.57	64.27
	Average	64.63	59.82	66.82	61.68	72.87	68.02	65.29	60.45
	Minimum	57.55	55.15	62.88	55.89	66.75	63.03	50.94	51.04

SRBP : Serial Recursive Binary Partitioning.

CRBP : Concurrent Recursive Binary Partitioning (Best of Bunch enabled).

(2, 4, 8 and 16) considered the KL-RBP has a time requirement roughly equal to $O(k^2)$. It is also observed that the execution time does not grow very rapidly with the increase in number of partitions. Fig. 4.10 shows the average partition cost (with the standard deviation shown as the lower and upper bound) and the corresponding average execution time as a function of the number of partitions. Two representative simulation graphs, 4x4 multiplier ckt. and frequency locked loop ckt. are used. It is observed that the partition cost obtained deteriorates with an increase of number of partitions.

4.2.2 Concurrent Recursive Binary Partitioning

A heuristic l -way graph partitioning algorithm requiring $O(k^2 \log l)$ is a great improvement in terms of practicality considering the NP-Hard nature of the problem. However, large graphs often require far too much time for convergence. Further reduction in solution time is always desired particularly in a VLSI design environment where a quick design turn around time is an absolute necessity. A parallel implementation of the graph partitioning algorithm on a general or special purpose multiprocessor system is an obvious choice. A multiprocessor system is thought to have the capability of further reducing the solution time for l -way graph partitioning problem. Ravikumar et al. [10] have exploited the micro or fine grain parallelism present in Kernighan-Lin's 2-way graph partitioning algorithm and devised a parallel implementation on an array of processors working in SIMD mode with shared memory. However, their algorithm is not suitable for an asynchronous message-passing, distributed memory system working in MIMD mode because of excessive synchronisation and communication overhead. The alternate approach i.e., the exploitation of macro or coarse grain parallelism is more favoured in such situations and is adopted here.

The recursive binary partitioning procedure lends itself to parallel implementation. Except for the first partitioning (level 1), partitioning at different stages in all other levels can be carried out independently of each other as can be visualised from the binary tree like structure of the overall recursive binary partitioning strategy as shown in Fig. 4.7. These independent partitioning procedures can be assigned to different processors

of the multiprocessor ensemble and can be run concurrently. A breadth-first partitioning would then take place. However, for a n -processor system, maximum processor utilisation is achieved only at the bottom (leaf) level.

A modification that utilises all the processor resources and improves the quality of the final solution can be made. This involves allowing more than one processor when available, to carry out the bi-partitioning in each stage of every level. The next is to accept the best partitions from the set of all partitions offered by the group of processors assigned to the stage and level concerned. The best partitions thus generated and accepted are moved forward to the next level and so on. We call this *best-of-bunch* technique. As an illustration we consider a 8 processor system and a 8-way graph partitioning problem (Fig. 4.11). At level 1 we have 8 separate bi-partitioning operations on the same input graph all running in parallel and only the best resulting partitions are accepted for level 2 partitioning, whereby at level 2, two concurrent partitioning operations are required because there are now two input sub-graphs. Out of the total 8 processors, 4 processors can thus be allocated for each of the two partitioning stages. At the bottom level (level 3) 2 processors can thus be allocated for each of the 4 partitioning stages.

The modification described above provides faster descent and guarantees a better solution that can be achieved otherwise. Fig. 4.12 shows the performance comparison between the serial and concurrent recursive binary partitioning (SRBP & CRBP respectively) procedures for four representative graphs. In all, cases the concurrent version where the best-of-bunch technique is employed outperformed its serial counterpart.

It can be seen from the above that minimum processor utilisation is achieved at the top level (level 1) and it increases by a factor of 2 as the partitioning process progresses. This accounts for the speed-up factor $\log n$ and since the same number of processors as the number of partitions are employed i.e., $n = l$, the total run-time requirement of the concurrent recursive binary partitioning procedure becomes $O(k^2)$. In a message-passing, distributed memory multiprocessor system the processors communicate via a high speed communication channel. In this concurrent implementation a total of k data objects, k being the node count of the input graph need to be

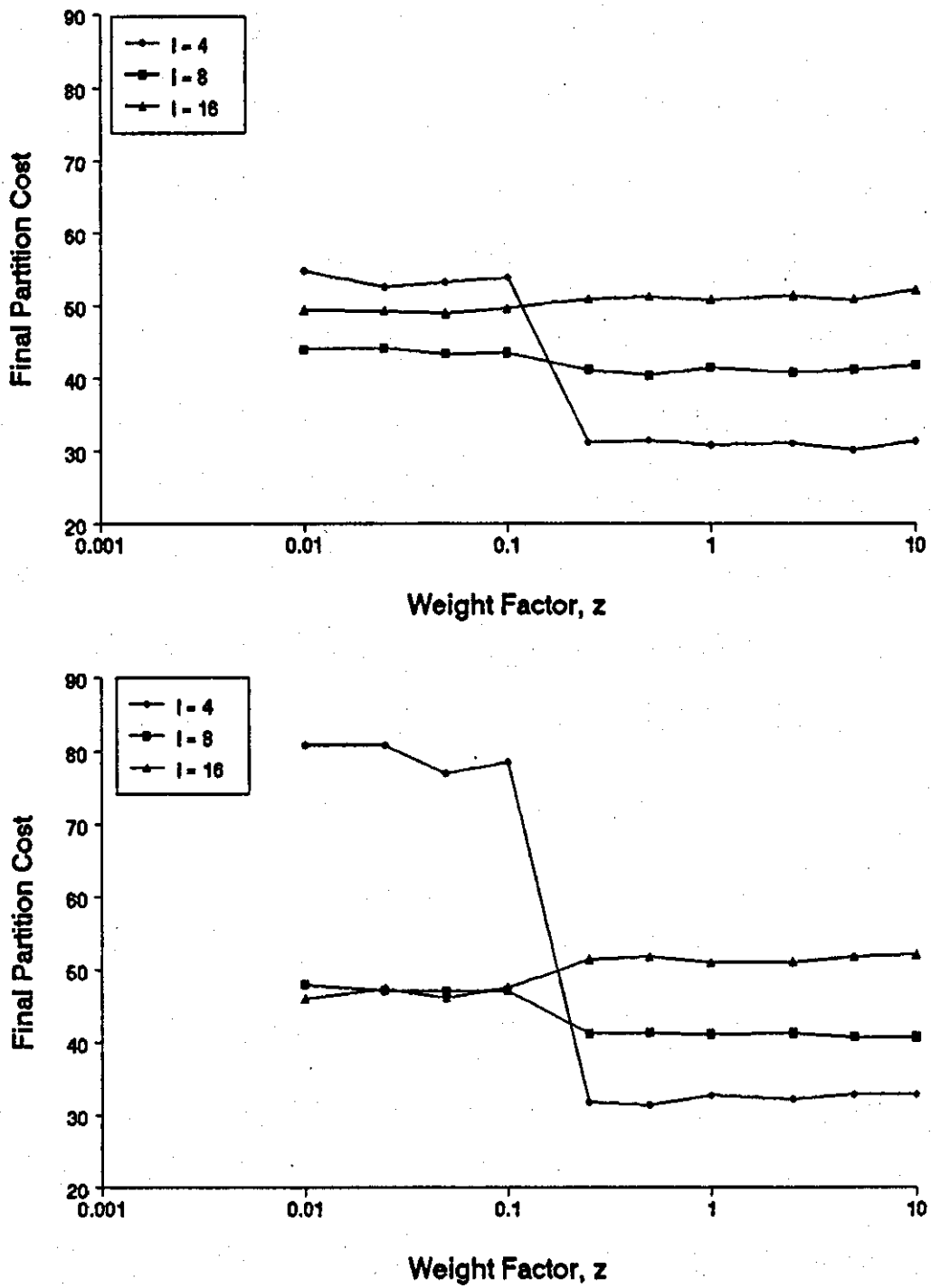


Fig. 4.13 Variation in final partition cost with the weight factor, z for partition sizes 4, 8 and 16. Adjustments are in level 1 (top) and levels 1 & 2 (bottom) only. Data flow graph instance for the 4x4 multiplier ckt. is used.

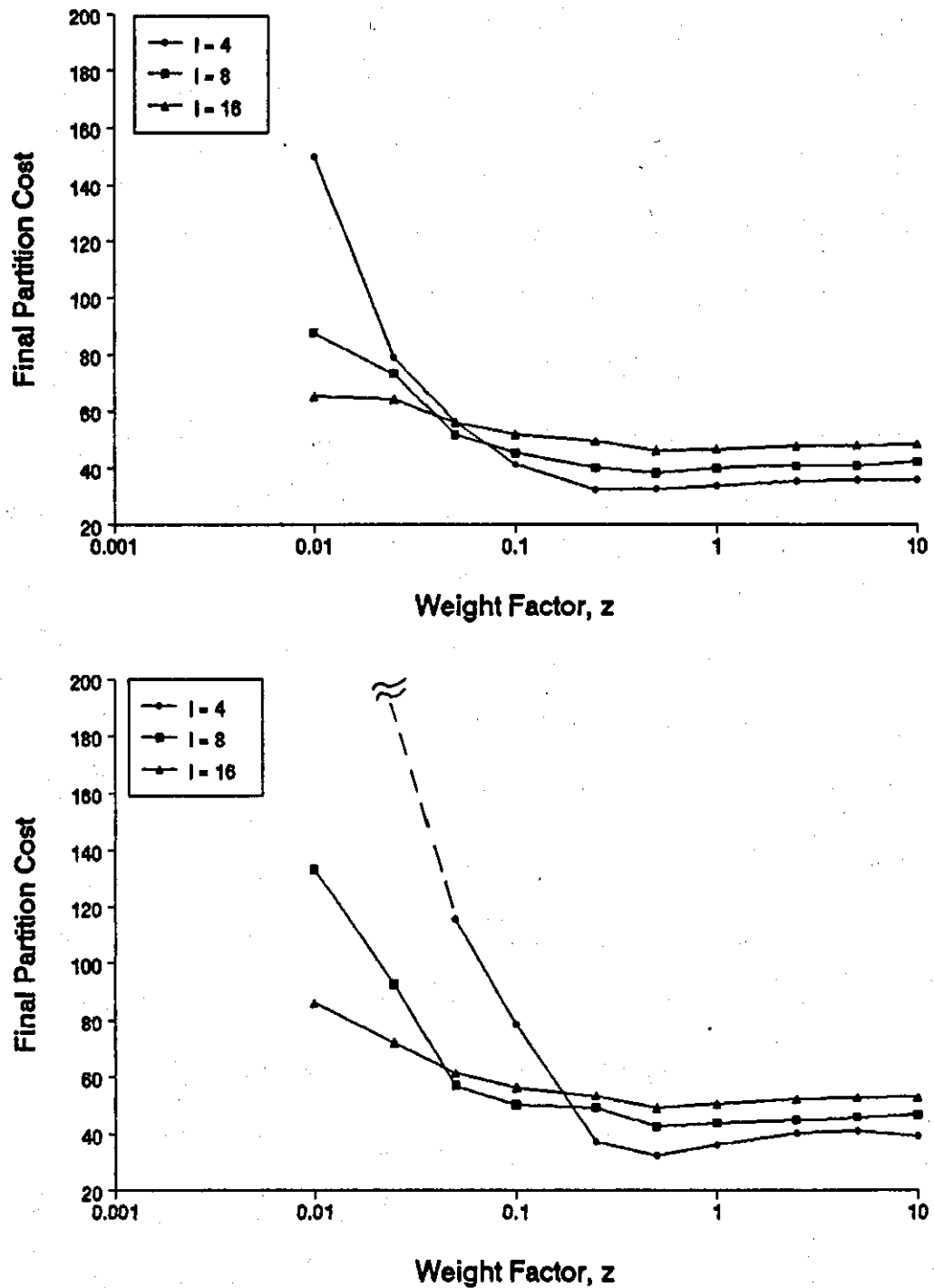


Fig. 4.14 Variation in final partition cost with the weight factor, z for partition sizes 4, 8 and 16. Adjustments are in level 1 (top) and levels 1 & 2 (bottom) only. Data flow graph instance for the 16x16 multiplier ckt. is used.

transferred between the processors at the start of each level. Since, the depth of partitioning is $\log l$, the total communication overhead becomes $O(k \log l)$.

4.2.3 Performance of Recursive Binary Partitioning

The major drawback of the recursive binary partitioning heuristic is its dependence on its top level partitioning. As pointed out in [1], a bad result in the first partitioning may bias the second and so on, with the largest occurring for large l . Also the first partitioning will try to minimise the number of edges between the first two blocks, thus tending to maximise the number of edges inside these blocks and making it harder to obtain good partitions thereafter. A general decline in the solution quality is thus expected with increasing l . This is clearly seen in the graphs of Fig. 4.12, where the final partition cost is expressed as a percentage of the average random initial partition cost. The decline in solution quality becomes more noticeable when the number of partitions l exceeds 8.

The objective (cost) function used throughout is a combination of both the communication cost due to the edges cut in the partition and cost due to load imbalance.

$$C_t = C_c + z.C_e \quad 4.11$$

where z is the weight factor maintaining a balance between the two components. The value of z was held at 1 for all the results obtained so far which signifies an equal emphasis placed on both the two cost components. However, by changing the value of z the roles that the two cost components play can be altered. A small z would mean greater emphasis on communication cost as compared to the cost due to load imbalance. A very small $z (< 0.1)$ would practically ignore load-balance from the optimisation criterion and in such cases, the 2-way partitioning algorithm would find an pseudo-optimal solution very easily where all the nodes are placed in one block only with the other going empty.

It has already been mentioned that the first level partitioning in a l -way recursive binary partitioning dictates the final partition cost by minimising the number of edges between the first two blocks. This action leaves the heavily connected nodes inside these blocks and subsequent partitioning of

these as a result fails to produce good partitions. An alternate scheme can be thought of where the 2-way partitioning heuristic in the first (and second) level can be directed to put more emphasis on load-balance than to minimise the number of edges cut. It is expected that this modification will produce slightly inferior first (and second) level partitions compared to which can be obtained otherwise and as such, will leave less heavily connected nodes inside the resulting partitions. These on subsequent partitioning thus stand better chance to produce good l -way final partitions. Changing the value of the weight factor z can bring in the above modification. The results are shown in Figs. 4.13 and 4.14 for two representative graphs with 58 and 415 nodes respectively.

The value of z is varied between 0.01 and 10 and three different partition sizes of 4, 8 and 16 are considered. For each graph the value of z is changed in level 1 alone and also in levels 1 and 2. For very small $z(< 0.1)$ the final partition cost is predominantly due to load imbalance and for large $z(> 2.5)$ it is mostly due to communication cost. In all cases the final partition cost is found to secure a steady low value for z between 0.25 and 1. This shows that the roles of the communication and load-balance are equally important in l -way recursive binary partitioning procedure and also proves that the gain obtained by placing more emphasis on load-balance in early partitioning levels is defeated by the bad results from these levels being propagated throughout.

References :

1. Kernighan, B.W. and Lin, S., *An Efficient Heuristic Procedure for Partitioning Graphs*, Bell Syst. Tech. J., Feb. 1970., pp. 291-307.
2. Garey, M.R. and Johnson, D.S., *Computers and Intractability : A Guide to the Theory of NP-Completeness*, Freeman, San Fransisco, 1979.
3. Barnes, E.R., *An algorithm for partitioning the nodes of a graph*, Tech. Rept. IBM T.J. Watson Research Centre, Feb. 1981.
4. Bui, T., Chaudhuri, S., Leighton, T. and Spiser, M., *Graph bisection algorithms with good average case behaviour*, Tech. Rep. 85-236, Dept. Elect. Engg & Comp. Sci., MIT, 1985.
5. Stone, H.S., *Multiprocessor Scheduling With the Aid of Network Flow Algorithms*, IEEE Trans. Software Engg., Vol. SE-3, No.1, Jan 1977, pp. 85-93.
6. Fiduccia, C.M. and Matthyyses, R.M., *A linear-time heuristic for improving network partitions*, Proc. 19th Design Automation Conf., 1982, pp. 175-181.
7. Berger, M.J. and Bokhari, S.H., *A partitioning strategy for non-uniform problems on multiprocessors*, IEEE Trans. Comp. Vol.36, No.5, May 1978, pp. 570-580.
8. Sadayappan, P., Ercal, F. and Ramanujam, J., *Cluster partitioning approaches to mapping parallel programs onto a hypercube*, Parallel Computing, Vol. 13, No.1, Jan 1990, pp. 1-16.
9. Iqbal, M.A., Saltz, J.H., and Bokhari, S.H., *A Comparative Analysis of Static and Dynamic Load Balancing Strategies*, Proc. Parallel Processing, 1986, pp. 1040-1047.
10. Ravikumar, C.P., Sastry, S. and Patnaik, L.M., *Parallel circuit partitioning on a reduced array architecture*, Computer Aided Design, Butterworth & Co (Publishers) Ltd., Vol.21, No.7, Sept 1989, pp. 447-455.

CHAPTER 5

The Simulated Annealing Algorithm

In this chapter we examine a heuristic task scheduling algorithm which has its roots in classical statistical physics. *Simulated annealing* (SA), is a technique modelled on the annealing process of physical matter and closely follows a probabilistic mechanism, similar to Boltzmann statistics used to analyse its physical counterpart. Simulated annealing has been found to be a robust tool for the solution of many difficult combinatorial optimisation problems. In this chapter, we first present the background information of SA heuristic, with due emphasis on the main factors that affect its performance. Two different cooling schedules that guide the SA algorithm to the convergence are described and their relative performances are compared. The parameters that influence the performance of the SA algorithm are also highlighted.

5.1 Combinatorial Optimisation and Simulated Annealing

Solving a combinatorial optimisation problem like the task scheduling problem amounts to finding the *best* or *optimal* solution among a finite or countably infinite number of alternative solutions. Over the past few decades, a wide variety of such problems has emerged from such diverse fields as management science, computer science, engineering, VLSI design etc. Over the years, it has been shown that many theoretical and practical combinatorial optimisation problems belong to the class of NP-Complete problems. However, large NP-Complete problems still must be solved and in this regard two different classes of algorithms exist. The first, known as the *optimisation* algorithms searches for and often provides the optimal solution but requires very large and possibly impractical amount of solution time. Well known examples in this class are enumeration methods using cutting plane, branch and bound or dynamic programming techniques. The other alternative known as *approximation* or *heuristic* algorithms provides solution in reasonably quick time but often sub-optimal solution results. Examples include *local search* or *randomisation methods*.

Furthermore, one may distinguish in both classes between *general* algorithms and *tailored* algorithms. General algorithms are applicable to a wide variety of problems and therefore may be called problem *independent*. Tailored algorithms use problem-specific information and their applicability is therefore limited to a restrictive set of problems. The Simulated Annealing (SA) algorithm, which is the main subject of this chapter is a high quality general algorithm. In nature it is a randomisation algorithm and its asymptotic behaviour can be viewed as that of an optimisation algorithm [1]. However, in any practical implementation it behaves as a heuristic algorithm.

5.1.1 The Simulated Annealing Algorithm

In solid state physics, *annealing* is known as a thermal process whereby low energy states of a solid in a heat bath can be obtained. The process can be summarised by the following two steps :

- a. increase the temperature of the heat bath to a maximum high at which the solid melts.
- b. decrease carefully the temperature of the heat bath until the particles arrange themselves in the ground energy state of the solid.

In the liquid phase all the particles of the solid arrange themselves randomly. In the ground energy state these particles are arranged in a highly structured lattice and the energy of the system is minimal. This ground energy state is obtained only if the maximum temperature is sufficiently high and the cooling is done sufficiently slowly. Otherwise, the solid will be frozen into a meta-stable state rather than into the ground state. If the temperature of the heat bath is lowered very quickly a meta-stable state will result. This is converse of annealing and is known as *quenching*.

The physical annealing process can be modelled successfully by using computer simulation methods from solid state physics. Metropolis et al. [2] introduced a simple algorithm (Metropolis algorithm) for simulating the evolution of solid in a heat bath to thermal equilibrium. This algorithm is based on *Monte-Carlo* technique and generates a sequence of states of the solid in the following manner. Given a current state i of the solid with energy E_i , then a subsequent state j with energy E_j is generated by applying a perturbation mechanism which transforms the current state i into the next state j by a small distortion. If the energy difference, $E_j - E_i$, is less than or equal to zero, the state j is accepted as the new current state. If, on the other hand the energy difference becomes greater than zero, the state j is accepted with a certain probability given by,

$$\exp \left(\frac{-(E_j - E_i)}{K_B T} \right) \quad 5.1$$

where T denotes the temperature of the heat bath and K_B , a physical constant known as *Boltzmann constant*.

If the heat bath is cooled sufficiently slow, the solid can reach thermal equilibrium at each temperature. In the Metropolis algorithm this is achieved by generating a large number of transitions (perturbations) at each temperature. Thermal equilibrium in such cases is characterised by the Boltzmann

```
Procedure Local_Search;  
  Initialise(i_start);  
  i := i_start;  
  Repeat  
    Generate(Configuration j from neighbourhood  $S_i$  of  
      configuration i);  
    If  $f(j) < f(i)$  Then  
      i := j;  
    EndIf;  
  Until  $f(j) \geq f(i), \forall j \in S_i$ ;  
EndProcedure;
```

Fig. 5.1 Pseudo-Pascal description of the local search algorithm.

distribution and gives the probability of the solid being in a state i with energy E_i at temperature T and is given by,

$$\mathbb{P}_T\{X = i\} = \frac{1}{Z(T)} \exp\left(\frac{-E_i}{K_B T}\right), \quad 5.2$$

where X is a stochastic variable denoting the current state of the solid. $Z(T)$ is the *partition function*, which is defined as,

$$Z(T) = \sum_j \exp\left(\frac{-E_j}{K_B T}\right), \quad 5.3$$

where the summation extends over all possible states.

Kirkpatrick et. al [3] assumed an analogy between a physical many-particle system and combinatorial optimisation problems based on the following equivalences,

- a. Solutions in a combinatorial optimisation problem are equivalent to states of a physical matter.
- b. The cost of a solution is equivalent to the energy of a state.

However, temperature in the physical system does not have a direct analog and as such a control parameter to be called *temperature* is used to play its role. The simulated annealing (SA) algorithm can now be viewed as an iteration of the Metropolis algorithm, evaluated at descending values of the control parameter, temperature, where a large number of candidate solutions are generated at each temperature.

The following formal definitions can be used to describe the SA algorithm in relation to the solution of combinatorial optimisation problems.

Let, (S, f) denote an instance of a combinatorial optimisation problem and i and j are two solutions with cost $f(i)$ and $f(j)$, respectively. The *acceptance criterion* determines whether j is accepted from i by applying the following acceptance probability :

$$\mathbb{P}_i\{\text{accept } j\} = \begin{cases} 1 & ; \text{ if } f(j) \leq f(i) \\ \exp\left(\frac{f(i)-f(j)}{t}\right) & ; \text{ if } f(j) > f(i), \end{cases} \quad 5.4$$

```
Procedure SimulatedAnnealing;
  Initialise(istart);
   $\lambda := 0$ ;
   $i := i(\text{start})$ ;
  Repeat
    For  $l := 1$  To  $L_\lambda$  Do
      Generate(Configuration  $j$  from neighbourhood  $S_i$ 
        of configuration  $i$ );
      If  $f(j) \leq f(i)$  Then
         $i := j$ 
      Else
        If  $\text{Exp}((f(j) - f(i))/C_\lambda) > \text{Random}[0,1]$  Then
           $i := j$ ;
        EndIf;
      EndFor;
     $\lambda := \lambda + 1$ ;
    Calculate_Length( $L_\lambda$ );
    Calculate_Control( $C_\lambda$ );
  Until Stop_Criterion;
EndProcedure;
```

Fig. 5.2 Pseudo-Pascal description of the simulated annealing algorithm.

where $t \in \mathbb{R}^+$ denotes the control parameter, temperature.

The generation mechanism in the SA algorithm corresponds to the perturbation algorithm in the Metropolis algorithm, whereas the acceptance criterion corresponds to the Metropolis criterion.

Let, t_λ denote the value of the control parameter and L_λ , the length of the number of transitions generated at the λ^{th} iteration of the Metropolis algorithm. Then the SA algorithm can be described in pseudo-Pascal as in Fig. 5.2. Fig. 5.1 shows similar description of its pre-cursor, the local search algorithm. A notable feature of the SA algorithm is that besides accepting transitions that improves the cost, it also, to a limited extent, accept cost deteriorating transitions. This is what is known as the so called hill climbing capability of an optimisation algorithm. Initially, at large values of t , large number of cost deteriorating transitions are accepted and as t approaches zero, almost all bad transitions are rejected. This controlled hill climbing capability of the SA algorithm allows it to escape from local minima while still exhibiting the favourable features of local search (simple iterative improvement) algorithms, i.e., simplicity and general applicability.

SA can be viewed as a generalisation of the local search algorithm. In fact, for cases where the value of the control parameter is set to zero, the former algorithm behaves similar to the latter. In the physical analogy, the local search algorithm can be viewed as the quenching process and like its physical counterpart though quick, often produces inferior quality results.

5.2 Markov Chain Model of SA

The SA algorithm can be modelled mathematically by using the theory of Markov chains. It is, therefore, possible to predict the asymptotic behaviour of the SA algorithm by using different properties of Markov chain. However, before going further into the discussion some definitions relating to the SA algorithm and combinatorial optimisation are first presented[1].

Definition 5.1 : An instance of a combinatorial optimisation problem can be formally declared as a pair (S, f) , where the solution space S

denotes the finite set of all possible solutions and the cost (objective) function f is a mapping defined as,

$$f : S \rightarrow \mathbb{R}. \quad 5.5$$

In the case of minimisation, the problem is that of finding the solution $i_{opt} \in S$ which satisfies,

$$f(i_{opt}) \leq f(i), \quad \forall i \in S. \quad 5.6$$

Such a solution i_{opt} is the globally-optimal solution, or simply optimum; $f_{opt} = f(i_{opt})$ denotes the optimal cost, and S_{opt} the set of optimal solutions.

Similar definition for maximisation problem can be made without loss of generality since maximisation is equivalent to minimisation with the sign of the cost function reversed. In the multiprocessor task scheduling problem, which is an instance of combinatorial optimisation problem, every feasible schedule is a solution and the goal is to find the schedule with minimum overhead. Henceforth, in all subsequent discussions unless explicitly stated, we consider combinatorial optimisation problems as minimisation problems.

Definition 5.2 : We consider an instance of combinatorial optimisation problem (S, f) . A *neighbourhood structure* can then be defined as the mapping

$$N : S \rightarrow 2^S, \quad 5.7$$

which defines for each solution $i \in S$, a set $S_i \subset S$ of solutions that are close to i in some sense. The set S_i is called the neighbourhood of solution i , and each $j \in S_i$ is called a neighbour of i . Furthermore, it is also assumed that $j \in S_i \iff i \in S_j$.

Definition 5.3 : An instance of combinatorial optimisation problem (S, f) and also its neighbourhood structure N are considered. A *generation mechanism* is then defined as a means of selecting a solution j from the neighbourhood S_i of i .

Comparison of combinatorial optimisation problems and thermodynamic behaviour of physical systems yields an expression for the probability distribution of the solutions due to SA algorithm similar to that given in eq.5.2. Therefore, given an instance (S, f) of a combinatorial optimisation problem and a suitable neighbourhood structure then, after a sufficiently large number of transitions at a fixed value of the temperature t , applying the acceptance probability of eq.5.4, the SA algorithm will find a solution $i \in S$ with a probability equal to,

$$\begin{aligned} \text{IP}\{X = i\} &\triangleq q_i(t) \\ &= \frac{1}{N_0(t)} \exp\left(\frac{-f(i)}{t}\right), \end{aligned} \quad 5.8$$

where X is a stochastic variable representing the current solution resulting from the SA algorithm, and

$$N_0(c) = \sum_{j \in S} \exp\left(\frac{-f(j)}{t}\right), \quad 5.9$$

represents a normalisation constant.

The above probability distribution (eq.5.8) is called the *stationary* or *equilibrium* distribution and is the equivalent of the Boltzmann distribution of eq.5.2. The normalisation constant $N_0(c)$ is equivalent to the partition function of eq.5.3.

If, in the SA algorithm sufficient number of transitions are allowed at each value of the temperature t so that the stationary distribution (eq.5.8) is attained, and that the temperature t is slowly reduced to very close to zero, it is then expected that the SA algorithm will find an optimal solution. The corollary[1] below represent the above and it's result is very important since it guarantees *asymptotic convergence* of the SA algorithm to the set of globally optimal solutions under the above mentioned conditions.

Corollary 5.1 : An instance (S, f) of a combinatorial optimisation problem, a suitable neighbourhood structure and the stationary distribution of eq. 5.8 are considered. We can then have,

$$\begin{aligned} \lim_{t \downarrow 0} q_i(t) &\triangleq q_i^* \\ &= \frac{1}{|S_{opt}|} \chi_{S_{opt}}(i), \end{aligned} \quad 5.10$$

where S_{opt} represents the set of globally optimal solutions. ¹

The proof of the above corollary is given in Appendix B.

As mentioned earlier, the SA algorithm can be modelled mathematically by using the theory of Markov chains. In such cases, the current solutions at any temperature t obtained by the SA algorithm are considered as stochastic variables (eq.5.8), the corresponding *transition probabilities* are then defined as follows :

$$\begin{aligned} \forall i, j \in S: \quad P_{ij}(\lambda) &= P_{ij}(t_\lambda) \\ &= \begin{cases} G_{ij}(t_\lambda) A_{ij}(t_\lambda) & \text{if } i \neq j \\ 1 - \sum_{\substack{l \in S \\ l \neq i}} P_{il}(t_\lambda) & \text{if } i = j, \end{cases} \end{aligned} \quad 5.11$$

where $G_{ij}(t_\lambda)$ represents the *generation probability*, i.e., the probability of generating a solution j from a solution i , and $A_{ij}(t_\lambda)$ represents the *acceptance probability*, i.e., the probability of accepting the solution j , once it is generated from solution i .

The $G_{ij}(t_\lambda)$ and $A_{ij}(t_\lambda)$ are both conditional probabilities and are defined as follows :

a. Generation Probability :

$$\forall i, j \in S: \quad G_{ij}(t_\lambda) = G_{ij} = \frac{1}{\Theta} \chi_{S_i}(j), \quad (5.12)$$

¹ Let, A and $A' \subset A$ be two sets. Then the characteristic function $\chi_{(A')}: A \rightarrow \{0, 1\}$ of the set A' is defined as $\chi_{(A')}(a) = 1$ if $a \in A'$, and $\chi_{(A')}(a) = 0$ otherwise.

where $\Theta = |S_i|$, for all $i \in S$.

b. Acceptance Probability :

$$\forall i, j \in S : \quad A_{ij}(t_\lambda) = \exp\left(-\frac{(f(j) - f(i))^+}{t_\lambda}\right), \quad (5.13)$$

where, for all $a \in \mathbb{R}$, $a^+ = a$ if $a > 0$, and $a^+ = 0$ otherwise.

Thus, the generation probabilities are independent of the control parameter t_λ and spread uniformly over the neighbourhoods S_i , where it is assumed that all neighbourhoods are of equal size, i.e. $|S_i| = \Theta$, for all $i \in S$. The acceptance probabilities are given by the acceptance criterion of eq.5.4 and is thus identical to the Metropolis criterion. The matrices corresponding to transition and generation probabilities, $P(t_\lambda)$ (transition matrix) and $G(t_\lambda)$ (generation matrix) respectively are stochastic, but the acceptance matrix $A(t_\lambda)$ corresponding to the acceptance probability $A_{ij}(t_\lambda)$ is not.

The definitions of the generation and acceptance probabilities (eqs. 5.12 & 5.13) supporting the Markov chain model of the SA algorithm, correspond fully to the original definition of the algorithm and closely follow the physical analogy discussed earlier in Sec.5.1.1. Furthermore, the above definitions formalising the SA algorithm generally apply to all forms of combinatorial optimisation problems.

Formulation of a set of conditions that ensures asymptotic convergence of the SA algorithm for general class of generation and acceptance probabilities would then give a sound basis for the Markov chain model of the SA algorithm. These conditions apply totally or partially to the following properties of the SA algorithm.

- a. *Reachability of the set of global optimum.* The set of global optima is reached from every starting solution with probability 1.
- b. *Asymptotic independence of starting solution.* The dependence of the distribution of $f_i(\lambda)$ for all $i \in S$ and at temperature t_λ with respect to the starting solution vanishes as $\lambda \rightarrow \infty$.
- c. *Convergence in distribution.* $f_i(\lambda)$ converges in distribution.

- d. *Convergence to a global optimum.* The algorithm converges to the set of global optima with probability 1.

Sufficient conditions for all or some of the above convergence properties were independently obtained by various researchers [4–9]. As for the convergence to the set of global optima, the SA algorithm finds with probability one an optimal solution if, after a large number of trials, we have

$$\mathbb{P}\{X(\lambda) \in S_{opt}\} = 1. \quad 5.14$$

The condition for asymptotic convergence of the SA algorithm to the set of optimal solutions can then be formulated as,

$$\lim_{\lambda \rightarrow \infty} \mathbb{P}\{X(\lambda) \in S_{opt}\} = 1. \quad 5.15$$

The asymptotic convergence of the SA algorithm has been proved for both homogeneous and inhomogeneous Markov chain models[1,9]. The proof for the homogeneous chain model requires an infinite number of transitions to approximate a stationary distribution arbitrarily close. Thus, implementation of the SA algorithm along this line would require generation of a sequence of infinitely long homogeneous Markov chains at descending values of temperature. This is visibly impractical. However, some moderations can be made whereby, the SA algorithm can be described as a sequence of homogeneous Markov chains of finite length, generated at descending values of temperature. This in turn converts the homogeneous Markov chains into one single inhomogeneous Markov chain. In this way, it is possible to reduce the sequence of infinitely long homogeneous Markov chain to a single inhomogeneous Markov chain of infinite length. The practical implementations to be discussed later are approximations of this inhomogeneous Markov chain model of the SA algorithm.

5.3 Cooling Schedule

A practical implementation of the SA algorithm involves an implementation in which a sequence of Markov chains is generated at descending values of the control parameter, temperature. To achieve this, the set of parameters that governs the convergence of the algorithm must be specified.

The combination of these parameters are known as cooling schedule . A cooling schedule specifies :

- a finite sequence of values of the control parameter, i.e.
 - an *initial value* of the control parameter t_0 ,
 - a *decrement function*, α for decreasing the value of the control parameter,
 - a *final value* of the control parameter, t_{end} specified by a stop criterion, and
- a finite number of transitions at each value of the control parameter, i.e.
 - a finite length of each homogeneous Markov chain.

In this section, we discuss some general features and characteristics of a cooling schedule and also present two simple but effective cooling schedules. However, we first introduce a new term the *quasi equilibrium* [1] which can be defined as follows.

Definition 5.4 : Let L_λ represent the length of the λ^{th} Markov chain and t_λ the corresponding value of temperature. Then *quasi equilibrium* is achieved if $a(L_\lambda, t_\lambda)$, i.e. the probability distribution of the solutions after L_λ trials of the λ^{th} Markov chain, is 'sufficiently close' to $q(t_\lambda)$, the stationary distribution at t_λ , defined by eqs.5.8 and 5.9, i.e.

$$\|a(L_\lambda, t_\lambda) - q(t_\lambda)\| < \varepsilon, \quad 5.16$$

for some specified positive value of ε .

A very large number of transitions quadratic in the size of the solution space will be required to hold for arbitrarily small values of ε . This leads for most combinatorial optimisation problems to an exponential-time execution of the SA algorithm. Thus, a practical implementation of the algorithm requires a relaxation of the rigid quantification of the quasi equilibrium condition.

