Load Balancing in Parallel and Distributed Systems

A thesis by: David Sinclair B.E., M.Sc.

Supervisors: Mr. Mícheál Ó hÉigeartaigh

Prof. Michael Ryan

Submitted to

Dublin City University

Computer Applications

for the degree of

Doctor of Philosophy

September 1993

Declaration: No portion of this work has been submitted in support of an application for another degree or qualification in the Dublin City University or any other University or Institute of Learning.

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: David Sinclair Date: 30/9/93

Candidate

Date: 30/9/93

Acknowledgements

There are two reasons why I have been looking forward to writing this part of the thesis. The first reason is that typically this is one of the last sections an author has to write. That means that I can see a light at the end of the tunnel which I know is not an onrushing train!

The second reason, and to me the most important reason, is that I can publicly acknowledge those people, whose support and encouragement, have made this thesis possible. I would like to thank my supervisors, Mr. Mícheál Ó hÉigeartaigh and Prof. Michael Ryan, for all their help. Mícheál's willingness to accept me as a student, his open door, his careful criticism and his guiding hand has made the journey down this tunnel very pleasurable. When you are immersed in a subject it is very easy to get absorbed by intricacies, to see the leaf and not the tree, let alone the forest. My conversations with Prof. Michael Ryan always reminded me of the forest and taught me a simple, but invaluable lesson. Namely, that there is an elegance to simplicity, and a simplicity to elegance.

To all the staff and postgrads in the School of Computer Applications here in D.C.U., thank you for your support, assistance, coffee break discussions and most of all your friendship.

Life does not end at the gates of D.C.U., and I am fortunate to have many friends. Among those I am especially grateful to the hard core gang: Brian, Colin, Colm, Derek, Kevin, Niall, Paul, Stephen,

Tony, Trevor, their alter-egos and better halves. I have known them for more years than they care to remember, and they have helped me in more ways than they know. A special vote of thanks to Kevin who proof-read this thesis and showed me which obvious points were not obvious.

The seeds of investigation were planted and nurtured by my parents, Noreen and George, who have always allowed my imagination to fly, even when they did not understand it. On those occasions when my flight failed, they were always there to soften the landing. For all you have done, thank you Mom and Dad. To Jennifer, Joanne, George and Scott, my sisters and brothers, and Oscar, the dog, thank you for all your love and support.

And finally I would like to dedicate this thesis to my ever loving wife, Edel. I had often thought of returning to university to pursue the ideas in my head, but never had the courage. The first time Edel heard me muse on the idea, she sat me down, and told me to write an application letter. Without her love, support and encouragement, this thesis would never have been possible. So this thesis is from me to you, Edel, with all my love.

Table of Contents

1.0 Introduction 1

1.1 The Load Balancing Problem 1

1.2 The Motivation for Solving the Load Balancing Problem 4

1.3 Overview of Thesis 7

2.0 Review of Research into Load Balancing 9

2.1 Summary 13

3.0 Static Communications Load Balancing 14

3.1 The Elastic Force Algorithm 14

3.2 Definitions 15

3.2.1 A Task 15

3.2.2 A Processing Node 16

3.2.3 The Communications Matrix ,[C] 16

3.2.4 The Allocation Matrix, [A] 16

3.2.5 The Modified Communications Matrix, [Ca] 17

3.2.6 The Force Matrix, [F] 17

3.2.7 Total Stored Energy, E_S 18

3.2.8 Inertia of a Task, I 18

3.2.9 Task Move Gain, M 19

3.2.10 Task Swap Gain, S 19

3.3 The Algorithm 20

3.4 Maximum (k-1) Sum Algorithm for Equal Computational Loads 23

3.4.1 Example 24

3.5 The General Maximum (k-1) Sum Algorithm 27

3.5.1 Maximum (k-1) Sum Algorithm with Computational Loads 33

3.5.2 Example 36

3.5.3 Experimental Evaluation of Maximum (k-1) Sum Algorithm 45

4.0 Non-Symmetric Formulation of Task Allocation Problem 49

4.1 Definitions 50

4.1.1 A Task 50

4.1.2 A Program 50

4.1.3 A Processing Node 50

4.1.4 Precedence Level 51

4.2 Formulation of Non-Symmetric Mathematical Programming Model 52

4.2.1 Relaxing Quadratic Constraints 58

4.2.2 Calculation of Bounding Constant C 62

4.3 Sciconic Implementation 64

4.3.1 Problem formulation in Sciconic 66

4.3.2 Report Writer 67

5.0 Pseudo-Dynamic Load Balancing 68

5.1 Definitions 70

5.1.1 A Task 70

5.1.2 A Program 70

5.1.3 A Processing Node 70

5.1.4 A Program Graph 71

5.1.5 Precedence Level 72

5.1.6 A Network Graph 74

5.2 The Pseudo-Dynamic Allocation Heuristic 74

5.2.1 The Computational Load Component, C_1 79

5.2.2 The Communications Load Component, C_c 80

5.2.3 The Precedence Component, C_p 81

5.2.4 The Global Scheduling Table (GST) 81

5.3 Local Schedulers 83

5.4 The Pseudo-Dynamic Load Balancing Algorithm 84

6.0 Experimental Evaluation of Pseudo-Dynamic Load Balancing 89

6.1 Experimental Test Results 92

6.1.1 Pseudo-Dynamic Load Balancing vs. Relaxed Non-Symmetric Mathematical Formulation 92

6.1.2 Pseudo-Dynamic Load Balancing vs. Simulated Annealing and Tabu Search 97

7.0 Worst Case Analysis of Pseudo-Dynamic Load Balancing 101

7.1 Worst Case Program Graph Structure 102

7.2 Optimal Allocation for Worst Case Structure 106

7.3 Worst Case Ratio, R, for Pseudo-Dynamic Load Balancing 110

7.4 Estimating the Worst Case Ration, R^{PD}, for a Given Graph 111

7.5 Worst Case Ratio, R^{PD}, as a Function of the Communications to Computations Ratio, a 113

7.6 Asymptotic Bounds on Worst Case Ratio, R^{PD} 114

8.0 Conclusions 116

8.1 Goals 116

8.2 Results and Achievements 117

8.3 Topics for further Research 120

8.3.1 Enhancements to the Pseudo-Dynamic Load Balancing Algorithm 120

8.3.2 Distributing the Pseudo-Dynamic Load Balancing Algorithm 121

8.4 Concluding Remarks 123

REFERENCES 127

APPENDIX A Maximum (k-1) Sum Algorithm Source Code A1

APPENDIX B: Sciconics Implementation of the Relaxed Non-Symmetric Formulation of the Task Allocation Problem B1

B.1 Introduction B1

B.1.1 Suffices B1

B.1.2 External Values B1

B.1.3 Internal Values B2

B.1.4 Declarations B2

B.1.5 Variables B3

B.1.6 Problem B3

B.1.7 Elements B6

B.1.8 Functions B6

B.2 MGG Problem Formulation B8

B.3 User Supplied Report Routime B13

B.4 Problem Data File B21

B.5 Sciconic Run Streams B25

APPENDIX C: Pseudo-Dynamic Load Balancing algorithm Source Code C1

APPENDIX D: Simulated Annealing Source Code D1

APPENDIX E: Tabu Search Source Code E1

Abstract

Title: Load Balancing in Parallel and Distributed Systems

Author: David Sinclair

Two major barriers prevent the widespread, common usage of parallel and distributed computing systems:

(1) A language which expresses parallelism without reference to the underlying hardware configuration.

(2) A user invisible method for effectively distributing the tasks that form the parallel/distributed program among the available processing nodes. This is known as the load balancing problem.

This thesis examines the load balancing problem. This problem of allocating *n* inter-communicating tasks among *m* processing nodes is formulated as a non-symmetric mathematical programming problem, which minimises the makespan, and is shown to be quadratic and discrete. A novel relaxation is developed which exploits the discrete nature of the problem, and this relaxed formulation is used to generate strong upper bounds.

Two novel heuristic algorithms are proposed. A static load balancing algorithm, the Maximum (k-1) Sum algorithm, is developed for maximising the throughput of tasks in a parallel or distributed system. This algorithm is compared with recently published results. An on-line load balancing algorithm, the Pseudo-Dynamic Load Balancing algorithm, is developed from the mathematical analysis of the problem. This algorithm seeks to minimise the makespan of a program, and is compared with standard combinatorial optimisation techniques, such as Simulated Annealing and Tabu Search, as well as the upper bounds set by the relaxed non-symmetric mathematical formulation. Both of these new algorithms are shown to provide efficient allocations of *n* tasks among *m* processing nodes.

Finally the Pseudo-Dynamic Load Balancing algorithm is analysed to determine its worst case scheduling ratio, R^{PD}, and the conditions under which this worst case occurs.

1.0 Introduction

1.1 The Load Balancing Problem

In the **concurrent programming** model of parallel and distributed processing a parallel or distributed program consists of a set of n inter-communicating tasks. Each one of the n tasks may communicate with any of the other $(n-1)$ tasks. The amount of communications between task i and task j is characterised by $c_{i,j}$, the communications load between task i and task j. In addition, task i is also characterised by a computational length l_i that represents the time taken to process its data. Tasks may also be bound by precedence constraints so that task i may not execute until task k is completed. Therefore the program can be represented as a program graph in which a node represents a task and the weight of the node represents the computational length of the task. The communications between tasks i and j is represented by an edge between the nodes representing tasks i and j, with the weight of the edge representing the amount of communications between the two tasks. Precedence constraints may be represented an edge of zero weight. If task i must be completed before task j, this may be represented by a zero weight from node i to node j.

The parallel or distributed computing system, on which this program will run, can be characterised as a set of processing nodes connected together by communications links in a given topology. Each processing node can be characterised by a relative processing speed and the communications distance from the processing nodes to which it

is connected. The communications distance between processing nodes *i* and *j* measures the time it takes a unit message to be transferred from processing node *i* to processing node *j*. It represents the speed of the communications medium between processing nodes *i* and *j*, as well as the physical distance between processing nodes *i* and *j*. A graph can be used to represent the parallel or distributed computing system. Each node in the network graph represents a processing node, where the weight of the node represents the relative speed of the node and the weight of the edge between nodes *i* and *j* represents the communications distance between processing nodes *i* and *j*.

Typically data transferred between two tasks on the same processing node occurs through memory, while data transferred between two tasks on different nodes is through one or more communications links. Since the access time of memory is negligible in comparison to the access time of an inter-node communications medium, e.g. LAN, high speed serial links, etc., data transfer between two tasks on the same node is considered to be instantaneous. The communications distance from a node to itself is zero.

The **load balancing problem** is that of allocating a set of *n* inter-communicating tasks among *m* processing nodes, arranged in a given topology, in order to minimise or maximise some criteria. The most common criteria are:

(a) The minimisation of the maximum task completion time, **makespan**, C_{max}.

(b) **The minimisation of the sum of the task completion times.**

(c) The minimisation of the sum of the inter-task communications.

Let $d_{i,j}$ equal 1 if task i is allocated to processing node j, and 0 otherwise. If the completion time for task i is C_i, the time at which task i starts is y_i, and we assume, without loss of generality, that task 1 is the root task and task n is the terminal task, then the most common criteria are defined as:

(a) **The minimisation of the maximum completion time, makespan, C_{max}.**

$$\min\{C_{\max}\} = \min\left\{\max_{1 \le i \le n} C_i\right\}$$

(b) **The minimisation of the sum of the completion times.**

$$\min\left\{\sum_{i=1}^{n} C_i\right\}$$

(c) The minimisation of the sum of the inter-task communications.

$$\min\left(\sum_{i,j,k} |(d_{i,k} - d_{j,k})| c_{i,j}\right)$$

In addition to these common criteria, this thesis will use another criterion:

(d) The minimisation of the total run time of the program as defined by the difference between the start times of the first and last task.

$$\min\{y_n - y_1\}$$

Criteria (a) and (d) are equivalent, since they only differ by the computational length of the last task, task n.

1.2 The Motivation for Solving the Load Balancing Problem

Since the late 1970's, it became apparent to some researchers that current, and envisaged technology, could not solve the "grand challenges" of computing, nor meet the users' growing demands for increased performance at an affordable price. These researchers proposed models of computing in which many processors co-operatively solved a problem. Since that time a lot has happened, and a lot has not. Parallel and distributed computers still remain behind the doors of research labs, and outside the price range of most users. These machines struggle with the "grand challenges". At the same time basic technology has advanced at an unbelievable rate, causing researchers to revise the predicted growth in performance approximately every five years. Standard sequential models of computing have increased in computing power and reduced in size and cost, providing most users with the performance they desire at a price they can afford. However, clouds are appearing on the horizon. As users' expectations continue

to grow many researchers are predicting an abrupt halt in the advancement of technology that underlies modern computers. This technology will meet fundamental physical limits imposed by quantum mechanical effects, such as quantum tunnelling, as the geometry of the integrated circuits continues to shrink. Unless new technologies are discovered, parallel and distributed computers will have to leave their labs and enter the mainstream of computing. Given the obvious performance/price advantages and users' growing use of computer networks, why has parallel and distributed computing not yet become widespread among the general computing community?

Originally many problems were associated with parallel and distributed computing, such as synchronisation, exclusion etc., but most of these have been solved. What parallel and distributed computing needs now is a revolution, similar to that which occurred for personnel computers, which will make parallel and distributed computers accessible to the general user and developer. One of the popular models of parallel and distributed computing among researchers is **concurrent programming**, where the program consists of a set of communicating sequential tasks. Two central issues still prevent this model of parallel and distributed computing from entering into the mainstream of computing. These are:

(1) A language which expresses the fundamental parallelism of a problem without reference to the underlying computing system.

(2) An effective method for automatically partitioning the tasks which form the parallel or distributed program among the available processors.

The solution to the first problem, the expression of parallelism, requires the software developer to change his/her thought processes, during the design of a program, from the artificial sequential environment of existing sequential programming languages to an environment similar to the world in which we live, where actions can occur simultaneously. Object oriented programming is providing a useful intermediate stage in enabling this change in the thought processes of the designer. Object oriented program designers no longer think of a program as a series of sequential actions, but as a set of inter-related objects. The next step for the software designers is to think of the program as a set of cooperative concurrent processes. For a parallel or distributed programming language to be commercially successful its structure should help reveal the fundamental parallelism in a problem without requiring developers to make a radical change in their thought processes during design.

The second problem, the load **balancing problem**, is the subject of this thesis. Load balancing is the means by which the expressed parallelism is exploited to improve the performance of the program, and/or computing environment, usually in terms of reduced response time or increased throughput. As well as being a fundamental problem in parallel and distributed processing, the load balancing problem belongs to a class of problem which is currently believed to be

intractable. The problem of determining if there exists an allocation of *n* arbitrarily inter-communicating tasks, constrained by precedence relationships, to an arbitrarily interconnected network of *m* processing nodes, which meets a given deadline is an NP-complete problem [1]. The problem of minimising the makespan, of a set of tasks, where any task can execute on any processing node and is allowed to preempt another task, is NP-complete even when the number of processing nodes is limited to two [2].

This thesis examines the load balancing problem and presents new algorithms for effective partitioning a set of *n* inter-communicating tasks among *m* processing nodes.

1.3 Overview of Thesis

Section 2 reviews the research into load balancing in parallel and distributed computing. Section 3 presents two new algorithms, the **Elastic Force algorithm** and **Maximum (k-1) Sum algorithm**, for static load balancing. The Maximum (k-1) Sum algorithm is compared with recent results. Section 4 presents a non-symmetric mathematical formulation of the task allocation problem. This formulation shows that the task allocation problem is quadratic as well as discrete. A new relaxation method is developed to relax the problem into a mixed integer linear programming (MILP) problem. The relaxed formulation is then implemented in the Sciconics mathematical programming package. Section 5 examines the non-symmetric formulation to develop an

on-line heuristic technique called **Pseudo-Dynamic Load Balancing**. Section 6 evaluates the quality of the allocations produced by Pseudo-Dynamic Load Balancing against the relaxed mathematical formulation and standard combinatorial optimisation techniques, such as Simulated Annealing and Tabu Search. Section 7 examines the Pseudo-Dynamic Load Balancing algorithm to determine under which conditions the Pseudo-Dynamic Load Balancing algorithm produces low quality allocations. Then an upper bound on the worst case scheduling ratio for Pseudo-Dynamic Load Balancing is derived. Section 8 presents the conclusions of the thesis and recommends areas for future research.

2.0 Review of Research into Load Balancing

There have been many different methods used to find a solution to the load balancing problem. These can be generally classified using the taxonomy presented by Casavant and Kuhl [3] (figure 1). This is a hierarchical classification, but the two most important distinctions are those between **static** and **dynamic** techniques and between **optimal** and **sub-optimal** techniques.

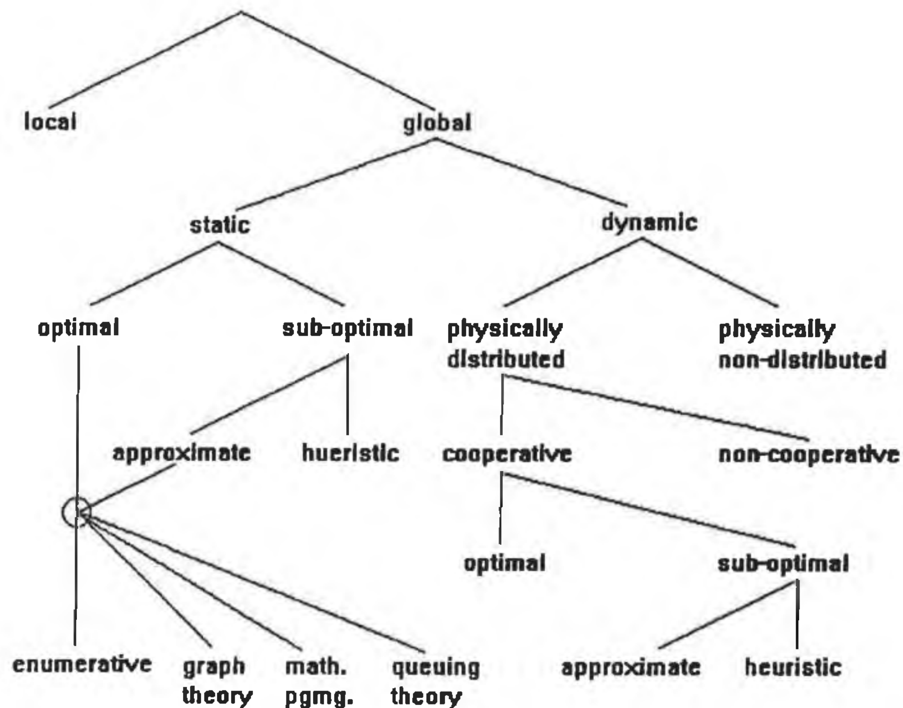


Figure 1: Casavant and Kuhl's Taxonomy of Load Balancing Characteristics

Static load balancing methods commit the allocation of tasks to processing nodes when the program is compiled. The information regarding the tasks which comprise the program, determined by an analysis of the program, and information regarding the hardware

configuration are combined to determine the allocation of tasks to processing nodes. All the information about the program and the parallel or distributed computing system on which the program will be executed is known, ab initio. Any changes to the hardware configuration require the program to be recompiled in order to determine the best task to processing node allocation. Static load balancing is also referred to as *deterministic scheduling* [4] or *task scheduling* [5]. **Dynamic** load balancing methods are on-line methods which use very little *a priori* knowledge of the program, and leave the decision of where to allocate a task to the run-time system when the task becomes available. This decision can be reached by the processing nodes agreeing on a task allocation, *cooperative dynamic scheduling*, or by independent decisions by individual processing nodes, *noncooperative dynamic scheduling*.

Given that the load balancing problem is NP-complete in the general sense, approaches to solving the problem are either based on (a) restricting the problem definition and finding an optimal solution for the restricted problem in polynomial time, or (b) finding a near-optimal solution to the general problem.

Optimal methods allocate tasks to processing nodes based on some condition of optimality, as determined by an objective function such as minimising the maximum completion time of any task, the **makespan**, or maximising resource utilisation etc. These optimal methods either perform an enumerated search of the solution space, or use graph theoretic, mathematical programming or queuing theoretic approaches. Many optimal methods apply restrictions to the problem

formulation so that the problem is no longer NP-complete and can be solved in polynomial time. Hu's algorithm [6], for example, is a graph theoretic method which finds the optimal solution to the load balancing problem if all tasks have equal computational length and the program graph is a tree structure. Coffman and Graham devised an algorithm similar to Hu's algorithm which finds the optimal allocation for an arbitrary program graph of equal computational length tasks on a 2 processor system [7]. Bokhari has extended this to graph theoretic methods for finding the optimal allocation of arbitrary program graphs with tasks of different computational length on a 2 processor system [8].

In essence, the load balancing problem is a discrete optimisation problem. Therefore any mathematical based solution must either transform the problem into a linear, or restricted quadratic, optimisation problem after applying some simplifying restrictions to the initial problem, or restrict the size of the feasible solution space.

When the tasks have equal computational loads and do not communicate with each other, Epstein, Wilamowsky and Dickman [9] showed that minimising the makespan, which is an integer linear problem (ILP), can be transformed into a classical assignment problem which is linear. Gaudioso and Legato [10] also showed how such "open-shop" models can be formulated as linear programming problems. Lawler and Labetoulle [11] showed that even when tasks where allowed to have different computational loads, and preemption is permitted, the minimisation of the makespan can be formulated as a linear

program, and an upper bound on the number of preemptions in the optimal schedule can be determined. Billionnet, Costa and Sutter [12] proposed a relaxed solution for the case of no precedence or sequencing constraints and when the objective function is the minimisation of the sum of the inter-node communications plus the sum of the task execution times. Their approach was based on solving the Legrangean dual of the task allocation problem and then proving that no duality gap existed.

Guiding the search of the solution space or reducing the size of the solution space is another approach to the problem. Barnes, Vannelli and Walker [13] developed a heuristic to guide their search for an optimally balanced communications load by estimating a linear cost matrix from a quadratic cost matrix. This linear cost matrix was used to determine which set of tasks need to be moved from one node to another. Holm and Sørensen [14] used techniques such as cutting planes, and branch and bound to reduce the symmetry and size of the solution space.

Heuristic methods involve limiting, or guiding, the search of the solution space using some function that quickly evaluates the "value" of the current candidate solution and guides the selection of the next candidate solution. These heuristics have their motivation in both graph theory and mathematical programming. Kernighan and Lin [15] proposed a heuristic based on the *max flow-min cut* algorithm of Ford and Fulkerson [16] which allocates tasks by first partitioning them into two sets, and then successively 2-way partitioning each subset until the allocation is completed. Another common approach is

to treat the load balancing problem as a graph isomorphism problem, which is also NP-complete, and to use heuristics to guide the mapping of the program graph on the network graph. These heuristic methods are based on identifying chains in the program graph [17] or identifying clusters in the program graph [18].

2.1 Summary

The approaches to the load balancing problem can be classified using Casavant's and Kuhl's taxonomy. There are two important distinctions in this taxonomy, the distinctions between static and dynamic techniques, and the distinction between optimal and sub-optimal techniques. Static load balancing techniques commit the allocation of tasks to processing nodes when the program is complied. Dynamic load balancing methods are on-line methods that use very little *a priori* knowledge of the program, and leave the decision of where to allocate a task to the run-time system when the task becomes available.

Since the load balancing problem is NP-complete in the general sense, the approaches to solving this problem are either based on restricting the problem such that an optimal solution can be found in polynomial time, or finding a near-optimal solution to the general problem. Both approaches are generally based on graph theory, mathematical programming theory or queuing theory.

3.0 Static Communications Load Balancing

The algorithms presented in this section are designed to find allocations of tasks to processing nodes that reduce the time a node spends receiving data, processing data and transmitting data. This increases the throughput of tasks through the parallel or distributed computing system. This optimality condition does not necessarily produce allocations with the lowest makespan.