A cooling schedule using the concept of quasi equilibrium can be devised on the basis of the following arguments. Let, the acceptance probability in the SA algorithm be given by eq.5.13, then, for $t \rightarrow \infty$, the stationary distribution is given by the uniform distribution on the set of solutions S , i.e.

$$\lim_{t \rightarrow \infty} q(\lambda) = \frac{1}{|S|} \mathbf{1}, \quad 5.17$$

where $\mathbf{1}$ denotes the $|S|$ -vector with all components equal to 1. The above equation can also be derived from eqs.5.8 and 5.9. Therefore, by choosing the value of t_λ sufficiently large — allowing acceptance probability of virtually all proposed transitions — quasi equilibrium is directly achieved at these values of temperature, since in this case all solutions occur with equal probability given by the uniform distribution of eq.5.17. The length of the Markov chain and the decrement function must then be chosen so that quasi equilibrium is restored at the end of each individual Markov chain. The equilibrium distribution for the various Markov chains will thus be 'closely followed', so as to arrive eventually, as $t_\lambda \downarrow 0$, close to q^* , the uniform distribution on the set of optimal solutions.

From the above, it is evident that large decrements in t_λ will require longer Markov chain lengths in order to restore quasi equilibrium at the next value of the temperature, $t_{\lambda+1}$. Thus, there is a trade-off between large decrements of the control parameter and small Markov chain lengths. The popular option is for small decrements in t_λ to avoid long chains, or alternatively for large values for L_λ in order to be able to make large decrements in t_λ .

Many different cooling schedules have so far been reported in search of an efficient and adequate schedule. Reviews are given by Collins, Eglese and Golden [10] and also by Van Laarhoven and Aarts [11]. In the following two sub-sections we present two cooling schedules as illustrations.

5.3.1 A Simple Cooling Schedule

The cooling schedule presented here is proposed by Kirkpatrick, Gelatt and Vecchi [3]. This particular schedule is based on a number of conceptually simple empirical rules and is expected to run in polynomial time.

* Initial value of the control parameter : As already stated earlier, the value of t_0 should be large enough to allow virtually all transitions to be accepted. This can be achieved by setting the *initial acceptance ratio* $\zeta_0 = \zeta(t_0)$ close to 1. The setting of this initial high temperature is equivalent to raising the temperature of the heat bath so that the solid in the bath completely melts.

* Decrement of the control parameter : From the discussion above, the decrement function can be defined as,

$$t_{\lambda+1} = \alpha t_{\lambda}, \quad \lambda = 1, 2, \dots \quad 5.18$$

where α is a constant smaller than but usually close to 1. Typical values of α where very small changes in the value of the control parameter are favoured, lies between 0.8 and 0.99.

* Final value of the control parameter : The algorithm is stopped from further execution when the value of the cost function of the solution obtained in the last trial of the Markov chain remains unchanged for a number of consecutive chains.

* Length of the Markov chain : The length of the Markov chain is calculated from the requirement that at each value t_{λ} of the control parameter quasi equilibrium is restored. The number of necessary transitions to achieve this is calculated from the intuitive argument, that, quasi equilibrium will be restored after acceptance of at least some fixed number of transitions. However, since transitions are accepted with decreasing probability, one would obtain $L_{\lambda} \rightarrow \infty$ for $t_{\lambda} \downarrow 0$. As a result, L_{λ} is usually bounded by some constant \bar{L} to avoid extremely long Markov chains for small values of t_{λ} .

5.3.2 A Polynomial-Time Cooling Schedule

The cooling schedule proposed by Aarts and Van Laarhoven [11] leads to a polynomial time execution of the SA algorithm. To differentiate it from the simple schedule of Sec.5.3.1 we shall call it *Polynomial-Time Schedule* in all future references following the original nomenclature even though both the schedules are expected to run in polynomial time. The polynomial-time

schedule is by its very design, more attuned with the statistical behaviour of the problem instance and as such is expected to perform better than the simple schedule of Sec.5.3.1. However, as is the norm of a heuristic algorithm this schedule fails to give any guarantee for the deviation in cost between the final solution obtained by the algorithm and the optimal cost.

* Initial value of the control parameter : The initial value t_0 of the control parameter should be such that at this temperature virtually all proposed transitions are accepted. For a sequence of trials generated at a certain temperature t , we assume that m_1 , represents the number of proposed transitions from i to j for which $f(j) \leq f(i)$, and, m_2 the number of transitions for which $f(j) > f(i)$. Also let, $\overline{\Delta f}^+$ be the average difference in cost over the m_2 cost-increasing transitions. Then, the acceptance ratio can be approximated as,

$$\zeta \approx \frac{m_1 + m_2 \exp(-\frac{\overline{\Delta f}^+}{t})}{m_1 + m_2}, \quad 5.19$$

from which it is easy to obtain,

$$t = \frac{\overline{\Delta f}^+}{\ln(\frac{m_2}{m_2\zeta - m_1(1-\zeta)})}. \quad 5.20$$

The initial temperature t_0 can be calculated from eq.5.20 in the following manner. Initially, t_0 is set equal to zero and then a sequence of m_0 trials is generated. After each trial a new value of t_0 is calculated from eq.5.20, where ζ is set to ζ_0 , the initial acceptance ratio. The values of m_1 and m_2 correspond to the cost-decreasing and cost-increasing transitions respectively and also $m_0 = m_1 + m_2$. The new value of t_0 is used in the next trial. The value of the control parameter thus slowly converges to the desired starting value t_0 , that produces the specified acceptance ratio ζ_0 .

* Decrement of the control parameter : It has already been stated that when the temperature, t is decreased very slowly the resulting stationary distribution of the homogeneous Markov chains will be close to each other. As a result, after decreasing t_λ to $t_{\lambda+1}$, a small number of

transitions are needed to restore the quasi equilibrium at $t_{\lambda+1}$, provided quasi equilibrium holds at t_{λ} . The condition for quasi equilibrium can thus be assumed to be,

$$\forall \lambda \geq 0: \quad \|q(t_{\lambda}) - q(t_{\lambda+1})\| < \varepsilon, \quad 5.21$$

for some positive value of ε . Thus, we assume that quasi equilibrium is maintained throughout the optimisation process if eq.5.21 holds for all λ . This requires that quasi equilibrium is achieved at t_0 .

For two successive values of the control parameter, the stationary distributions need to be close to each other. This can be quantitatively expressed as,

$$\forall i \in S: \quad \frac{1}{1+\delta} < \frac{q_i(t_{\lambda})}{q_i(t_{\lambda+1})} < 1+\delta, \quad \lambda = 0, 1, \dots \quad 5.22$$

for some small positive number δ , which can be related to eq.5.21. The above inequalities of eq.5.22 are satisfied if the the following condition holds,

$$\forall i \in S: \quad \frac{\exp(-\frac{\delta_i}{t_{\lambda}})}{\exp(-\frac{\delta_i}{t_{\lambda+1}})} < 1+\delta, \quad \lambda = 0, 1, \dots \quad 5.23$$

where $\delta_i = f(i) - f_{opt}$.

Equation 5.23 can be rewritten to give the following condition on two successive temperature values,

$$\forall i \in S: \quad t_{\lambda+1} > \frac{t_{\lambda}}{1 + \frac{t_{\lambda} \ln(1+\delta)}{f(i) - f_{opt}}}, \quad \lambda = 0, 1, \dots \quad 5.24$$

Using the empirical knowledge[1] that the probability distribution of the cost values of the solutions is 'normal' near the average value of the cost function and 'exponential' in the region close to the optimal value of the cost function, eq.5.24 can be simplified for 99% and 95% confidence limits of the normal and exponential distributions as

$$t_{\lambda+1} > \frac{t_{\lambda}}{1 + \frac{t_{\lambda} \ln(1+\delta)}{\langle f \rangle_{t_{\lambda}} - f_{opt} + 3\sigma_{t_{\lambda}}}}, \quad \lambda = 0, 1, \dots \quad 5.25$$

For many instances of combinatorial optimisation problems the value of f_{opt} is not known. However, the average value ($\langle f \rangle$) and the spreading (σ) of the cost function typically exhibit a similar behaviour as a function of the control parameter. $\langle f \rangle_{t_\lambda} - f_{opt} + 3\sigma t_\lambda$ can thus be replaced by $3\sigma t_\lambda$ and the omission of the term $\langle f \rangle_{t_\lambda} - f_{opt}$ can be counterbalanced by choosing smaller value of δ . Thus eq.5.25 can be rewritten as

$$t_{\lambda+1} = \frac{t_\lambda}{1 + \frac{t_\lambda \ln(1+\delta)}{3\sigma t_\lambda}}, \quad \lambda = 0, 1, \dots \quad 5.26$$

The amount by which the value of t is decreased by the decrement function of eq.5.26 is determined by the value of the *distance parameter* δ . Small δ -values lead to small decrements and large δ -values lead to large decrements in t .

* Final value of the control parameter : The execution of the SA algorithm can be terminated by extrapolating the expected cost $\langle f \rangle_{t_\lambda}$ for $t_\lambda \downarrow 0$. Let,

$$\Delta \langle f \rangle_t = \langle f \rangle_t - f_{opt}, \quad 5.27$$

then execution of the algorithm is terminated if $\Delta \langle f \rangle_t$ is very small compared to the expected cost at t_0 , $\langle f \rangle_{t_0}$. For sufficiently large values of t_0 , $\langle f \rangle_{t_0} \approx \langle f \rangle_\infty$. Hence, $\Delta \langle f \rangle_t$ can be approximated for $t \ll 1$ as,

$$\Delta \langle f \rangle_t \approx t \frac{\partial \langle f \rangle_t}{\partial t}. \quad 5.28$$

Therefore, the algorithm can be reliably terminated when,

$$\forall \lambda \geq 0: \quad \frac{t_\lambda}{\langle f \rangle_\infty} \frac{\partial \langle f \rangle_t}{\partial t} \Big|_{t=t_\lambda} < \epsilon_s, \quad 5.29$$

where ϵ_s is a small positive number. ϵ_s is the stop parameter and eq.5.29 is the *stop criterion*.

* Length of the Markov chain : The discussion for the simple cooling schedule (Sec.5.3.1) applies here as well. However, in practice, the length of the Markov chain is made equal to the size Θ of the neighbourhoods, i.e.

$$L_\lambda = L = \Theta, \quad \lambda = 0, 1, \dots \quad 5.30$$

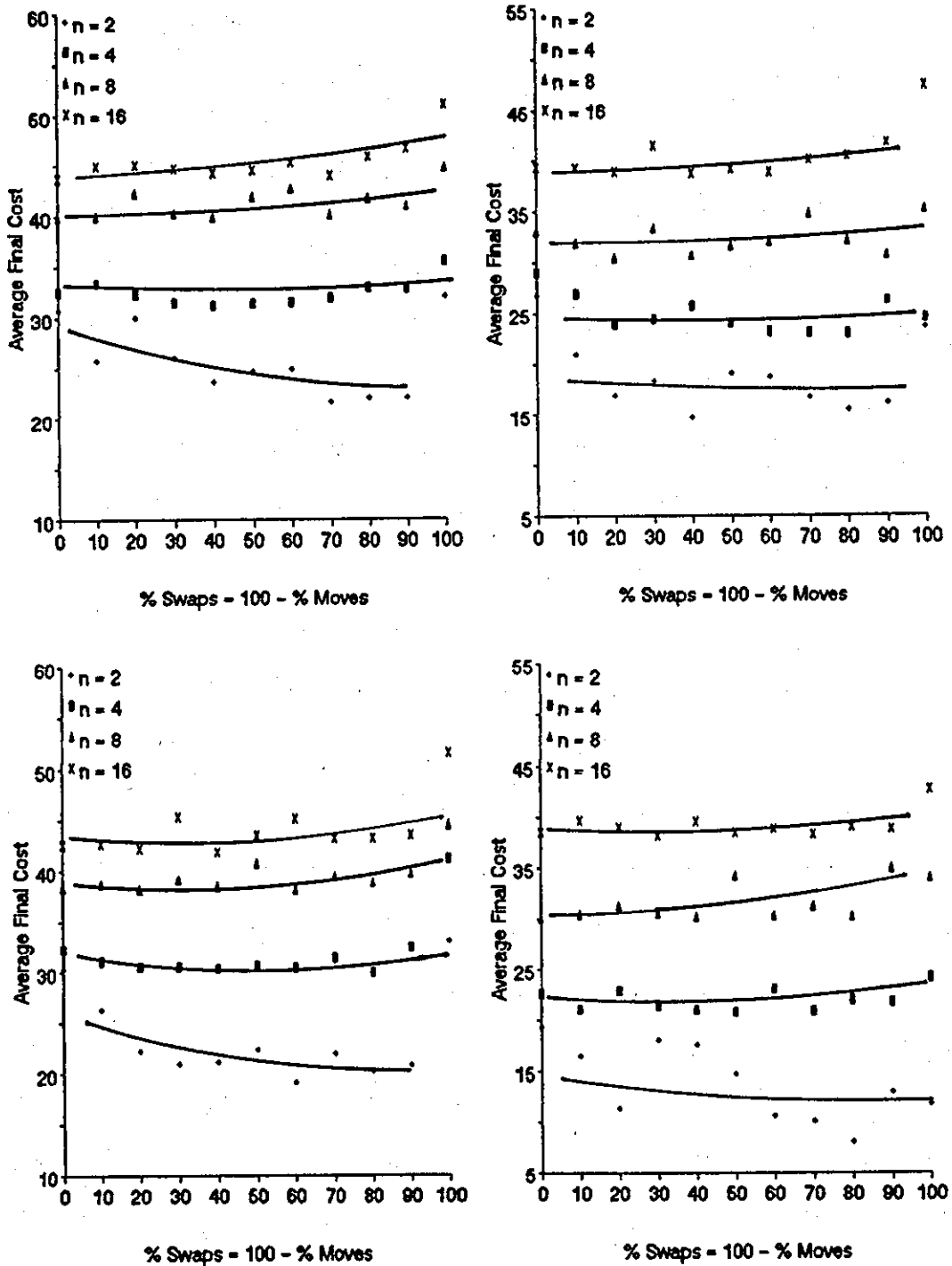


Fig. 5.3 Effect of different *Swap-Move* composition on the final cost. Both simple cooling schedule (top) & polynomial-time cooling schedule (bottom) are used. Graph data instances used are 4x4 multiplier ckt. (left) & frequency locked loop ckt. (right).

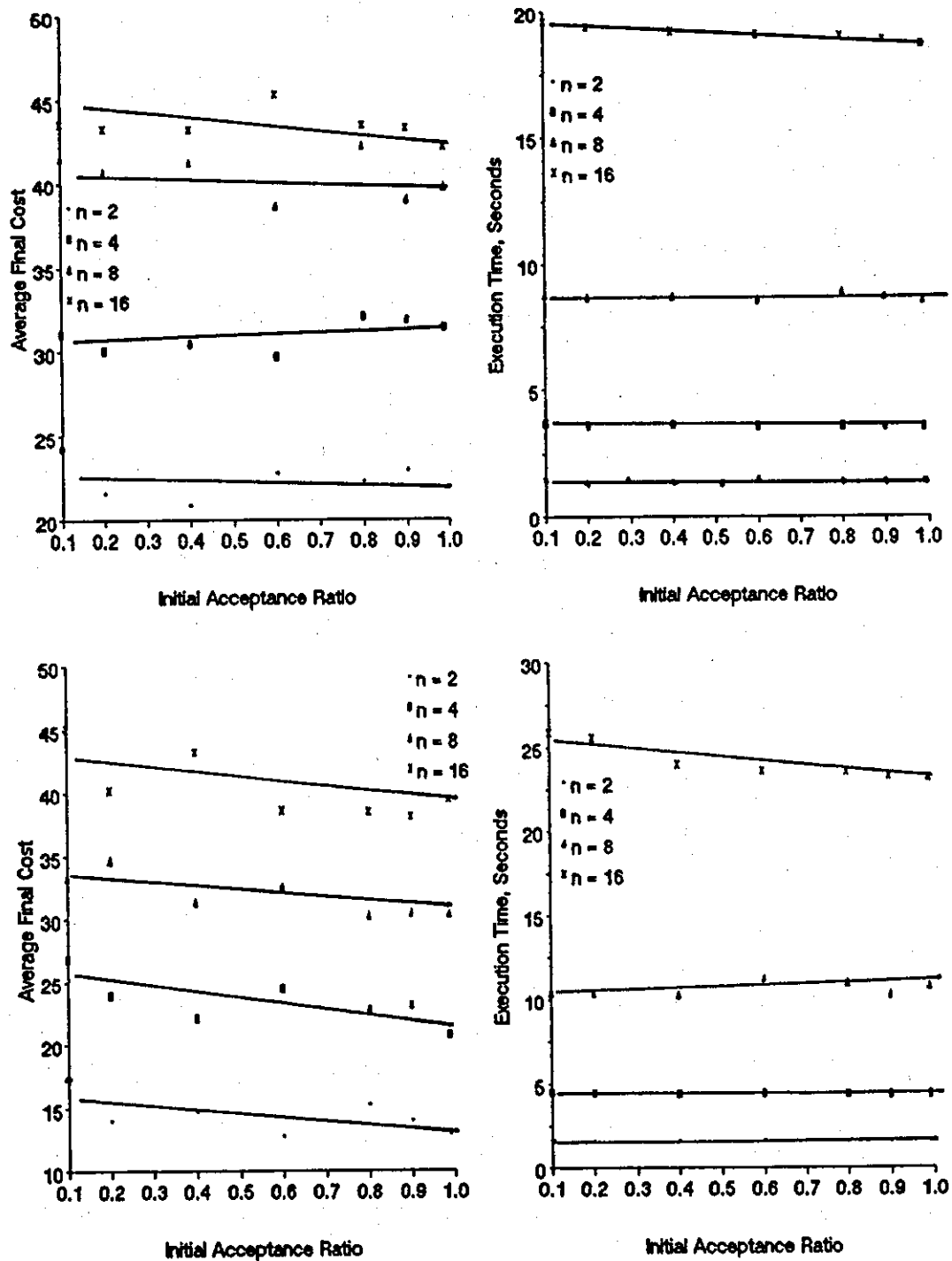


Fig. 5.4 Average final cost (left) & average execution time (right) as functions of the initial acceptance ratio, ζ'_0 . The simple cooling schedule is in use. Graph data instances used are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

5.4 Implementation of the SA algorithm

In this section, we shall discuss a simple single processor implementation of the SA algorithm for the solution of the multiprocessor task scheduling problem as encountered in concurrent VLSI timing simulation (c.f. Chapter 3). Implementation of the SA algorithm requires a sequence of Markov chains to be generated at descending values of the control parameter, temperature.

A generation mechanism is devised, and individual Markov chains are generated, by attempting to transform a current solution into a subsequent one, by repeatedly applying the generation mechanism and the acceptance criterion. An implementation of the SA algorithm requires the specification of the following three items :

1. a concise problem representation,
2. a transition mechanism, and
3. a cooling schedule.

The above three items are now discussed in detail in the following sub-sections.

5.4.1 Concise Problem Representation

A suitable concise description of the problem is necessary to represent the solution space, and also to evaluate an expression for the cost (objective) function. The cost function must be able to represent the cost effectiveness of the various solutions with respect to the objective of the optimisation process. Sections 3.4, 3.5 and 3.6 cover these issues for the present problem.

5.4.2 Transition Mechanism

Three distinct steps are involved in the process of generating trials for transforming a current solution into a subsequent one. At first, a generation mechanism is used to generate a new solution. Secondly, the difference in cost between the two solutions is calculated and finally, based on the result of the second step a decision is made whether or not to accept the new solution as the current solution.

The generation mechanism as described earlier is used to generate a new solution from the current solution by bringing in a simple modification of the current configuration of the problem instance. For the multiprocessor task scheduling problem modelled on a directed acyclic graph (DAG), this modification can be easily accomplished by utilising any one of two simple processes. The first is the *move* and the other is the *swap* (c.f. Section 3.7). A move involves changing the processor allocation of any one node of the graph representing the problem instance from one processor to another while in swap, processor allocations of any two nodes with differing allocations are interchanged. A swap, thus, can be viewed as the equivalent of two moves. In a practical implementation of the SA algorithm or for that matter any heuristic algorithm based on the iterative improvement method, a mix of moves and swaps is deemed better than using any of these two techniques alone.

The evaluation of trials is the most important time consuming part of the SA algorithm and therefore, needs to be done as efficiently as possible. The two techniques, move and swap, allow generation of new solutions by simple re-arrangement of the current configuration of the problem instance. Calculation of the cost difference is needed to be done quickly and as such methods that calculate the incremental cost difference are preferred. In Sec.3.7.2 and 3.7.3 these issues are covered.

The decision to accept new solutions is based on the Metropolis criterion,

$$IP_t\{\text{accept } i\} = \begin{cases} 1 & ; \text{ if } \Delta f \leq 0 \\ \exp(-\frac{\Delta f}{t}) & ; \text{ if } \Delta f > 0, \end{cases} \quad 5.31$$

where t represents the control parameter, temperature and Δf the difference in cost between a new and a current solution.

5.4.3 The Cooling Schedule

Carrying out the optimisation based on the annealing process requires specification of the parameters determining the cooling schedule. These are the initial value of the control parameter, temperature; a decrement function of this control parameter, the length of the individual Markov chains and a stop criterion. Two different cooling schedules are examined. The first based

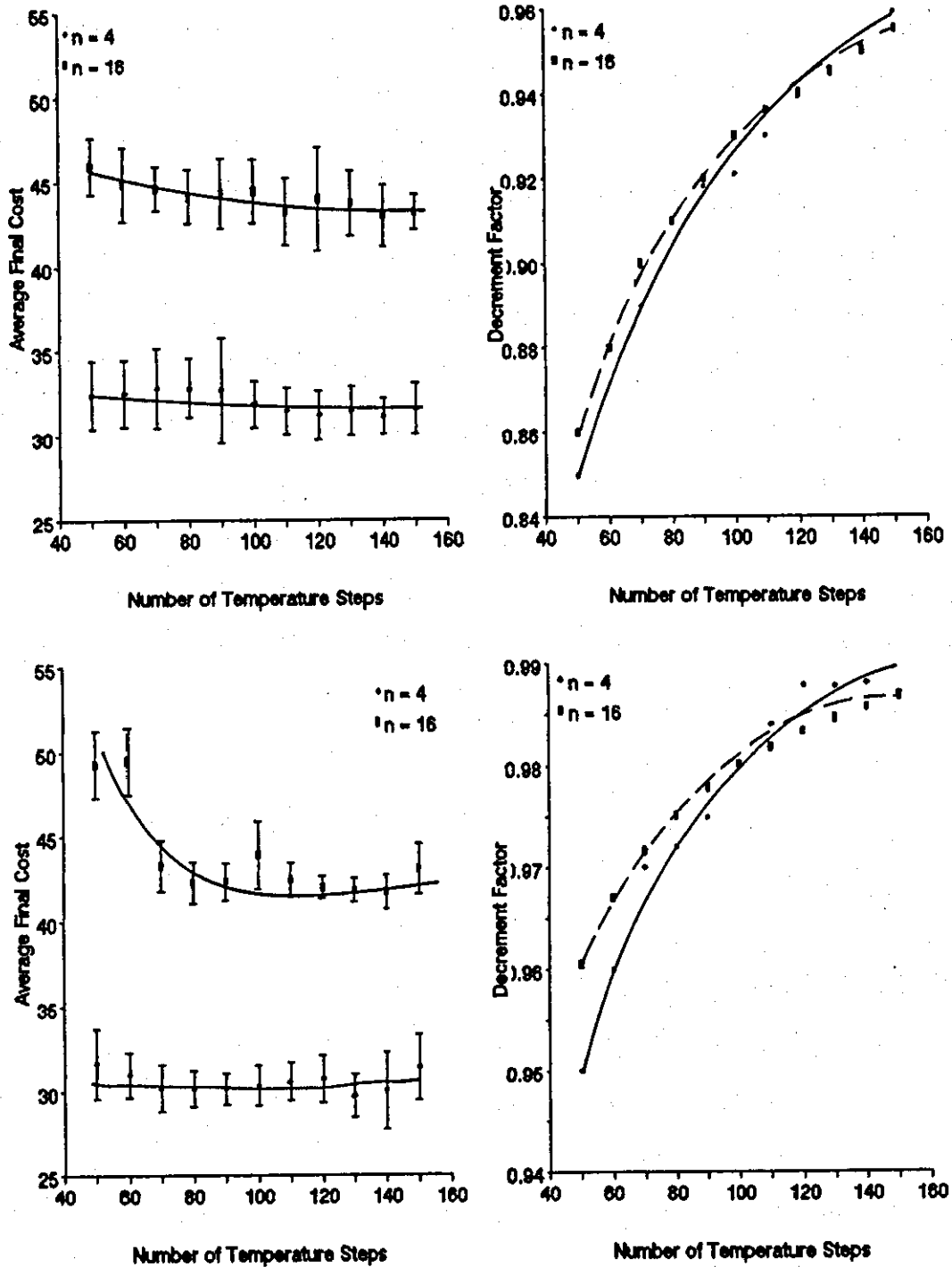


Fig. 5.5 Average final cost (left) and the decrement factor (right) as functions of the number of temperature steps. The simple cooling schedule is in use. Data flow graph for the 4x4 multiplier ckt. is used. Initial acceptance ratio, $\zeta'_0 = 0.99$ (top) & $\zeta'_0 = 0.10$ (bottom).

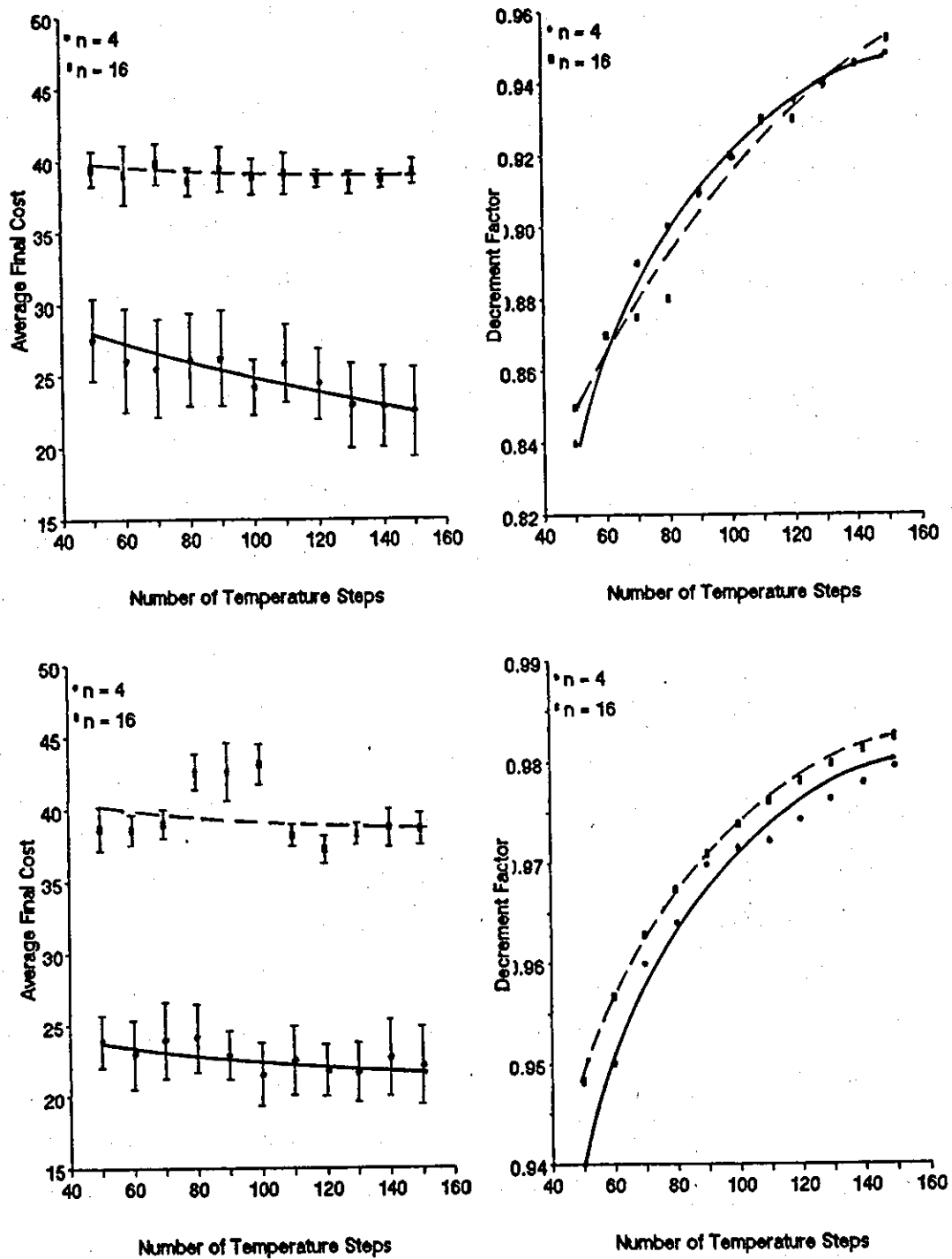


Fig. 5.6 Average final cost (left) and the decrement factor (right) as functions of the number of temperature steps. The simple cooling schedule is in use. Data flow graph for the frequency locked loop ckt. is used. Initial acceptance ratio, $\zeta'_0 = 0.99$ (top) & $\zeta'_0 = 0.10$ (bottom).

on empirical experience and is very similar to that of Sec.5.3.1. The other is an implementation of Aarts and Van Laarhoven's (Sec.5.3.2) polynomial-time cooling schedule. It is to be noted that both of these schedules run in polynomial time and is expected to provide near-optimal solutions.

Simple Cooling Schedule

Here we adopt a very simple definition for the initial acceptance ratio, which is necessary to set the initial high temperature. The initial acceptance ratio ζ'_0 is defined as follows :

$$\zeta'_0 = \exp \left(-\frac{\overline{\Delta f}}{t_0} \right), \quad 5.32$$

where as before, t_0 is the corresponding initial high temperature and $\overline{\Delta f}$ is average difference in cost for all cost-increasing and cost-decreasing transitions. From the above, the value of t_0 can be obtained as,

$$t_0 = -\frac{\overline{\Delta f}}{\ln(\zeta'_0)}. \quad 5.33$$

Though the above is a static definition as compared to the more accurate one of eq.5.19, the results obtained are acceptable.

Unlike the simple schedule of Sec.5.3.1 we here, place a bound on the number of temperature steps, i.e. the number of homogeneous Markov chains at descending values of temperature. From experience, it is found that an upper bound of 100 temperature steps is acceptable for almost all small to medium sized instances of the task scheduling problem. In order to make this implementation of the SA algorithm independent of the problem size, it is decided to set the number of temperature steps to 100. It is also assumed that the probability of accepting a cost-increasing transition at the last temperature step is very small. This assumption leads to the calculation of the decrement function. We let,

$$\begin{aligned} \zeta'_{end} &= \text{probability of accepting a cost-increasing transition} \\ &\quad \text{at the last temperature step,} \\ t_{end} &= \text{the corresponding temperature (c.f. eq.5.33),} \\ &\quad \text{and,} \end{aligned}$$

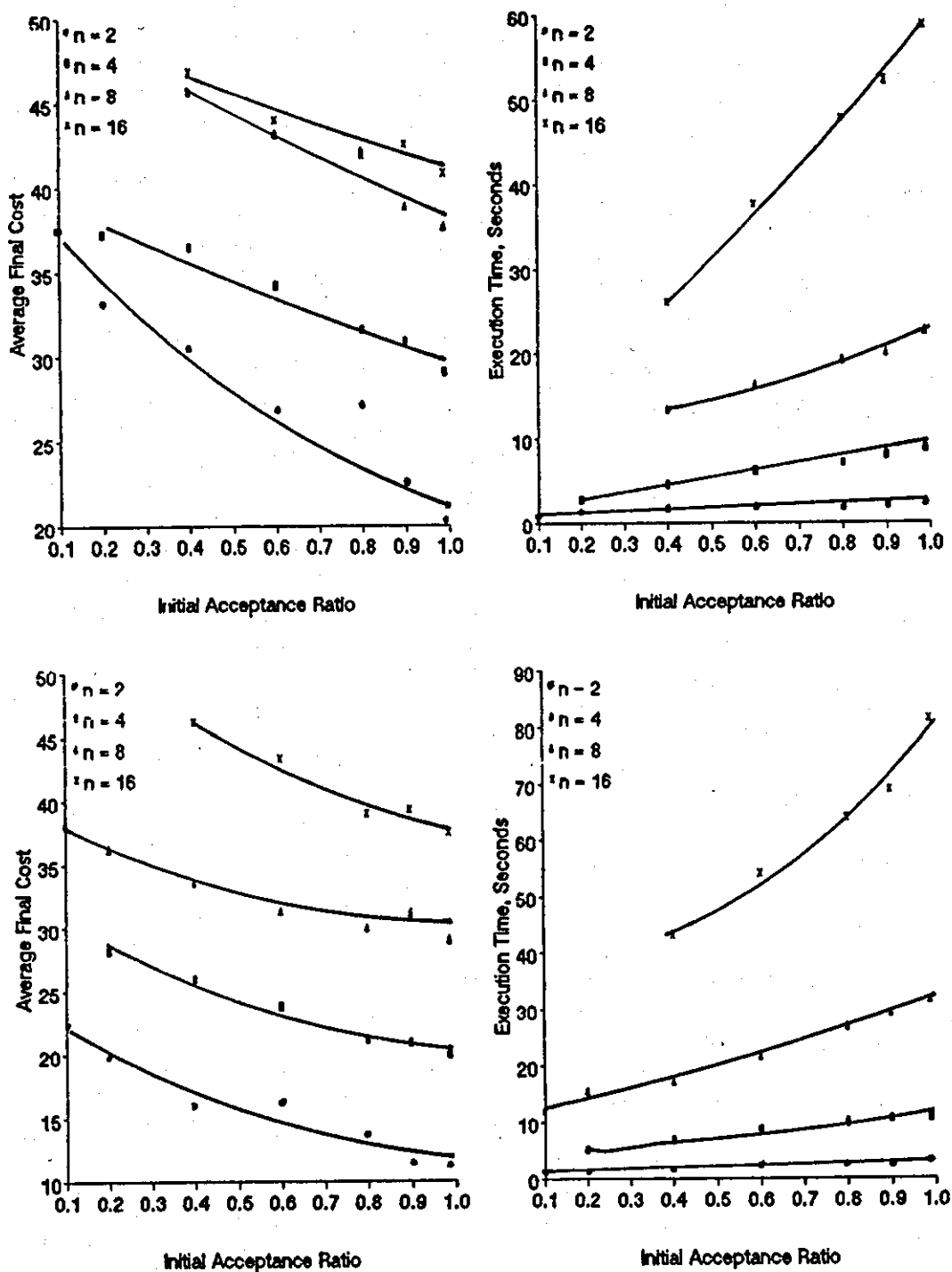


Fig. 5.7 Average final cost (left) & average execution time (right) as functions of the initial acceptance ratio, ζ_0 . The polynomial-time cooling schedule is in use. Graph data instances used are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

N_T = the total number of temperature steps.