3.1 The Elastic Force Algorithm

Consider a system of n tasks allocated among m processing nodes. The m processing nodes are equidistant as regards communications and the time taken for a unit message to be transferred from node i to node j is independent of i and j, $\forall i \neq j$. Let each task exerts a distance dependent force on every other task. This force is equal to a measure of the communications between the tasks multiplied by the distance between the tasks. The distance between tasks allocated to the same processing node is zero. The distance between tasks allocated to different processing nodes is one. The total energy stored in the system is proportional to the sum of the forces in the system. If the n tasks are equally distributed among the m processing nodes initially, then the basis of the algorithm is to keep swapping pairs of tasks until the total energy in the system reaches a minimum. The selection of which pair of tasks to swap is made by calculating the total force exerted on each task

by each processing node, i.e. the sum of the forces exerted by the tasks currently allocated to that node on the task in question. The task which experiences the greatest force and one of the tasks allocated to the processing node which exerts that force are the candidates for swapping. Prior to swapping the reduction in system energy, i.e. the gain from the swap, is calculated. If this gain is greater than zero the swap is made, otherwise the swap is eliminated and the algorithm searches for another candidate pair for swapping. When there is no swap which will reduce the system energy, then the algorithm terminates with the current task allocation.

Prior to applying the algorithm, the communications between tasks are examined to find the tasks that experience a force from only one other task. Each of the tasks that experience a force from only one other task is then swapped with another task on the node from which the force was exerted.

3.2 Definitions

3.2.1 A Task

A **task** is a computational entity that is created, receives data, performs a computation in a time l_i, transmits data to other tasks and then ceases to exist. A task can communicate with any other task. The amount of data transmitted from task i to task j is $c_{i,j}$. A computational entity that receives and transmits data before all

computations are completed can be decomposed into tasks which obey this definition.

3.2.2 A Processing Node

A processing node is a computational engine capable of running one task at a time. Tasks assigned to a node may be run either to completion or in a round robin manner. Communications between tasks assigned to the same node are considered to occur instantaneously, while communications between tasks on different nodes occur in a time which is a function of the amount of data transferred between the two tasks.

3.2.3 The Communications Matrix, [C]

The communications matrix [C] is an n x n matrix where n is the number of tasks to be balanced. Each element of [C], $c_{i,j}$, is a measure of the communications from task i to task j.

3.2.4 The Allocation Matrix, [A]

The allocation matrix [A] is an n x m matrix where n is the number of tasks to be allocated among m processing nodes. Each

element of [A], $a_{i,j}$, is 1 if task i is allocated to node j, and is 0 if task i is not allocated to node j.

3.2.5 The Modified Communications Matrix, [Ca]

The **modified communications matrix** [Ca] is an n x n matrix where n is the number of tasks to be balanced. The modified communications matrix [Ca] is equal to the communications matrix [C] modified by allocation matrix [A]. Each column of [A] is examined and the rows which are set to 1 are noted. The communications between the tasks that correspond to these rows are set to 0. For example, if column 1 of [A] contains elements set to 1 in rows 1,4 and 5, which corresponding to tasks 1,4 and 5, then $c_{1,4}$, $c_{4,1}$, $c_{4,5}$, $c_{5,4}$, $c_{1,5}$ and $c_{5,1}$ are set to 0.

3.2.6 The Force Matrix, [F]

The **force matrix** [F] is an n x m matrix where n is the number of tasks to allocate among m processing nodes. Each element of [F], $f_{i,j}$, is a measure of the total force processing node j exerts on task i.

$$[F] = [Ca][A]$$

Therefore, $f_{i,j}$, is the total communications task i receives from all the tasks allocated to node j.

3.2.7 Total Stored Energy, E_s

The total energy stored in the system is $E_s = KF_c$ where F_c is the total of the forces exerted on each task, and K is a constant of proportionality.

$$F_c = \sum_{i,j} f_{i,j}$$

Therefore, F_c, is the total of all the communications between tasks on different nodes.

$P(i,j)$ is the force task j exerts on task i.

$$P(i,j) = c_{i,j}$$

3.2.8 Inertia of a Task, I

The inertia of task i, $I(i)$, is equal to the sum of the forces experienced by task i.

$$I(i) = \sum_{j} c_{i,j}$$

3.2.9 Task Move Gain, M

The gain from moving task i from node x to node y is (the force exerted on task i by node y) + (the force exerted by task i on the tasks allocated to node y) - (the force exerted on task i by the tasks allocated to node x) - (the force exerted by task i on the task allocated to node x).

$$M(i,x,y)=\left(\sum_{k=1}^{m}(c_{i,k}.a_{k,y})+\sum_{k=1}^{m}(c_{k,i}.a_{k,y})\right)-\left(\sum_{k=1}^{m}(c_{i,k}.a_{k,x})+\sum_{k=1}^{m}(c_{k,i}.a_{k,x})\right)$$

3.2.10 Task Swap Gain, S

The gain from swapping task i on node x with task j on node y is

$$S(i,x,j,y)=M(i,x,y)+M(j,y,x)-2(c_{i,j}+c_{j,i})$$

i.e., it is the gain from moving task i from node x to node y plus the gain from moving task j from node y to node x minus twice the sum of the communications between tasks i and j.

3.3 The Algorithm

The Elastic Force load balancing algorithm for tasks of equal computational load is as follows.

Step 1: Allocate the n tasks among the m processing nodes equally (creating as many null processes as required) without reference to the inter-task communications. This gives the initial allocation matrix [A].

Step 2: Examine the communications matrix [C] for rows which contain only one non-zero element. For such elements, $c_{i,j}$, swap task i with a task, other than task j on the processing node to which task j is allocated, which experiences the greatest force from a task, other than task i, on the processing node to which task i is allocated.

Step 3: Derive the modified communications matrix [Ca] from [C] using [A].

Step 4: Calculate the force matrix [F] and the measure of the total force stored in the system, F_c.

Step 5: Examine [F] to find the rows and columns of [F] which contains the largest element in [F]. Note these (task, node) pairs.

Step 6: For each (task i, node j) pair, find the tasks currently allocated to node j, e.g. tasks k and l. Then calculate the force exerted on task i by these tasks.

$$P(i,k) = c_{i,k}$$

Step 7: If there is more than one minimum P value P(i,k) proceed to step 8. Otherwise, the candidate swap is task i with task k. The gain from swapping task i on node x with task k on node j is calculated, $S(i,x,k,j)$. If $S(i,x,k,j)$ is not greater than zero examine [F] to find the next highest elements in [F]. Note these (task, node) pairs, and return to step 6. If $S(i,x,j,y)$ is greater than zero, swap tasks i and j, and proceed to step 10.

Step 8: If there is more than one minimum P value $P(i_C,k_C)$, calculate the inertia of each task k_C,

$$I(k_c) = \sum_{h} c_{k_c,h}$$

Step 9: Find the minimum inertia value $I(k_C)$. The task swap associated with this minimum inertia value is swapping tasks i_C and k_C. Where there is more than one minimum $I(k_C)$, choose the swap which has the lowest $I(i_C)$ value. Calculate the gain from swapping task i_C on node x with task k_C on node j. If $S(i_C,x,k_C,j)$ is not greater than

zero, find the next lowest inertia value $I(k_d)$ and calculate the gain $S(i_d,x,k_d,j)$. If $S(i_d,x,k_d,j)$ is not greater than zero continue selecting the next lowest inertia values until an inertia value $I(k_e)$ is found such that $S(i_e,x,k_e,j)$ is greater than zero. If no such $I(k_e)$ is found, examine [F] to find the next highest elements in [F]. Note these (task, node) pairs and return to step 6. When a pair of tasks, i_e and k_e, are found such that $S(i_e,x,k_e,j)$ is greater than zero, swap tasks i_e and k_e.

Step 10: Derive the new allocation matrix [A].

Step 11: Repeat steps 3 to 10 until no task swap is found which decreases the measure of the system energy F_c.

This algorithm seeks to minimise the energy of the system and hence maximise the throughput of tasks through the parallel or distributed system. The speed at which the algorithm finds a solution depends on the probability that the swap indicated by $\max_{i,j}(f_{i,j})$ has a gain $S(i,x,k,j)$ greater than zero. Even if this probability is high the algorithm is very complicated. It involves many steps and calculations, and is not a realistic algorithm, due to its complexity, for large numbers of tasks. However, the algorithm does lead to a much more efficient algorithm, the Maximum (k-1) Sum algorithm.

$$[C] = \begin{bmatrix} 0 & 1 & 2 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 1 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

where $C_{a,b} \equiv C_{1,2}$, $C_{c,d} \equiv C_{3,4}$, etc.

The maximum j element sum of task i will be denoted as jS(i).

First, create a null task, ϕ, so that the number of tasks can be evenly divided among the 3 nodes. This modifies the [C] matrix as follows:

[C] =	ϕ	a	b	c	d	e	I(i)
ϕ	0	0	0	0	0	0	0
a	0	0	1	2	2	0	5
b	0	1	0	0	0	0	1
c	0	2	0	0	1	1	4
d	0	2	0	1	0	0	3
e	0	0	0	1	0	0	1

where task ϕ is the null task.

Therefore,

[Cu] =		ϕ	a	b	c	d	e	1S(i)
	ϕ	-	0	0	0	0	0	0
	a		-	2	4	4	0	4
	b			-	0	0	0	4
	c				-	2	2	4
	d					-	0	4
	e						-	2

The largest 1S(i) corresponds to task a. Tasks c and d contribute equally to 1S(a), but since I(d) < I(c), allocate tasks a and d to node 1. Eliminating the rows and columns corresponding to tasks a and d gives

[Cu] =		ϕ	a	b	c	d	e	1S(i)
	ϕ	-	0	0	0	0	0	0
	a		-	0	0	0	0	0
	b			-	0	0	0	0
	c				-	0	2	2
	d					-	0	0
	e						-	2

The largest 1S(i) corresponds to task c. Task e contributes to 1S(c), so therefore allocate tasks c and e to node 2. Allocate the remaining tasks to node 3.

The allocation produced by the algorithm is:

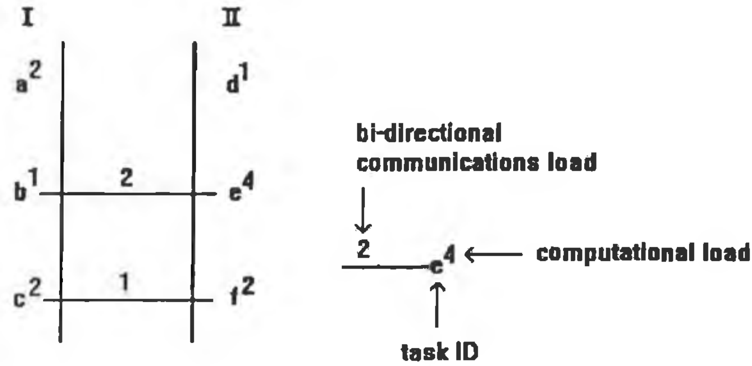
Node	Tasks
1	a, d
2	c, e
3	b, ϕ

3.5 The General Maximum (k-1) Sum Algorithm

The Maximum (k-1) Sum algorithm developed so far assumes that all the tasks have equal computational length. It is possible to extend the Maximum (k-1) Sum algorithm to include tasks with different computational loads. Consider a system of n tasks of different computational loads l_1, l_2, ..., l_n where the measure of the communications from task i to task j is given by $c_{i,j}$. Assume that the n tasks have been evenly allocated among m processing nodes without considering the inter-task communications. Then the ideal execution time due to the computational loads of the n tasks allocated among m nodes is

$$P = \frac{\sum_{j=1}^{n} l_j}{m}$$

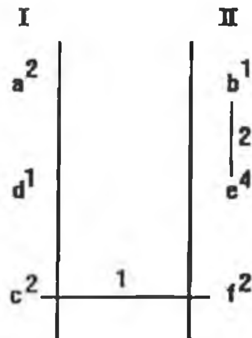
In this ideal computationally balanced system, consider the following portions of 2 nodes.



The total execution time on node I , R_t, is $P + R_c$, where R_c = execution time due to communications.

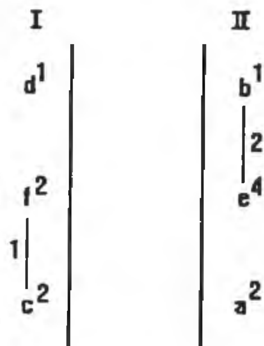
$$R_t(I) = P + 3 \text{ and } R_t(II) = P + 3$$

If tasks *b* and *d* are swapped, we then have



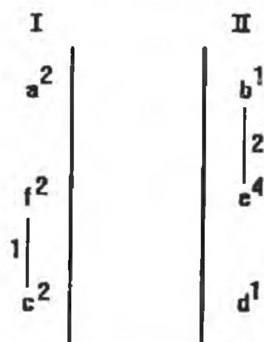
Since $l_b = l_d$, we have $R_t(I) = P + 1$ and $R_tI(I) = P + 1$.

If tasks *a* and *f* are swapped now, we have



Since $l_a = l_f$, we have $R_t(I) = P + 0$ and $R_tI(I) = P + 0$.

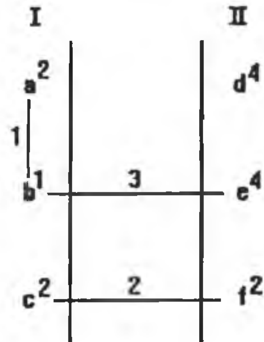
If instead of swapping tasks *a* and *f* we swapped tasks *d* and *f*, we would have the following arrangement.



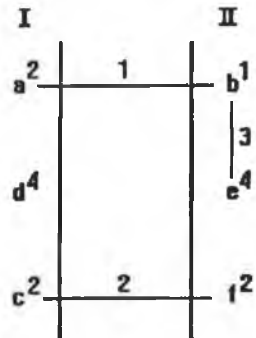
Now since $l_d \neq l_f$ the execution time due to computational loading has changed on both nodes by $|l_d - l_f| = 1$. Hence $R_t\ (I) = (P + 1) + 0$ and $R_t\ (II) = (P - 1) + 0$.

The saving in R_t due to the reduction in communications between nodes I and II has been penalised by the difference in the computational loads of the two tasks swapped, namely tasks *d* and *f*.

Consider another ideal computationally balanced system. Examining portions of 2 nodes we have $R_t(I) = P+5$ and $R_tI(I) = P+5$.



If we swapped tasks *b* and *d*, we would have:



Since $l_b \neq l_d$ the change in execution time due to task computational load on both nodes is $|l_b - l_d| = 3$, then

$$R_t(I) = (P + 3) + 3 = P + 6$$

$$R_t(II) = (P - 3) + 3 = P$$

Swapping task *b* and *d* has not reduced $R_t(I)$ and $R_t(II)$ evenly, but has caused an imbalance between $R_t(I)$ and $R_t(II)$. While an imbalance is not necessarily something which does not reduce the overall execution time of a program on a set of processing nodes, it should be avoided as it may increase the execution of some processing node in the system. In this example the execution time of node I, $R_t(I)$, has increased by 1.

Any swap between two tasks on different processing nodes which reduces the inter-node communications will cause a computational imbalance of:

$$2|l_i - l_j|$$

since both nodes are imbalanced by $|l_i - l_j|$.

Therefore the saving in total execution time by swapping task *i* on node *x* with task *j* on node *y* is

$$S(i,x,j,y) - 2|l_i - l_j|$$

The previous Maximum (k-1) Sum algorithm selected the *k* tasks whose sum of inter-task communications was maximised. When tasks with different computational loads are considered, the Maximum (k-1) Sum algorithm is modified to select the *k* tasks which maximise the expression

$$\{(cu_1 + \ldots + cu_{(k-1)}) - |P - (l_1 + \ldots + l_k)|\}$$

where P is the ideal computational load per node,

$$P = \frac{\sum_{i=1}^{n} l_i}{m}.$$

To maximise the expression

$$\{(cu_1 + \ldots + cu_{(k-1)}) - |P - (l_1 + \ldots + l_k)|\}$$

we need to maximise

$$(cu_1 + \ldots + cu_{(k-1)})$$

while minimising

$$|P - (l_1 + \ldots + l_k)|.$$

The expression

$$|P - (l_1 + \ldots + l_k)|$$

can be partitioned out among the k tasks as

$$\left|\sum_{i=1}^{k}\left(\frac{P}{k} - l_i\right)\right|.$$

Where the sum of the k computational loads is not equal to the ideal computational load per node P, this expression partitions the

cost of allocating k tasks to the same node among the k tasks. Using it we can formulate a variation of the Maximum (k-1) Sum algorithm which examines gain and cost of each task in turn.

3.5.1 Maximum (k-1) Sum Algorithm with Computational Loads

The following is the Maximum (k-1) Sum algorithm for allocating n inter-communicating tasks, with different computational loads, among m processing nodes.

Step 1: Initialise the variable *Deviation Sum* as zero.

Step 2: Calculate ideal computational load per node, P.

Step 3: Calculate [Cu].

Step 4: Calculate $\left(\frac{P}{k} - l_i\right)$, the load deviation, for each task.

Step 5: FOR each row (task) j DO

Step 6: FOR each element i which resides on the row and column that intersects the diagonal element of the row j DO

Step 7: Calculate G_i as follows,

If $(Deviation\ Sum)\left(\frac{P}{k}-l_i\right)>0$ or

$Deviation\ Sum=0$, then

$$G_i=c_i-\left|\frac{P}{k}-l_i\right|$$

else if $\left(\frac{P}{k}-l_i\right)\leq 2|Deviation\ Sum|$, then

$$G_i=c_i+\left(|Deviation\ Sum|-\left|Deviation\ Sum+\left(\frac{P}{k}-l_i\right)\right|\right)$$

else

$$G_i=c_i-\left(\left(\frac{P}{k}-l_i\right)+2|Deviation\ Sum|\right)$$

where c_i = total communications load between task i and task j.

Step 8: Record (k-1) largest G_i values, G'_i, and the tasks which they correspond to. Note G_j = min (G'_i).

Step 9: Calculate

$$Deviation\ Sum = \sum_i \left(\frac{P}{k} - l_i \right)$$

where i corresponds to the tasks which generate the (k-1) largest G_i values.

End of Step 6 FOR loop

Step 10: Calculate

$$S_j = \left(\sum_i c_i - \left| P - \sum_i l_i \right| \right)$$

where task i is one of the (k-1) tasks which correspond to the (k-1) largest G_i values of task j.

End of Step 5 FOR loop

Step 11: Find maximum S_j and allocate all tasks which contribute to S_j to the same node.

Step 12: Remove the rows and columns which correspond to the allocated tasks from [Cu].

Step 13: Repeat steps 5 to 12 until only k tasks remain. Allocate these tasks to the final node.

Notes:

(i) When selecting (k-1) largest G_i values, G'_i, and $G_j = G_k = \min(G'_i)$, choose G_j over G_k if $c_j > c_k$.

(ii) Inertia of task i is $I(i) = \sum_j c_{i,j} + \left| \frac{P}{k} - l_i \right|$

(iii) The computational order of the algorithm is

$$O(n,m) = (n-1)n + \left(n - \frac{n}{m} - 1\right)\left(n - \frac{n}{m}\right) + \cdots + \left(\frac{2n}{m} - 1\right)\left(\frac{2n}{m}\right)$$

$$= \sum_{i=0}^{m-2} \left(n - \frac{in}{m} - 1\right)\left(n - \frac{in}{m}\right)$$

$$\approx \sum_{i=0}^{m-2} \left(n - \frac{in}{m}\right)^2 \approx \sum_{i=0}^{m-2} \left\{ n^2 - 2\frac{in^2}{m} + \frac{i^2 n^2}{m^2} \right\}$$

Since $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ and $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ then,

$$O(n,m) \approx mn^2$$

3.5.2 Example

Allocate tasks a,b,c,d,e,f,g,h and i among 3 processor nodes given that the communications matrix [C] for these tasks is:

[C] =

	a	b	c	d	e	f	g	h	i
a	0	1	0	2	0	0	0	0	5
b	1	0	0	0	0	2	0	0	0
c	0	0	0	0	1	0	0	2	1
d	2	0	0	0	0	2	0	0	0
e	0	0	1	0	0	0	0	0	0
f	0	2	0	2	0	0	3	0	0
g	0	0	0	0	0	3	0	0	0
h	0	0	2	0	0	0	0	0	0
i	5	0	1	0	0	0	0	0	0

and that the computational load for each task is:

Task	Computational Load
a	1
b	3
c	2
d	2
e	3
f	2
g	1
h	4
i	5

Firstly, calculate the ideal load per node, P, and k.

$$P = \frac{\sum_i l_i}{m} = 23/3 = 7.6666$$

$$k = \frac{\#tasks}{\#nodes} = 9/3 = 3$$

The [Cu] matrix is:

[Cu] =	a	b	c	d	e	f	g	h	i
a	-	2	0	4	0	0	0	0	10
b		-	0	0	0	4	0	0	0
c			-	0	2	0	0	4	2
d				-	0	4	0	0	0
e					-	0	0	0	0
f						-	6	0	0
g							-	0	0
h								-	0
i									-

The load deviation, $\left\{\frac{P}{k} - l_i\right\}$, for each task is:

Task	Load Deviation
a	+1.5555
b	-0.4444
c	+0.5555
d	+0.5555
e	-0.4444
f	+0.5555
g	+1.5555
h	-1.4444
i	-2.4444

To find which tasks are allocated to node 1 we calculate the maximum (k-1) element sum for each task, which in this example is the maximum 2 element sum for each task.

Line 1 - Task a

Task	G_i	*Deviation Sum*
b	2-0.4444	
c	0+0.3333	-0.4444
* d	**4+0.3333**	**+0.5555**
e	0+0.4444	
f	0-0.5555	
g	0-1.5555	
h	0-0.3333	
* i	**10-1.3333**	

Line 2 - Task *b*

Task	G_i	*Deviation Sum*
c	0-0.5555	
d	0-0.5555	+0.5555
e	0+0.4444	-0.4444
* f	**4+0.3333**	**+0.5555**
g	0-1.5555	
h	0-0.3333	
i	0-1.3333	
* a	**2-1.5555**	

$S_1 = (4+10) - |7.6666 - (1+2+5)|$

$= 13.6666$

$S_2 = (4+2) - |7.6666 - (3+2+1)|$

$= 4.3333$

Line 3 - Task *c*

Task	G_i	*Deviation Sum*
d	0-0.5555	
* e	2+0.4444	-0.4444
f	0+0.3333	
g	0-0.6666	
* h	4-1.4444	-1.4444
i	2-2.4444	
a	0+1.3333	
b	0-0.4444	

$S_3 = (2+4) - |7.6666 - (2+3+4)|$

$= 4.6666$

Line 4 - Task *d*

Task	G_i	*Deviation Sum*
e	0-0.4444	
* f	4+0.3333	+0.5555
g	0-1.5555	
h	0-0.3333	
i	0-1.3333	
* a	4-1.5555	
b	0+0.4444	
c	0-0.5555	

$S_4 = (4+4) - |7.6666 - (2+2+1)|$

$= 5.3333$

Line 5 - Task *e*

Task	G_i	*Deviation Sum*
f	0-0.5555	
g	0-1.5555	+0.5555
h	0-0.3333	-1.4444
i	0-2.4444	
* a	0+1.3333	+1.5555
b	0+0.4444	
* c	2-0.5555	
d	0-0.5555	

$$S_5 = (0+2) - |7.6666-(3+1+2)|$$

$$= 0.3333$$

Line 6 - Task *f*

Task	G_i	*Deviation Sum*
* g	6-1.5555	
h	0+1.4444	+1.5555
i	0-0.6666	
a	0-1.5555	
* b	4+0.4444	
c	0-0.5555	
d	4-0.5555	
e	0+0.4444	

$$S_6 = (6+4) - |7.6666-(2+1+3)|$$

$$= 8.3333$$

Line 7 - Task *g*

Task	G_i	*Deviation Sum*
h	0-1.4444	
i	0-2.4444	-1.4444
* a	0+1.3333	+1.5555
b	0+0.4444	
c	0-0.5555	
d	0-0.5555	
e	0+0.4444	
* f	6-0.5555	

Line 8 - Task *h*

Task	G_i	*Deviation Sum*
i	0-2.4444	
* a	0+1.5555	+1.5555
b	0+0.4444	
* c	4-0.5555	+0.5555
d	0-0.5555	
e	0+0.4444	
f	0-0.5555	
g	0-1.5555	

$S_7 = (0+6) - |7.6666 - (1+1+2)|$

$= 2.6666$

$S_8 = (0+4) - |7.6666 - (4+1+2)|$

$= 3.3333$

Line 9 - Task *i*

Task	G_i	*Deviation Sum*
* a	10-1.5555	
b	0-0.4444	+1.5555
* c	2-0.5555	
d	0-0.5555	
e	0+0.4444	
f	0-0.5555	
g	0-1.5555	
h	0+1.4444	

$S_9 = (10+2) - |7.6666 - (5+1+2)|$

$= 11.6666$

The largest maximum 2 element sum, $\max(S_i)$, is S_1 which corresponds to task *a*. The tasks which contribute to S_1 are tasks *d* and *i*. Therefore allocate tasks *a*, *d* and *i* to node 1.