Then the decrement function can be expressed as,

$$\alpha = \left(\frac{t_{end}}{t_0} \right)^{\frac{1}{N_T}}. \quad 5.34$$

The total number of transitions allowed i.e. the length of the Markov chain at each value of temperature is bounded by the size of the neighbourhood structure. We recall that for the task scheduling problem the suitable transition mechanisms are moves and swaps. For simplicity, we here take the case of move. When a node of the graph representing the problem instance is selected for a move, the number of available processors to which it can be allocated (moved) to is $n - 1$, where n is the number of processors. For k nodes of the graph the total number of choices for a single move is thus, $k(n - 1)$ and this is the number of neighbouring solutions of any arbitrary solution i . From this it is easy to derive an expression for the length of the Markov chain (i.e. the number of transitions allowed) at each value of temperature, t_λ as,

$$L_\lambda = m k (n - 1), \quad \lambda = 0, 1, \dots \quad 5.35$$

where m is a constant whose value determines a widening or shortening of the Markov chain. Evidently, a high valued m resulting in a long Markov chain would result in better solutions, as it would approach an infinitely long Markov chain more closely. But, this advantage could be inhibited by very long computing time. A sensible value of m is between 1 and 5.

For the practical implementation of the simple cooling schedule, the following parameter values are used

initial acceptance ratio, ζ'_0 = 0.99,
 final acceptance ratio, ζ'_{end} = 0.00001,
 total number of temp. steps, N_T = 100,
 neighbourhood size parameter, m = 1.

and the value of $\overline{\Delta f}$ is determined from a trial run involving L_λ transition with m set at 5.

The Polynomial-Time Cooling Schedule

This implementation adheres fully to the Aarts and Van Laarhoven's[11] polynomial-time schedule. The following parameter values are used

initial acceptance ratio, ζ_0	= 0.98,
distance parameter, δ	= 0.1,
stop parameter, ϵ_s	= 0.00001,
neighbourhood size parameter, m	= 1.

The lengths of the Markov chains (i.e. the total number of transitions attempted) at each value of the control parameter, temperature is similar to that of preceding cooling schedule.

The noticeable difference between these two cooling schedules is that the latter is more computationally intensive and also for very small but reasonable value of the stop parameter, ϵ_s , would require more temperature steps, thereby taking more time to reach the stop criterion of the SA algorithm. However, the perceptible benefit is that better solutions are expected.

5.4.4 Performance Analysis of the SA Algorithm

The two cooling schedules namely, the simple cooling schedule and the polynomial-time cooling schedule described earlier were implemented on a single processor in order to ascertain their performance in solving the multiprocessor task scheduling problem. The performance analysis described here involves the finite-time behaviour of the SA algorithm. This analysis is based on both the cooling schedules investigating the performance of the SA algorithm as a function of the following parameters :

- a. Simple cooling schedule: swap-move ratio, initial acceptance ratio (ζ'_0), the number of temperature steps (N_T) and size parameter (m).
- b. Polynomial-time cooling schedule: swap-move ratio, initial acceptance ratio (ζ_0), stop parameter (ϵ_s), distance parameter (δ) and size parameter (m).

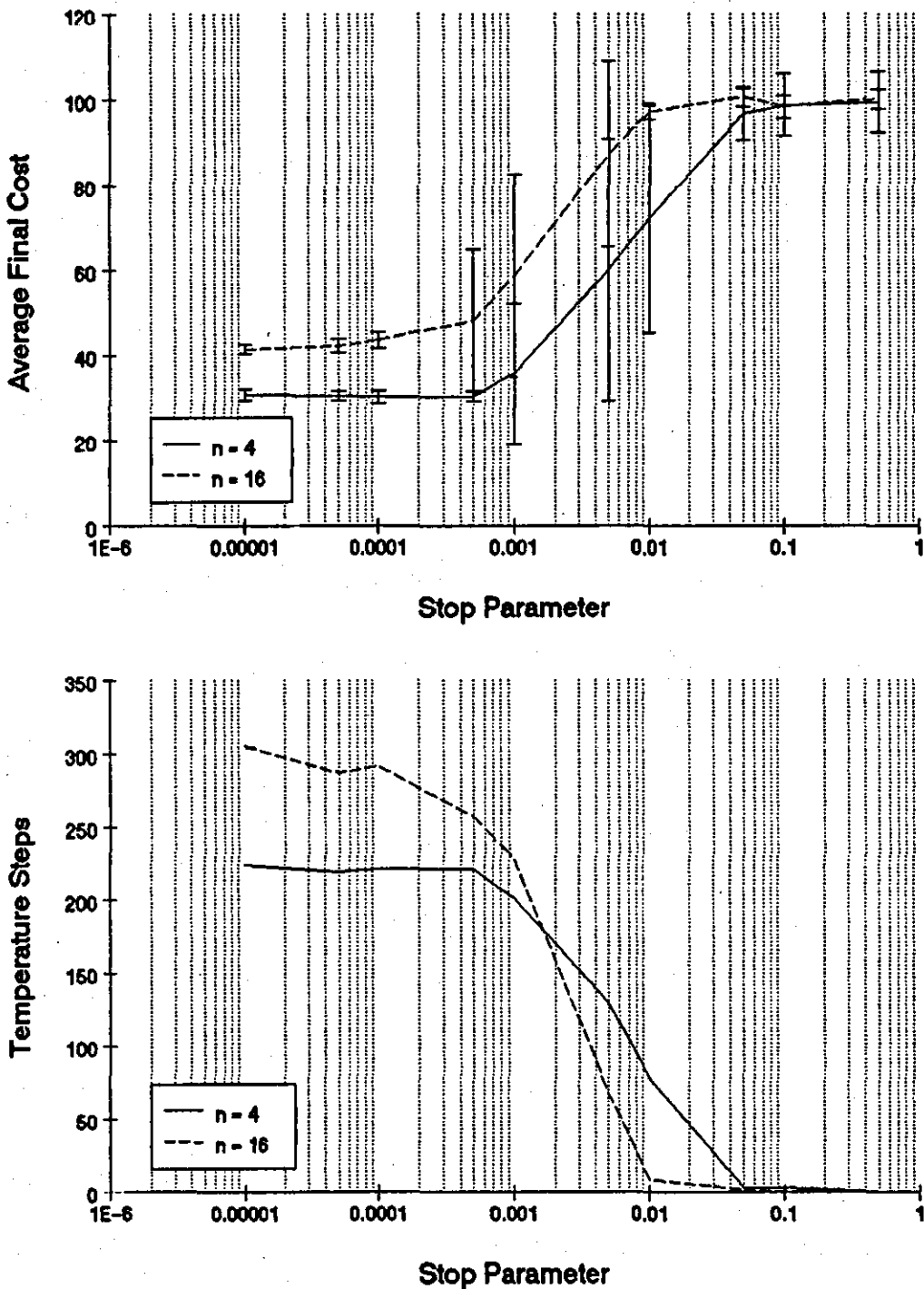


Fig. 5.8 Average final cost (top) & the number of temperature steps (bottom) as functions of the stop parameter, ϵ , for fixed values of $\zeta_0 (= 0.98)$ and $\delta (= 0.1)$. The polynomial-time cooling schedule & the graph instance for the 4x4 multiplier ckt. are used.

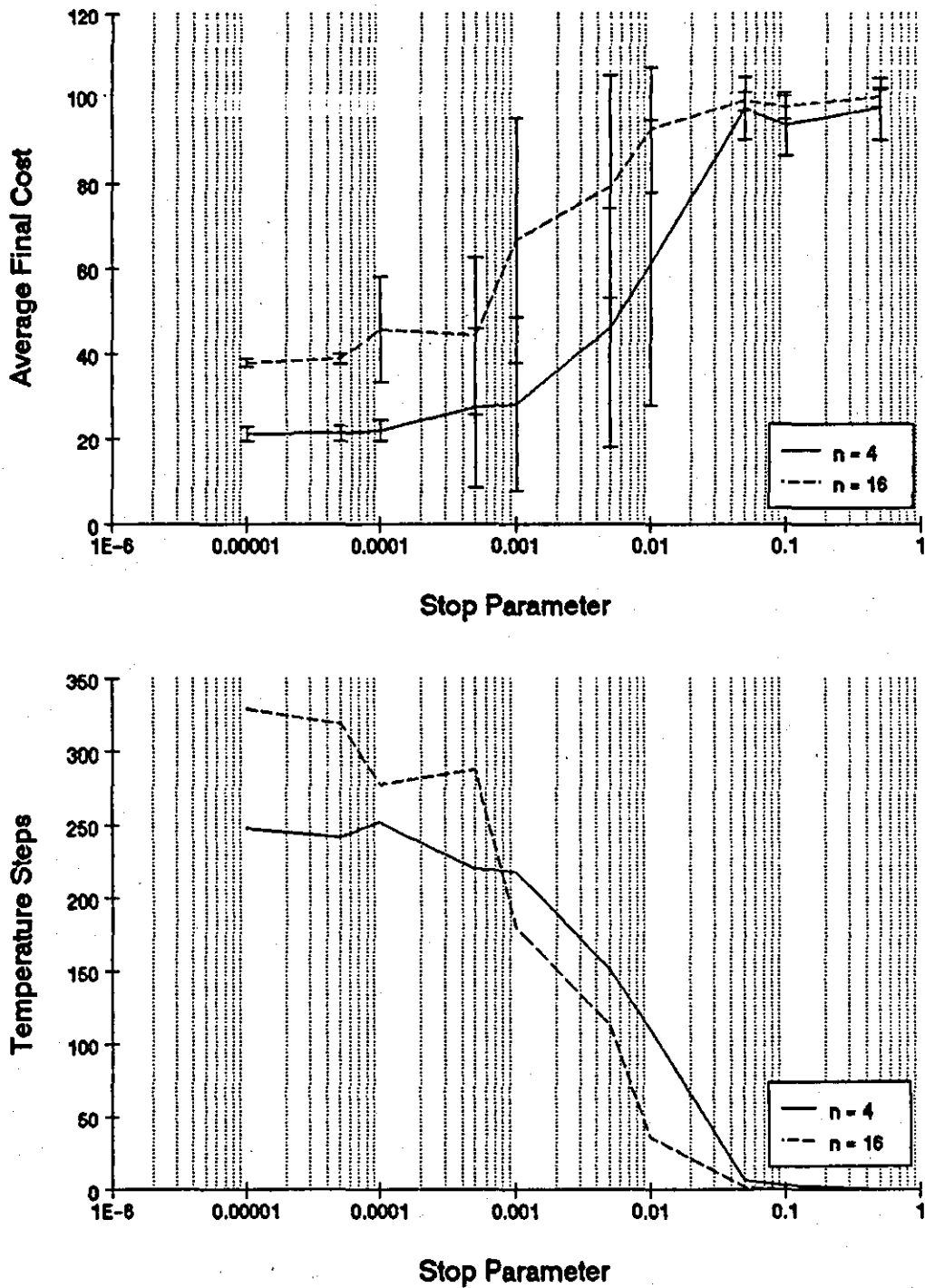


Fig. 5.9 Average final cost (top) & the number of temperature steps (bottom) as functions of the stop parameter, ϵ , for fixed values of $\zeta_0 (= 0.98)$ and $\delta (= 0.1)$. The polynomial-time cooling schedule & the graph instance for the frequency locked loop ckt. are used.

As is the norm, the performance of the heuristic algorithm is related to the quality of the final solution obtained by the algorithm and also the running time required. As in chapter 4, the final solution here is expressed as the percentage of the average initial random scheduling cost. The running-time of the algorithm for the polynomial-time schedule is in some cases represented as the number of temperature steps required. This is valid, as a direct relationship between the number of temperature steps and the actual running-time can be easily established.

The performance of the SA algorithm has been investigated by carrying out an average-case analysis, which relates to the average value of the final solution and the running-time (or the number of temperature steps) computed from the probability distribution over the set of final solutions that can be obtained by the algorithm for a given problem instance. This results directly from the probabilistic nature of the SA algorithm. The number of iterations for each investigation lies between 30 and 50.

The problem instances used here for the investigation are the graphs representing the 4x4 multiplier circuit (58 nodes) and the frequency locked loop circuit (68 nodes). The larger graph instances are avoided here for the sake of investigation time. The processor sizes (graph partition sizes) used are 2, 4, 8 and 16 similar to that used in chapter 4. Apart from the two graph instances mentioned above, some synthetic graphs of known sizes and connectivities are also used to find the time-complexity of the SA algorithm.

Fig. 5.3 shows the final cost as a function of the swap-move composition. The total number of transitions attempted for different processor sizes are governed by eq. 5.35. This number is the same irrespective of the nature (swap or move) of the attempted transition. No significant dependence of the final cost value on the swap-move composition is observed. However, the final cost values at the two extreme compositions (0% swap & 100% swap) are found, in most cases worse than at other values of the composition. When 100% swap is used the load-imbalance issue is not properly addressed and as a result the final solution contains significant load-imbalance. This result here substantiates the observation made in Sec.5.4.2. Wide statistical variation is also observed in all cases. As swaps take more time than the

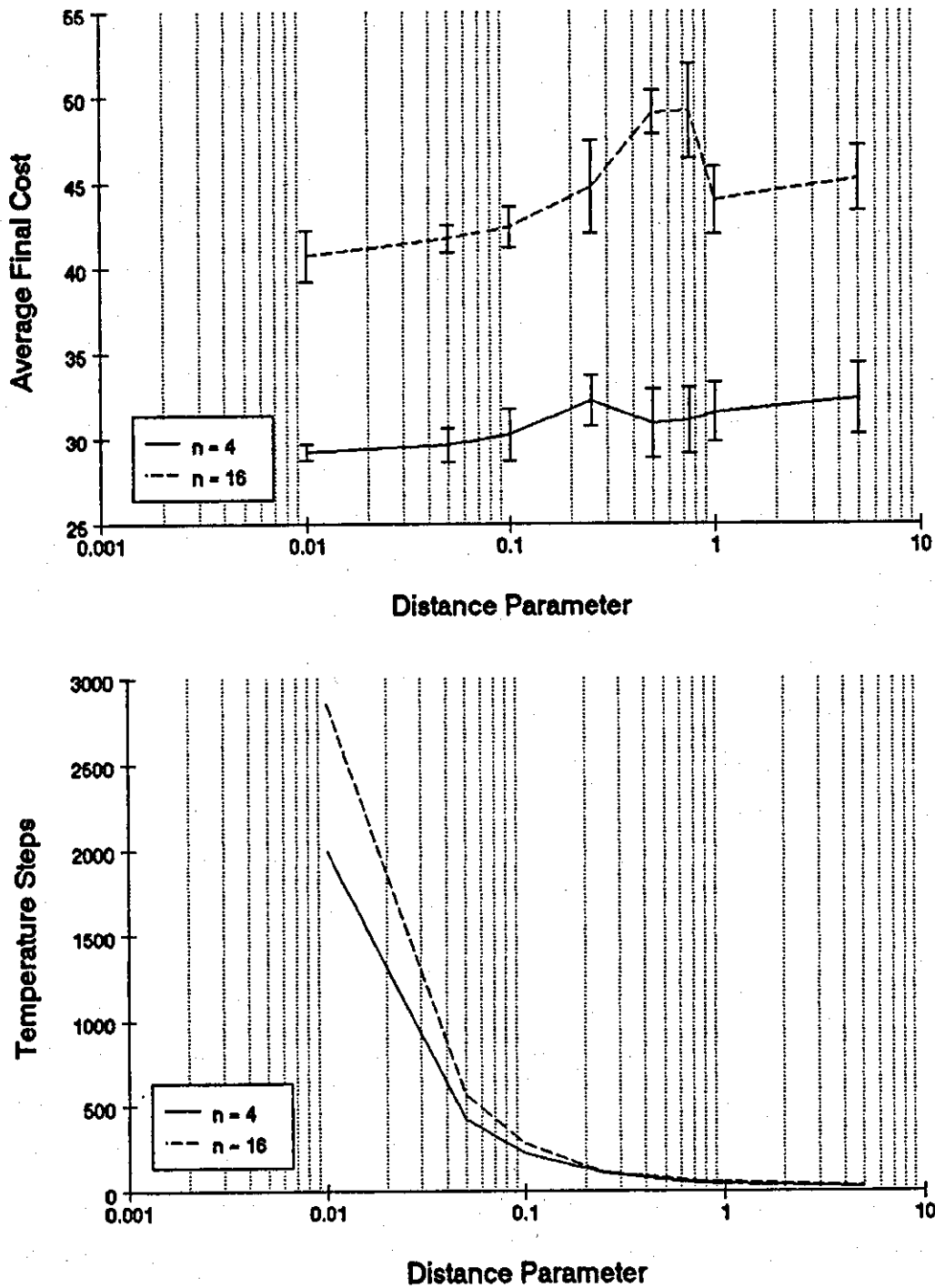


Fig. 5.10 Average final cost (top) & the number of temperature steps (bottom) as functions of the distance parameter, δ for fixed values of $\zeta_0 (= 0.98)$ and $\epsilon_s (= 10^{-5})$. The polynomial-time cooling schedule & the graph instance for the 4x4 multiplier ckt. are used.

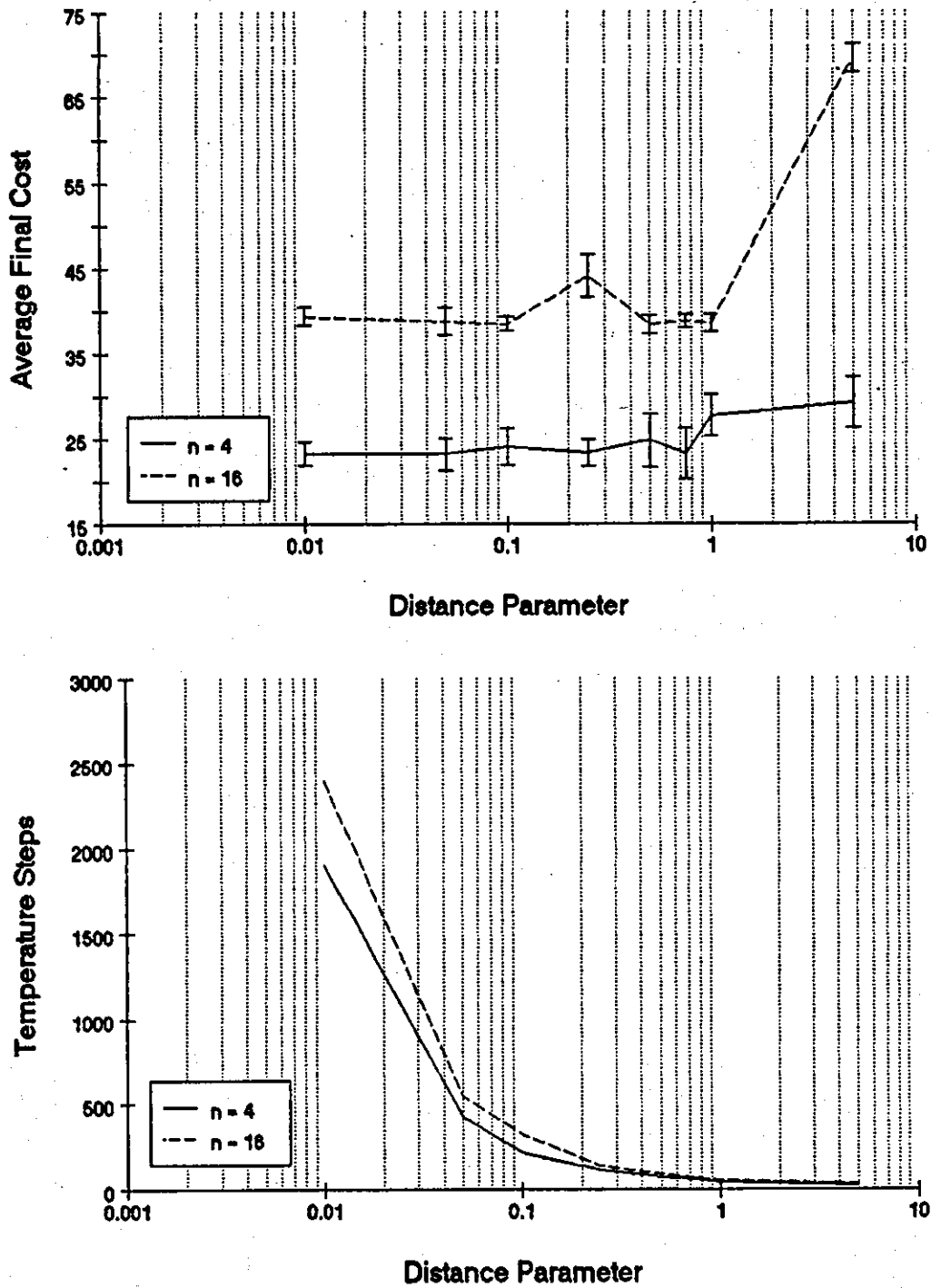


Fig. 5.11 Average final cost (top) & the number of temperature steps (bottom) as functions of the distance parameter, δ for fixed values of $\zeta_0 (= 0.98)$ and $\epsilon_s (= 10^{-5})$. The polynomial-time cooling schedule & the graph instance for the frequency locked loop ckt. are used.

moves, it is advantageous to use more moves than swaps. A percentage figure of 20%–30% of swaps in most cases produced acceptable results.

In Fig. 5.4, the average final cost and also the average running-time as a function of the initial acceptance ratio, ζ'_0 for the simple cooling schedule are shown. The observed results are quite unexpected. A deterioration of the final cost with the decrease in the initial acceptance ratio was anticipated. However, it is to be remembered that the implementation here is only a finite length approximation of the Markov theory of the SA algorithm which requires infinite length Markov chains to be generated at the descending values of temperature. Moreover, the simple cooling schedule always works with a fixed number of temperature steps (number of Markov chains) for all values of ζ'_0 . Smaller values of the initial acceptance ratio, ζ'_0 result in lower values of the initial temperature. Since, a fixed number of temperature steps, $N_T (= 100)$ and also a fixed terminating condition based on a pre-determined value of the final acceptance ratio ($\zeta'_{end} = 0.00001$) are used, a very slow cooling take place for smaller values of the initial acceptance ratio, ζ'_0 . On the other hand, higher values of the initial acceptance ratio forces a relatively much larger temperature decrement factor, α and since the number of transitions at each temperature step is not very large ($m = 1$), the resulting quasi equilibrium distribution can not come close enough to the stationary distribution (eq.5.8). The reason that the final cost obtained for smaller values of ζ'_0 is not much better than that for higher values of ζ'_0 for the same number of transitions, even though cooling is very slow in the former case is due to the fact that not many perturbations are allowed in that situation and also that the initial condition for quasi equilibrium (high initial temperature) is not met.

The running-times for all values of ζ'_0 for both the problem instances are fairly uniform. This is expected as the running-time of the algorithm is directly proportional to the product of the number of temperature steps and the number of attempted transitions at each temperature step. However, if the number of temperature steps are increased (or decreased), the decrement factor, α is changed accordingly. An improvement in the final cost is expected with an increase in the number of temperature steps. Figs. 5.5 & 5.6 illustrates the above in graph form. The change in average final cost with

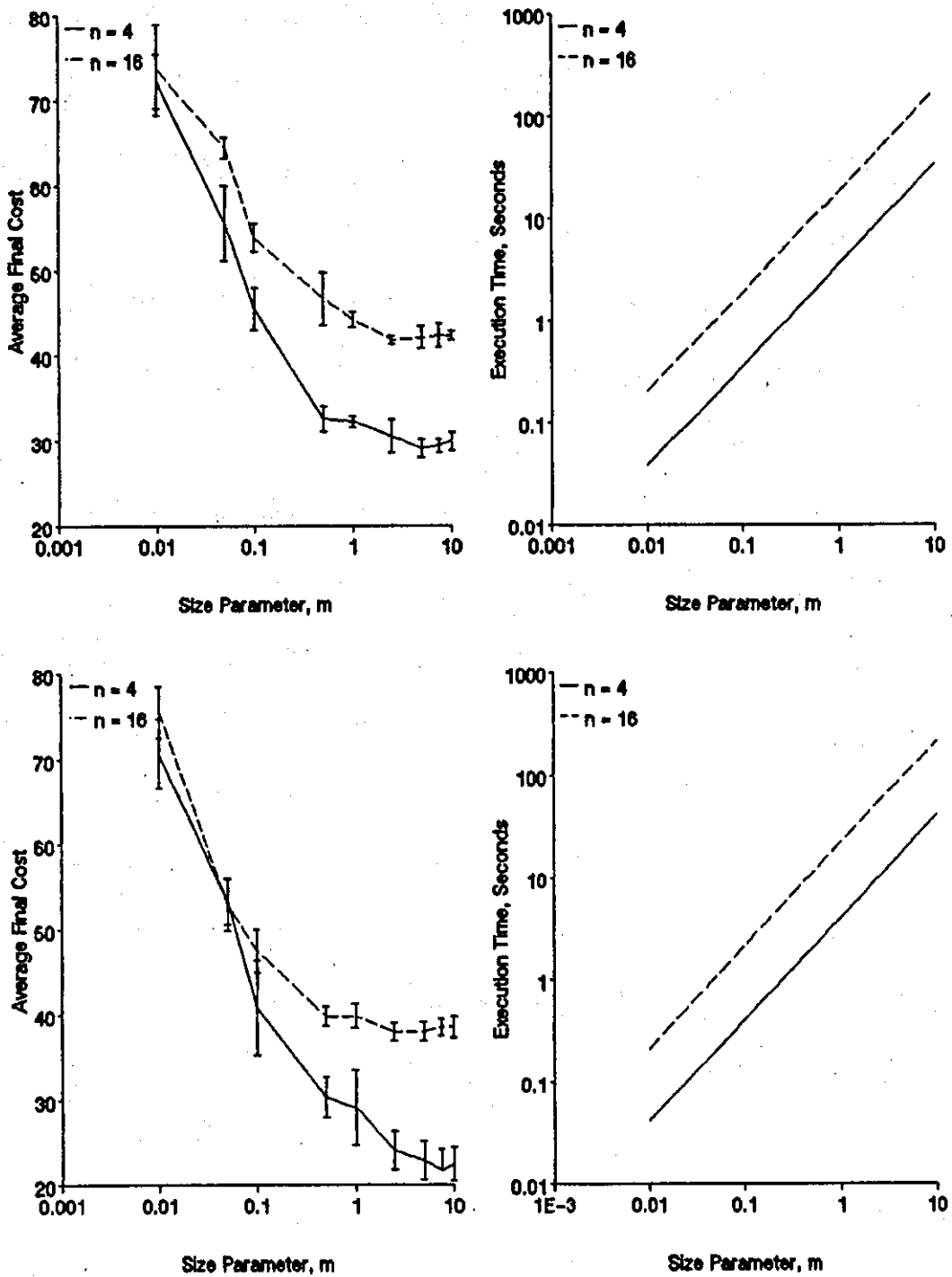


Fig. 5.12 Average final cost (left) & average execution time (right) as functions of the size parameter, m . The simple cooling schedule is in use. Graph data instances used are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

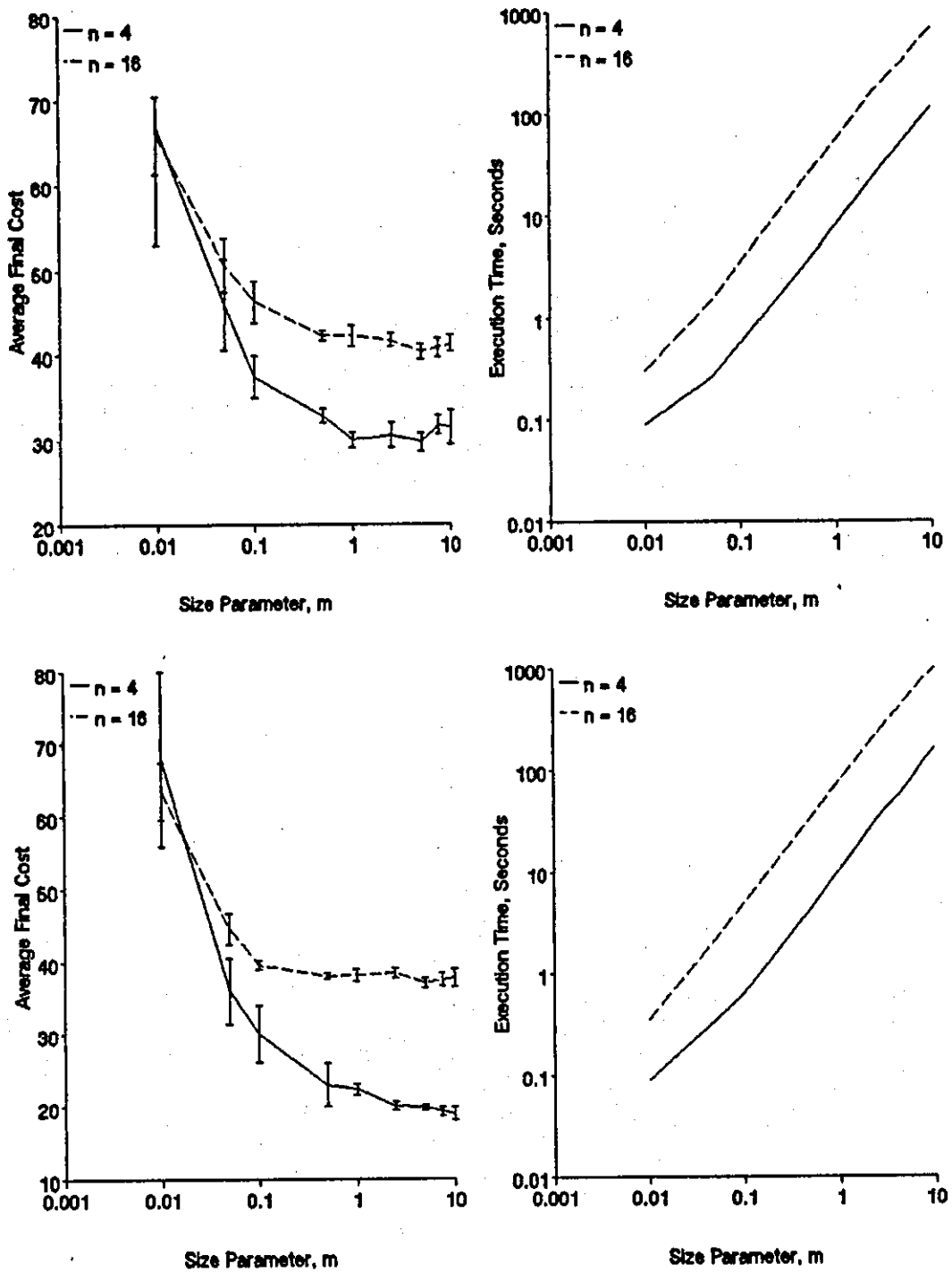


Fig. 5.13 Average final cost (left) & average execution time (right) as functions of the size parameter, m . The polynomial-time cooling schedule is in use. Graph data instances used are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

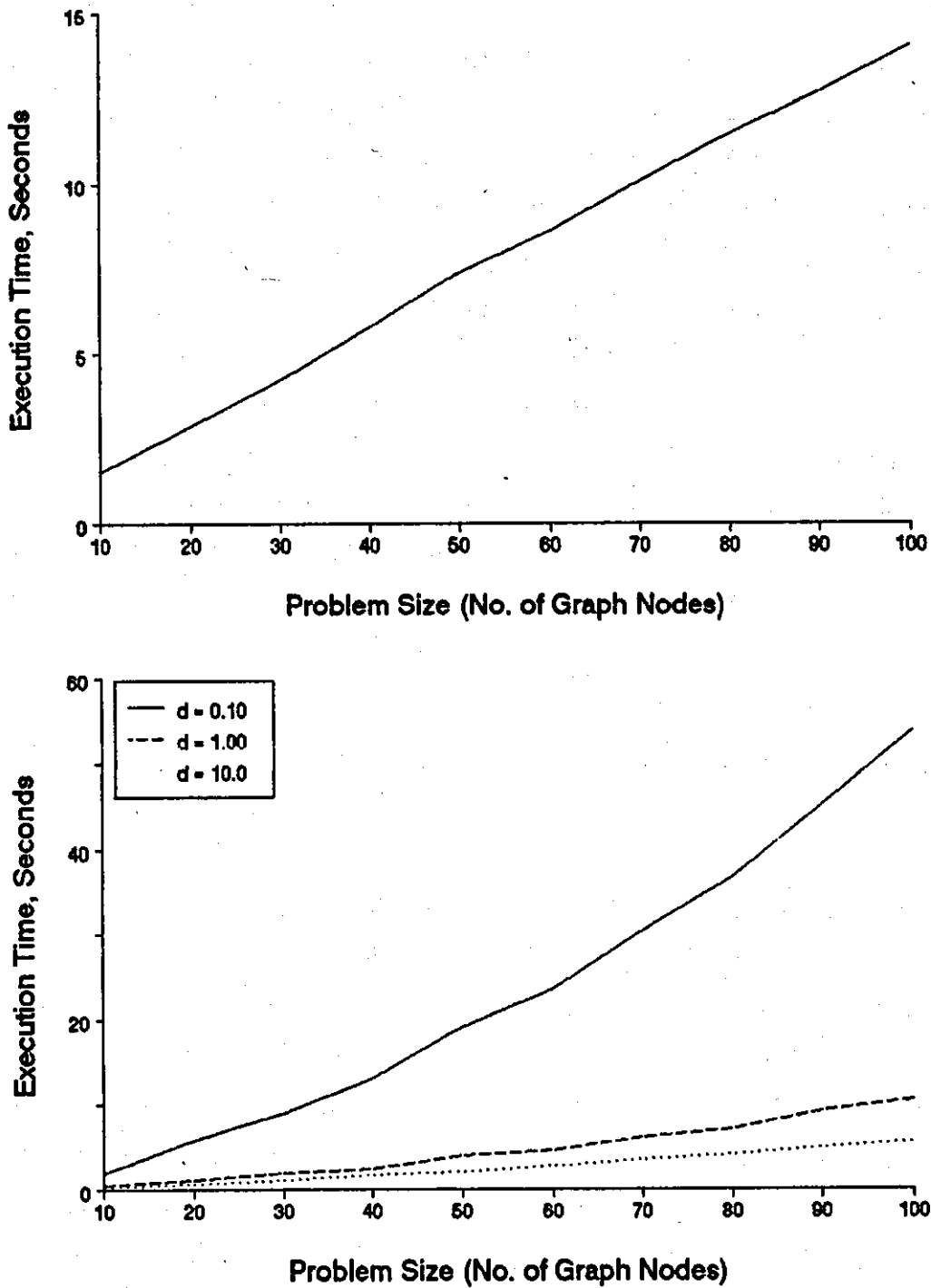


Fig. 5.14 Average execution time as a function of the problem size for a fixed number of temperature steps, $N_T (= 100)$ [simple schedule, top] and for different values of the distance parameter, d and fixed values of $\zeta_0 (= 0.98)$ & $\epsilon_s (= 10^{-5})$ [polynomial-time schedule, bottom].

the number of temperature steps however, in most cases, has not been found very significant.

Fig. 5.7 once again shows the average final cost and the average running time as a function of the initial acceptance ratio, ζ_0 . In this case, the polynomial-time cooling schedule is used and the values of the distance parameter, $\delta (= 0.1)$ and the stop parameter, $\epsilon_s (= 0.00001)$ are fixed. It is seen that smaller values of ζ_0 leads to faster execution time. This is expected since the algorithm starts off at smaller values of temperature while it terminates at approximately the same temperature. It is also observed that the average final cost deteriorates as ζ_0 decreases. This can be explained as follows. Smaller values of the initial acceptance ratio, ζ_0 result in lower values of the starting temperature. As a consequence, for these smaller values, the initial condition for quasi equilibrium, i.e., high starting temperature is no longer met. This predictably results in a deterioration of the quality of the final solution, since the stationary equilibrium distribution is no longer reached. An important observation here is that the SA algorithm (with polynomial-time cooling schedule) has failed to reach convergence for some very small values of ζ_0 .