Removing tasks *a*, *d* and *i* from [Cu] yields:

		b	c	e	f	g	h
[Cu] =	b	-	0	0	4	0	0
	c		-	2	0	0	4
	e			-	0	0	0
	f				-	6	0
	g					-	0
	h						-

To find the tasks to allocate to node 2 we calculate the maximum 2 element sum for each task.

Line 1 - Task *b*

Task	G_i	*Deviation Sum*
c	0-0.5555	
* e	0+0.4444	-0.4444
* f	4+0.3333	+0.5555
g	0-1.5555	
h	0-0.3333	

$$S_1 = (0+4) - |7.6666-(3+3+2)|$$

$$= 3.6666$$

Line 2 - Task *c*

Task	G_i	*Deviation Sum*
* e	2-0.4444	
f	0+0.3333	-0.4444
g	0-0.6666	
* h	4-2.4444	
b	0-0.4444	

$$S_2 = (2+4) - |7.6666-(2+3+4)|$$

$$= 4.6666$$

Line 3 - Task *e*

Task	G_i	*Deviation Sum*
f	0-0.5555	
g	0-1.5555	-0.5555
* h	0-0.3333	-1.4444
b	0-0.4444	
* c	2+0.5555	

$$S_3 = (0+2)-|7.6666-(3+4+2)|$$

$$= 0.3333$$

Line 4 - Task *f*

Task	G_i	*Deviation Sum*
* g	6-1.5555	
h	0+1.4444	+1.5555
* b	4+0.4444	
c	0-0.5555	
e	0+0.4444	

$$S_4 = (6+4)-|7.6666-(2+1+3)|$$

$$= 8.3333$$

Line 5 - Task *g*

Task	G_i	*Deviation Sum*
h	0-1.4444	
b	0-0.4444	-0.4444
c	0+0.3333	+0.5555
* e	0+0.4444	-0.4444
* f	6+0.3333	

$$S_5 = (0+6)-|7.6666-(1+3+2)|$$

$$= 4.3333$$

Line 6 - Task *h*

Task	G_i	*Deviation Sum*
b	0-0.4444	
* c	4+0.3333	+0.5555
* e	0+0.4444	
f	0-0.5555	
g	0-1.5555	

$$S_6 = (4+0)-|7.6666-(4+2+3)|$$

$$= 2.6666$$

$Max(S_i)$ is S_4, and task *f* corresponds to S_4. The tasks which contribute to S_4 are tasks *g* and *b*. Therefore allocate tasks *f*, *g* and *b* to node 2. Allocate the remaining tasks, *c*, *e* and *h*, to node 3.

Hence the "optimum" allocation produced by the Maximum (k-1) Sum algorithm is:

Node	Tasks
1	a, d, i
2	b, f, g
3	c, e, h

3.5.3 Experimental Evaluation of Maximum (k-1) Sum Algorithm

To verify the quality of the allocations produced by the Maximum (k-1) Sum algorithm, the algorithm was applied to an existing computer vision problem. The directed acyclic graph for this computer vision problem (figure 2) was initially presented by Kunii, Nishimura and Noma [19]. The Maximum (k-1) Sum algorithm was applied to this problem for varying number of processing nodes. Table 1 compares these results with the results produced by Sarkar's algorithm [20] on the unbounded case, followed by modulo allocation, and Darte's two heuristics [17]. The values in Table 1 are the maximum time spent by any node processing the tasks allocated to it and

receiving/transmitting data from/to tasks allocated to other processing nodes.

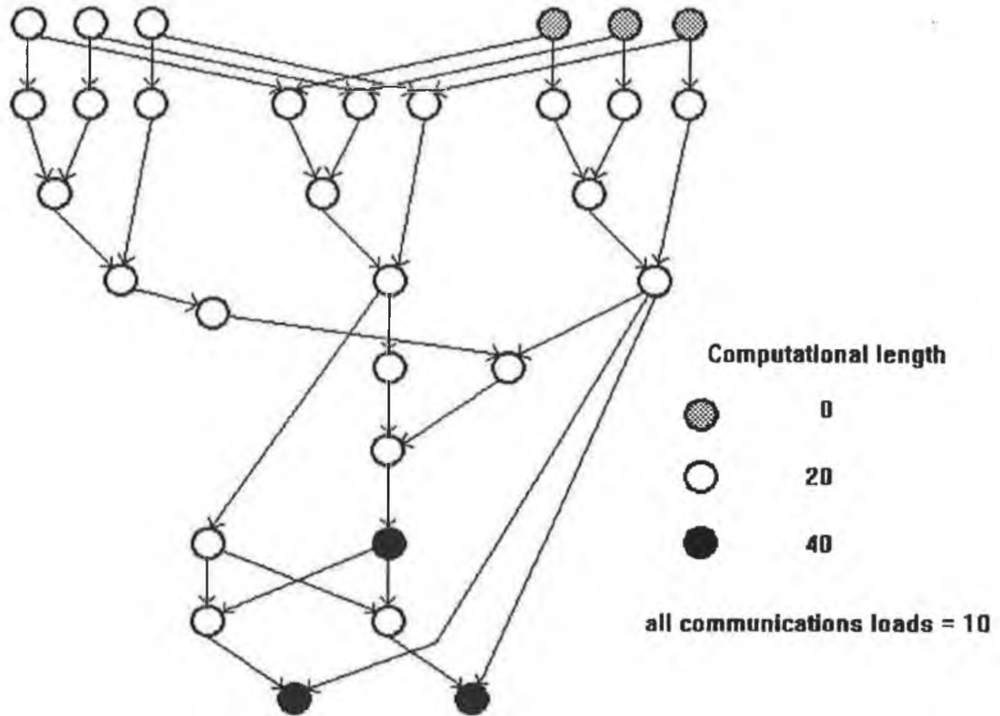


Figure 2: Directed Acyclic Graph for Computer Vision Program

This problem was chosen to test the quality of the Maximum (k-1) Sum algorithm because it was a real world problem, and because it is not particularly suited to the algorithm. The algorithm seeks to allocate a task to the same node as the tasks with which it directly communicates. The algorithm does not seek to allocate chains of communicating task to the same processing node. The direct acyclic graph for the computer vision problem contains many communicating

task chains, e.g. tasks 1, 7, 16, 19, 22, 23, 25, 27, 29, and 31 form a chain. The computer vision problem presented here has mixture of both types of graph structures. The Maximum (k-1) Sum algorithm will perform poorly on program graphs which have replicated sub-structures containing many chains.

Number of Processing Nodes	Sarkar + modulo allocation	Darte's Heuristic 1 + modulo allocation	Darte's Heuristic 2 + modulo allocation	Maximum (k-1) Sum algorithm
1	620	620	620	620
2	480	430	370	530
3	350	360	320	440
4	320	360	270	330
5	290	280	260	290
6	310	280	260	240
7	280	270	260	180
8	280	260	260	170
9	270	260	260	180
10	270	260	260	150

Table 1: Execution Times

From Table 1, we see that for small numbers of processing nodes Darte's second heuristic produced the best results. But as the number of processing nodes was increased the Maximum (k-1) Sum algorithm

produced superior results, producing allocations which executed in 73.33% faster when allocated to 10 processing nodes. Dartes' heuristics produces better results with low numbers of processing nodes because it is not restricted in the number of tasks it allocates to a given node. For example, when the computer vision problem is allocated to four processing nodes using Dartes' heuristics, it will allocate 11 tasks to one node and only 4 tasks to another. The Maximum (k-1) Sum algorithm always allocates an equal number of tasks to each node. However, as the number of processing nodes increases, Dartes' heuristics become restricted by the search for linear clusters in the program graph and fails to reduce the execution time further. The Maximum (k-1) Sum algorithm continues to reduce the execution time as the number of processing nodes increases and produces superior results when the computer vision problem is allocated to 6 or more processing nodes. However, the Maximum (k-1) Sum algorithm can be modified to allocated a different number of tasks to different processors by adding null tasks to the system. The algorithm will now allocate k' tasks to each processor, but some of these tasks will be null tasks that do not exist. This modification generated better allocations at the cost of extra processing time since n, the number of tasks in the system, has increased. For example, allocating the computer vision problem to four processing nodes, but allowing up to 12 tasks per node, generated 17 null tasks and produced an allocation with an execution time of 310. This is a 6% improvement on the result obtained in Table 1.

4.0 Non-Symmetric Formulation of Task Allocation Problem

This section presents a non-symmetric mathematical formulation for the problem of allocating n inter-communicating tasks among m homogeneous processing nodes when the optimisation criterion is to minimise response time, which is the elapsed-time for the execution of the whole set of tasks that constitute the program. This formulation shows that the problem is not only an integer optimisation problem, it is also quadratic in nature. A relaxation technique is presented which reduces the problem to a mixed integer linear problem (MILP). This relaxed MILP formulation is implemented in the Sciconic mathematical programming package and is applied to an example problem.

A difficulty with previous formulations is that they are symmetric. If an allocation produces s non-empty subsets of the n tasks, with each subset assigned to a different node, then there exists $\frac{m!}{(m-s)!}$ identical allocations of the s subsets of tasks among the m processing nodes since the processing nodes can be numbered arbitrarily. This implies that the gap between the relaxed linear solution and the mixed integer solution is large [14]. The aim of the non-symmetric formulation is to reduce the size of the mixed integer solution space by removing the symmetries.

4.1 Definitions

4.1.1 A Task

A task is a computational entity that is created, receives data, performs a computation in a time l_i, transmits data to other tasks and then ceases to exist. A task can communicate with any other task. The time taken to transmit data from task i to task j is $c_{i,j}$. A computational entity that receives and transmits data before all computations are completed can be decomposed into tasks which obey this definition.

4.1.2 A Program

A program consists of a set of inter-communicating tasks bounded by precedence constraints.

4.1.3 A Processing Node

A processing node is a computational engine capable of executing one task at a time. The processing nodes are assumed to be homogeneous and to be connected in a bus topology in which no queuing problems arise. This implies that:

(i) All processing nodes are similar as regards computational performance, i.e. task i will take al_i seconds to execute its computational load regardless of which processing node it is allocated to.

(ii) All processing nodes are equidistant as regards communications and the time it takes for task i to transmit data to task j is independent of the nodes to which tasks i and j are assigned, if tasks i and j are allocated to different nodes.

4.1.4 Precedence Level

The **precedence level of task *i*, prec (*i*)**, is defined as the computational load of task i plus the maximum, over all task i's successors, of the successor's precedence level plus the communications load between the tasks. Task i's **successor tasks** are those tasks that can not be executed until task i is completed. Assume, without loss of generality, that task n is the terminal task. Then,

$$prec(i) = l_i + \max_{j \in S_i}\{prec(j) + c_{i,j}\}; \qquad S_i = \{i \in \{1,..,n\}, j \in S_i | i \rightarrow j\}$$

$$prec(n) \equiv 0$$

4.2 Formulation of Non-Symmetric Mathematical Programming Model

Let l_i = the computational load of task i, the time task i spends processing data.

$c_{i,j}$ = the communications load from task i to task j, the time taken to transmit data from task i to task j.

$a_{i,j}$ = 1, if task i and task j are allocated to the same node,

0, if task i and task j are not allocated to the same node.

y_i = the start time for task i.

m_i = 0, if task i is the task with the lowest cardinal number on the processing node to which it is allocated,

1, if task i is not the task with the lowest cardinal number on the processing node to which it is allocated.

prec(i) = precedence value of task i.

We wish to find the allocation of n tasks among the m processing nodes which minimises the elapsed-time for the execution of the whole set of tasks,

$$\min\ (y_n - y_1) \qquad [4.1]$$

subject to the following constraints.