Figs. 5.8 & 5.9 show the average final cost and also the average number of resulting temperature steps, an indicator of the running-time as a function of the stop parameter, ϵ_s , the values of $\zeta_0 (= 0.98)$ and $\delta (= 0.1)$ are fixed. The expected behaviour is clearly in evidence. As the value of ϵ_s increases, the stop criterion of eq. 5.29 is more easily met and consequently the running-time decreases, while the quality of the final solution deteriorates. From the plots however, three distinct regions can be clearly seen, viz. (a) $0.00001 \leq \epsilon_s < 0.001$, (b) $0.001 \leq \epsilon_s < 0.01$ and (c) $\epsilon_s > 0.01$. The SA algorithm appears to show its calibre as ϵ_s assumes values less than 0.01. Within region (a) very little improvement in the final solution is achieved with commensurate little increase in the running-time.

The effect of the distance parameter, δ on the average final cost and also on the running time (number of temperature steps) is shown in Figs. 5.10 & 5.11. As can be seen, smaller values of δ result in better results but at the expense of much increased execution time. A value of δ lying between 0.1 and 0.5 is more acceptable for a practical implementation.

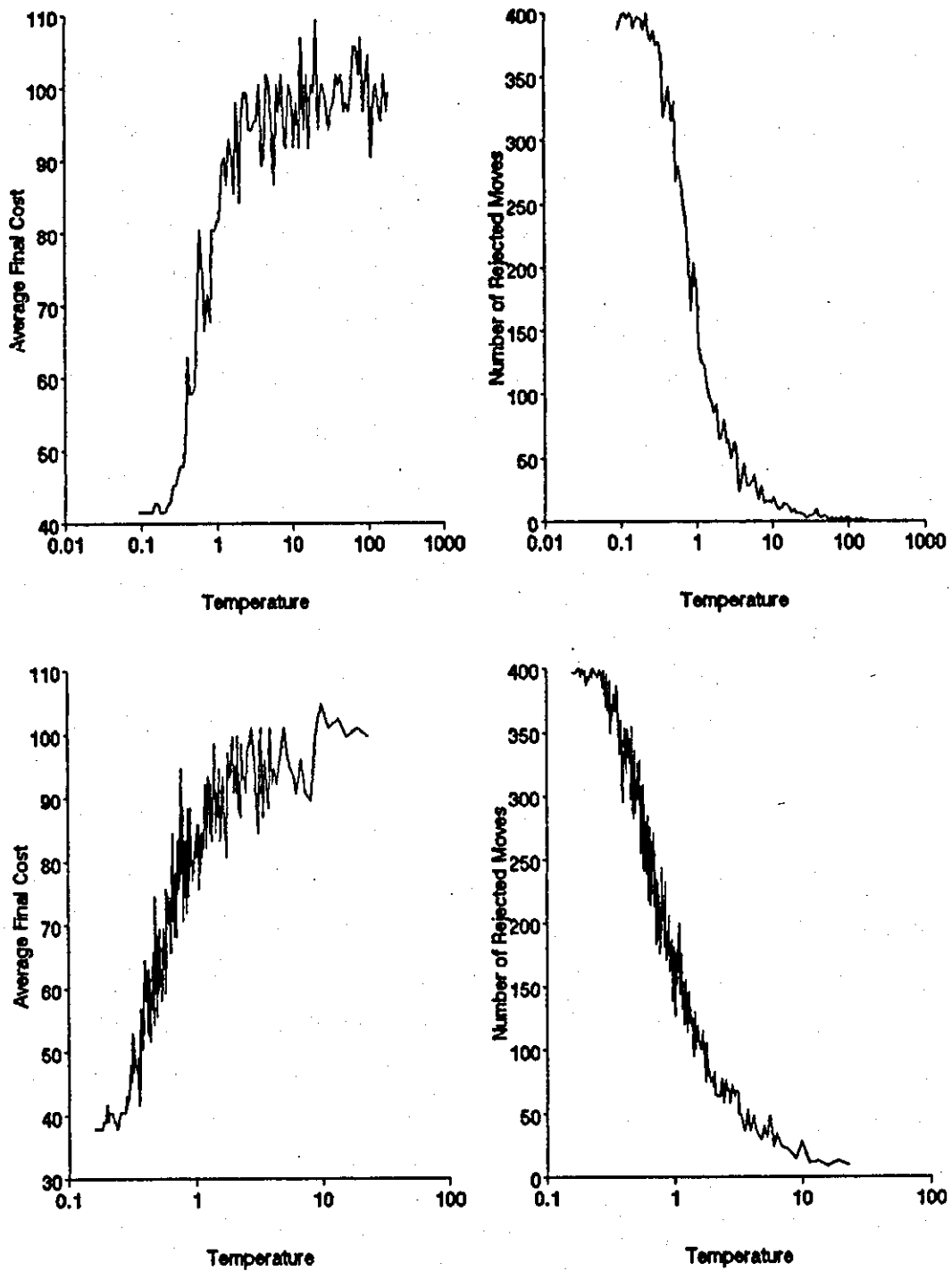


Fig. 5.15 Temperature profiles of the simple (top) and polynomial-time (bottom) cooling schedules. Graph data instance is 4x4 multiplier ckt.

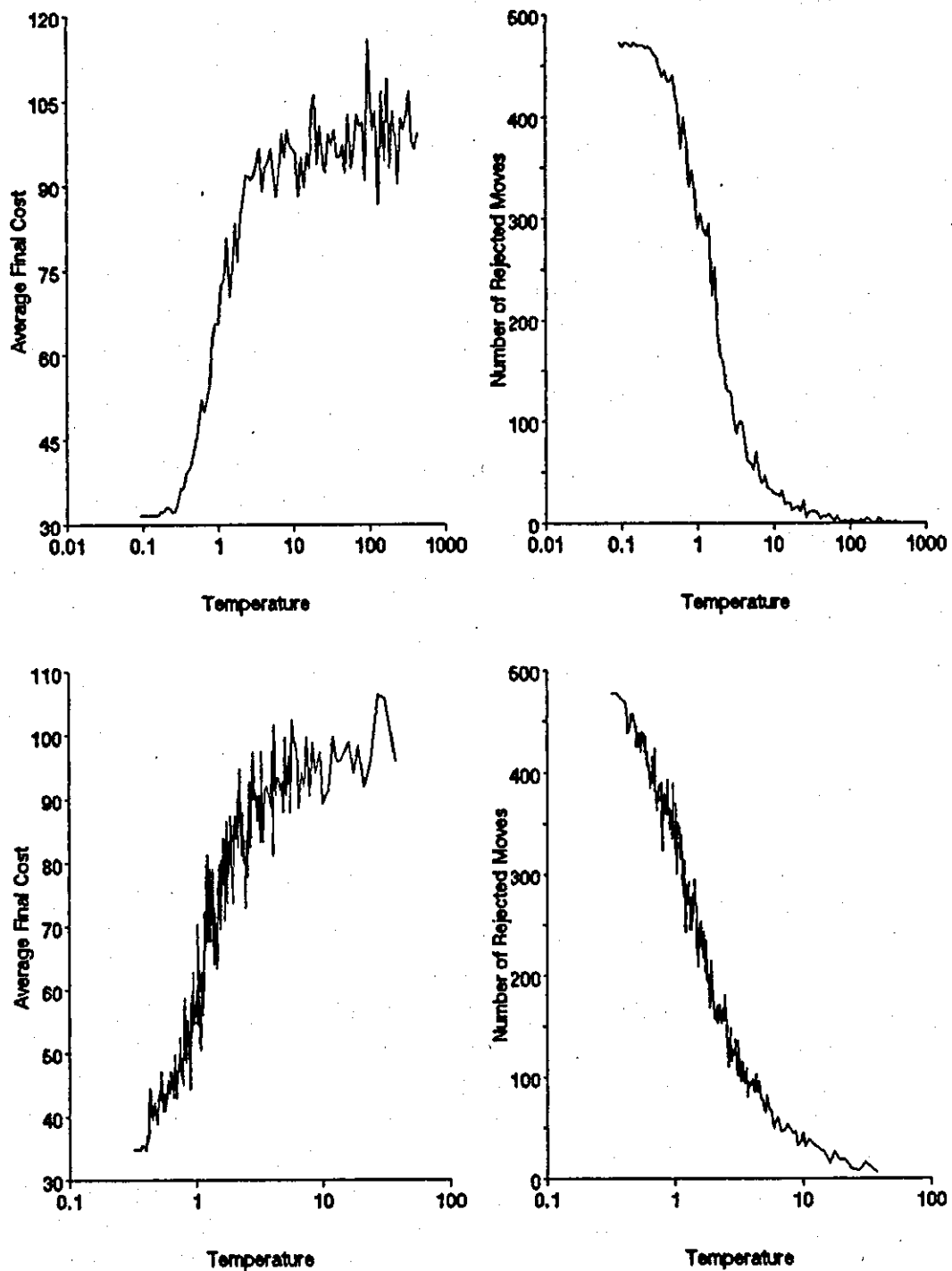


Fig. 5.16 Temperature profiles of the simple (top) and polynomial-time (bottom) cooling schedules. Graph data instance is frequency locked loop ckt.

Figs. 5.12 & 5.13 show the effect of the size parameter, m on the average final cost and the running time for both the schedules. The observed result is evidence of the Markov theory of the SA algorithm. An increased m result in a larger neighbourhood size and consequently more attempted transitions for each Markov chain (eq.5.35) bringing a closer approximation to the required infinite length Markov chain. The penalty of a better result with higher m is however the much increased execution time which grows linearly with m .

Fig. 5.14 shows the time dependence of the SA algorithm with problem size. As in chapter 4, the problem size is defined as the number of nodes of the graph representing a task system. For this investigation, the same set of synthetic graphs as used earlier in chapter 4 with nodes ranging from 10 to 100 with average degree ranging from 2.8 to 3.6 are used. The nodes and edges of these synthetic graphs are unity weighted. A processor size of 8 ($n=8$, number of partitions in the graph) is considered. The simple cooling schedule resulted in a linear time ($O(k)$) the problem size relationship with the problem size, a direct consequence of the very simplistic convergence condition.

The polynomial-time cooling schedule however, behaves slightly differently. It is observed that the average-case time complexity is the same for the different δ -values and is estimated to be slightly worse than $O(k \log k)$. Finally, it should be noted that the running times may be very large at small values of δ for the larger problem instances.

In Figs. 5.15–5.18 the performance of the two cooling schedules are compared. The default parameter values (Sec.5.4.3) were used. The profile of a single complete cooling schedule for both the graph instances are depicted in Figs. 5.15 & 5.16, where the number of transitions rejected at each temperature steps and also the cost value at that temperature are shown. It is observed that the simple cooling schedule starts off from a much higher temperature than the polynomial-time schedule, a result of differing formulations (eqs.5.32 & 5.19). The consequence of this is that the polynomial-time schedule cools very slowly, approximating the ideal physical annealing process more closely. It is also observed that there is more activity in the polynomial-time schedule which helps to explore the optimal solution more rigorously.

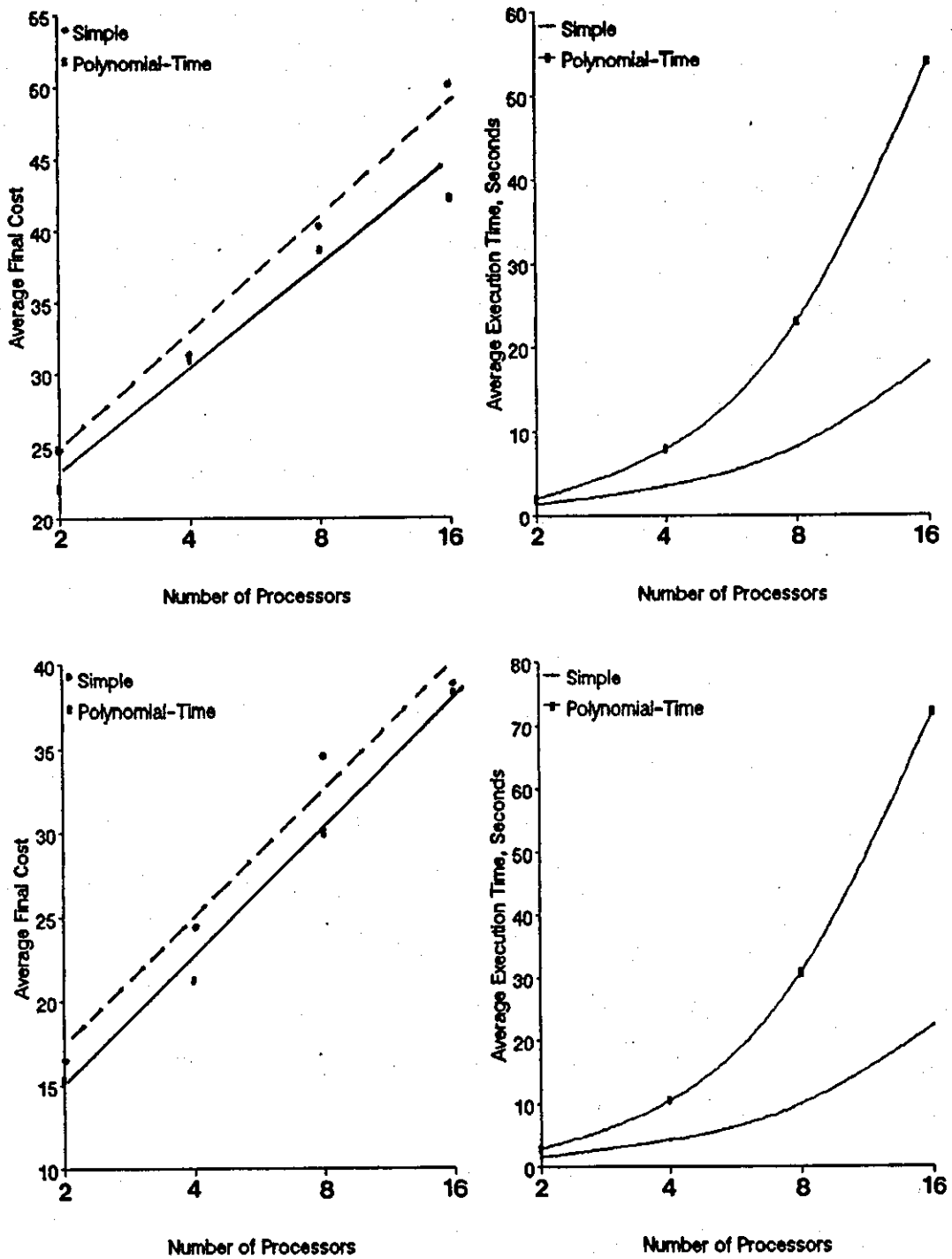


Fig. 5.17 Performance comparison of the two cooling schedules, viz. simple & polynomial-time. Graph data instances are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

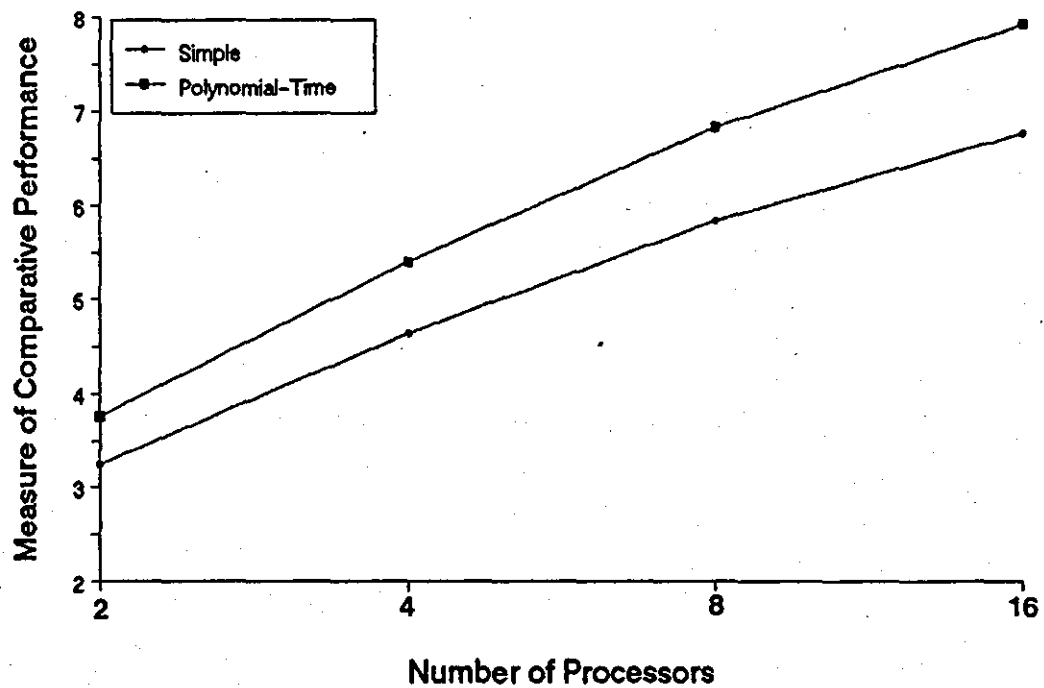
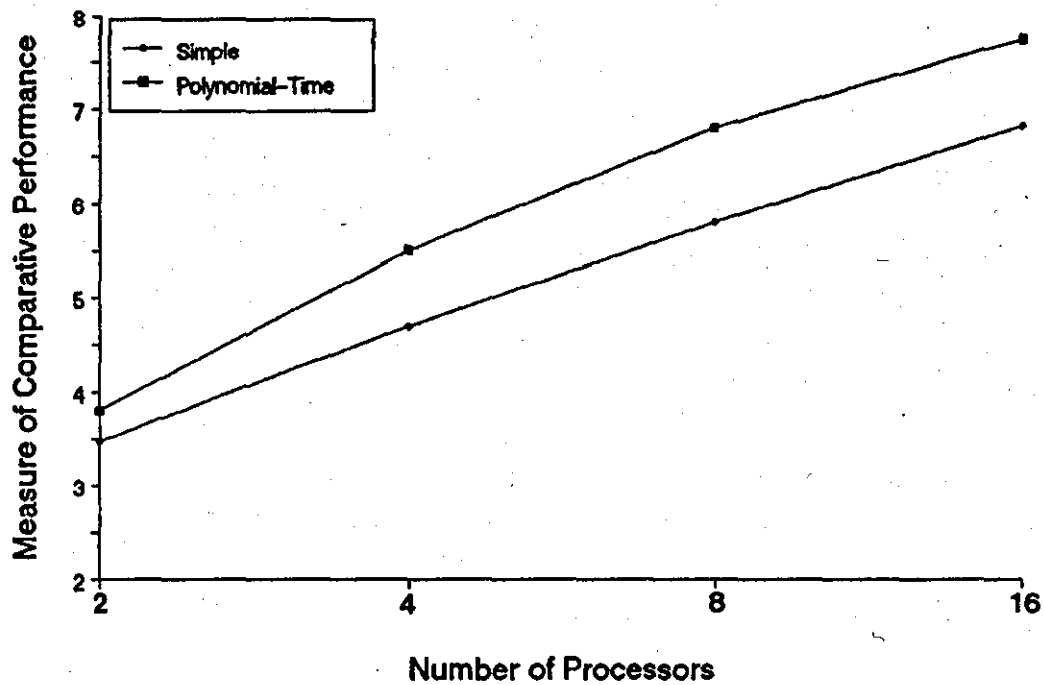


Fig. 5.18 Performance comparison of the simple & polynomial-time two cooling schedules using the measure of comparative performance. Graph data instances are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

Fig. 5.17 shows the performance comparison of the two cooling schedules for different processor (graph partition) sizes. The solutions created by the polynomial-time schedule always outperformed those created by the simple schedule, though not by a wide margin. However, its time requirement is also significantly higher compared to that of the simple schedule. In order to be able to make a subjective comparison of the two, we introduce a *measure of comparative performance*, η figure. This is defined as follows,

$$\eta = \ln(\overline{f_{end}} \overline{T_e}), \quad 5.36$$

where $\overline{f_{end}}$ is the average final cost value and $\overline{T_e}$ is the average execution time. This definition, though, clearly favours the polynomial-time cooling schedule (Fig. 5.18) nonetheless expresses the emphasis given on the ability of a schedule to produce better final solution even at the cost of increased running-time. It is seen in Fig. 5.18 that the simple cooling schedule based on empirical knowledge performed better throughout. The performance of the polynomial-time schedule is close to that of the simple schedule for small processor sizes (n) but starts to worsen as the processor size is increased.

In conclusion, it can be said that the polynomial-time cooling schedule is more robust and amenable to different problem instances than its counterpart because of its good statistical foundation. It is also found that for the polynomial-time schedule, the performance of the SA algorithm is more sensitive to the value of the stop parameter, ϵ_s , a judicious value of the distance parameter, δ would reduce the running time of the algorithm without sacrificing much in the solution quality and that starting the algorithm from a higher temperature would yield a better solution.

In its support it can be said that the simple cooling schedule is very easy to implement, runs relatively faster and generally performs very well.

However, in many practical situations the running-time demanded by the SA algorithm is considered far too long and this poses the major hindrance of the general applicability of the SA algorithm. The succeeding chapter discusses this issue and proposes a concurrent multiprocessor implementation of the SA algorithm.

References :

1. Aarts, E. and Korst, J., *Simulated Annealing and Boltzmann Machines*, John Wiley & Sons, 1989.
2. Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A and Teller. A., *Equation of state calculation by fast computing machines*, J.Chem. Physics, Vol. 275, 1953, pp. 1087-1092.
3. Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P., *Optimisation by Simulated Annealing*, Science, Vol. 220, 1983, pp. 671-680.
4. Aarts, E.H.L. and Van Laarhoven, P.J.M., *Statistical cooling: a general approach to combinatorial optimization problems*, Philips J. of Research, Vol. 40, 1985, pp. 193-226.
5. Anily, S. and Federgruen, A., *Simulated annealing methods with general acceptance probabilities*, J. Applied Probability, Vol. 24, 1987, pp. 657-667.
6. Romeo, F. and Sangiovanni-Vincentelli, A.L., *Probabilistic hill climbing algorithms: properties and applications*, Proc. Chapel Hill Conf. on VLSI, USA, 1985, pp. 393-417.
7. Geman, S. and Geman, D., *Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images*, IEEE Trans. Pattern Analysis and Machine Intelligence, Vol. 6, 1984, pp. 721-741.
8. Hajek, B., *Cooling schedules for optimal annealing*, Mathematics of Operations Research, Vol. 13, 1988, pp. 311-329.
9. Mitra, D., Romeo, F. and Sangiovanni-Vincentelli, A.L., *Convergence and finite-time behaviour of simulated annealing*, Advances in Applied Probability, Vol. 18, 1986, pp. 747-771.
10. Collins, N.E., Eglese, R.W. and Golden, B.L., *Simulated annealing — an annotated bibliography*, Cambridge Univ. Press, UK, 1987.
11. Aarts, E.H.L. and Van Laarhoven, P.J.M., *A new polynomial- time cooling schedule*, Proc. IEEE Intl. Conf. CAD, Santa Clara, USA, 1985.

CHAPTER 6

Concurrent Simulated Annealing

The advantage of the simulated annealing (SA) algorithm as a general tool for the solution of combinatorial optimisation problems are its potential to find near-optimal solutions, its general applicability, its flexibility and its ease of implementation. However, the major disadvantage it carries is the potential length of time required to converge to a near-optimal solution.

The amount of computational requirement of the SA algorithm strongly depends on the nature and size of the optimisation problem. It ranges from a few seconds, e.g. for small instances of the travelling salesman problem, up to a few days, e.g. for large instances of the VLSI cell placement problem [1]. Generally, the situation with respect to the computational requirement worsens as problems increase in size. An effort to speed-up the algorithm, in order to keep the computation times within acceptable limits is a reasonable thought. The increasing availability of multiprocessor

systems offers a suitable platform to explore the possibility of parallelisation of the SA algorithm.

6.1 Speeding-up the SA Algorithm

There exist many different means of speed-up of the SA algorithm. Design of a fast sequential SA algorithm results in a more efficient implementation of the algorithm. Also, use of special hardware accelerators produces good performance in many demanding situations. Last but not the least, parallelisation is an exciting possibility especially in the realm of present days's technology. These three major speed-up approaches are described below.

6.1.1 Fast Sequential Algorithms

The generation mechanism and/or the cooling schedule of the SA algorithm may be improved for an efficient implementation of the algorithm without deterioration of the quality of the final solution.

The state generation mechanism in Szu & Hartley's [2] Fast Simulated Annealing (FSA) uses Cauchy distribution instead of the usual Gaussian distribution resulting in an inverse linear cooling rate thereby requiring less time. The FSA algorithm was found very efficient for the solution of continuous valued functions. The usefulness of Szu & Hartley's approach for the solution of combinatorial optimisation problems is however debatable [4]. The main difficulty is with the different formalism and with the lack of adequate statistical knowledge of the problem instances required to generate moves that correctly follow the Cauchy distribution.

In the *rejectionless method* by Greene and Supowit [3], new solutions are generated with probability proportional to the effect of a transition on the cost function. This results in the subsequent solution being chosen directly from the neighbourhood of the current solution and thus the rejections are eliminated. This method leads to shorter Markov chains in a number of problems. However, the efficient use of this method requires some additional conditions to be met by the neighbourhood structure, which unfortunately can not be met by many combinatorial optimisation problems.

It is generally held that cooling schedules alone can not improve the efficiency of the SA algorithm [4]. However, it is also expected that certain schedules which are tailored to a given problem or a set of problems have the possibility of improving the efficiency of the SA algorithm. Cathoor, DeMan and Vanderwalle [5] proposed an efficient cooling schedule in which clustering occurs.

Derivatives of the SA algorithm have also been proposed. Hoptroff and Hall [6] have proposed a method of optimisation for learning in multilayer perceptron which is essentially the simulation of annealed diffusion process. A better than an order of magnitude of improvement in run-time and also a superior quality of solution is reported. Bart and Miller's [7] *Mean Field Algorithm*, which combines the characteristics of the SA algorithm and the Hopfield neural network has been used for the graph partitioning problems and is found to be as much as 50 times faster than the SA algorithm.

6.1.2 Hardware Accelerators

Dedicated hardware accelerators can be used to evaluate the time-consuming parts of SA algorithm. Iosupovici, King and Breuer's [8] point accelerator was used to evaluate the incremental wire lengths in a placement problem. Using a different approach, Spira and Hage [9] rewrote the time-consuming parts of the algorithm in micro code to be executed on a fast general purpose micro engine attached to a workstation host. Speed-up factor upto 20 was reported for the placement problem.

6.1.3 Design of Parallel Algorithms

The parallelisation of the SA algorithm involves the distribution of the execution of various parts of algorithm over a number of communicating parallel processors. This approach promises significant speed-up of the SA algorithm, but is by no means a trivial task. This is mainly due to the intrinsic sequential nature of the algorithm, where in order to hold the Markov property and the convergence criterion transitions are to be carefully carried out one after another.

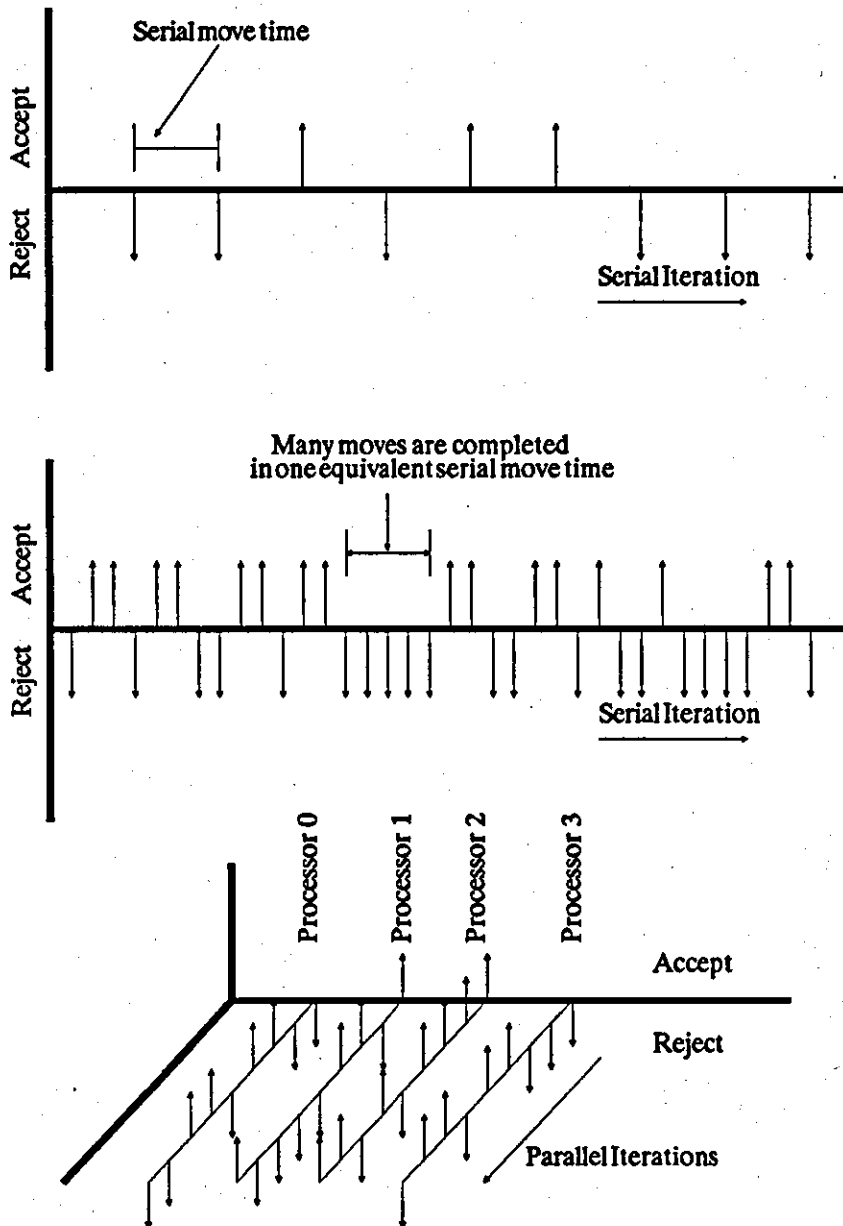


Fig. 6.1 Various approaches of *moves* for iterative improvement. Serial move (top), move decomposition (middle) and parallel moves (bottom). Adapted from [10].

Recently there is seen a growing interest in the research for the design and analysis of a parallel SA algorithm. In the next section we shall discuss the basic aspects related to the design and analysis of parallel SA algorithms, and also a brief review of various approaches will be presented.

6.2 Parallel Annealing Algorithms

The SA algorithm can be characterised by a sequence of accept/reject decisions on attempted trials that constitute a Markov chain (Fig. 6.1a). It can be easily seen that the algorithm spends most of the time in generating this sequence of trials. The length of the sequence and the computing time to propose, evaluate and accept/reject each trial determines the total solution time. An efficient parallel SA algorithm should concentrate on this part of the algorithm for the exploitation of maximum parallelisation. Decomposing each trial, we get the following four constituent tasks :

- a. selecting a new solution from the neighbours of the current solution,
- b. calculating the difference in cost between these two solutions,
- c. deciding whether or not the new solution is to be replaced,
- d. replacing the current solution by the new solution if it is accepted.

The sequence of trials can then be written in algorithmic form as:

Repeat

1. select new solution;
2. evaluate Δc , the cost difference;
3. decide to accept or reject the new solution;
4. if accept, execute REPLACE & UPDATE operations;

Until *ThisLoopConditionSatisfied*;

The four steps of the above loop must be executed sequentially because it is not possible to evaluate the cost of a *move* (transition) without having knowledge of the cost of the move and so on. Kravitz and Rutenbar [10] attempted to break each step into smaller subtasks, thus defining upto 15 different small jobs, some of which can be carried out in parallel. In this way,

the sequential nature of the SA algorithm is retained. This decomposition of a single move is referred to as *Functional Decomposition* of a move or simply *Move Decomposition* (Fig. 6.1b). Strong inherent precedence relationships in various steps of a complete move reduces the opportunity for parallelism. As such, scheduling of the subtasks must be synchronised carefully. Very little speed-up is expected with this technique, especially for larger multiprocessor systems as fine grain parallelism is difficult to achieve with this technique. The reported speed-up is less than 2 for 3 processors and is projected to increase only slightly with the addition of more processors.

The opposite to move decomposition is the *Parallel Move* (or *Multiple Move*) approach, where several complete moves are executed simultaneously in parallel (Fig. 6.1c). Different variations in this approach are possible. One such is the *Division Algorithm* proposed in [11], where decomposition of the serial SA is done at the Markov chain level. At each temperature step, the Markov chain is divided into subchains each of which can be generated by a different processor. Thus, the algorithm achieves parallelism by having the processors work on different copies of the data. The speed-up is obtained by spending less time at each temperature, carefully selecting the initial conditions for each stage of the cooling schedule in an attempt to preserve the quasi-equilibrium. In this respect, two variations of the division algorithm strategy exist. In the first strategy, there is no communication in between the generation of consecutive Markov chains, i.e. each processor continues the generation of a subsequent subchain with the solution given by the outcome of the last trial of the preceding subchain obtained by the same processor (Fig. 6.2a). In the second strategy, the best solution found by the processors that each generate their own subchain is used as the outcome of the Markov chain constituted by the n subchains where n is the number of processors. In between subsequent Markov chains the solution is communicated to all processors and used as initial solution for generating the subchains constituting the next Markov chain (Fig. 6.2b). The first strategy returns n solutions — one for each processor — the best of which is chosen as final solution. Aarts et. al [11] used both strategies for the solution of a 100-city travelling salesman problem and found very little performance difference. They reported speed-up of about 6 using 8 processors. However,

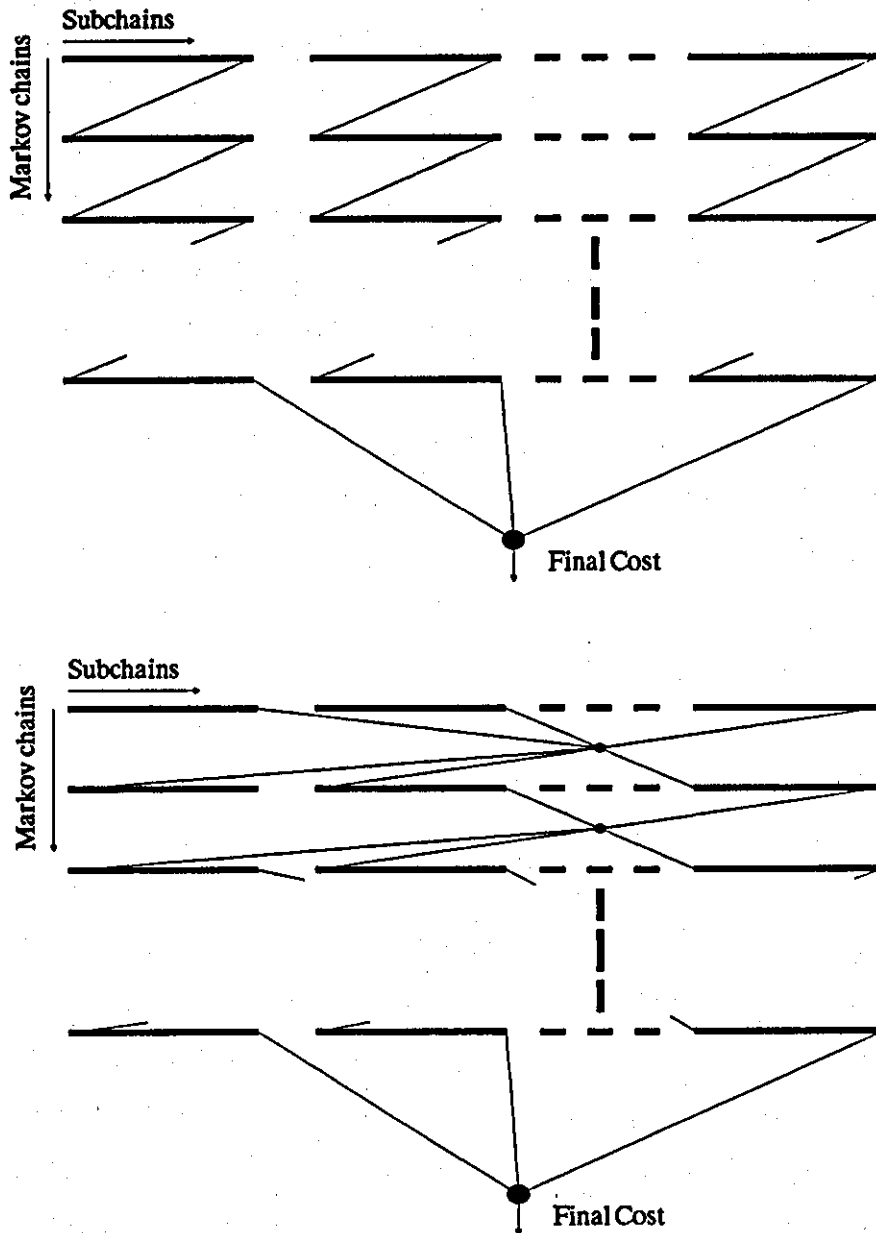


Fig. 6.2 Pictorial representation of the *division* algorithm, without communication (top) and with communication (bottom). Adapted from [4].

it was found that the efficiency of processor utilisation drops as the number of processors increases and extrapolation of results shows that no further gain can be obtained for about 30 processors. Woodhams and Price [12] adopted a similar approach (division algorithm) for their parallel SA implementation for the solution of VLSI cell placement problem. They also reported linear speed up upto 4 processors.