(i) The earliest start time of two communicating tasks

The earliest time a task j can start depends upon whether the task it is receiving data from, task i (predecessor task) is allocated to the same node as task j (successor task).

Case (a): Both predecessor and successor tasks allocated to the same node.

The successor task, j, can start to execute once the predecessor task, i, has terminated. Task i terminates when it has finished its computation and all its inter-task communications. Therefore,

$$y_j - y_i \geq l_i + \sum_{i,k} (1 - a_{i,k}) c_{i,k} \qquad \forall\ i \sim j \qquad [4.2]$$

Case (b): Predecessor and successor tasks allocated to different processing nodes.

The successor task, j, can start executing once it has received its data from the predecessor task, i, subject to the processing node being available for execution. The predecessor task is assumed to transmit data to tasks in the order of decreasing precedence level. Therefore,

$$y_j - y_i \geq l_i + \sum_{i,k} p'_{i,k}(1 - a_{i,k})c_{i,k} \quad \forall\, i \sim j \qquad [4.3]$$

where $p'_{i,k}$ = 1 , if prec(k) ≥ prec(j)

0 , if prec (k) < prec(j)

Combining case (a) and case (b) gives:

$$y_j - y_i \geq l_i + \sum_{i,k} p_{i,k}(1 - a_{i,k})c_{i,k} \quad \forall\, i \sim j \qquad [4.4]$$

where $p_{i,k}$ = 1 , if $a_{i,j}$ = 1 or prec(k) ≥ prec(j)

0 , if $a_{i,j}$ = 0 and prec(k) < prec(j)

The above notation can be simplified by ranking the precedence level of each task and renumbering the tasks such that $i < j$ if

rank(*prec*(*i*)) > rank(*prec*(*j*)). However, since the concept of precedence level will be used in chapter 5 the current notation will be retained.

(ii) Associativity

If task i is allocated to the same node as tasks j and k, then task j is allocated to the same node as task k.

$$a_{i,j} + a_{i,k} - a_{j,k} \leq 1 \qquad \forall\, i,j,k \qquad [4.5]$$

(iii) Symmetric nature of the adjacency matrix [A]

If task i is allocated to the same node as task j, then task j is allocated to the same node as task i.

$$a_{i,j} = a_{j,i} \qquad \forall\, i,j \qquad [4.6]$$

(iv) The diagonal of the adjacency matrix [A]

$$a_{i,i} = 1 \qquad \forall\, i \qquad [4.7]$$

(v) The binary nature of the adjacency matrix [A]

$$a_{i,j} \in \{0,1\} \qquad \forall\, i,j \qquad [4.8]$$

(vi) The number of processing nodes m

Since m_i, $1 \le i \le n$, equals zero if task i is the lowest cardinal number task allocated to its processing node, then the maximum number of m_i entries that are equal to zero is the number of processing nodes available, m. Therefore the sum of m_i must be greater than or equal to $(n-m)$.

$$m_i \in \{0,1\} \qquad \forall\, i \qquad [4.9]$$

$$m_1 = 0 \qquad [4.10]$$

$$m_i \le \sum_{j=1}^{i-1} a_{i,j} \qquad \forall\, 1 < i \le n \qquad [4.11]$$

$$m_i \ge a_{i,j} \qquad \forall\, 1 \le j \le i-1, i > 1 \qquad [4.12]$$

$$\sum_{i=1}^{n} m_i \ge n - m \qquad [4.13]$$

(vii) The number of tasks that can be executing on the same node at the same time.

Only one task can execute on a processing node at any given time. If task i has a higher precedence level than task j, and both tasks are allocated to the same processing node, then

$$y_j - y_i - l_i - \sum_k (1 - a_{i,k}) c_{i,k} \geq 0 \qquad \text{if } prec(i) \geq prec(j) \text{ and } a_{i,j} = 1$$

$$\Rightarrow \quad a_{i,j}\left(y_j - y_i - l_i - \sum_k (1 - a_{i,k}) c_{i,k} \right) \geq 0 \qquad \text{if } prec(i) \geq prec(j)$$

$$\Rightarrow \quad a_{i,j} y_j - a_{i,j} y_i - a_{i,j} l_i - \sum_k a_{i,j} (1 - a_{i,k}) c_{i,k} \geq 0 \quad \text{if } prec(i) \geq prec(j)$$

[4.14]

Two constraints in this non-symmetric formulation are quadratic and discrete. In equation [4.4] $p_{i,k} = a_{i,j} p'_{i,j}$, and therefore this constraint is quadratic, as well as discrete. Equation [4.14] contains 3 terms that are quadratic and discrete.

The approach taken in this thesis to relax the formulation is to "linearise" the quadratic constraints, solve the discrete optimisation problem using branch and bound techniques, and then to use the adjacency matrix produced to calculate the actual run time,

including quadratic effects, of the solution of the discrete optimisation problem.

4.2.1 Relaxing Quadratic Constraints

The constraint on the earliest start time of two communicating tasks allocated to the same processing node can be relaxed by ignoring the effects of the other (n-2) tasks and recognising that $p_{i,j}$ = 1 because *prec(i)* ≥ *prec(j)*. Therefore equation [4.4],

$$y_j - y_i \geq l_i + \sum_{i,k} p_{i,k}(1 - a_{i,k})c_{i,k} \quad \forall\, i \sim j$$

where $p_{i,k} = \begin{cases} 1, & \text{if } a_{i,j} = 1 \text{ or } prec(k) \geq prec(j) \\ 0, & \text{if } a_{i,j} = 0 \text{ and } prec(k) < prec(j) \end{cases}$

can be rewritten as,

$$y_j - y_i \geq l_i + (1 - a_{i,j})c_{i,j} \quad \forall\, i,j$$

$$\Rightarrow \quad y_j - y_i + a_{i,j}c_{i,j} \geq l_i + c_{i,j} \quad \forall\, i,j \qquad [4.15]$$

The constraint on the number of tasks allocated to the same processing node that execute at the same time is relaxed as follows. Firstly, ignore the last term in equation [4.14] since this only

influences the result when the task with the greater precedence level communicates with its successor tasks that are allocated to different nodes than the node to which it is allocated.

$$a_{i,j}y_j - a_{i,j}y_i - a_{i,j}l_i \geq 0 \quad \text{if } prec(i) > prec(j) \qquad [4.16]$$

$$\Rightarrow \quad z_{i,j} - z_{j,i} - a_{i,j}l_i \geq 0 \qquad \text{if } prec(i) > prec(j)$$

[4.17]

$$\text{where } z_{i,j} = a_{i,j}y_j \in \{0, y_j\} \quad \because \quad a_{i,j} \in \{0,1\}$$

Relax $z_{i,j}$ so that it is continuous over the interval $[0,y_j]$,

$$0 \leq z_{i,j} \leq y_j.$$

and force $z_{i,j}$ towards either 0 or y_j, which ever is nearest, by modifying the objective function [4.1] as follows.

$$\min\left\{C(y_n - y_1) - \sum_{i,j}\left(\min\{z_{i,j}, y_j - z_{i,j}\}\right)\right\} \qquad [4.18]$$

where C is a constant called the *bounding constant*.

Substitute $w_{i,j}$ for $\min\{z_{i,j}, y_j - z_{i,j}\}$ and then the objective function is:

$$\min\left\{C(y_n - y_1) - \sum_{i,j} w_{i,j}\right\} \qquad [4.19]$$

subject to

$$0 \leq w_{i,j} \leq z_{i,j} \qquad \forall\, i,j \qquad [4.20]$$

$$w_{i,j} \leq y_j - z_{i,j} \qquad \forall\, i,j \qquad [4.21]$$

The first term in equation [4.19] can be replaced by $C\sum_i y_i$ since $C\sum_i y_i$ is minimised when $C(y_n - y_1)$ is minimised. This reason for this substitution is to simplify the calculation of the *bounding constant, C*.

Lemma 1: *$C(y_n - y_1)$ is minimised when $C\sum_i y_i$ reaches it minimum value.*

Proof: The value of y_1 can be assumed to be zero without loss of generality. To prove that $C(y_n - y_1)$ is minimised when $C\sum_i y_i$ reaches it minimum value it is necessary and sufficient to show that given a set of valid y_i's which yield a minimum to $C\sum_i y_i$, it is impossible to reduce y_n.

When $C\sum_i y_i$ is minimised this implies that there is no idle time before the execution of any task. Without loss of generality task n can be assumed to be the terminal task, since a dummy task with zero computational and communications loading can be created and

assigned to the highest cardinal identification number. Therefore task n is a direct descendant of every other task in the task graph of the program. If y_n decreases then the start time for some other task in the program must also decrease. But since there is no task with idle time before it starts execution, this is impossible.

Q.E.D.

Therefore the task allocation problem can be relaxed and stated as follows.

$$\min\left\{C\sum_{i} y_i - \sum_{i,j} w_{i,j}\right\} \quad [4.22]$$

subject to

$$y_j - y_i + c_{i,j}a_{i,j} \geq l_i + c_{i,j} \qquad \forall\, i \sim j \quad [4.23]$$

$$a_{i,j} + a_{i,k} - a_{j,k} \leq 1 \qquad \forall\, i,j,k \quad [4.24]$$

$$a_{i,j} = a_{j,i} \qquad \forall\, i,j \quad [4.25]$$

$$a_{i,i} = 1 \qquad \forall\, i \quad [4.26]$$

$$a_{i,j} \in \{0,1\} \qquad \forall\, i,j \quad [4.27]$$

$$m_i \in \{0,1\} \qquad \forall i \qquad [4.28]$$

$$m_1 = 0 \qquad [4.29]$$

$$m_i \le \sum_{j=1}^{i-1} a_{i,j} \qquad \forall 1 < i \le n \qquad [4.30]$$

$$m_i \ge a_{i,j} \qquad \forall 1 \le j \le i-1, i > 1 \qquad [4.31]$$

$$\sum_{i=0}^{n} m_i \ge n - m \qquad [4.32]$$

$$z_{i,j} - z_{j,i} - a_{i,j} l_i \ge 0 \qquad \text{if } prec(i) > prec(j) \qquad [4.33]$$

$$0 \le z_{i,j} \le y_j \qquad \forall i,j \qquad [4.34]$$

$$0 \le w_{i,j} \le z_{i,j} \qquad \forall i,j \qquad [4.35]$$

$$w_{i,j} \le y_j - z_{i,j} \qquad \forall i,j \qquad [4.36]$$

4.2.2 Calculation of Bounding Constant C

For C to bound equation [4.22] from below

$$C \sum_i y_i - \sum_{i,j} w_{i,j} \ge \alpha \qquad \text{as} \quad y_i \to \infty \qquad [4.37]$$

where α is some arbitrary value.

Each $w_{i,j}$ is subject to the following constraints.

$$0 \leq w_{i,j} \leq z_{i,j};\quad w_{i,j} \leq y_j - z_{i,j};\quad 0 \leq z_{i,j} \leq y_j$$

Therefore,

$$\min\left\{C(y_n - y_1) - \sum_{i,j} w_{i,j}\right\}$$

will tend to maximise $\sum_{i,j} w_{i,j}$. Since $z_{i,j}$ is a free variable over $[0, y_i]$ and $0 \leq w_{i,j} \leq z_{i,j};\ w_{i,j} \leq y_j - z_{i,j}$ each $w_{i,j}$ will be maximised when $z_{i,j} = \frac{y_j}{2} \Rightarrow w_{i,j} \rightarrow \frac{y_j}{2}$

$$\Rightarrow \quad \max\left\{\sum_{i,j} w_{i,j}\right\} \leq \sum_j \frac{n y_j}{2}$$

since $z_{i,j} - z_{j,i} \geq a_{i,j} l_i \qquad \text{if } prec(i) > prec(j)$.

Substituting into equation [4.37] yields:

$$C\sum_i y_i - \sum_j \frac{ny_j}{2} \geq \alpha$$

$$\Rightarrow \left(C - \frac{n}{2}\right)\sum_i y_i \geq \alpha$$

$$\Rightarrow C > \frac{n}{2}$$

Therefore C should be greater than $n/2$. As C continues to increase the effects of each $w_{i,j}$ decreases until the constraint on the number of simultaneous executing task on a node becomes insignificant and all tasks allocated to a node try to execute simultaneously.