There is another variation of the parallel move approach to the problem of parallelising SA, whose distinctive feature is that only one copy of the data is shared among all the processors. An example in this class is reported in [10]. It uses the concept of serializable subset of moves. A subset M' of μ moves $M = \{m_1, m_2, \dots, m_\mu\}$ is serializable if all of the moves in M' do not interact with each other (c.f. Sec.6.3.1). This means that the decision of accepting or rejecting a particular move m_i does not depend on the order in which the moves in M' are executed. Only serializable moves are allowed in such move set. The authors found it difficult to determine large serializable subset of moves and as such, their algorithm restricts its attention to the *simplest serializable subsets*, i.e. subsets in which all but possibly one moves are rejected. This simplest serializable subsets of moves are obtained by attempting many moves in parallel, and by executing (i.e. if the move is accepted by replacing the current solution with the new solution) only the move that is accepted first and aborting all the rest. One drawback of this procedure is that it seems to favour those simple moves whose cost can be evaluated faster than the others. This approach shows a linear speed-up at very low temperatures, where most of attempted moves are expected to be rejected, but its performance is poor at high temperatures.

Darema, Kirkpatrick and Norton [13] proposed a different variety of multiple move parallel SA algorithm for the placement of gate arrays. They considered only pair-wise exchange of cells as valid moves. In order to minimise erroneous calculation of costs due to move interactions, care is taken to avoid situations in which the same cell is moved by more than one processor at the same time. Connected cells having common edges are also avoided. Each processor selects a pair of cells at random; if the pair (or any one cell of the pair) is found to be already taken by another processor, the attempt to move that pair is aborted and another pair is selected. When

a pair is found, flags are raised to mark the cells as locked, the processor can now attempt to interchange the cells of the pair, at the end of which the processor clears the flags and the loop continues. This procedure makes heavy use of synchronisation mechanisms such as *locks*. The probability of finding free pair of cells decreases as the number of processors is increased relative to the size of the problem instance. Consequently, the speed-up falls with larger multiprocessor system.

Another class of parallel SA algorithm adopting the parallel move approach is the *Error Algorithm*. This is so called as no explicit attempt is made to minimise the errors due to interaction between parallel moves. At high temperatures where most of the attempted moves are accepted, there is a very high probability of such errors to enter into the solution. However, as the system is gradually cooled to low temperatures such possibility is virtually eliminated and a near-optimal convergence is expected. In the error algorithm parallelism is achieved by letting all available processors cooperatively generate the same Markov chain. No division into subchains and no communication is used. Therefore, the algorithm is well suited for execution on an asynchronous MIMD machine.

Casotto et al's [14] implementation of parallel SA algorithm for the macro-cell placement problem is based on the error algorithm. The dataset is decomposed into as many nearly equal sized subsets as the number of available processors. Each processor is allocated one such subset and then allowed to execute SA asynchronously thereby allowing parallel moves that might result in erroneously calculated cost functions. In order to obtain a wider solution space the cells of the subsets are periodically allowed to migrate between the subsets. This migration is governed by another SA process satisfying a different optimisation criterion. It is found that the error reduces to almost zero near the freezing temperature for the processor configurations tested (2, 4 and 8). An 80% processor utilisation with 8 processors is reported. Vai's [15] parallel SA implementation is almost similar with the difference that it requires a serial SA preprocessor to create the subsets which are to be assigned to the available processors in such a way that the neighbour cells belong as much as possible to the same

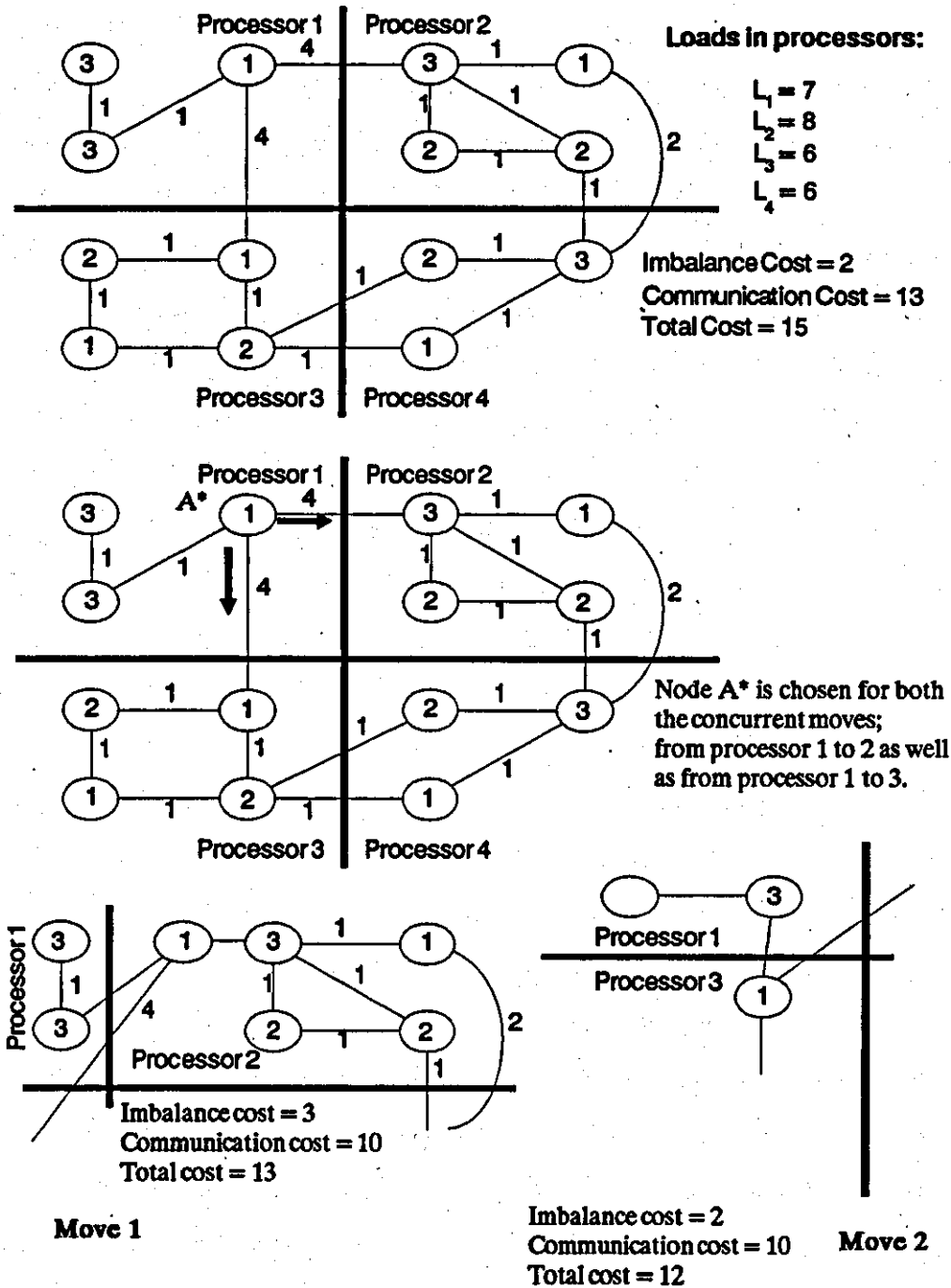


Fig. 6.3 A 14-node, 4-processor task system (top). Node A^* is involved in both the concurrent moves (middle) which when calculated independently (bottom) result in accepting decisions. This brings in contradiction, as the same node can not be moved simultaneously to two different places.

subset. No migration of cells between subsets are allowed in this particular implementation.

The advantage of the error algorithm is the absence of communication requirements, enabling simple and straightforward implementation on asynchronous MIMD machine. However, as a result of the presence of erroneously calculation of moves it no longer follows the SA serial algorithm strictly and consequently, the asymptotic convergence properties of serial SA presented in Chapter 5 can no longer be assumed to hold.

6.3 Concurrent Simulated Annealing

The *Concurrent Simulated Annealing* (CSA) [16] follows the parallel move approach. CSA uses move sets which are conceptually similar to those presented in [10]. However, the main difference is that, where in [10] the serializable subsets of moves are discarded because of difficulty in determining them and a very simple approximation is used, in CSA an explicit attempt is made to create non-interacting sets of parallel moves. The main thrust in the design of CSA lies in the determination of such non-interacting move sets. In the subsections to follow, some salient features of CSA are presented.

6.3.1 Parallel Moves and Move Interactions

The simplest way to execute (evaluate cost difference, accept/reject decision, possible update) several moves in parallel is to generate several random independent moves and then to execute them in parallel. But, the major problem is that such concurrently executed moves often interact. The interaction of concurrently executed moves limits the effective parallelism of the SA algorithm.

When moves are executed in parallel it is important to control how, moves that have been accepted by the normal annealing criterion are accepted and applied to the problem database. At the lowest level, moves which are attempted in parallel should not be contradictory among themselves (e.g. when concurrent moves involving the same node are accepted in more than one processors). Furthermore, erroneous accept/reject decisions are possible when moves are executed in parallel. During the execution of each

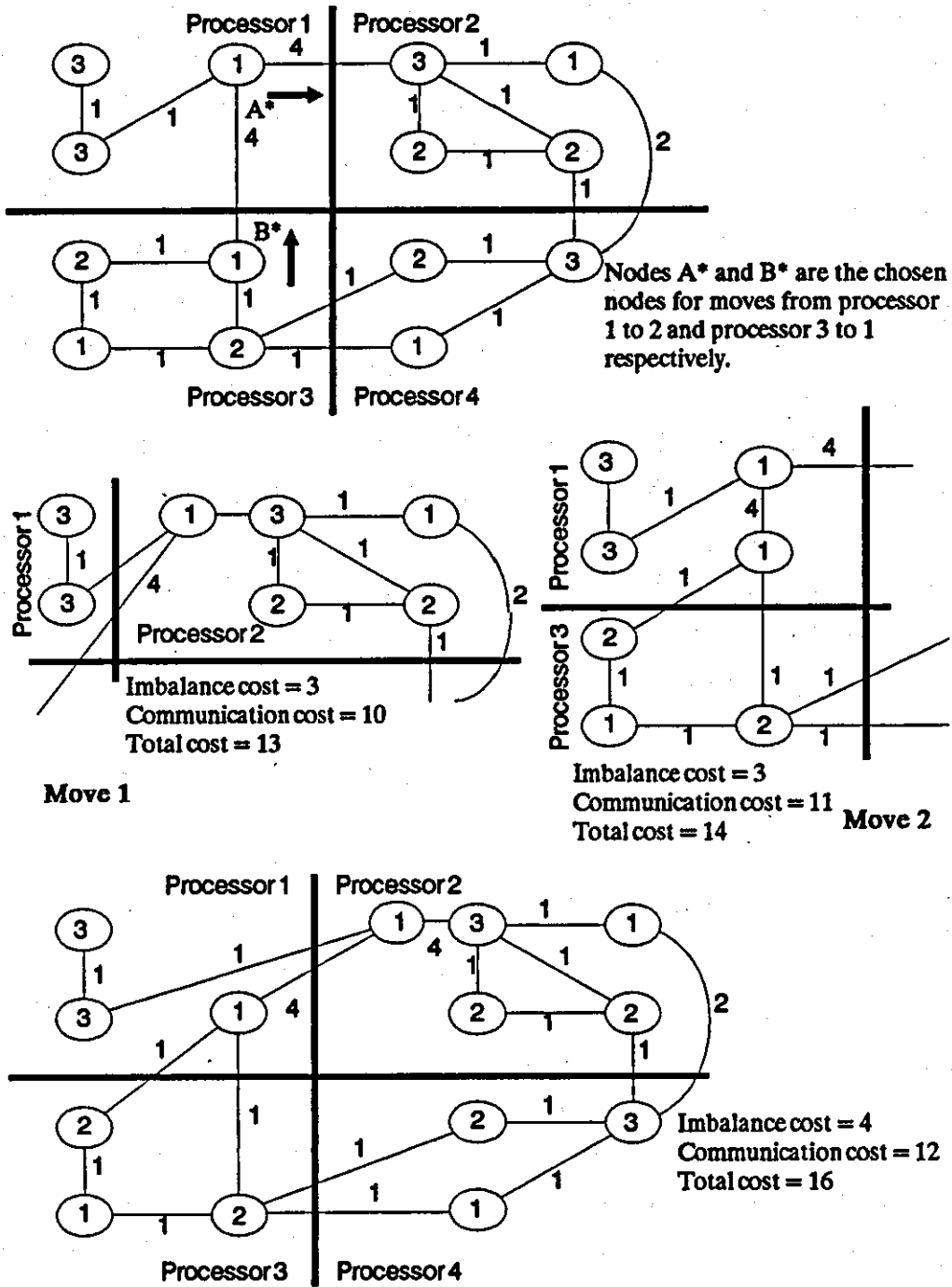


Fig. 6.4 Nodes A* & B* are selected for the concurrent moves (top). When calculated independently (middle) both results in accepting decisions. However, the final result after accepting both moves turns out worse than the original.

move, the processor executing the move works within its own domain and therefore, can not foresee or predict the effect of its own decision on other activities being performed concurrently. As a result, a correct local decision of accepting a move might prove wrong in the global scene. These erroneous decisions may result in oscillations in the system and also most importantly disrupt the proven convergence of the SA algorithm. The erroneous decisions which result from concurrent execution of moves in turn result in uncontrolled hill-climbing.

Two examples in which concurrently executable moves interact are shown in Figs. 6.3 and 6.4. In both cases, both the shown moves when carried out independently show cost improvement but their net effect is that of cost increase. The example in Fig. 6.3 shows moves which are contradictory and highlight the dilemma of handling these effectively.

6.3.2 CSA Parallel Move Algorithm

CSA works with non-interacting serializable move sets. Moves from each such move set can be executed in parallel without the risk of them interacting and thereby disrupting the convergence to a near-optimal final solution. As mentioned earlier, the main thrust in the design of CSA is in the determination of such move sets.

It is difficult to generalise exactly how and under what conditions independent moves executed in parallel would interact. As mentioned in the previous subsection contradictory moves (those involving the same node) need to be avoided. Furthermore, by limiting the interdependency between moves in a move set one would hope that the resulting move set will improve. In order to achieve that, we slightly extend the scope of the definition of move interaction and postulate that *move interaction would occur if the nodes of the graph within the different moves of the move set are connected through common edges and also if such moves cause concurrent read/write access conflict*. One advantage of this extended definition is that a non-interacting move set will not require use of any operating system assisted mutual exclusion primitives (which usually results in costly overheads). This however, does not address the overheads imposed by the hardware

```
Procedure Algorithm_A;  
    MaxMove := NoOfProcessors Div 2;  
    Select any random node for the move;  
    Propose a new but different allocation for the node;  
    For i := 2 To MaxMove Do  
        Repeat  
            Pick a random node for move;  
            Has this node already been selected?;  
            Is it a direct neighbour of any node already selected?;  
        Until Both of the above are non-affirmative;  
        Propose a new but diferent allocation for the node;  
    EndFor;  
EndProcedure;
```

Fig. 6.5 General structure of Algorithm A.

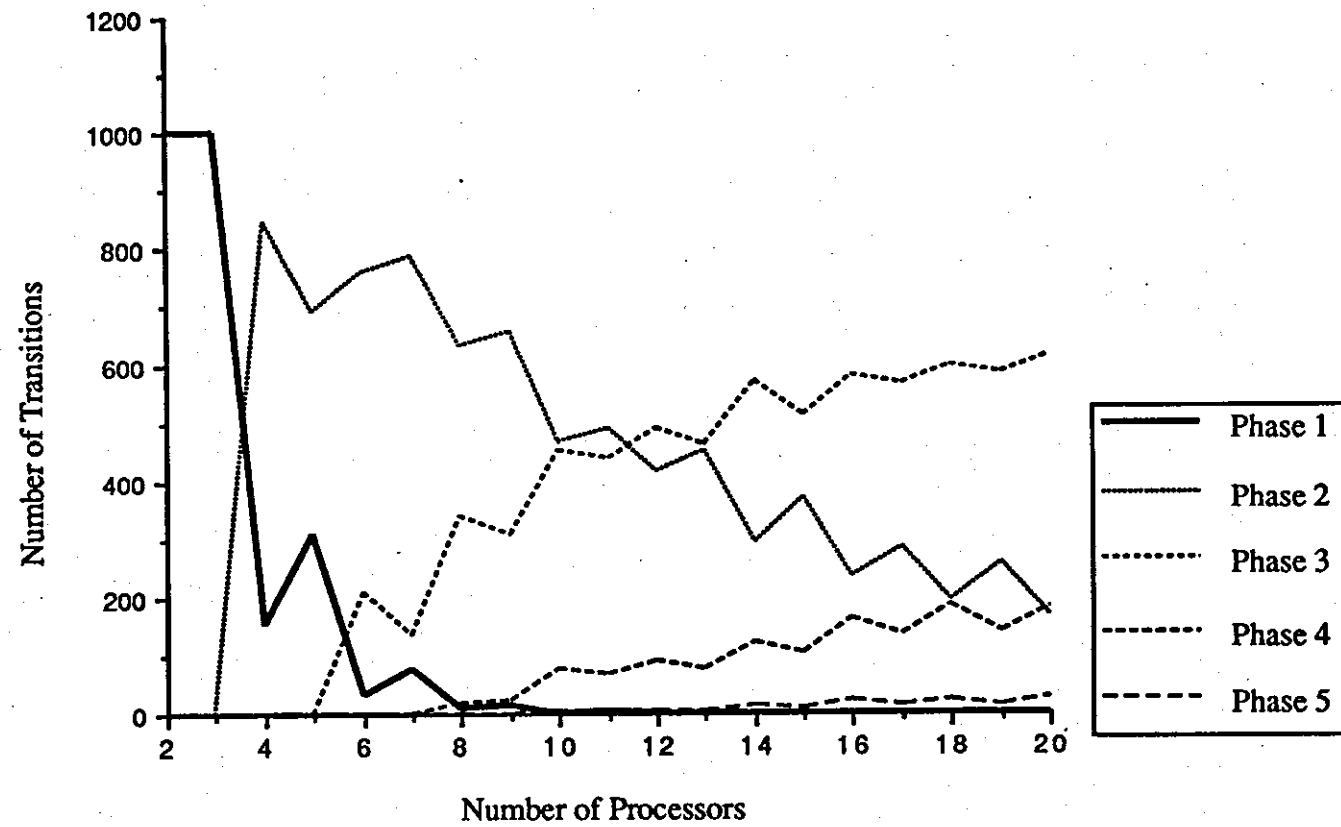


Fig. 6.6 Efficacy of Algorithm A.

architecture of the machine (e.g. overheads of accessing the same page/block of shared memory).

In CSA it is assumed that the execution of the parallel moves (evaluating cost change, accept/reject decision, possible update) are if not totally synchronous, synchronous in a way that not any single move in any set of parallel moves starts before all the moves in the previous set are completed. It is also noted that, when a move is executed, loads in only the two processors involved are affected. Since, it has been already pointed out that each move is executed independently, then in order to minimise interaction it can be safely proposed that $\lfloor n/2 \rfloor$, where n is number of available processors is the maximum number of moves in a move set. Furthermore, it is also noted that selecting the same node in more than one move will result in contradiction and as such needs to be avoided. Additionally, it is also observed that selecting a node which is neighbour (sharing a common edge) to any other already selected node will cause interaction.

The starting points for the necessary control procedure for the determination of non-interacting move sets are thus :

- a. size of the move set is $\lfloor n/2 \rfloor$, where n is the number of available processors;
- b. Neighbour nodes can not be chosen.

In the following, three move generation algorithms are examined and their comparative results are presented. The performance of these three algorithms A, B and C are studied using a specially written simulation software. The results to be presented here are for demonstration purpose only and shows the relative efficacies of the algorithms A, B and C. Different processor sizes ranging from 2 to 20 processors are considered. For each processor size, 1000 move sets are generated using the three algorithms separately. Moves in these move sets are then classified according to how they interact with other moves. Those moves, which do not interact with any other move are placed in group 1. Group 2 consists of moves which interact with only one other move, whereas moves which interact with two other moves are placed in group 3 and so on. The sizes of these resulting groups can thus

```
Procedure Algorithm_B;  
    MaxMove := NoOfProcessors Div 2;  
    Select any random node for the move;  
    For i := 2 To MaxMove Do  
        Repeat  
            Pick a random node for move;  
            Has this node already been selected?;  
            Is this a direct neighbour of any node already selected?;  
        Until Both of the above are non-affirmative;  
    EndFor;  
    Repeat  
        Select a random new allocation for the 1st node;  
    Until none of the current allocations of the moves in the move set  
        is repeated;  
    For i := 2 To MaxMove Do  
        Repeat  
            Propose a new random allocation for the ith selected node;  
            Is any current allocation of the nodes in the chosen move  
                set repeated?;  
            Is this a repeat of any newly chosen allocations?;  
        Until Both of the above are non-affirmative;  
    EndFor;  
EndProcedure;
```

Fig. 6.7 General structure of Algorithm B.

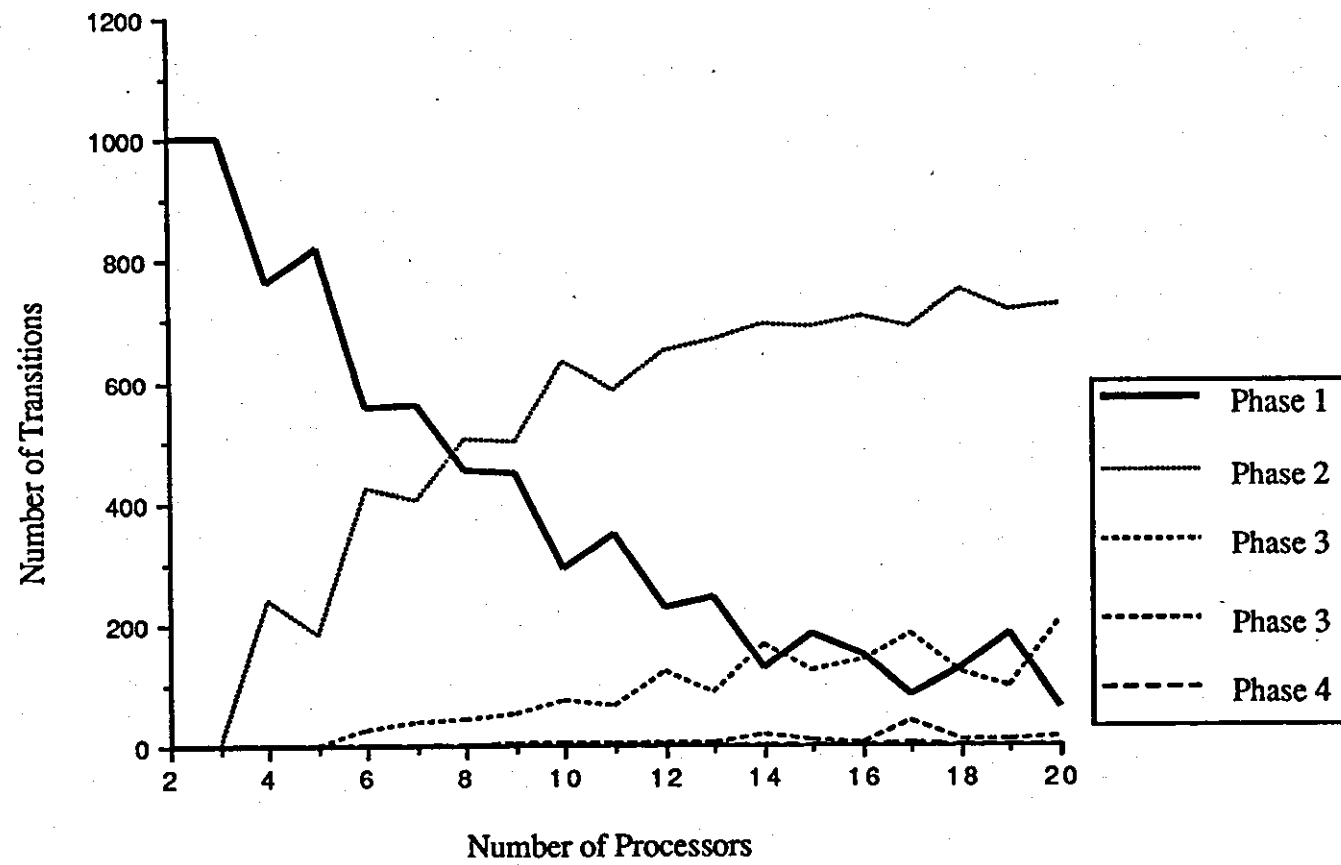


Fig. 6.8 Efficacy of Algorithm B.

be taken as indicators for the efficacy of the move generating algorithm used. For very high processor utilisation, one would hope to have all the moves in the move sets to be in group 1, i.e., only non-interacting moves are desirable for such demands. For cases where significant number of moves in groups higher than 1 is resulted, processor utilisation, efficiency and consequently the throughput of the multiprocessor system would be affected. In order to execute all the moves in such circumstances, one would require more than one *phase* of concurrent move operations. In the first such phase all moves from group 1, some from group 2 and some more from other groups will be executed. Only those moves from higher groups which can be executed without causing any interaction with any other moves are to be considered in the first phase. Similarly, for second phase, all the remaining moves needs to be examined and only those satisfying the above criterion will have a realistic chance to be included. Higher phases of concurrent operations can be carried out in similar fashion so long as there are free moves available. It is clear that the upper limit of the maximum number of phases required is M , where M is the size of each move set. In the efficacy graph to be presented later, we show the number of moves in different phases (according to their degree of interaction) plotted against different processor sizes and in order to keep the graph uncluttered, data values upto phase 5 are plotted. The data flow graph instance of the VLSI logic simulation graph of 4x4 multiplier circuit is used.

Algorithm A :

This is the simplest of the three algorithms tested. In fact in Algorithm A is a over simplification of the move selection criterion deriving from the two starting presumptions already established. The structure of the algorithm is shown in Fig.6.5. Fig.6.6 shows the plot of the results from the simulation run of Algorithm A. It is clearly seen that the efficacy of the algorithm deteriorates very quickly as the number of processors is increased. The high interaction for large processor systems make Algorithm A unsuitable for practical use.

```
Procedure Algorithm_C;  
  MaxMove := NoOfProcessors Div 2;  
  Select any random node for the move;  
  For i := 2 To MaxMove Do  
    Repeat  
      Pick a random node for move;  
      Has this node already been selected?;  
      Is this a direct neighbour of any node already selected?;  
      Is this node's present allocation is the same as that of any  
        other node already selected?;  
    Until All of the above are non-affirmative;  
  EndFor;  
  Repeat  
    Select a random new allocation for the 1st node;  
  Until none of the current allocations of the moves in the move set  
    is repeated;  
  For i := 2 To MaxMove Do  
    Repeat  
      Propose a new random allocation for the  $i^{th}$  selected node;  
      Is any current allocation of the nodes in the chosen move  
        set repeated?;  
      Is this a repeat of any newly chosen allocations?;  
    Until Both of the above are non-affirmative;  
  EndFor;  
EndProcedure;
```

Fig. 6.9 General structure of Algorithm C.

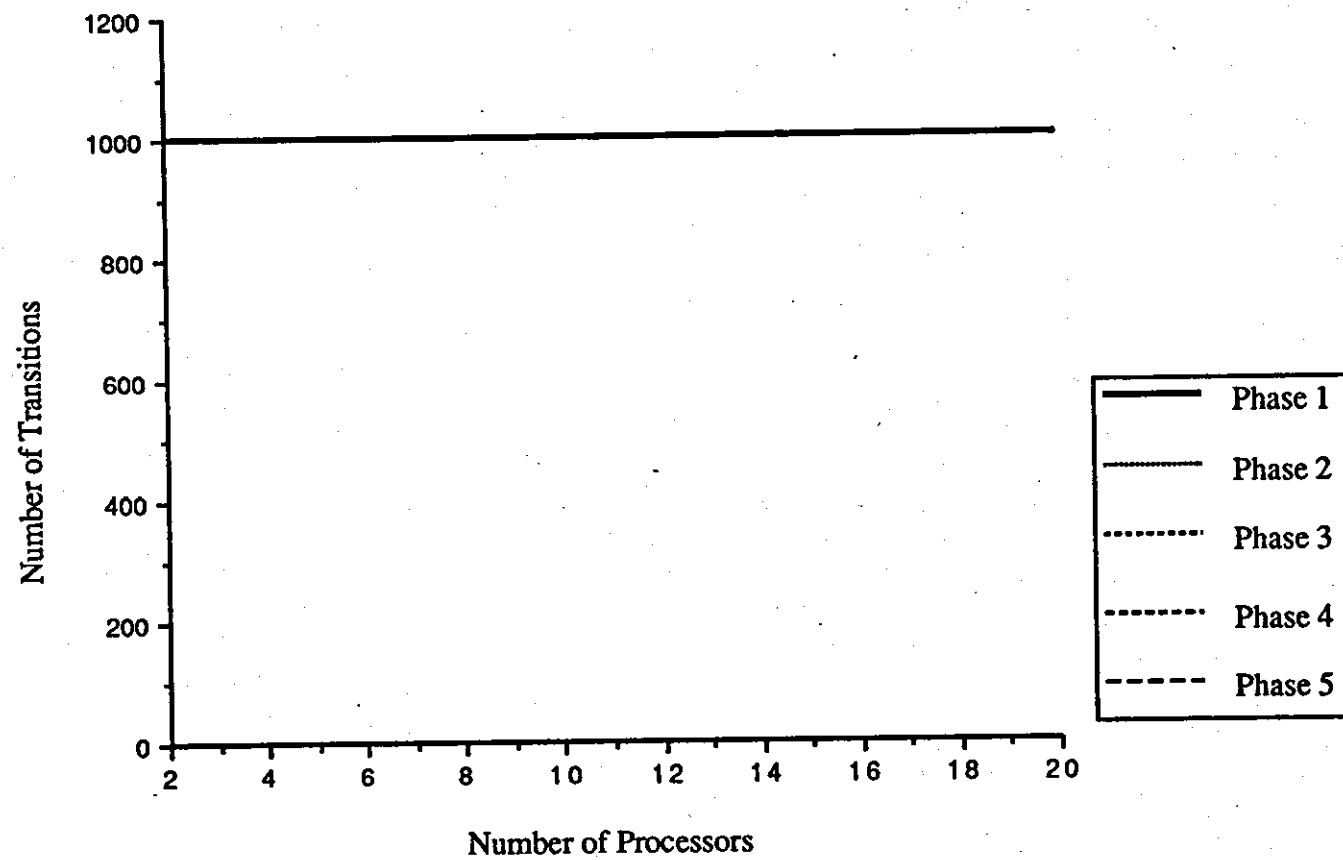


Fig. 6.10 Efficacy of Algorithm C.

Algorithm B :

This is almost similar to Algorithm A with an additional restriction incorporated into it. As before, the nodes are first selected one at a time following the rule we have set before. But, this time the new allocations for each node are proposed only after all the nodes to be moved in the move set are first selected. The new allocations are proposed one by one in such a way that not any one of the old allocations or new allocations already proposed are repeated. This way interaction between the moves in a move set are further reduced.

The simulation results of Algorithm B performed for the same graph instance with processors ranging from 2 to 20 are presented in Fig.6.8. As before, a total of 1000 move sets were generated in each processor case. An improvement is clearly seen. But, again the result is far from the ideal one.

Algorithm C :

In this, we incorporate more control procedures in the way a move is proposed. Here, in addition to the rules we have already used for the selection of a node in a move set, we also make sure that it's present allocation is not the same as that of any already chosen node in the same move set. Once all the nodes in the move set are chosen, we then propose the new allocations for the nodes chosen in the same way as in Algorithm B.

Simulation results of Algorithm C is presented in Fig.6.10. As is evident, this algorithm produces the ideal result, i.e. move sets without interacting moves and obviously best suited for parallel implementation. The moves thus selected are optimal concurrent moves. The price for the excellent performance of Algorithm C is however the increased complexity and the resulting higher execution time.

6.3.3 Parallel Accept/Reject Decisions

The accept/reject decision of a move is dependent on the change of cost (Δc) resulting from the move. This decision is governed by the acceptance criterion of eq.5.4. In a parallel implementation of SA based on parallel move approach, the decision to accept or reject a new solution can be done globally or locally. In the global decision scheme, the individual processors

report their cost changes associated with the moves they were executing to a master processor. The master processor sums up the costs and decides whether or not to accept the summed up new solution.

The other possible scheme for accepting a solution generated by multiple moves is to consider the cost changes separately for each move. The accept/reject decision is locally made by the processors involved.

Since the moves are evaluated locally and independently, it is possible that not all, but part of the moves attempted will be accepted and appear in the new solution. We consider a solution produced by the local decision scheme and using simple statistical analysis similar to [15] compare this with its probability of being accepted in a multiple move SA process using global decision scheme. We suppose that M is the size of the move set and let c_i indicate the cost changes associated with each move $i = 1, \dots, M$. Three different cases are possible and we here explore these to compare the relative suitability of the two decision schemes.

Case A:

The cost changes due to the moves are $\Delta c_i > 0$, for $i = 1, \dots, M$, which suggests that all of them are cost increasing moves. In the local decision scheme, the probability of accepting the move i is $\exp(-\Delta c_i/t)$. The probability of accepting all the moves is the same regardless of whether the decision is made locally or globally. This probability can be expressed as,

$$\begin{aligned} P_1 &= \prod_{i=1}^M \exp(-\Delta c_i/t) \\ &= \exp\left(-\sum_{i=1}^M \Delta c_i/t\right). \end{aligned} \quad 6.1$$

In the local decision scheme, there is also a finite probability of accepting x moves out of M moves in the resulting solution. These can be

numbered as 1 to x . The probability of accepting these moves in local decision scheme is,

$$\begin{aligned} P_2 &= \prod_{i=1}^x \exp(-\Delta c_i/t) \prod_{j=x+1}^M (1 - \exp(-\Delta c_j/t)) \\ &= \exp\left(-\sum_{i=1}^x \Delta c_i/t\right) \prod_{j=x+1}^M (1 - \exp(-\Delta c_j/t)). \end{aligned} \quad 6.2$$

Case B:

The cost changes resulting from the moves are $\Delta c_i \leq 0, i = 1, \dots, M$. These changes indicate that none of the moves has worsened the cost function. Since, $\Delta c_i \leq 0$, a solution with these moves will be accepted in both the schemes.