Hence C should be set marginally greater then $n/2$.

4.3 Sciconic Implementation

Sciconic is a mathematical programming package for solving linear, integer and non-linear problems. It consists of four major sections.

(i) MGG, an ultra-high level language for problem formulation.

(ii) MG, an MPS matrix generator.

(iii) Sciconic, a mathematical programming engine.

(iv) RW, a report writer.

MGG is a specialised ultra-high level language for the formulation of mathematical programming problems. Using MGG the user can specify a model without reference to any instance of the problem. As well as the MGG language, MGG allows the user to supply, as part of the problem specification, their own FORTRAN routines for calculating coefficients and the conditions under which a constraint exists. In addition to these routines the user can supply FORTRAN routines for initialising internal variables. The MGG program compiles the MGG problem formulation and produces matrix generator (MG) source code, report writer (RW) source code and MG data file specification file, MGGOP.LIS. The MG source code is linked together with user supplied initialisation routines by the MGCL program to produce a matrix generator program, MG.

Once the problem has been specified, the user writes an MG data file which contains the data for a given instance of the problem. The format of the MG data file is strictly defined by the MG data file specification file MGGOP.LIS. When the user runs the matrix generator, MG, with a specified data file, an output file in standard MPS format is produced.

The Sciconic program is the mathematical programming engine which takes a file, which contains a matrix in standard MPS format, finds the optimal solution and writes it to a solution file. The

action of Sciconic in solving the problem is defined by a set of *run stream commands*. These commands break down into classes:

(i) **Agenda** which invoke built-in Sciconic procedures to load a problem, initialise a problem, find the linear primal and dual optimal solutions, perform a branch and bound search for the integer optimal solution, etc.

(ii) **System state variables** (SSVs) which control the action of Agenda, such as specifying the problem, solution and temporary files, setting options and determining the state of processes and end conditions.

In order to interpret the solution file, the report writer program, RW, is run on the solution file. The report writer source code is produced by the MGG compiler, based on the problem specification file. A user supplied FORTRAN routine REPORT is linked with the source code using the RWCL program. It is the responsibility of the REPORT routine to extract the data from the solution file, process it, and display the results to the user and/or store the results in a file. Figure 3 illustrates the procedure.

4.3.1 Problem formulation in Sciconic

Appendix B contains the MGG formulation of the non-symmetric task allocation model encapsulated in equations [4.22] to [4.36].

4.3.2 Report Writer

The report writer is used to extract the relevant data from the solution file produced by Sciconic and produce the results. The non-symmetric formulation of the task allocation problem in equations [4.22] to [4.36] produces a solution file in which the earliest task start times are relaxed. In order to generate a real allocation with valid start times, the report writer was used to reconstruct the actual start times for each task based on the generated adjacency matrix. The source code for the report routine is in appendix B.3.

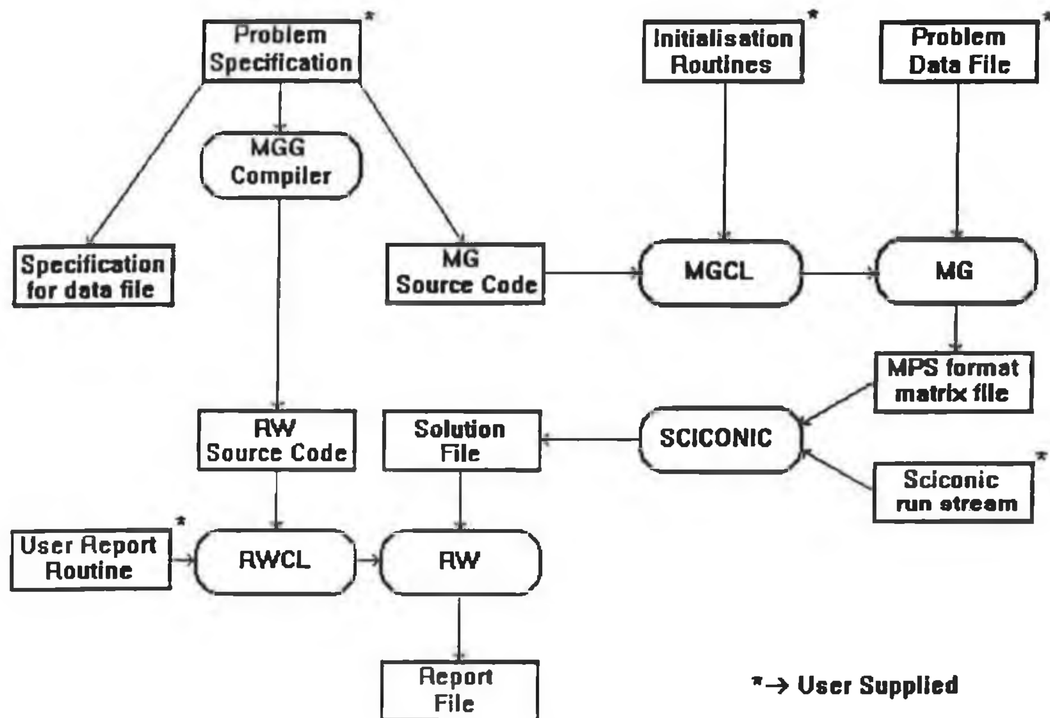


Figure 3: Sciconic Development Environment

5.0 Pseudo-Dynamic Load Balancing

This chapter proposes an extension to Casavant and Kuhl's taxonomy by further defining the distinction between static and dynamic load balancing to include a new classification **Pseudo-Dynamic Load Balancing**. Pseudo-Dynamic Load Balancing takes the best from both the static and dynamic approaches and merges them. Static load balancing analyses the program and hardware configuration of the parallel or distributed computing system, and commits itself to an allocation of tasks to processing nodes at compilation time. Dynamic load balancing makes the allocation of a task to a processing node at run time, based primarily on the state of the parallel or distributed computing system at that point in time. Both methods have advantages and disadvantages. Pseudo-Dynamic Load Balancing takes the data from the analysis of the program graph at compilation time and the state of the parallel or distributed computing system at run time, and makes an allocation of tasks to processing nodes, at run time, based on the current state of the parallel or distributed computing system and the program graph data from analysis at compilation time.

Pseudo-Dynamic Load Balancing is an **on-line** technique that produces high quality near-optimum task to node allocations where the optimality condition is to minimise the run time of the program. This heuristic technique uses information about the program extracted at compilation time, together with run-time information to determine the "best" allocation of tasks to processing nodes. The compilation time information is a program graph which can be generated by the compiler or pre-processor. The run-time information is a graph of the parallel

or distributed computing system and information on the current loading of the processing nodes.

The advantages of the pseudo-dynamic approach over the purely static and dynamic approaches are:

(1) Static load balancing algorithms cannot react to changes in the parallel or distributed computing system prior to, or during, run-time. Any change in the environment requires a recompilation of the program.

(2) Static load balancing algorithms are dependent on the accuracy of the estimation of the computational load of each task and the amount of communications between each task. If the estimations are incorrect the static load balancing algorithm cannot make any corrections during run-time.

(3) Pure dynamic load balancing algorithms tend to react mainly to the current state of the parallel or distributed computing system, since very little *a priori* information is included about the program. These algorithms tend to chase the optimum allocation as the state of the system changes. They do not use any available a priori knowledge to improve the task to processing node allocation.

5.1 Definitions

5.1.1 A Task

A **task** is a computational entity that is created, receives data, performs a computation in a time l_i, transmits data to other tasks and then ceases to exist. A task can communicate with any other task. The amount of data transmitted from task i to task j is $c_{i,j}$. A computational entity that receives and transmits data before all computations are completed can be decomposed into tasks which obey this definition.

5.1.2 A Program

A **program** consists of a set of inter-communicating tasks bounded by precedence constraints.

5.1.3 A Processing Node

A **processing node** is a computational engine capable of executing one task at a time. Multiple tasks can be assigned to the same node at any given time. The order in which a node executes the tasks allocated to it is given by a predefined rule. This rule state that the task currently executing is the task with the highest

precedence that is not waiting for any data from other tasks. The node may preempt a task with another task if the other task has a higher precedence and is not waiting for data. Processing nodes are not required to multi-task by time slicing tasks ready to execute.

The processing nodes in the parallel or distributed computing system are heterogeneous Q-class processing nodes [21], .i.e. the processing nodes are similar but the processing speed of different nodes in the system are related by a linear speed-up function.

5.1.4 A Program Graph

A program graph is a graph where the vertices represent tasks and the edges between vertices represent the communications between the corresponding tasks. The weight of each vertex represents the computational load, processing time, of the task associated with that vertex. The weight of an edge between two vertices is the communications load between the tasks represented by the vertices. The communications load represents the amount of data transferred between tasks. Edges are directional and represent the transfer of data from one task to another. Precedence constraints between non-communicating tasks can be represented by an edge of zero, or minimal, weight.

5.1.5 Precedence Level

The precedence level of task *i*, prec(*i*), is defined as the computational load of task *i* plus the maximum, over all task *i*'s successors, of the successor's precedence level plus the communications load between the tasks. Assume, without a loss of generality, that task *n* is the terminal task. Then,

$$prec(i) = l_i + \max_{j \in S}\{prec(j) + c_{i,j}\}; \qquad S = \{i \in \{1,..,n\}, j \in S | i \rightarrow j\}$$

$$prec(n) \equiv 0$$

For the program graph in figure 4, the precedence levels are:

prec(7) = 0, prec(6) = 3, prec(5) = 9, prec(4) = 6, prec(3) = 19, prec(2) = 16, prec(1) = 25

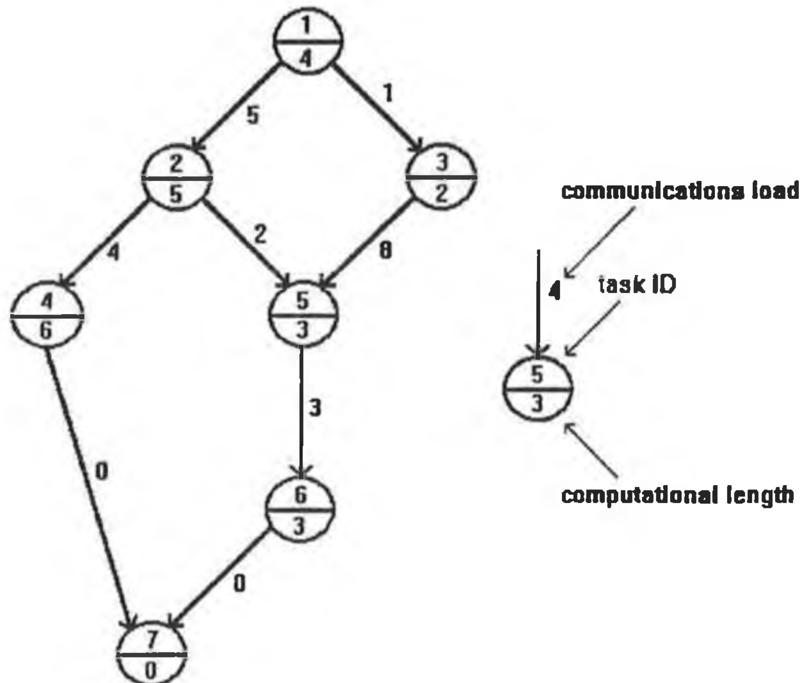


Figure 4 : Program Graph

A compiler, or pre-processor, can estimate the computational load of each task and the communications load between tasks. Using this information the compiler, or pre-processor, can calculate the precedence level of each task, and produce a program graph for the program.