Case C:

Mixed cost changes are produced in this case. Of the M total moves, y moves numbered from 1 to y are cost increasing moves while the rest improve or at least do not deteriorate the cost value. These cost changes can be written as, $\Delta c_i > 0, i = 1, \dots, y$ and $\Delta c_i \leq 0, j = y+1, \dots, M$. In the local decision scheme, the moves $j = y+1, \dots, M$ will definitely be accepted. In addition, we assume that x out of the y cost increasing moves are accepted. The probability of accepting such a solution through local decisions is,

$$\begin{aligned} P_3 &= \prod_{i=1}^x \exp(-\Delta c_i/t) \prod_{j=x+1}^y (1 - \exp(-\Delta c_j/t)) \\ &= \exp\left(-\sum_{i=1}^x \Delta c_i/t\right) \prod_{j=x+1}^y (1 - \exp(-\Delta c_j/t)). \end{aligned} \quad 6.3$$

If the cost is evaluated globally and $\sum_{i=1}^x \Delta c_i + \sum_{j=x+1}^y \Delta c_j \leq 0$, the cost improving moves dominate the cost evaluation and the probability of accepting all moves is 1. On the other hand, if $\sum_{i=1}^x \Delta c_i +$

$\sum_{j=x+1}^y \Delta c_j > 0$, the global effect of all the moves is cost increasing and probability of accepting them is,

$$P_4 = \exp \left(- \left(\sum_{i=1}^x \Delta c_i / t + \sum_{j=x+1}^y \Delta c_j / t \right) \right). \quad 6.4$$

The global decision scheme considers all the moves at the same time and they are either totally accepted or totally rejected. This eliminates the possibility of accepting erroneously calculated cost values. In the local decision scheme, moves that improve the cost will definitely be accepted, while the cost increasing moves are handled using probability of eq.5.4. This is an advantage of the latter scheme. Since, it will not discard the cost improving moves thus ensuring enough perturbations to satisfy the quasi-equilibrium condition of the SA process. This feature is especially useful at the lower temperatures and thus the capability of locating the optimal solutions at the final stages of the SA process is enhanced.

Since, in CSA non-interacting parallel moves are always used, the advantage of global decision scheme is rendered redundant and as such, the local decision scheme employed ensures that sufficient perturbations are used.

6.3.4 CSA Implementation Model

The concurrent simulated annealing (CSA) algorithm makes use of only one copy of the problem database and also no data decomposition is used. This necessitates the use of a shared-memory closely-coupled MIMD multiprocessor system. This requirement contrasts with the message-passing distributed memory MIMD machine required by the concurrent recursive binary partitioning (CRBP) algorithm discussed in chapter 4.

From Sec.6.3.2 we find that CSA relies on pseudo-synchronous concurrent execution of moves and also that the maximum number of concurrent moves in a single move set is at best only half of the available processors. This frees the other half of available processors to carry out the move generation tasks. The available processors can thus be divided into two separate groups. One group is given the task of generating the moves to be executed by the

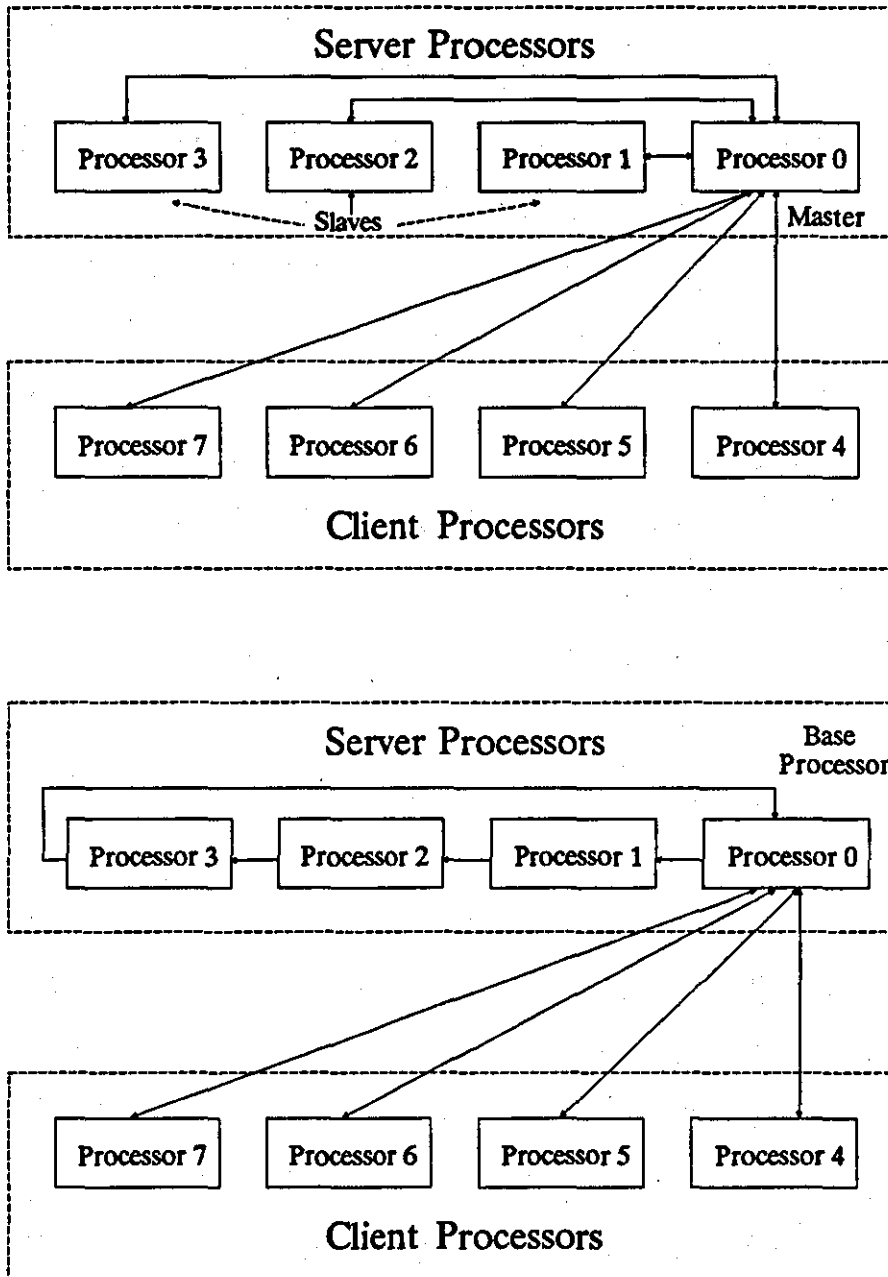


Fig. 6.11 CSA implementation models. Server is in master-slave mode (top) and server is in pipe-lined mode (bottom).

other group and can be referred to as *move server*. One other vital task to be performed by this group is the task of coordinating the whole CSA process. The processors in the other group receive move instructions from the move server and execute them. This group can thus be referred to as the *move client*. CSA can thus be considered as based on the client-server model (Fig.6.11). The transfer of moves between the server and the client is accomplished through shared variables and synchronised through the use of shared flags and counters.

The processors in the client group execute the moves handed over to them by the server group independently and non-cooperatively. The code that each of the client processors execute can be represented as :

Forall clientprocessors in Parallel Do

Repeat

1. Wait for the move instruction from server;
2. Evaluate cost difference, Δc ;
3. Make an accept/reject decision;
4. If accepted, update the database;

Until StopCriterionSatisfied;

EndForall;

Steps 2 to 4 above can be carried out by each client processor at full speed using the shared global memory and no mutual exclusion operation would be necessary for these steps. However, step 1 may become a potential performance constraint for CSA. The efficiency of CSA depends on the wait time at step 1 and in turn depends on the performance of the move server. The wait operation can be realised by using a P(Wait) or V(Signal) on a semaphore. This however, requires an operating system involvement and also process rescheduling. Furthermore, the time that the waiting process remain idle can not be used effectively by another process in the current situation. Instead, simple *spin-lock* synchronisation primitive using a shared lock variable can be used. A client processor clears the lock when it is ready to execute a move and would then loop indefinitely until the server sets the

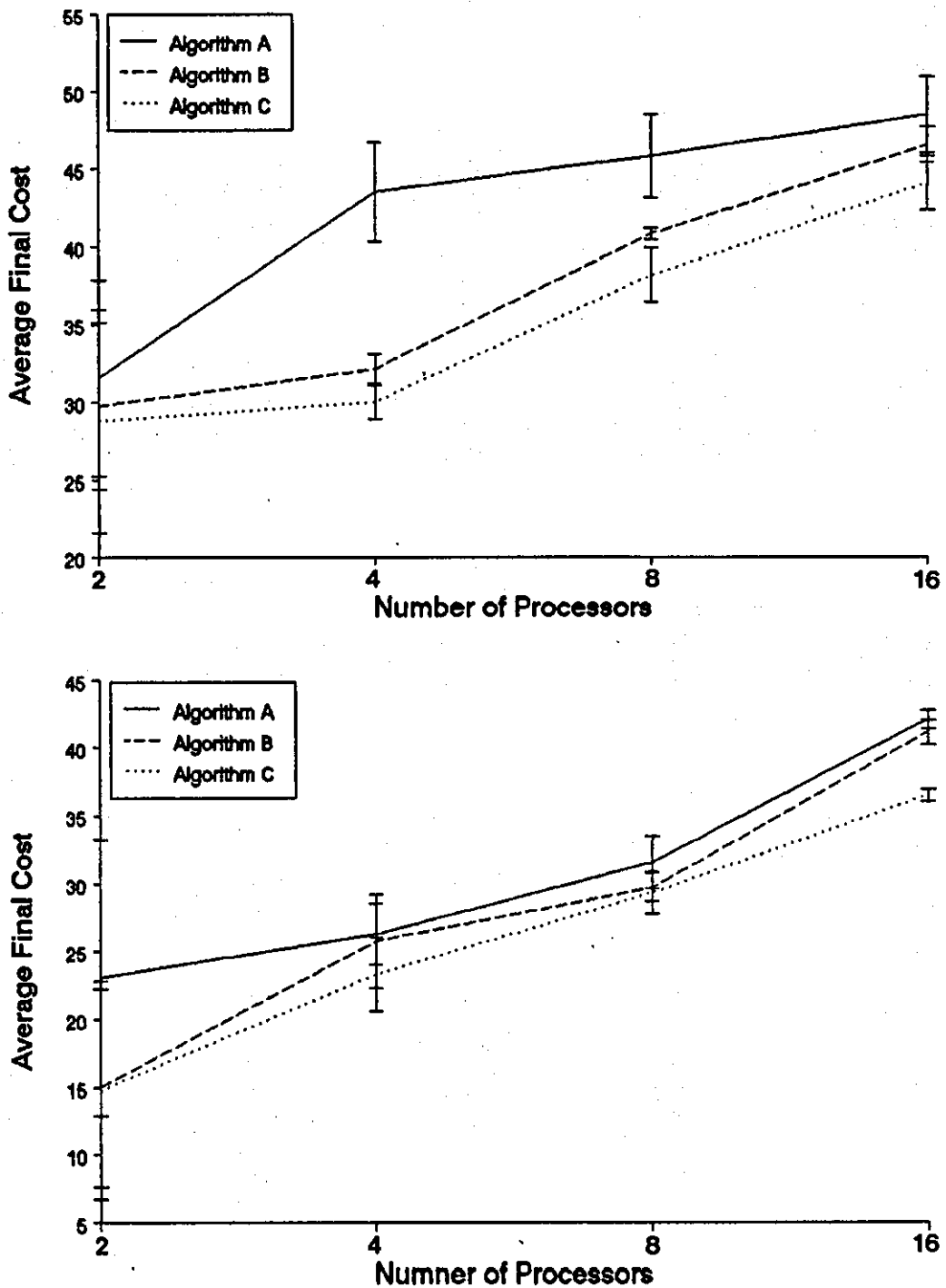


Fig. 6.12 Performance comparison of the three move generating algorithms A, B & C. The polynomial-time cooling schedule is used. Graph data instance are 4x4 multiplier circuit (top) and frequency locked loop circuit (bottom).

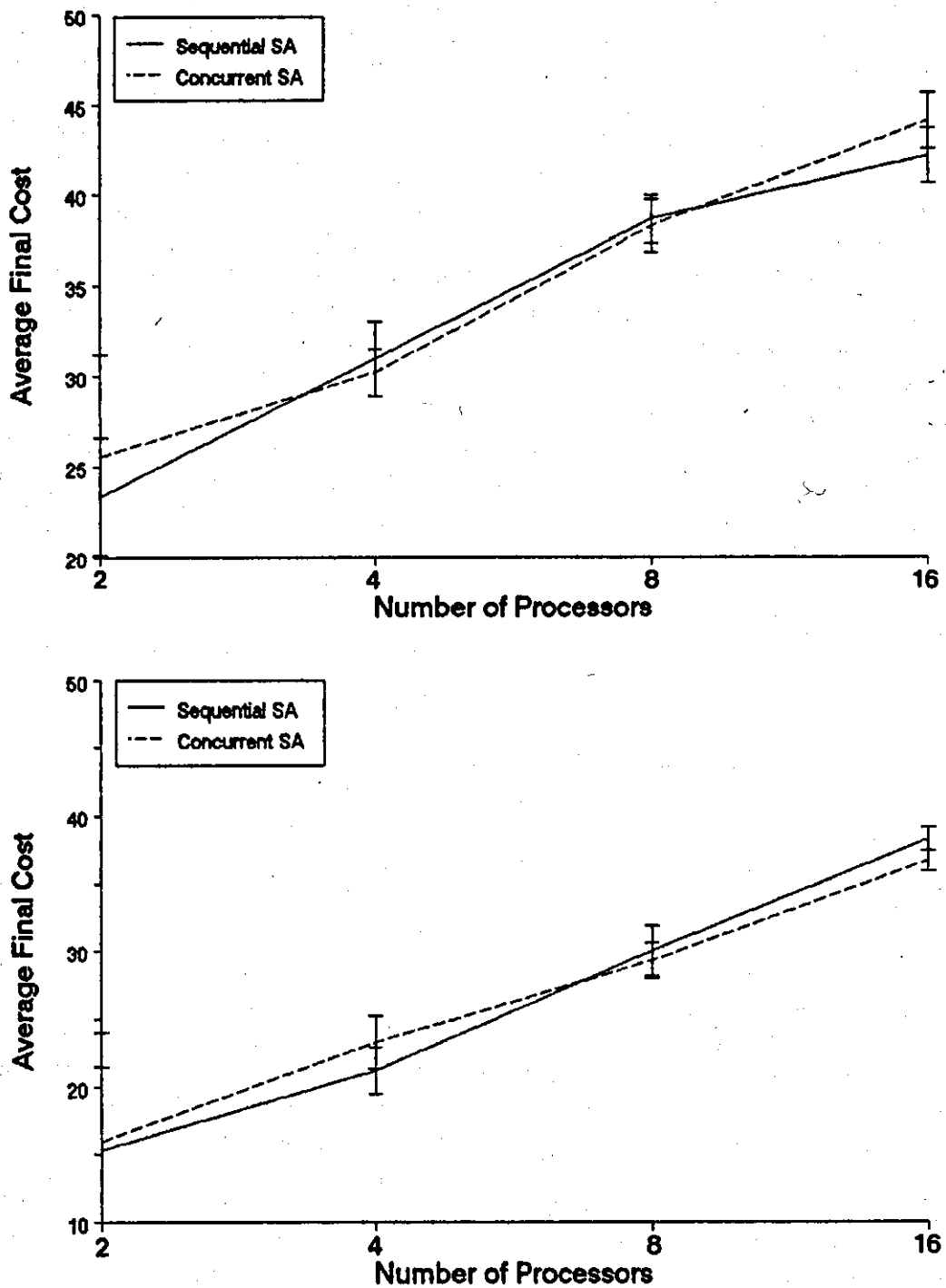


Fig. 6.13 Performance comparison of the sequential (SSA) and the concurrent (CSA) implementations of the simulated annealing algorithm. Data instances are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom). The polynomial-time cooling schedule is used.

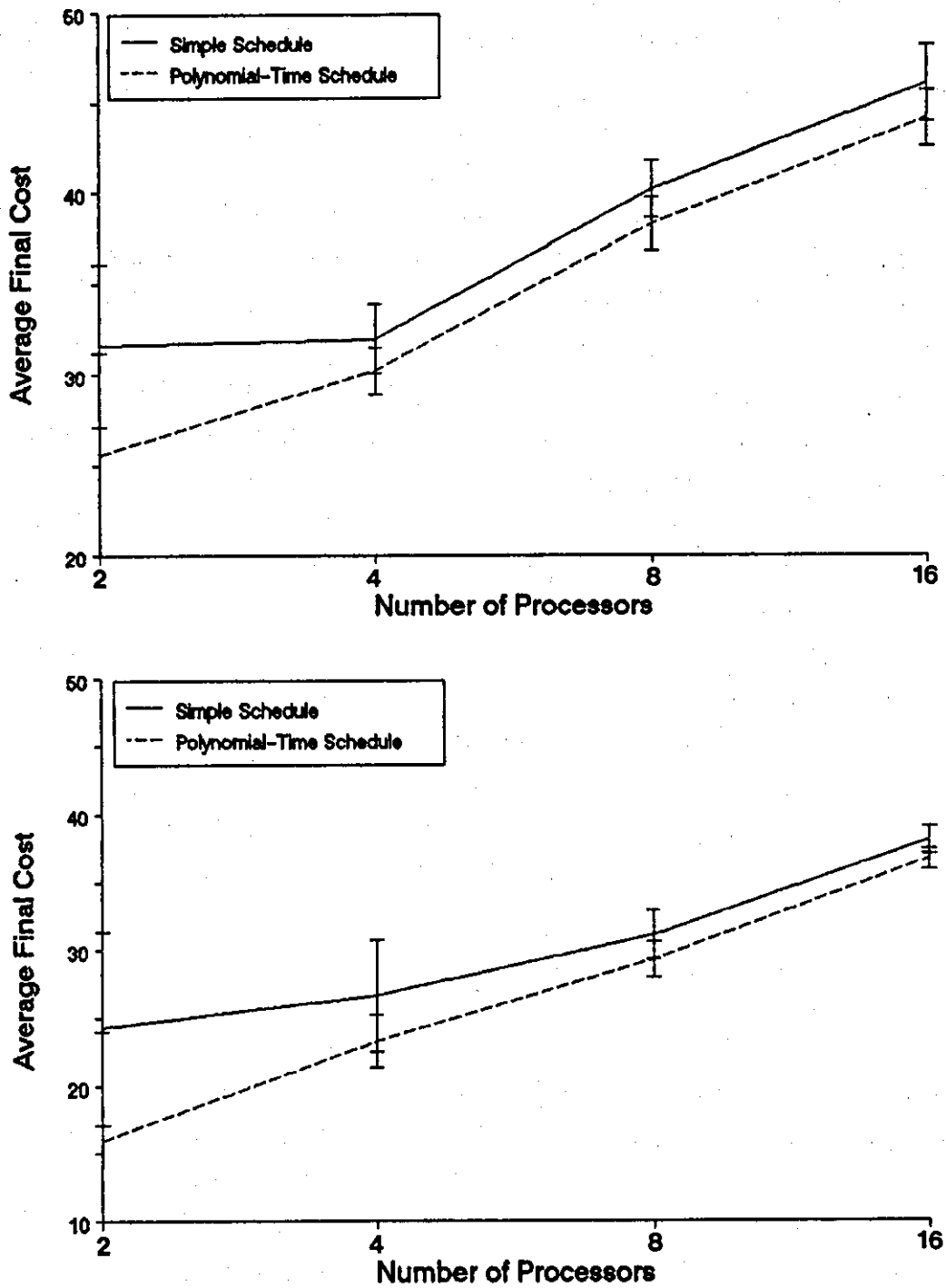


Fig. 6.14 Performance comparison of the two cooling schedules for the concurrent simulated annealing (CSA) algorithm. Data instances are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

lock at which point it is assumed that a move instruction has arrived. The latter method is simple to implement and as the synchronisation times are assumed short, the CPU cycles wasted by the spin locks will be less than the cycles wasted by all the operating system traps generated by the semaphore operation [17].

The processors in the move server group carry out the task of generating non-interacting concurrent moves to be executed by the client processors. The code for move generation is decomposed and assigned to the server processors. Exact decomposition is not shown here, as it depends on the number of processors used. However, the two possible decomposition models which can be used are,

- a. master-slave operation (Fig.6.11a), and
- b. pipe-lined operation (Fig.6.11b).

The efficiency of CSA largely depends on the performance of the move server. As such, the decomposition of move generation task requires careful attention. In this connection, the rich repertoire of parallel algorithms in [17, 18] can be consulted. The move generation task requires a good number of search for neighbouring nodes (Algorithms A,B and C, Sec.6.3.2). Since, the average degree of practical VLSI simulation graphs generally lie between 2 and 5 [19], it is hoped that the search for neighbouring nodes can be accomplished sufficiently quickly. With large number of nodes in the problem graph instance relative to the number of processors, using the pipe-lined operation model, it is also possible to generate the moves in truly pipe-lined fashion and thereby improving the performance of the move server.

6.4 Simulation Results

Simulation was carried out to assess the performance of the CSA algorithm. Because of the non-availability of a suitable multiprocessor system or a simulator, CSA was simulated on a sequential computer. The CSA simulation program executes sequentially different concurrent processes of CSA in appropriate sequence and avoids overlappings or interactions between them. The concurrent moves are first generated in one module

and are then fed to the second module where these moves are executed in sequence. This simulation program carries out only the functional simulation of the CSA algorithm and as such records only the scheduling cost and other related parameters. Parameters such as processor utilisation, message queue statistics etc. are not thus available.

Fig.6.12 shows the performance comparison between the three move generating algorithms A, B and C. Unlike Figs. 6.6, 6.8 and 6.10, here the average final costs of the resulting solutions due to these algorithms used in CSA implementations are reported. It is to be noted that, as CSA is proposed, interaction between concurrently executable moves are not at all accepted and so by that account algorithms A and B are to be avoided (as very few of their resulting moves are interaction free). However, in order to show their efficiency, the CSA simulation program executes all the moves generated by all the three move generating algorithms nonetheless. The results obtained are interesting in that, for all the processor sizes considered (2, 4, 8 and 16) CSA implementation based on algorithm C always outperformed the other two implementations. The margin of difference is however variable for different processor sizes and also different data instances considered. We recall here that, algorithm C is very selective in the way it generates moves resulting in non-interacting moves in move sets. The likelihood is then that, these non-interacting moves in turn cause perturbations in the solution space which are not localised in normal sense and thereby searches the entire solution space more exhaustively for an optimal solution.

However, the above hypothesis does not seem to hold good in Fig.6.13, where the resulting average scheduling costs from CSA (using algorithm C) and the sequential SA are compared. The performances of both the implementations are very similar. By being selective for the generation of moves, CSA with algorithm C, deviates slightly from the true spirit of SA where truly random moves are used to simulate the physical process of metal annealing. On the other hand, however, CSA conforms to the asymptotic convergence properties outlined in chapter 5.

Finally, average final (scheduling) costs from the simple cooling schedule and the polynomial-time cooling schedule are compared in Fig.6.14. As before, the latter schedule is found to produce more acceptable solutions, however at the cost of increased running time.

References :

1. Sechen, C. and Sangiovanni-Vincentelli, A.L., *The Timber Wolf placement and routing package*, IEEE J. Solid State Ckts., Vol. 30, 1985, pp. 510-522.
2. Szu, H. and Hartley, R., *Fast simulated annealing*, Physics Letters, A 122, 1987, pp. 157-162.
3. Greene, J.W. and Supowit, S.J., *Simulated annealing without rejected moves*, IEEE Trans. CAD, Vol. CAD-5, 1986, pp. 221-228.
4. Aarts, E. and Korst, J., *Simulated Annealing and Boltzmann Machines*, John Wiley & Sons, 1987.
5. Cathoor, F., De Man, H. and Vanderwelle, J., *SAMURAI: a general and efficient simulated annealing schedule with fully adaptive annealing parameters*, Integration, Vol. 6, 1988, pp. 147-178.
6. Hoptroff, R.G. and Hall, T.J., *Learning by Diffusion for Multilayer Perceptron*, Elect. Letters, Vol. 25, No. 8, 1989, pp. 531-532.
7. Bout, D.E. Van den and Miller, T.K., *Graph Partitioning using Annealed Neural Networks*, IEEE Conf. Neural Networks, 1989, pp. 521-528.
8. Iosupovici, A.C., King, C. and Breuer, M., *A module interchange placement machine*, Proc. IEEE 20th Design Automation Conf., 1983, pp. 171-174.
9. Spira, C. and Hage, C., *Hardware acceleration of gate array layout*, Proc. IEEE 22nd Design Automation Conf., 1985, pp. 359-366.
10. Kravitz, S.A. and Rutenbar, R.A., *Multiprocessor-based placement by simulated annealing*, IEEE Trans. CAD, Vol. CAD-6, July 1987, pp. 534-549.
11. Aarts, E.H.L., De Bont, F.M.J., Habors, J.H.A. and Van Laarhoven, P.J.M., *Parallel implementation of the statistical cooling algorithm*, Integration, Vol. 4, 1986, pp. 209-238.
12. Woodhams, F.W.D. and Price, W.L., *Optimising accelerator for CAD workstation*, IEE Proc. Vol. 135, Pt. E, July, 1988, pp. 214-225.
13. Darema, F., Kirkpatrick, S. and Norton, V.A., *Parallel algorithms for chip placement by simulated annealing*, IBM J. Research Dev., Vol. 31, No. 3, May 1987, pp. 391-402.

14. Casotto, A., Romeo, F. and Sangiovanni-Vincentelli, A.L., *A Parallel Simulated Annealing Algorithm for the Placement of Macro Cells*, IEEE Trans. Cad, Vol. CAD-6, No. 5, Sept. 1987, pp. 838-847.
15. Vai, M.K., *Acceleration of Simulated Annealing Building Block Placement Process*, Ph.D. Thesis, Michigan State Univ., USA, 1987.
16. Rahin, M.A. and Sheild, J., *Concurrent Partitioning of VLSI Simulation Graphs*, Proc. Conf. Parallel Computing '89, Leiden, The Netherlands, 1989.
17. Quinn, M.J., *Efficient Parallel Algorithms*, McGraw Hill Book Co., USA, 1987.
18. Fox, G., Johnson, M., Lyzenga, G., Otto, S., Solman, J. and Walker, D., *Solving Problems for Concurrent Processors*, Prentice-Hall, USA, 1988.
19. Goldberg, M.K., and Burstein, M., *Heuristic improvement technique for bisection of VLSI networks*, IEEE Proc. Int. Conf. on Computer Design, Port Chester, New York, USA, 1983.

CHAPTER 7

Conclusions & Discussion

This chapter provides an overview of the work as a whole. The results obtained so far are reviewed and the performances and suitability of the algorithms investigated are discussed. Finally, a conclusion of the overall work is made and those aspects requiring further research are highlighted.

In this thesis, the problem of scheduling VLSI simulation systems on a general or special purpose multiprocessor systems is considered. The concurrent VLSI timing simulation system considered is based on the data-flow computation model. A simple directed acyclic graph (dag) model is adopted for the VLSI simulation system. This allows easy mapping of the simulation system onto the target multiprocessor system. However, in order to achieve an efficient and near-optimal mapping scheduling algorithms are used. Because of the NP-Hardness of scheduling problem, heuristic procedures are favoured over exhaustive enumeration search procedures. Two heuristic procedures are investigated in detail and their concurrent

implementations are proposed. The first of these two heuristics involve a hierarchical partitioning of an input graph and the other one has its root in classical statistical physics.

7.1 Review of Results

The focus of the research reported in this thesis is on heuristic algorithms. Two heuristic algorithms - the recursive binary partitioning (RBP) and the simulated annealing (SA) are investigated in detail. Being heuristic and approximate in nature, the above two algorithms are not expected to provide optimal solutions for all cases of the multiprocessor task scheduling problems all the time. Indeed, for most of the test cases considered, their optimal solutions are not known. The performance analysis of the RBP and SA algorithms as such involves their average performance. Furthermore, all the results unless explicitly stated are expressed as the percentage of the average (random) initial scheduling cost of the task system.

The recursive binary partitioning (RBP) algorithm is based on Kernighan-Lin's [1] graph bi-partitioning procedure. The basic bi-partitioning procedure used is a slightly modified version of KL's original heuristic. This is needed to bring the load-imbalance criterion in the objective function of the multiprocessor task scheduling problem presently posed as a graph partitioning problem. The modification involves repetitive move operations as opposed to the original repetitive swap operations. The move operations are accomplished by transferring a single node from one of the sub-graphs of the tentative partition to the other sub-graph. Three different move selection rules are explored. Among these three rules considered, Rule C is found to be most efficient as far as the graph partitioning objectives are considered (Sec. 4.1.2) and as such this rule is used in all subsequent versions of RBP and also in its concurrent implementation, CRBP algorithm.

The multi-way partition in chapter 4 is accomplished through the application of hierarchical partitioning process. At each level of partitioning, starting from level 1, the modified KL bi-partitioning process is applied and the resulting partitions are then propagated to the next higher level. This is repeated until the desired number of partitions (a power of 2) are obtained. This major criticism of the recursive binary partitioning process is that for

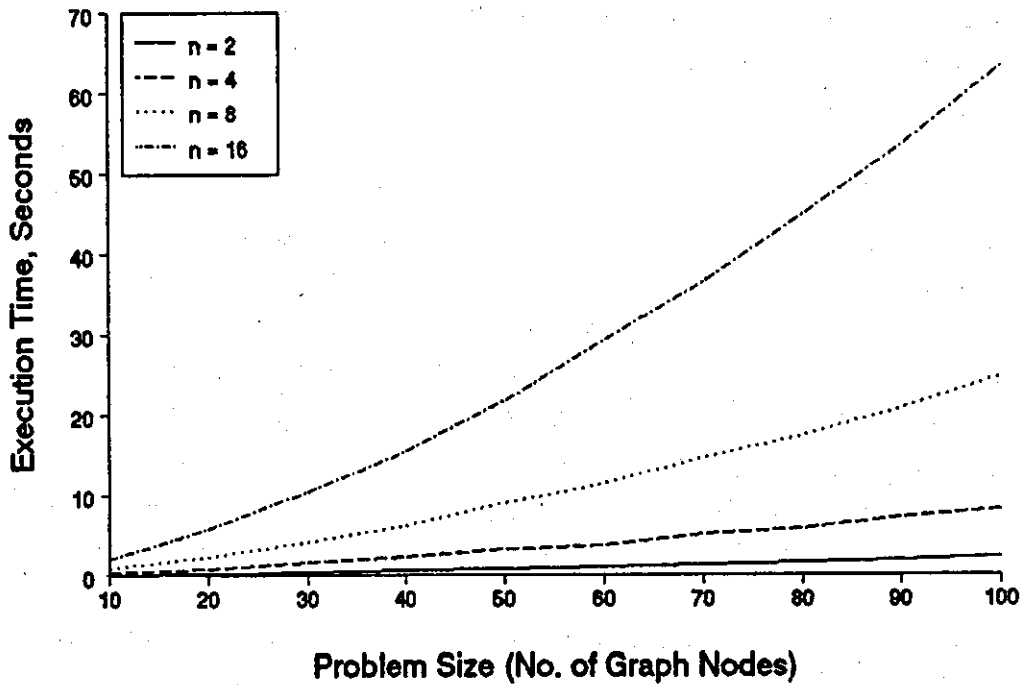
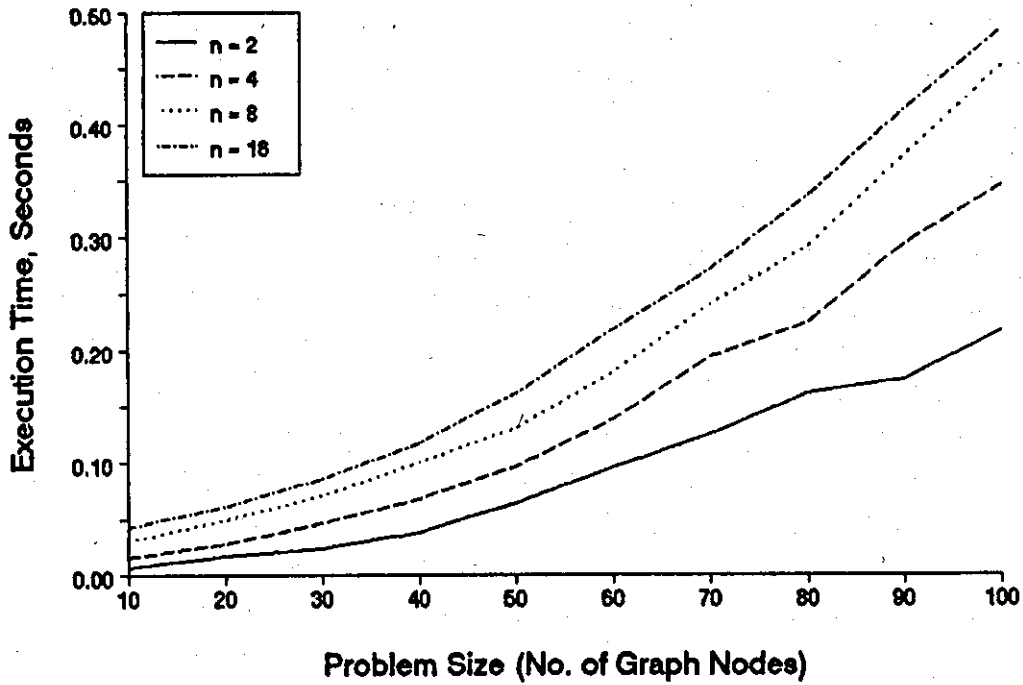


Fig. 7.1 Execution time as function of the problem size for different processor sizes. Results from the Recursive Binary Partitioning algorithm (top) & the Simulated Annealing algorithm with polynomial-time cooling schedule (bottom) are presented. Synthetically generated graph instances are used.

cases where larger than quite a few partitioning levels are required, the final partition cost often turns out to be of inferior quality. This is explained by the fact that the early levels of partitions (mostly levels 1 & 2) tend to minimise the number of edges cut at the cost of keeping more heavily connected nodes of the input graph within the single sub-graphs and these sub-graphs then as a result have less chance of attaining a near-optimal cut in the next bi-partitioning process. This amounts to a greedy approach at earlier levels. In our experiment, it has been found that, for cases where partition levels upto 3 (three) resulting in 8 partitions are required, the resulting final solutions are quite acceptable. A marked deterioration of the quality of the final partition cost is noticed when 16 partitions are created.

In a separate experiment, an attempt is made to reduce the adverse affect of the result of the earlier (levels 1 & 2) partitions on the final partition cost for multi-way (≥ 4) partitions. This is accomplished by varying the relative emphasis on the two cost components, viz. communication and load-imbalance costs. However, the results obtained failed to show any improvement achieved. It is found that the final partition cost is practically invariant beyond a certain value of the weight factor, z (eq. 4.11). For smaller z , the final partition cost is mostly made up of load-imbalance cost as expected and larger values of z result in final partition costs with a significant communication overhead. It is however, to be noted that the above findings need to be judged with the actual formulation of the partition cost, which for the present case places equal emphasis on both the two cost components.

The concurrent recursive binary partitioning (CRBP) procedure is a logical progression of the RBP algorithm. In the CRBP algorithm presented in chapter 4, the *best-of-bunch*, which means engaging a group of processors whenever available, to perform independently the bi-partitioning procedure at various stages at different levels of a multi-way partitioning process but accepting the best result only is used. This resulted in an improved performance (Fig. 4.12) for all the cases considered.

The simulated annealing (SA) [2] algorithm and its concurrent implementation (CSA) are investigated in detail in chapters 5 and 6 respectively. In chapter 5, the SA algorithm is introduced as modeled on Markov chains. Two cooling schedules are investigated. Both of these are approximations of

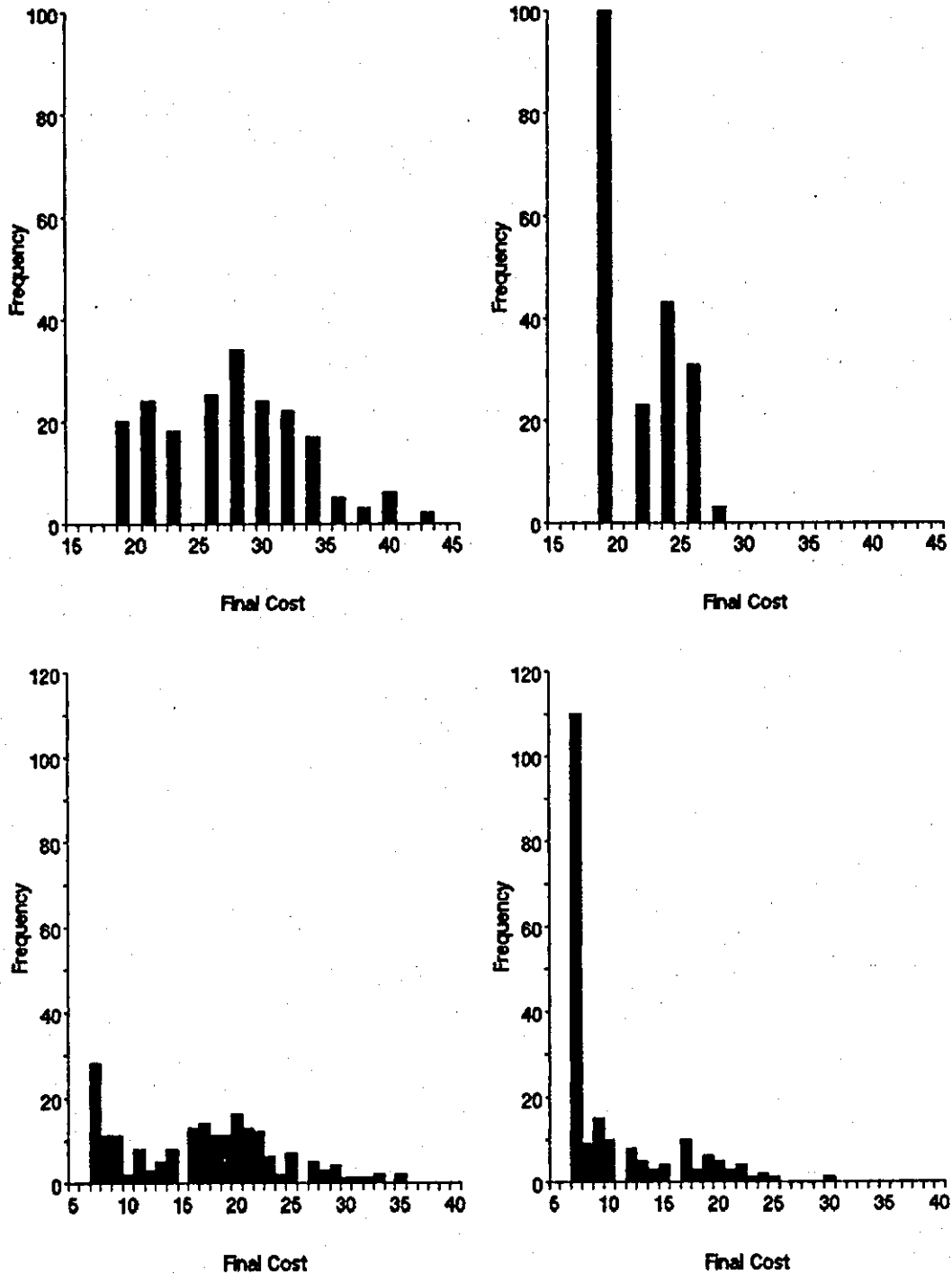


Fig. 7.2 Frequency distribution of final scheduling costs for 2-processor system. Results from CSA are on the left & that from CRBP are on the right. Graph data instances used are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

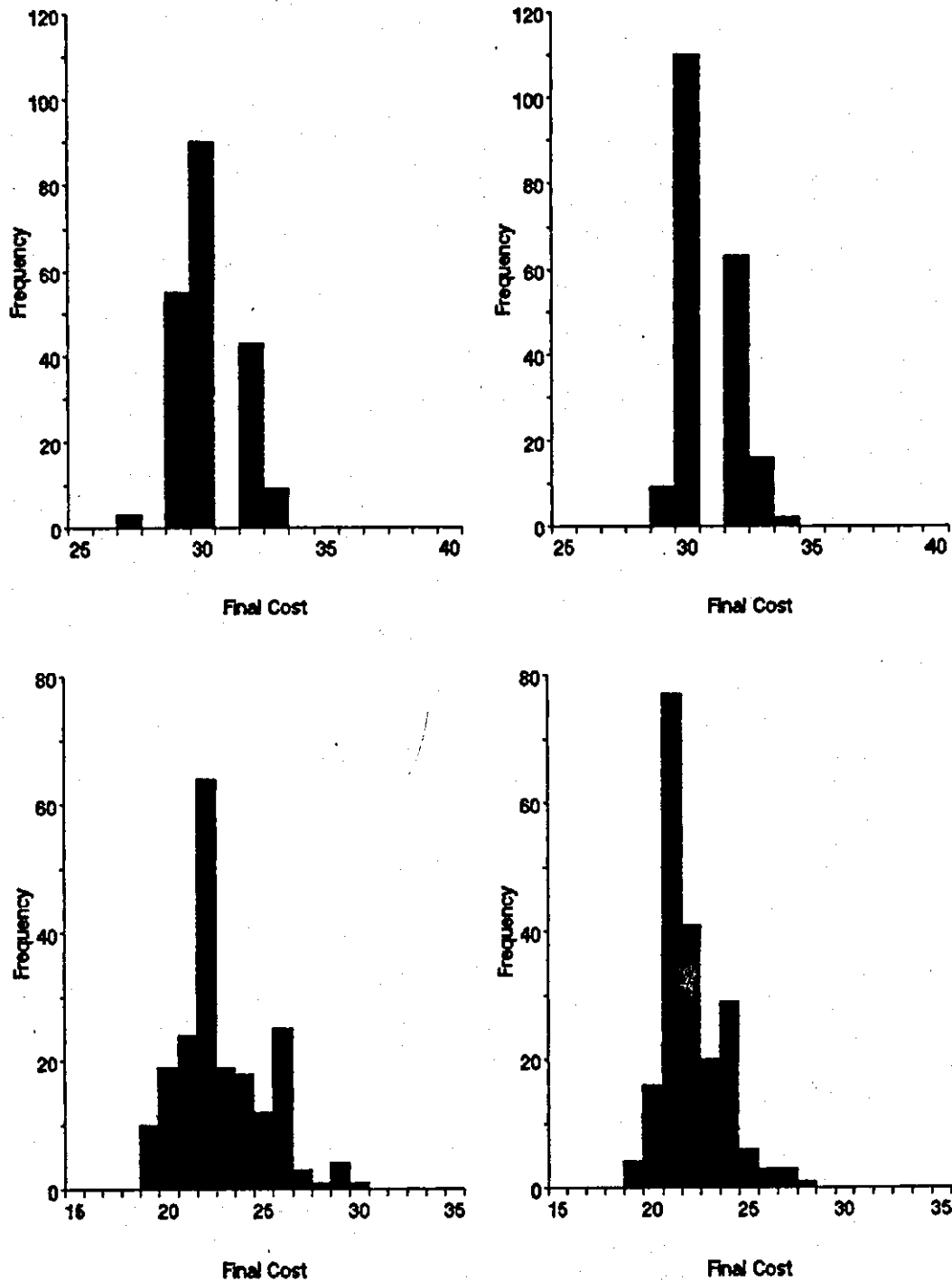


Fig. 7.3 Frequency distribution of final scheduling costs for 4-processor system. Results from CSA are on the left & that from CRBP are on the right. Graph data instances used are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

the inhomogeneous Markov chains and are expected to run in polynomial-time. Two transition mechanisms, swap and move and also various compositions of these two are considered. It has been found that for most of the cases the final cost obtained has very little dependence on the choice of transition mechanism (or their compositions). However, 100% swap definitely have a deteriorating effect on the final schedule. This is so because of the inability of swaps to address the load-imbalance issue properly.

The performances of the two cooling schedules are analysed by varying different control parameters as applicable. As expected, both schedules provided improved results with the increase in the size parameter, m (i.e., when number of attempted transitions are increased) but at the cost of a commensurate linear growth in solution time. The performance of the simple cooling schedule is found to be practically invariant with the choice of initial acceptance ratio, whereas the polynomial-time cooling schedule showed a marked dependence with this control parameter. Overall, the simple cooling schedule is found to be quicker in reaching a solution compared to the polynomial-time cooling schedule. However, the latter consistently provided better quality solutions than the former and also the latter is more robust and amenable to the variation in problem instance by virtue of its design. In retrospect, however, the choice of any of these two cooling schedules is difficult and needs to be carefully weighed in relation to the problem instance.

The concurrent simulated annealing (CSA) algorithm is discussed in detail in chapter 6. One unique feature of CSA is that it works with totally interaction free parallel moves thus ensuring conformance with the asymptotic convergence properties of the SA algorithm. Three move generation algorithms are explored from which Algorithm C is found to be 100% successful in producing totally interaction free move sets. Comparing the performances of these three move generation algorithms it is also found that Algorithm C produces the best scheduling cost. This is attributed to the non local perturbation activities within the solution space. Finally, the two cooling schedules, viz. simple and polynomial-time schedules are applied in CSA and as before, the latter is found to produce better solutions.

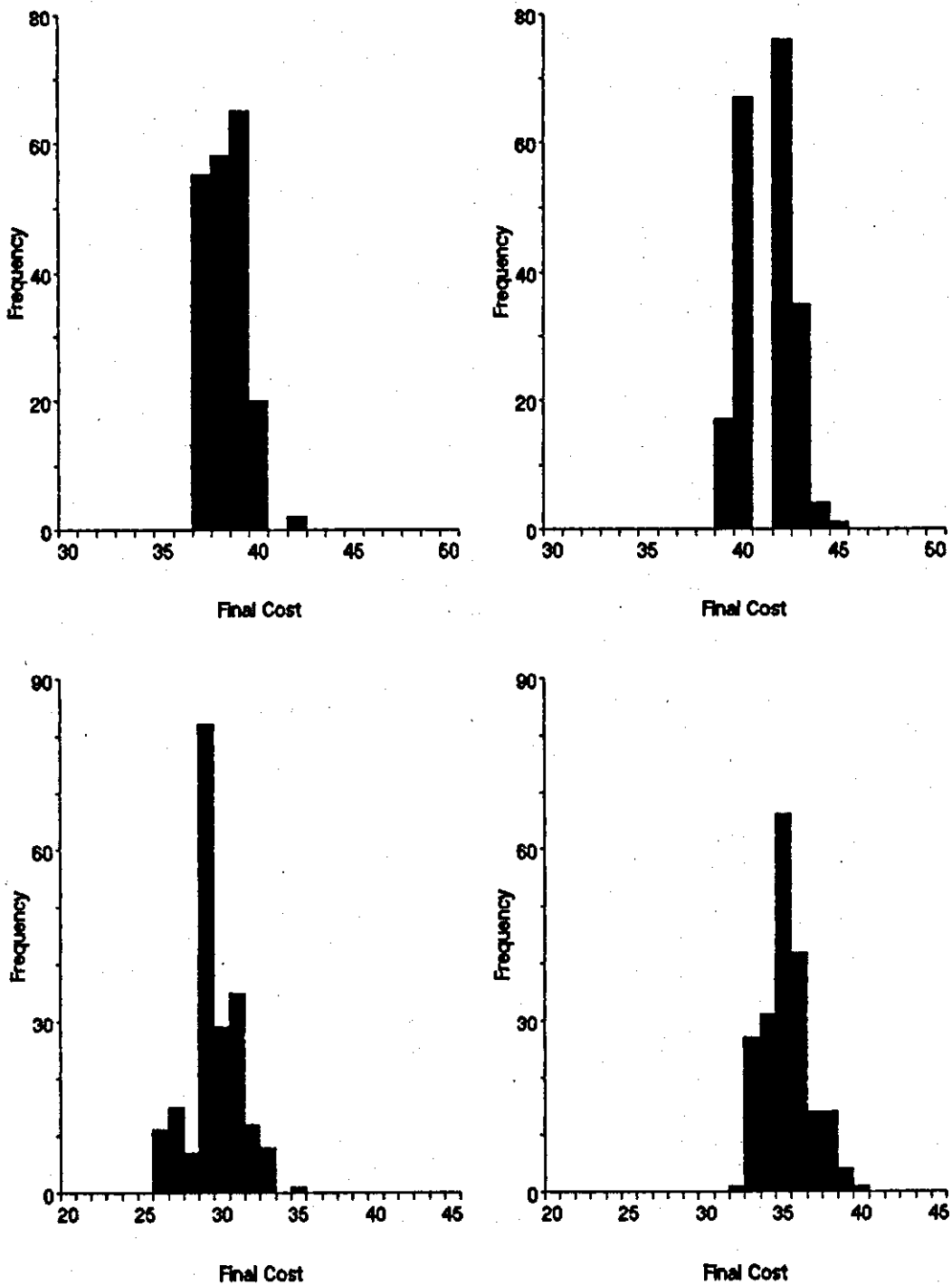


Fig. 7.4 Frequency distribution of final scheduling costs for 8-processor system. Results from CSA are on the left & that from CRBP are on the right. Graph data instances used are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

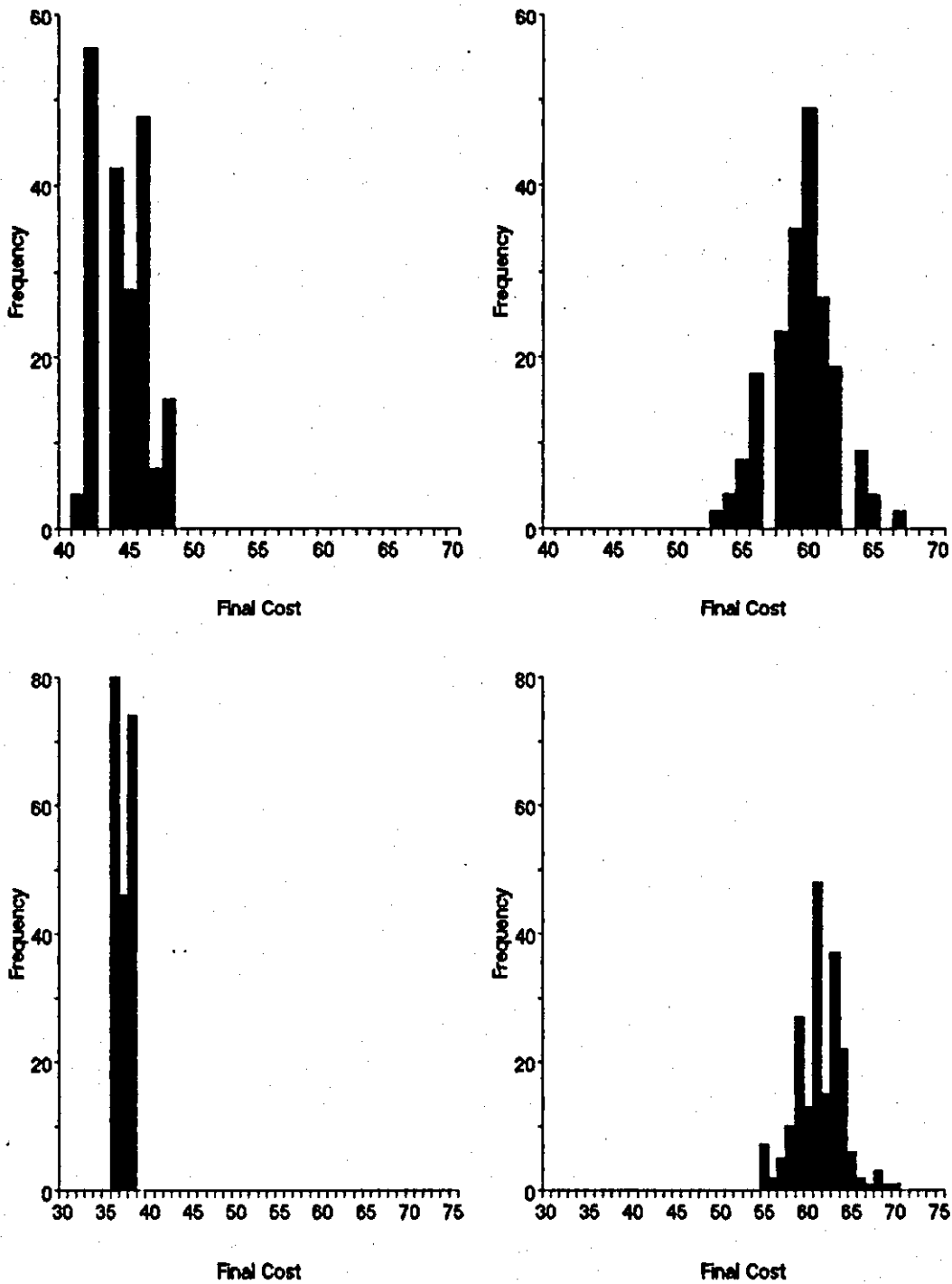


Fig. 7.5 Frequency distribution of final scheduling costs for 16-processor system. Results from CSA are on the left & that from CRBP are on the right. Graph data instances used are 4x4 multiplier ckt. (top) & frequency locked loop ckt. (bottom).

7.2 Recursive Binary Partitioning vs. Simulated Annealing

In this section the two heuristic algorithms studied so far are compared. Fig. 7.1 compares the time requirement of both the algorithms. Since, the concurrent implementations are simulated with a functional simulator, the exact determination of their time requirement is thus difficult and as such Fig 7.1 depicts the time requirement of their serial implementations. For the SA algorithm the polynomial-time cooling schedule is used with the following parameter set, $\zeta_0 = 0.98$, $\epsilon_s = 0.00001$, $\delta = 0.1$ and $m = 1$. Synthetically generated graph instances with number of nodes ranging from 10 to 100 with roughly uniform connectivities are used. Four different processor sizes 2,4,8 and 16 are considered.

It is interesting to note that the observed time complexity of both RBP and SA algorithms for the different processor sizes considered differs quite significantly. The RBP algorithm requires roughly $O(k^2)$ time and the SA algorithm's requirement is slightly worse than $O(k \log k)$, where k is the number of nodes in the input graph instance. However, the actual time taken by the SA algorithm is much higher than it's counterpart. RBP is found to be in cases about 130 times faster than SA. Furthermore, in SA the time requirement is very strongly dependent on the processor size. This is a direct consequence of the formulation of the length of the Markov chain used (eq. 5.35). On the other hand, in RBP, there is significantly lower dependence of time requirement on processor size. This can be explained as follows. Even though, for large processor sizes, more partitioning levels are required thereby causing increased demand for completion time, the sub-graphs ready to be bi-partitioned at the last level become comparatively much smaller in size to warrant a significant increase in completion time.

Figs. 7.2 to 7.5 give a performance comparison between the CRBP and CSA algorithms. The figures show the frequency distribution of the schedules obtained through the two algorithms. A total of 200 iterations are made for each of the two algorithms and for different processor sizes considered. The data instances considered are the simulation graph data for a 4x4 multiplier ckt. and also for a frequency locked loop ckt. The histograms in Figs. 7.2 to 7.5 show the final schedule cost expressed as percentage of the average

(random) initial scheduling cost against their frequency of occurrences. It is observed that for smaller processor size, the spread of the final schedule cost for CSA is comparatively larger than that for CRBP. However, this is reversed as the processor size is increased. Also, the mode (the most frequent result) for CRBP tend to be nearer to the lower edge of the spread as compared to the CSA suggesting that the CRBP is statistically superior than its counterpart. The most important observation is that the overall performance of CRBP is comparatively much superior than that of CSA for very small processor size (2). But, as the processor size is increased, CSA slowly outperforms CRBP and for processor size 16, CSA comes out as the winner with a significant margin. This behaviour is, however, expected, as the result of successive partitioning in CRBP leaves the final result less acceptable and is now vindicated in a comparative study.

The variation in the pattern of these histograms strongly suggest a dependence of the graph structure on the final scheduling cost. Where the graph for the 4x4 multiplier ckt. has most of the nodes more or less uniformly connected except for few branches showing strong sequential activity (Appendix A), the graph for the frequency locked loop ckt. shows wide variance in the connectivity pattern, where one node is very heavily connected in relation to the other nodes. This variance in graph structures, it is believed, favour a certain natural partitions in some cases and offer difficulty in attaining non-natural partitions.

Tables 7.1 and 7.2 summarize the results depicted in the above histograms. Here, the maximum, minimum and average scheduling cost as well as the standard deviation is expressed as the actual scheduling cost. The last column, however, expresses the figures as percentage of the average (random) initial scheduling cost as expressed throughout this thesis.

The final phase of this comparative study of CRBP and CSA algorithms involve subjecting these algorithms to problem instances with known optimal scheduling cost. These problem instances are synthetically generated from some basic graph instances. Multiple (the same number as the number of partitions or processor size is required), but disjoint copies of the basic graph instance is used as a single graph data instance. The resulting optimal scheduling cost thus becomes zero. Three basic graph instances are used

— (a) a small synthetic graph with 6 nodes and two visible clusters, (b) 4x4 multiplier ckt. and (c) frequency locked loop ckt. Table 7.3 summarizes the result. The final scheduling cost is once again expressed as the percentage of average (random) initial scheduling cost. The total success column shows the number of times either algorithm achieves the optimal solution. The relative success column shows the number of times either algorithm performed better than the other. It is clearly seen that for the basic graph instances (a) and (b), optimal solutions are more easily achievable. Basic graph instance (c) proved to be much more difficult to partition optimally. For the first two basic graph instances and for processor sizes upto 8, CRBP outperformed CSA. The variation in graph structure in the basic graph instance (c) as stated in the preceding paragraph is thought to have influenced a change in the pattern of results. This again is another demonstration of structures of certain graph data instances affecting the natural partition. Overall it is observed that CRBP is more at home in finding the natural partitions for simple graphs, whereas for difficult graphs like the basic graph (c) CSA has shown a definite edge.

7.3 Conclusions

Summarizing the above results, we present the following conclusions :

- a. Concurrent heuristic algorithms proposed in this thesis offer a viable alternative for the solution of a class of difficult combinatorial optimisation problems.
- b. In view of the improvements in scheduling overheads obtained, it is fair to expect a general speed-up of the concurrent VLSI timing simulation system in an actual run.
- c. For the heuristic algorithms studied, concurrency does not affect the solution quality. Concurrency in many instances enhances the solution quality.
- d. Structure of the graph data instance is thought to have an influence on the solution quality.
- e. For the data instances considered, CRBP algorithm is found to be faster than the CSA algorithm. However, the latter is algorithmically superior to the former.

Number of Processors	Iterations	Average Initial Scheduling Cost	Final Scheduling Cost				
			Maximum	Average	Minimum	Standard Deviation	As % of Initial Scheduling Cost
2	200	46.42	13	10.07	9	1.21	21.69
4	200	69.85	24	21.46	20	0.75	30.72
8	200	79.49	36	32.72	31	0.95	41.17
16	200	83.35	56	49.60	44	2.09	59.51

Number of Processors	Iterations	Average Initial Scheduling Cost	Final Scheduling Cost				
			Maximum	Average	Minimum	Standard Deviation	As % of Initial Scheduling Cost
2	200	46.93	42	13.06	9	3.28	27.84
4	200	69.39	23	21.00	19	0.85	30.26
8	200	79.25	33	30.28	29	1.01	38.21
16	200	83.24	35	34.30	34	0.45	41.20

Table 7.1 Table comparing the performances of the Concurrent Recursive Binary Partitioning (CRBP) (top) & the Concurrent Simulated Annealing (CSA) (bottom) algorithms for different processor sizes. Graph data instance is the 4x4 multiplier ckt.

Number of Processors	Iterations	Average Initial Scheduling Cost	Final Scheduling Cost				
			Maximum	Average	Minimum	Standard Deviation	As % of Initial Scheduling Cost
2	200	90.54	27	9.09	6	4.66	10.04
4	200	131.58	37	28.87	25	2.14	21.94
8	200	146.58	59	51.82	47	2.03	35.35
16	200	151.96	106	93.07	83	4.10	61.24

Number of Processors	Iterations	Average Initial Scheduling Cost	Final Scheduling Cost				
			Maximum	Average	Minimum	Standard Deviation	As % of Initial Scheduling Cost
2	200	90.34	71	15.20	6	7.48	16.83
4	200	129.05	39	29.42	24	2.87	22.80
8	200	147.04	99	43.74	38	4.78	29.74
16	200	152.04	57	55.40	54	0.66	37.52

Table 7.2 Table comparing the performances of the Concurrent Recursive Binary Partitioning (CRBP) (top) & the Concurrent Simulated Annealing (CSA) (bottom) algorithms for different processor sizes. Graph data instance is the frequency locked loop ckt.

- f. The CRBP though gives very good quality solution for smaller processor sizes, a deterioration in the solution quality is observed with the increase in processor size. CRBP however is well suited for finding the natural partition present in some graph instances.
- g. The CSA algorithm has a consistent performance behaviour.
- h. The idealised speed-up of CRBP is not expected to grow linearly with the increase of processors, but is expected to stay within the lower bound of $\log n$.
- i. The maximum speed-up with CSA is only half of the processors engaged, but is expected to grow linearly with the increase of processors.

7.4 Discussion

Through the simulation results presented so far, it is established that the recursive binary partitioning (RBP) algorithm offers a fast and reasonable means for the solution of multiprocessor task scheduling problems. However, where slightly larger multiprocessor systems are considered, the advantage of RBP, especially its ability to provide reasonable quality solutions slowly disappears. The simulated annealing algorithm, on the other hand, though comparatively slower is much more effective for handling larger multiprocessor systems. It is however, to be appreciated that, the discussion here applies for coarse grain model of parallel processing. The complexity of the scheduling problem for the other competing model, viz. fine grain parallelism is much too high for the two heuristics considered here to possibly make them unsuitable. For this scenario a completely different strategy needs to be considered.

Apart from the speed-up of SA algorithm through concurrency as suggested in CSA, there exists other potential means — namely by utilising a more efficient cooling schedules, pre-processing of problem instance for which the RBP algorithm is eligible, improved approximations of the inhomogeneous Markov chain etc. These, however, were not considered in the current study in order to keep the problem of parallelising SA algorithm simple. It would provide an interesting exercise for the future to incorporate some of these ideas into CSA.

Number of Processors	Number of disjoint sub-graphs	Optimal Solution	Total Success		Relative Success		Average Scheduling Cost		Standard Deviation	
			CSA	CRBP	CSA	CRBP	CSA	CRBP	CSA	CRBP
2	2	0	20	20	-	-	0.00	0.00	0.00	0.00
4	4	0	20	20	-	-	0.00	0.00	0.00	0.00
8	8	0	20	20	-	-	0.00	0.00	0.00	0.00
16	16	0	19	0	20	0	0.14	39.77	0.61	3.47

Number of Processors	Number of disjoint sub-graphs	Optimal Solution	Total Success		Relative Success		Average Scheduling Cost		Standard Deviation	
			CSA	CRBP	CSA	CRBP	CSA	CRBP	CSA	CRBP
2	2	0	0	20	0	20	6.96	0.00	3.21	0.00
4	4	0	0	20	0	20	5.06	0.00	1.71	0.00
8	8	0	0	20	0	20	1.80	0.00	1.04	0.00
16	16	0	0	0	20	0	0.34	49.54	0.52	1.10

Number of Processors	Number of disjoint sub-graphs	Optimal Solution	Total Success		Relative Success		Average Scheduling Cost		Standard Deviation	
			CSA	CRBP	CSA	CRBP	CSA	CRBP	CSA	CRBP
2	2	0	0	0	8	12	14.00	10.36	2.86	3.65
4	4	0	0	0	15	5	8.70	10.76	1.89	1.76
8	8	0	0	0	19	1	6.52	12.33	1.09	1.29
16	16	0	0	0	20	0	4.96	55.88	0.21	1.30

Table 7.3 Table comparing the relative performances of the algorithms CSA & CRBP when the optimal scheduling cost is known. Data instances are made of multiple but disjoint copies of a 6-node, two cluster graph (top), 4x4 multiplier ckt. (middle) & frequency locked loop ckt. (bottom).

It has been found that with CSA, free from interactions between parallel moves, the upper limit of the achievable speed-up is only half of the processors engaged. However, a linear growth of speed-up with processors is expected. The above restriction is a direct result of the identification of interaction between parallel moves being executed simultaneously and is a consequence of the way the problem is enclosed within its own bound, i.e., the processors trying to find an optimal mapping of nodes onto themselves. This speed-up limitation however, does not apply to the VLSI cell layout problem which bears a close resemblance with the multiprocessor task scheduling problem.

In order to achieve further speed-up with CSA an adaptive strategy can be employed. This involves the merging of CSA with another concurrent variant of SA perhaps modeled on the error algorithms. CSA can be applied at the early high temperature stages when there is a greater probability of errors occurring and then to use the alternate algorithm which engages all the available processors to take over the annealing process at lower temperatures. This hybrid strategy has the potential to offer increased speed-up performance, but is riddled with experimental difficulty. The switch over point needs to be very carefully determined so as not to affect the final solution. The speed-up advantage needs to be carefully weighed against the effect on the final solution, which might be adulterated by accepting erroneous transitions at lower temperatures.

Apart from the two heuristics investigated there are many other heuristic algorithms which might have the potential to offer acceptable solution at reasonable speed for the current problem. Preliminary investigations with the Genetic (or Evolution) Algorithm (GA) [3] and Artificial Neural Network (ANN) [4] were made. The initial results were not very encouraging. However, both promise easy parallelisation and modifications in the algorithm or setting up right parameters could improve their performance.

The scheduling problem considered in this thesis is that of static scheduling. This is made possible as the problem and the graph data instances used have sufficient *a priori* information to support static scheduling. It has been shown that for multiprocessor task scheduling problem, when sufficient *a priori* execution profile information is available the static scheduling always performs better than dynamic scheduling [5]. However, the success of the

static scheduling depends on the accuracy or gathering of enough execution profile information. Though, the dynamic scheduling approach gives rise to unwanted execution overhead, it is better suited to handle dynamic variations in the program. A case that can be thought of is when variable simulation time step is demanded by the timing simulation system. Also for many actual parallel programs, it is very difficult to predict actual execution profile information. A static scheduling algorithm that provides some degree of support for dynamic scheduling would be very useful in these circumstances and presents an interesting but challenging future research.

In the present thesis, precedence constraints within the concurrent VLSI timing simulation modules is not considered. This is valid as the simulation system considered is based on true data-flow computation model. However, most of the practical problems involving scheduling parallel programs onto multiprocessor systems encounter precedence constraints. An appropriate formulation of the cost function taking into account of this thus need to be devised. Fortunately, both the RBP using KL heuristic and SA algorithm can be made to work with this revised cost function. It would thus be interesting to note the performance of these two algorithms in such circumstances. Also, the cost function used in this thesis does not consider the effect of multiprocessor architecture, especially hardware communication overhead. In the light of previous work in this area [6], it would thus be desirable to investigate more on this aspect.

And last but not the least, the immediate future research would be to port the CRBP and CSA algorithms on appropriate hardware.

References :

1. Kernighan, B.W. and Lin, S., *An Efficient Heuristic Procedure for Partitioning Graphs*, Bell Sys. Tech. J., Feb. 1970, pp. 291-307.
2. Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P., *Optimisation by Simulated Annealing*, Science, Vol. 220, 1983, pp. 671-680.
3. Booker, L.B., Goldberg, D.E. and Holland, J.F., *Classifier Systems and Genetic Algorithms*, Artificial Intelligence, Vol. 40, 1989, pp. 235-282.
4. Hopfield, J.J. and Tank, D.W., *Neural computation of decisions in optimisation problems*, Biological Cybernetics, Vol. 52, 1985, pp. 141-152.
5. Sarker, V., *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Pitman, London, 1989.
6. Pathak, G.C. and Agarwal, D.P., *Task division and multicomputer systems*, IEEE 5th Intl Conf. on Distributed Computing Systems, 1985, pp. 273-280.

APPENDIX A

Appendix A contains various data and pictorial representations of the graph data instances used in this thesis. The subsequent pages in this appendix carry these data and figures.

Table A.1 Statistics of various concurrent simulation graph instances.

	4x4 multiplier	Frequency locked loop	16x16 multiplier	Vector coder
Vertices	58	68	415	899
Nodes in circuit	168	167	2577	1746
Communication Links	82	139	1102	2034
Average Nodes/Vertex	2.89	2.46	6.02	1.94
Average Degree	1.41	2.04	2.65	2.26

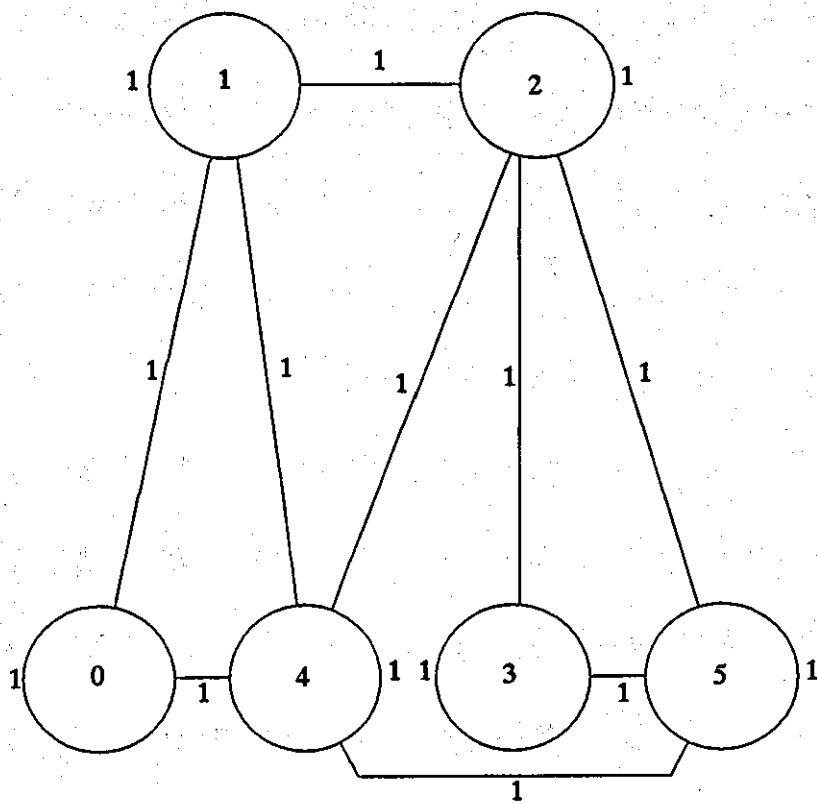


Fig. A.1 Concurrent simulation graph for the synthesised graph instance 6n2c.grf.

Table A.2 Data for the synthesised graph 6n2c.grf. Each line corresponds to the data structure shown in Fig. 3.9

2 1 0 0	1 0 1	4 1 1		
3 1 0 1	0 0 1	2 0 1	4 1 1	
4 1 0 2	1 0 1	3 0 1	4 1 1	5 1 1
2 1 0 3	2 0 1	5 1 1		
4 1 1 4	0 0 1	1 0 1	2 0 1	5 1 1
3 1 1 5	2 0 1	3 0 1	4 1 1	

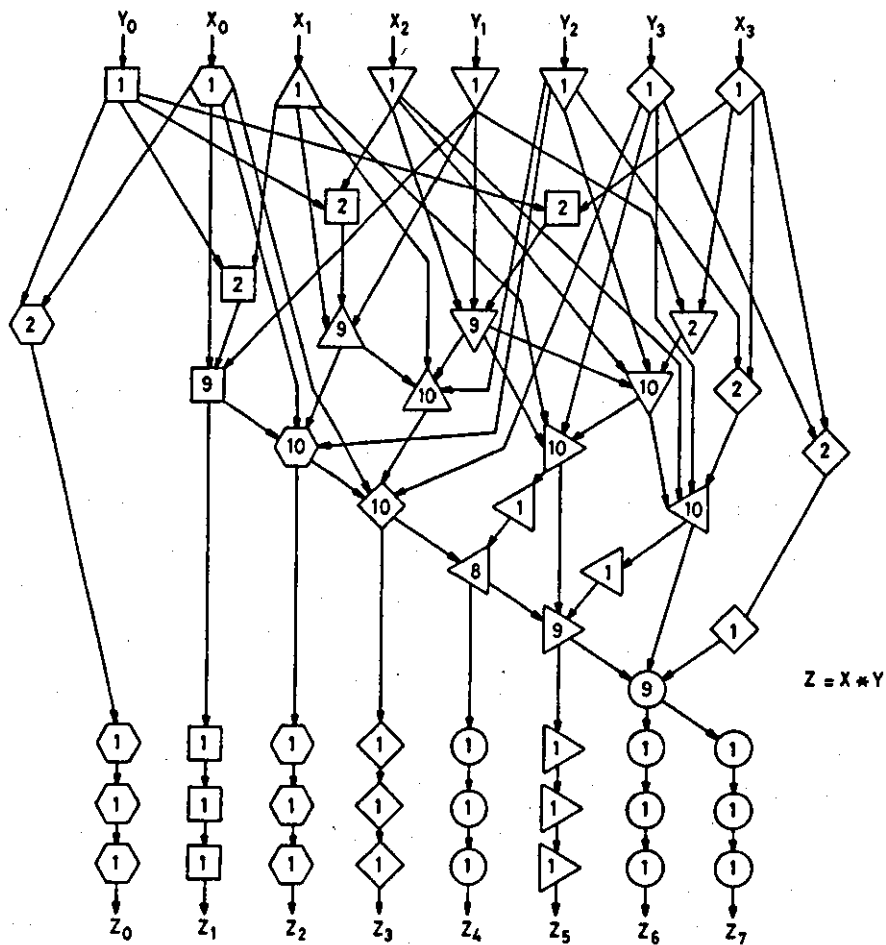


Fig. A.2 Concurrent simulation graph for the 4x4 Multiplier Circuit showing processor allocation at the nodes.

Contd. . .

APPENDIX A

5 1 3 32	29 6 1	28 0 1	36 7 1	54 3 1	56 1 1
2 1 6 33	39 1 1	37 7 1			
3 1 6 34	30 4 1	50 6 1	57 6 1		
3 1 1 35	31 2 1	52 7 1	57 6 1		
3 1 7 36	32 3 1	53 7 1	57 6 1		
3 1 7 37	33 6 1	54 3 1	57 6 1		
3 1 0 38	26 5 1	50 6 1	56 1 1		
2 1 1 39	10 4 1	33 6 1			
3 1 3 40	22 2 1	50 6 1	55 3 1		
2 1 5 41	9 7 1	29 6 1			
3 1 4 42	15 2 1	50 6 1	51 3 1		
2 1 6 43	8 5 1	25 2 1			
2 1 1 44	7 7 1	21 5 1			
0 1 1 45					
2 1 4 46	2 3 1	0 1 1			
2 1 7 47	3 0 1	12 3 1			
2 1 2 48	4 5 1	13 7 1			
2 1 7 49	5 1 1	14 7 1			
4 1 6 50	42 4 1	40 3 1	38 0 1	34 6 1	
4 1 3 51	42 4 1	24 1 1	23 6 1	22 2 1	
4 1 7 52	35 1 1	30 4 1	26 5 1	22 2 1	
4 1 7 53	36 7 1	31 2 1	27 6 1	23 6 1	
4 1 3 54	37 7 1	32 3 1	28 0 1	24 1 1	
4 1 3 55	40 3 1	28 0 1	27 6 1	26 5 1	
4 1 1 56	38 0 1	32 3 1	31 2 1	30 4 1	
4 1 6 57	37 7 1	36 7 1	35 1 1	34 6 1	

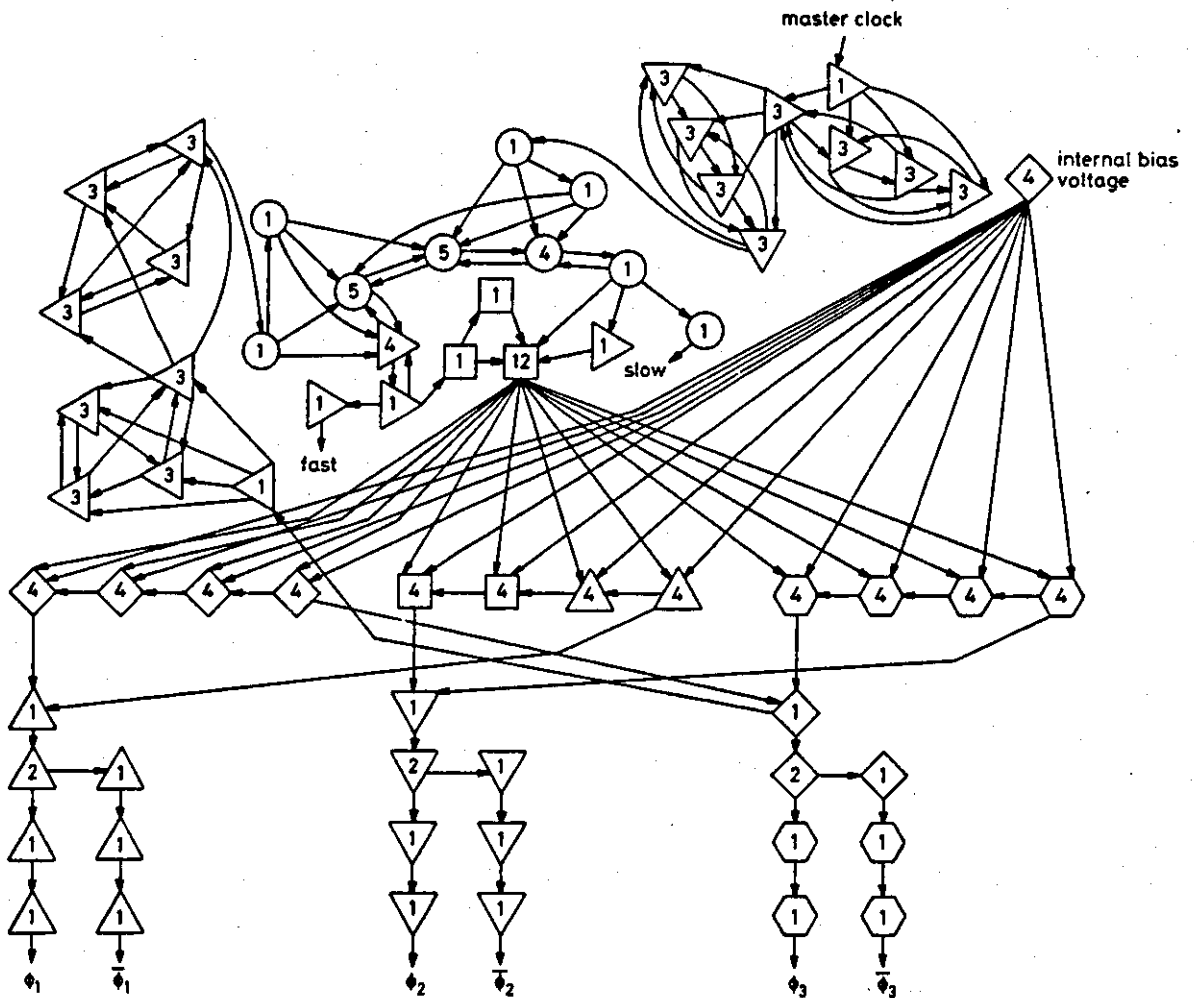


Fig. A.3 Concurrent simulation graph for the Frequency Locked Loop Circuit showing processor allocation at the nodes.

Table A.4 Data for the VLSI simulation graph Frequency Locked Loop
Circuit. Each line corresponds to the data structure of Fig. 3.9

5	3	1	0	60	5	1	59	2	2	52	7	1	57	6	1	58	5	1
4	1	1	1	64	0	1	63	0	1	62	1	1	61	4	1			
5	3	3	2	67	7	1	66	4	2	50	6	1	64	0	1	65	3	1
16	12	0	3	27	6	1	26	5	1	25	2	1	24	1	1	23	6	1
				22	2	1	21	5	1	20	3	1	19	1	1	18	5	1
				17	4	1	16	1	1	14	7	1	43	6	1	44	1	1
				45	1	1												
3	1	5	4	46	4	2	44	1	1	6	1	1						
1	1	1	5	14	7	1												
1	1	1	6	4	5	1												
1	1	7	7	28	0	1												
1	1	5	8	30	4	1												
1	1	7	9	32	3	1												
1	1	4	10	34	6	1												
1	1	2	11	36	7	1												
1	1	3	12	38	0	1												
5	1	7	13	57	6	1	56	1	1	55	3	1	54	3	1	42	4	1
4	1	7	14	48	2	2	43	6	1	5	1	1	3	0	1			
12	4	2	15	27	6	1	26	5	1	25	2	1	24	1	1	23	6	1
				22	2	1	21	5	1	20	3	1	19	1	1	18	5	1
				17	4	1	16	1	1									
4	4	1	16	17	4	1	3	0	1	15	2	1	42	4	1			
4	4	4	17	19	1	1	3	0	1	15	2	1	16	1	1			
4	4	5	18	41	5	1	3	0	1	15	2	1	19	1	1			
4	4	1	19	18	5	1	3	0	1	15	2	1	17	4	1			
4	4	3	20	21	5	1	3	0	1	15	2	1	41	5	1			
4	4	5	21	23	6	1	3	0	1	15	2	1	20	3	1			
4	4	2	22	40	3	1	3	0	1	15	2	1	23	6	1			
4	4	6	23	22	2	1	3	0	1	15	2	1	21	5	1			
4	4	1	24	25	2	1	3	0	1	15	2	1	40	3	1			
4	4	2	25	27	6	1	3	0	1	15	2	1	24	1	1			
4	4	5	26	42	4	1	3	0	1	15	2	1	27	6	1			
4	4	6	27	26	5	1	3	0	1	15	2	1	25	2	1			
2	1	0	28	7	7	1	29	6	1									
3	2	6	29	31	2	1	28	0	1	41	5	1						
2	1	4	30	8	5	1	31	2	1									
2	1	2	31	30	4	1	29	6	1									
2	1	3	32	9	7	1	33	6	1									

Contd...

APPENDIX A

3	2	6	33	35	1	1	32	3	1	40	3	1						
2	1	6	34	10	4	1	35	1	1									
2	1	1	35	34	6	1	33	6	1									
2	1	7	36	11	2	1	37	7	1									
3	2	7	37	39	1	1	36	7	1	42	4	1						
2	1	0	38	12	3	1	39	1	1									
2	1	1	39	38	0	1	37	7	1									
3	1	3	40	33	6	1	24	1	1	22	2	1						
3	1	5	41	29	6	1	20	3	1	18	5	1						
4	1	4	42	37	7	1	16	1	1	13	7	1	26	5	1			
2	1	6	43	3	0	1	14	7	1									
3	1	1	44	45	1	1	3	0	1	4	5	1						
2	1	1	45	3	0	1	44	1	1									
4	4	4	46	47	7	2	4	5	2	52	7	1	53	7	1			
5	5	7	47	49	7	2	46	4	2	51	3	1	52	7	1	53	7	1
4	4	2	48	49	7	2	14	7	2	50	6	1	51	3	1			
5	5	7	49	48	2	2	47	7	2	50	6	1	51	3	1	53	7	1
4	1	6	50	51	3	1	49	7	1	48	2	1	2	3	1			
4	1	3	51	49	7	1	48	2	1	47	7	1	50	6	1			
4	1	7	52	53	7	1	47	7	1	46	4	1	0	1	1			
4	1	7	53	49	7	1	47	7	1	46	4	1	52	7	1			
4	3	3	54	57	6	1	56	1	2	13	7	1	55	3	1			
4	3	3	55	57	6	2	54	3	1	13	7	1	56	1	1			
4	3	1	56	55	3	1	54	3	2	13	7	1	57	6	1			
8	3	6	57	60	5	1	59	2	1	58	5	1	56	1	1	55	3	2
				0	1	1	13	7	1	54	3	1						
4	3	5	58	60	5	2	0	1	1	57	6	1	59	2	1			
4	3	2	59	58	5	1	0	1	2	57	6	1	60	5	1			
4	3	5	60	59	2	1	58	5	2	0	1	1	57	6	1			
4	3	4	61	64	0	1	63	0	2	1	1	1	62	1	1			
4	3	1	62	64	0	2	61	4	1	1	1	1	63	0	1			
4	3	0	63	62	1	1	61	4	2	1	1	1	64	0	1			
8	3	0	64	67	7	1	66	4	1	65	3	1	63	0	1	62	1	2
				2	3	1	1	1	1	61	4	1						
4	3	3	65	67	7	2	2	3	1	64	0	1	66	4	1			
4	3	4	66	65	3	1	2	3	2	64	0	1	67	7	1			
4	3	7	67	66	4	1	65	3	2	2	3	1	64	0	1			

The concurrent VLSI timing simulation graphs for the circuits 16x16 multiplier ckt. (416 nodes) and the vector coder ckt. (900 nodes) are much too large to be placed in figures or tables in this thesis.

APPENDIX B

In Appendix B the proof of Corollary 5.1 used in Chapter 5 is presented. This proof is due to Aarts and Korst (Ref. [1] in Chapter 5.)

Corollary 5.1 : An instance (S, f) of a combinatorial optimisation problem, a suitable neighbourhood structure and the stationary distribution of eq. 5.8 are considered. We can then have,

$$\begin{aligned}\lim_{t \downarrow 0} q_i(t) &\triangleq q_i^* \\ &= \frac{1}{|S_{opt}|} \chi_{(S_{opt})}(i)\end{aligned}$$

where S_{opt} represents the set of globally optimal solutions.

Proof : It is known that,

$$\forall a \leq 0, \quad \lim_{x \downarrow 0} \exp^{\frac{a}{x}} = \begin{cases} 1 & ; \text{ if } a = 0 \\ 0 & ; \text{ otherwise} \end{cases}$$

Using this fact we get,

$$\begin{aligned}
 \lim_{t \downarrow 0} q_i(t) &= \lim_{t \downarrow 0} \frac{\exp\left(\frac{-f(i)}{t}\right)}{\sum_{j \in S} \exp\left(\frac{-f(j)}{t}\right)} \\
 &= \lim_{t \downarrow 0} \frac{\exp\left(\frac{f_{opt} - f(i)}{t}\right)}{\sum_{j \in S} \exp\left(\frac{f_{opt} - f(j)}{t}\right)} \\
 &= \lim_{t \downarrow 0} \frac{1}{\sum_{j \in S} \exp\left(\frac{f_{opt} - f(j)}{t}\right)} \chi_{(S_{opt})}(i) \\
 &\quad + \lim_{t \downarrow 0} \frac{\exp\left(\frac{f_{opt} - f(i)}{t}\right)}{\sum_{j \in S} \exp\left(\frac{f_{opt} - f(j)}{t}\right)} \chi_{(S \setminus S_{opt})}(i) \\
 &= \frac{1}{|S_{opt}|} \chi_{(S_{opt})}(i) + 0,
 \end{aligned}$$

which completes the proof.

TASC

TEACHING AS A CAREER

Teaching offers a challenging and rewarding career

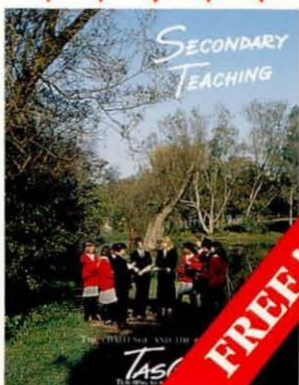
Some 16,000 places are available annually for post-graduate courses for **Secondary Teaching**.

Most are full time and include teaching practice. They cover *all* secondary curriculum subjects but there is a particular need for candidates to train to teach the sciences, maths, technology and modern languages.

An appropriate degree with English language and maths to GCSE grade C, or equivalent, is required.

Why not find out more about this important and satisfying career? Send for the FREE brochure on Secondary Teaching by returning the coupon.

**TASC Publicity Unit,
6th Floor,
Sanctuary Buildings,
Great Smith Street,
London SW1P 3BT**



Postcode:

Address:

Name:

BLOCK CAPITALS PLEASE

**Please send to:
TASC Publicity Unit,
6th Floor,
Sanctuary Buildings,
Great Smith Street,
London SW1P 3BT**

TASC
TEACHING AS A CAREER

TEACHING

OFFERS A

CHALLENGING

AND

REWARDING

CAREER

TEACHING AS A CAREER

TEACHING AS A CAREER

DILLONS
DIRECT
0345 125704

The Bookstore To Your Door

DS
DILLONS
DIRECT
0345 125704

TEACHING AS A CAREER

Parallel Partitioning of Concurrent VLSI Simulation Graphs.

M.A. Rahin and J. Sheild
Department of Electronic & Electrical Engineering.
Loughborough University of Technology.
Loughborough, LE11 3TU. ENGLAND.

The concurrent electrical activity occurring in a VLSI circuit can be evaluated by running a suitable simulation model on a computer. These simulations are usually very time consuming and this has led to the use of parallel computers for acceleration. The underlying electrical model of the simulation may be represented as a concurrent data flow graph, which when optimally partitioned and assigned to the participating processors of a multiprocessor system ensure maximum achievable acceleration. In this paper two parallel graph partitioning algorithms (CKL & CSA) are reported and their simulation results are compared.

1. Introduction

Electrical circuit simulation is an important part of the VLSI design process in CAD as tentative designs can be confidently verified before expensive manufacture. A timing simulator offers better accuracy than a switch level logic simulator but is not as computation intensive as detailed analysis like SPICE [1]. However, the increasing growth of VLSI design complexities demand further reductions in simulation time whilst retaining simulation accuracy. Consequently concurrent versions of timing simulators like CEMU [2] have been developed for possible execution on a multiprocessor system.

A timing simulator generally models the electronic circuit as a weighted graph, where a vertex represents a set of capacitive nodes of the circuit which share a common bidirectional voltage controlled current source. The vertex weights in the graph represent the number of circuit nodes that are grouped together to form the vertices and so represent the amount of simulated electrical activity of the set of circuit nodes. The edges in the graph represents the discrete voltage values that are passed between groups of circuit nodes along the interconnection at any simulation time step. This leads to a simple concurrent implementation on a multiprocessor system as the rich inherent data flow representation of the graph model can be easily exploited. The overall task here is to optimally partition the circuit to be simulated into different sub-units which are then assigned to different processors. This partitioning and assignment procedure is however, quite sophisticated and closely resembles VLSI cell placement and layout problems. Problems of this kind fall into the class of combinatorial optimisation and have been identified as NP-Hard [3].

Heuristic solutions for NP-Hard problems have been favoured over exhaustive search algorithms for their ability to provide approximate near optimal solutions in polynomial time. Iterative improvement is one of the more common types of heuristic algorithms in use. In this paper, a parallel implementation of such an algorithm based on the 2-way partitioning procedure due to Kernighan and Lin [4] is reported. The results obtained were compared with a concurrent version of the Simulated Annealing [5] algorithm and was found favourable for smaller multiprocessor configurations.

2. Problem description

For the partitioning of a VLSI logic simulation graph and subsequent assignment on to a set of P processors of a homogeneous multiprocessor system, we assume a perfect synchronous concurrent data-flow representation in the input graph and consequently there are no precedence relations. Mathematically, the graph can thus be represented by a weighted and undirected network flow graph $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_n\}$ are the N weighted vertices of the graph and $E = \{e_{i,j}, i, j = 1..n\}$ are the weighted edges between the vertices representing the amount of data-flow between the vertices. For any positive integer K , a K -way partitioning of G is a set of non-empty, disjoint sub-sets (or blocks) of G , g_1, g_2, \dots, g_k such that

$$\bigcup_{x=1}^k g_x = G$$

The 'cutset' of the partition is the sum of all the weighted edges with vertices in more than one sub-set and accounts for the communication cost in the final task scheduling. The 'load-imbalance' is the maximum difference between the total weights of any two sub-sets, thus accounting for the completion time of the final scheduling. The optimal partition cost is the partition where assignment of tasks to the P sets of processors results in the minimum data communication between the processors and minimal load imbalance.

3. 2-way and K -way partitioning algorithm

In [4], Kernighan and Lin described a heuristic procedure for graph partitioning. Their algorithm dealt with the problem of partitioning a graph with N vertices, where N is even, into two disjoint sub-sets of $N/2$ vertices each. The algorithm functions by successively choosing all possible pairs of vertices, taking one from each sub-set, and keeping aside the pair which if swapped would produce the best cost improvement. This procedure is repeated for the remaining pairs, keeping a record of the point when the best cost improvement is seen, until all $N/2$ pairs have been set aside. Those pairs set aside preceding and including the point when the best cost improvement was recorded are then actually swapped to produce a new starting partition. The whole procedure or pass is then repeated again from this new partition. The algorithm terminates when no cost improvement can be generated from swapping any pairs. The running time of each pass of the algorithm is $O(N^2 \log N)$.

However, for a graph with an odd number of vertices, non-uniform vertex weights, edge weights and requiring load balancing in the final partition some modifications are necessary in the original Kernighan-Lin algorithm. In the modified algorithm, instead of swapping two chosen vertices, a single vertex from a randomly chosen sub-set that would give the best cost improvement if moved across, moved over to the other side and kept aside. This is repeated until all the vertices have been moved across and then as before, the set of moves that produced the minimum partition cost is selected, the two sub-sets updated accordingly and used as the starting partition for the next pass.

Kernighan & Lin's 2-way partitioning procedure can easily be extended to produce K number

of partitions. This uses the recursive 2-way partitioning algorithm until the desired number of sub-sets, which must be an integer power of 2, is obtained. This method promises good run-time behaviour but, as mentioned in [4, 6] can produce a bad result in the first partition (level 0 partition) which may bias the second and so on. Also the level 0 partition will try to minimise the number of connections between the first 2 blocks thus tending to maximise the connections inside these blocks, making it harder to obtain good partitions thereafter. This would suggest that a good level 0 partition will always result in a bad final partition (for $K > 2$). However, experimental results for smaller partition sizes were found favourable and thus needed further exploration.

4. Concurrent K-way partitioning (CKL)

The binary dissection procedure lends itself to parallel implementation. Except for the first partition (level 0), partitioning at all other levels can be carried out independently of each other as can be visualised from the binary tree like structure of the overall dissection strategy as shown in Fig. 1. These independent partitioning procedures can be assigned to different processors and can be run concurrently. A breadth-first partitioning would then take place. However, for a P-processor system, maximum processor utilisation is achieved only at the bottom level (leaf level).

A modification that utilises all the processor resources and improves the quality of the final solution is made. A multiple dissection operation, limited by the available number of processors, is carried out in each partitioning stage of every level and only the one giving the best result is accepted. The best partitions thus generated are moved forward to the next level and so on. So for an 8 processor system, at level 0 we have 8 separate partitioning operations on the same input graph all running in parallel and only the best resulting partitions are accepted for level 1 partitioning, whereby at level 1, two concurrent partitioning operations are required because there are now two input sub-graphs. Out of a total of 8 processors, 4 processors can thus be allocated for each of the two partitioning stages. At the bottom level (level 2) 2 processors can thus be allocated for each of the 4 partitioning stages. This modification provides a fast descent and guarantees a better solution that could be achieved otherwise.

The run-time requirement of each pass of Kernighan-Lin's 2-way partitioning algorithm is $O(N^2 \log N)$. The number of passes required for the convergence has always been found to be between 2 and 4 and thus is not strongly dependent on the size of the graph. Again, by employing better search algorithms for selecting the candidate vertex for a move over to the other side, a lower bound run-time $O(N^2)$ can be obtained. For a K-way partition using the binary dissection method the run-time requirement becomes $O(N^2 \log K)$. In the parallel implementation, minimum processor utilisation is obtained at the top level and it increases by a factor of 2 as the partitioning progresses from one level to the other. This provides a speed-up factor of $\log P$, giving run-time complexity of $O(N^2)$, as $P = K$.

5. Concurrent Simulated Annealing (CSA)

Simulated Annealing (SA) [5] has been found to be a powerful and robust tool for the solution of many different types of difficult NP-Hard combinatorial optimisation problems. While this technique provides good quality solutions, the computation time requirement is very high. Several

accelerating techniques using multiprocessors have been reported [7, 8,9].

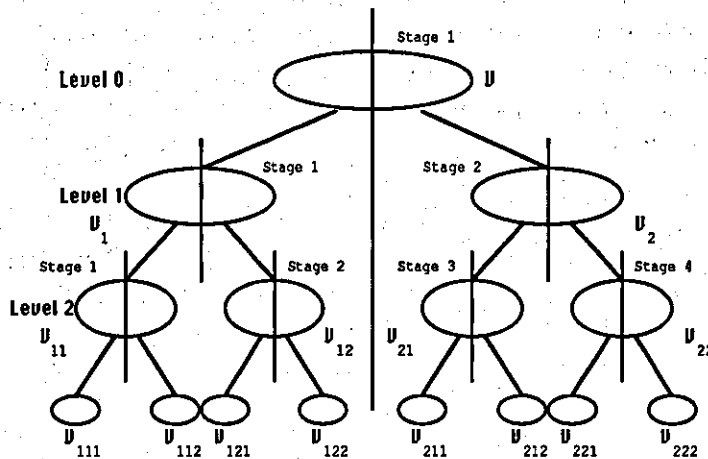


Figure 1. Overall 8-way dissection structure.

Our concurrent implementation of the SA algorithm (CSA) is based on the parallel move approach due to Kravitz & Rutenbar [7], which however has the problem of interaction between parallel moves giving rise to possible erroneous solutions. Unlike Kravitz & Rutenbar, where they have used the 'simplest serialisable sub-set' of the parallel move set obtained by attempting many moves in parallel, and by executing only the move that is accepted first (cost improving move) and aborting all other, our CSA algorithm works with a move set consisting of an optimal number of non-interacting parallel moves. This eliminates the error due to interacting parallel moves.

The annealing schedule adopted is a simple one and is as follows: a move acceptance probability of 0.999 at starting maximum temperature and 0.001 at final minimum temperature obtained by 100 geometric reductions in temperature. The total number of moves allowed at each temperature step is dependent of the problem size and is given by $N_{\text{moves}} = M \cdot N \cdot (P - 1)$ where, the constant M is defined in [10] and other notations have their usual meanings. A value of 2 for M was used in the simulation runs.

Table 1. Statistics of the four chosen concurrent VLSI simulation graph instances.

	4x4 multiplier	Frequency locked loop	16x16 multiplier	Vector coder
Vertices	58	68	415	899
Nodes in circuit	168	167	2577	1746
Communication Links	82	139	1102	2034
Average Nodes/Vertex	2.86	2.46	6.02	2.34
Average Degree	2.83	4.09	5.31	4.53

6. Results and Discussion

Parallel programs to implement the concurrent Kernighan-Lin (CKL) and concurrent Simulated Annealing (CSA) algorithms were written for the Intel iPSC/2 hypercube, a message-passing, distributed memory multiprocessor system. The programs were written in C and Pascal languages and tested with an Intel iPSC/2 simulator on a SUN 3 minicomputer. As the programs were run under a simulated environment the true run-times could not be perfectly ascertained and only the simulation results obtained are reported.

Four examples of VLSI logic simulation graphs were used to evaluate the concurrent heuristic algorithms. These were chosen on the variety of their size, complexity and functional behaviour. The sizes and the statistics of these four logic simulation graphs are given in Table. 1.

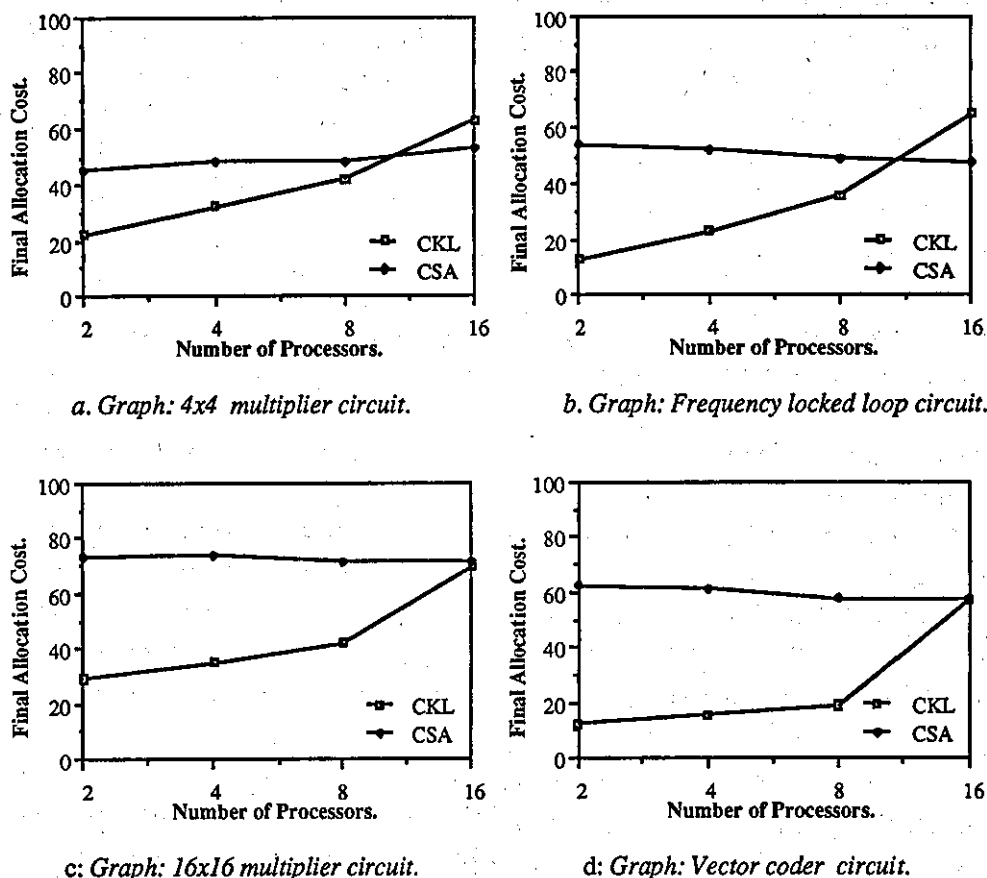


Fig. 2 Graphs showing the performance comparison between algorithms CKL and CSA for different processor configurations.

Each of the two programs (CKL & CSA) were performed 50 times on each of the four graph instances. Processor configurations with 2, 4, 8 and 16 processors were considered. The average partition costs for each graph instance and each system configuration were noted. The partition cost was then expressed as a percentage of the average cost of the random starting partitions. The results obtained are presented in Fig. 2.

The graphs in Fig. 2 reveal that the CKL algorithm is better than CSA for small processor configurations. However, a gradual decline in the performance of CKL is observed as the number of processors (same as the number of partitions) is increased and asymptotically fares worse compared to CSA for larger configurations (≥ 16). This observation is in line with the disadvantage of the recursive binary dissection method for K-way partitioning as discussed in [4, 6]. The intrinsic disadvantage of the recursive binary dissection method gradually becomes more noticeable as the depth of partition increases.

On the other hand CSA performed more or less consistently throughout. However, for the two large graph instances the results are comparatively inferior than the other two. This is perhaps due to the behaviour and nature of the two large graph instances. For the frequency locked loop circuit, a steady and gradual improvement in the performance of CSA is observed, which however is absent in the other cases. For the 4x4 multiplier circuit the performance curve has a uneven profile possibly due to the random behaviour of the SA algorithm coupled with some long sequential data flow modules present in the graph.

A fairly adequate number of moves per temperature step ($M = 2$) were allowed for CSA. However, if more generous number of moves were permitted, CSA might have produced much better and improved solutions. The very long computing time demand of CSA however, precluded us to permit more moves. Though an exact run time could not be ascertained for reasons described earlier, CSA was always found to be approximately 100 times slower than CKL in their simulation runs. CKL is thus more suitable than CSA for smaller system configuration (≤ 8 processors) by virtue of its speed and quality of result. One of the ways to improve the quality of results obtained from CKL is to post process the partitions so that they become mutually and pairwise optimal. Essentially pairwise optimality is a necessary condition for global optimisation. This can be achieved by repetitively selecting any two partitions and moving some chosen vertices from one to another, so that the cost of partition between these two is minimised.

References :

- [1]. D.M. Lewis, 'Hardware Accelerators for Timing Simulation of VLSI Digital Circuits,' *IEEE Trans. CAD*, Vol. 7, No. 11, pp. 1134-1149, Nov 1988.
- [2]. B. Ackland, S. R. Ahuja, T.L. Lindstorm & D. J. Romero, 'CEMU - A concurrent timing simulator,' *Proc. IEEE Intl. Conf. CAD*, 1985.
- [3]. M. R. Garey & D. S. Johnson, 'Computers and Intractability - A guide to theory of NP-Completeness,' W.H. Freeman Company, San Fransisco, USA, 1979.
- [4]. B.W. Kernighan and S.Lin, 'An efficient heuristic procedure for partitioning graphs,' *Bell Syst. Tech. J.*, Vol. 49, pp. 291-307, Feb. 1970.
- [5]. S.Kirkpatrick, C.D. Gelatt and M.P. Vecchi, 'optimisation by Simulated Annealing,' *Science*, Vol. 220, pp. 671-680, 1983.
- [6]. L.A. Sanchis, 'Multiple-Way Network Partitioning,' *IEEE Trans. Comp.*, Vol. 38, No. 1, pp. 62-81, Jan 1989.
- [7]. S.A. Kravitz and R.A. Rutenbar, 'Multiprocessor-based placement by simulated annealing,' *Proc. of Decisions and Control*, June 1986.
- [8]. F. Dorem-Rogers, S. Kirkpatrick and V.A. Norton, 'Parallel VLSI Placement by Simulated Annealing,' *IBM J. Res. & Dev.*, May 1988.
- [9]. A. Cosotto, F. Romero and A. Sangiovanni-Vincentelli, 'A parallel Simulated Annealing Algorithm for the Placement of Macro-Cells,' *IEEE Trans. CAD*, Vol. CAD-6, No. 5, pp. 838-847, Sept 1987.
- [10]. J. Sheild, 'Partitioning concurrent VLSI simulation programs onto a multiprocessor by simulated annealing,' *IEE Proc.*, Vol. 134, Pt. E, No. 1, pp. 24-30, Jan 1987.