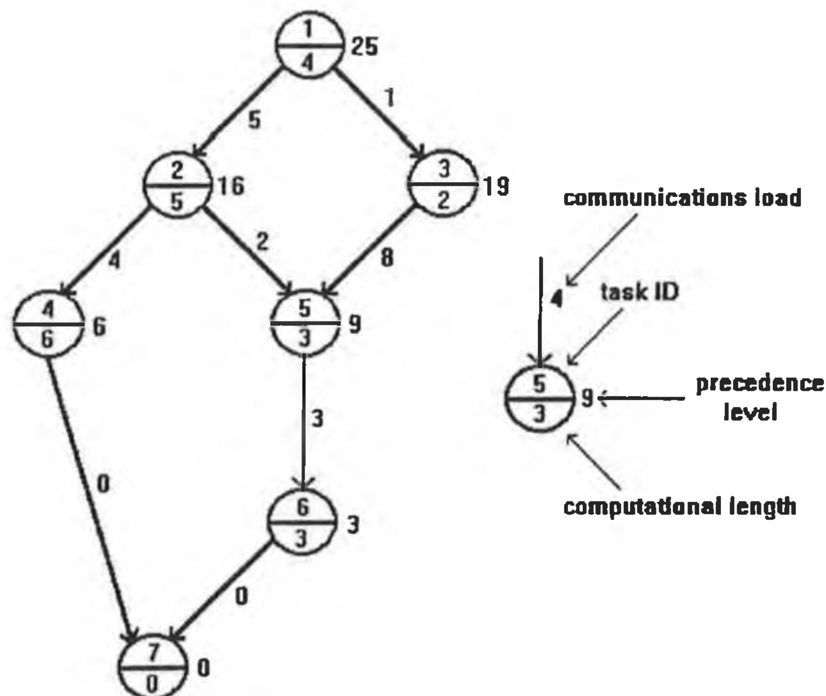


Figure 5 : Program Graph with Precedence Levels

The precedence level of a task assigns a ranking to the task in terms of the worst case path from the task, through the program graph, to the last task in the program. It is similar to the concept of positional weight used in the Ranked Positional Weight Heuristic [22], but does force the Pseudo-Dynamic Load Balancing algorithm to process the tasks in order of their precedence levels.

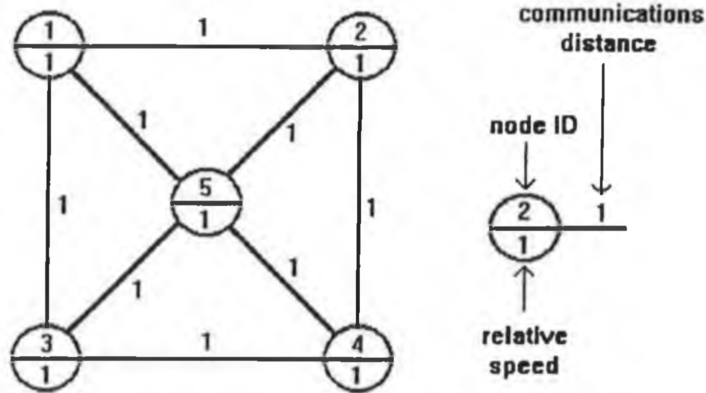
5.1.6 A Network Graph

A **network graph** is a graph where the vertices represent processing nodes and an edge between two vertices represents a communication link between the nodes represented by the two vertices. The weight of a vertex represents the relative processing speed of the associated processing node to some reference value. The weight of an edge between two vertices represents the distance or speed of the communications medium between the associated processing nodes.

The **communications distance matrix** [D] is an n x n matrix whose elements, $d_{i,j}$, are the communications distances from node i to node j. Since tasks assigned to the same processing node communicate with each other through on-node memory, $d_{i,i}$ is taken to be zero. Figure 6 shows a network graph and it associated communications distance matrix.

5.2 The Pseudo-Dynamic Allocation Heuristic

By examining the equations ([4.1] to [4.14]) which define the non-symmetric formulation of the task allocation problem, the key factors that affect the allocation can be identified. From these factors a heuristic value function can be generated which seeks to minimise the makespan of the allocated program graph.



$$[D] = \begin{bmatrix} 0 & 1 & 1 & 2 & 1 \\ 1 & 0 & 2 & 1 & 1 \\ 1 & 2 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Figure 6 : Network Graph and Communications Matrix

The objective function $\sum_i y_i$ is equivalent to $\min(y_n - y_1)$, by lemma 1. $\sum_i y_i$ reaches its minimum when each of the individual y_i reached its minimum value. Ignoring the effects of inter-task communications, and assuming that the earliest time a task on node j can execute is α_j then:

$$\min\left\{\sum_i y_i\right\} = \min\left\{\sum_j \left\{\alpha_j + \sum_k l_k\right\}\right\}; \qquad k = \{k | task\ k\ is\ assigned\ to\ node\ j\}$$

Since a processing node can only execute one task at a time, this is effectively an uncapacitated bin-packing problem of n objects into m bins such that the largest value held by any bin is minimised. Therefore, the heuristic should seek to average the computational loads over all the m processing nodes.

But inter-task communications have an opposite effect on the makespan. Consider constraint [4.23], the relaxed form of constraint [4.4]:

$$y_j - y_i \geq l_i + (1 - a_{i,j})c_{i,j} \qquad \forall i \sim j$$

Seeking to minimise each y_j implies that $a_{i,j} = 1$, i.e. that tasks i and j are allocated to the same node. Therefore the heuristic should seek to allocate communicating tasks to the same node while averaging the computational loads over all the processing nodes. For an on-line heuristic this can be restated as seeking to allocate the currently "visible" tasks onto the same nodes as their predecessors while averaging the computational loads of the tasks capable of executing at that given instant over all the processing nodes.

When these conditions conflict the allocation decision should be made by comparing the reduction in $\sum_i y_i$ that each condition contributes.

Allocating inter-communicating tasks i and j to the same node, node x, will reduce y_j by $c_{i,j}$. However, allocating task i and task j to the same processing node x will increase

$$\sum_k l_k \qquad k = \{k | task\ k\ is\ assigned\ to\ node\ j\}$$

by l_j, and increase the makespan, C_{max}, by

$$\left(\sum_k l_k - P\right) \quad \text{if } \sum_k l_k > P$$

where $P = \max\left\{\alpha_j + \sum_k l_k\right\}$

and $k=\{k \mid$ task k is allocated to processing node $x\}$.

So allocating task j to the same processing node as task i contributes $\left(P - \sum_k l_k\right)$ to the reduction in y_j if $\sum_k l_k > P$.

From constraint [4.14]

$$y_j - y_i - l_i - \sum_k (1 - a_{i,k}) c_{i,k} \geq 0 \qquad \text{if prec}(i) \geq \text{prec}(j) \text{ and } a_{i,j}=1$$

in order to reduce y_j tasks with higher precedence levels should start before other tasks. Therefore the heuristic should incorporate the difference between different tasks' precedence levels. In an on-line heuristic this can be recorded as the difference between the task's precedence level and the minimum precedence level of any currently "visible" task.

5.2.1 The Computational Load Component, C_1

The computational load component, $C_1(i,j)$, of task i allocated to node j is defined as:

$$C_l(i,j) = l_i \qquad , if\left(\frac{l_i}{s_j} + L(j) - L_p(i,j)\right) \leq P$$

$$= P - \left(\frac{l_i}{s_j} + L(j) - L_p(i,j)\right) \quad , otherwise$$

where s_j = relative processing speed of node j,

and:

(a) The **average computational load per node**, P, is the sum of the computational load of all active tasks divided by the number of processing nodes.

(b) An **active task** is a task that is either currently running on a processing node or ready to execute, i.e. has received all the messages from its predecessor tasks.

(c) The **load level of node j, $L(j)$,** is the sum of the computational loads of all the tasks currently assigned to processing node j, divided by the relative processing speed of node j.

(d) The **precedence load of task i on node j, $L_p(i, j)$,** is the sum of the computational loads of the decedents of task i that are currently assigned to processing node j, divided by the relative processing speed of node j.

The reason for calculating $L_p(i,j)$ is that there is no possibility of task i's decedents executing on processing node j when task i is ready to execute. Therefore, we need to remove their computational loads from the calculation of the effects of allocating task i to node j.

5.2.2 The Communications Load Component, C_c

If task i, allocated to node j, communicates with task k, $\{k|k \in (1...n), c_{i,k} \neq 0\}$, allocated to node l then the **communications load component,** $C_c(i,j)$, of task i allocated to node j is:

$$C_c(i,j) = -\sum_k d_{j,l} c_{i,k}$$

where $d_{j,l}$ is the communications distance from node j to node l.

5.2.3 The Precedence Component, C_p

The minimum active precedence, t_p, is the minimum precedence value of any task in the global scheduling table.

The precedence component, $C_p(i)$, of task i is given by:

$$C_p(i) = prec(i) - t_p$$

5.2.4 The Global Scheduling Table (*GST*)

At each point in a parallel or distributed computing system from which an application can be initiated a global scheduling table is maintained. This table contains a list of all the tasks currently waiting to be assigned, and their allocation heuristic values, $h(i,j)$, for each task i and processing node j in the system. Because it contains information relating to the whole parallel or distributed computing system, it is termed a **global** table. However in section 8.3.2, a method of distributing the table is described, in which each table only contains information about a restricted set of processing nodes called a neighbourhood. These neighbourhoods are connected via gateway nodes which exist in both neighbourhoods.

		P=14	t_p=61	L[1]=9	L[2]=4	L[3]=10	L[4]=14	L[5]=3
Task	holds	comp	prec level	h[i,1]	h[i,2]	h[i,3]	h[i,4]	h[i,5]
k	1	4	79	4+18-5	4+18-10	4+18-5	-4+18-8	4+18-9
e	0	6	76	-1+15-4	6+15-4	-2+15-4	-6+15-8	6+15-4
h	0	4	76	4+15-4	4+15-4	4+15-2	-4+15-12	4+15-8
i	2	3	72	3+11-1	3+11-3	3+11-3	-3+11-4	3+11-6
j	1	15	70	-10+9-3	-5+9-5	-11+9-6	-15+9-3	-4+9-2
n	1	8	69	-3+8-8	8+8-8	-4+8-10	-8+8-6	8+8-8
o	3	5	64	5+3-4	5+3-4	-1+3-2	-5+3-4	5+3-4
q	0	6	61	-1+0-3	6+0-3	-2+0-3	-6+0-3	6+0-3

comp = computational load

prec level = precedence level

communications load factor
precedence load factor
computational load factor

Figure 7 : Sample Global Scheduling Table

Figure 7 represents a sample *GST*. The table contains the following information:

(1) The minimum active precedence, t_p.

(2) The average computational load per node, P.

(3) The load level, L_j, for each node j in the parallel or distributed computing system or the neighbourhood.

(4) A list of tasks currently waiting to be scheduled.

For each task *i*, the following information is recorded:

(1) The number of holds remaining on task *i*.

(2) The allocation heuristic, *h(i,j)*, for each node *j* in the parallel or distributed computing system or the neighbourhood.

The number of holds on a task in the *GST* is initially equal to the number of predecessor tasks the task has. The first time a task receives data from any its predecessors, it decrements the number of holds on all of its successor tasks. Subsequent data transmissions received by a task have no effect on its successors in the *GST*.

5.3 Local Schedulers

Each processing node has a local scheduler that determines which of the tasks currently allocated to it will be executed at any given time. The local schedulers are simple schedulers that obey the following rules.

(1) Local schedulers execute the task with the highest precedence level which is not waiting for messages from any of its predecessor tasks.

(2) A task with a higher precedence level can preempt an executing task when it has received all its messages from its predecessor tasks.

(3) Tasks that wish to transmit data to a task that has not yet been allocated, enter a *waiting state* until the receiving task is allocated. Once the receiving task has been allocated, the waiting task leaves the *waiting state* and is available for scheduling. When the task resumes execution, the message is transmitted and the task will terminate.

(4) Local schedulers do not multi-task by time slicing tasks that are ready to execute.

If a task is transmitting messages to more than one task, then it will sequence the messages in order of decreasing precedence levels. This can be enforced by the compiler.

5.4 The Pseudo-Dynamic Load Balancing Algorithm

The Pseudo-Dynamic Load Balancing algorithm has three phases, **initialisation**, **allocation** and **update**. The state of the *GST* is also modified by two events external to the table, **message transmission completed** and **task termination**.

Phase 1 : Initialise

(a) Initialise the global scheduling table , *GST*, with the root tasks and their immediate successor tasks.

(b) Initialise the number of holds on each root task in the table to zero.

(c) Initialise the load level, $L(j)$, for each processing node to zero.

(d) Calculate the average computational load per node, P.

(e) Calculate the heuristic value function, $h(i,j)$, for each task i in the *GST* and each node j.

Phase 2 : Allocation

(a) Find the task in the *GST* with largest heuristic value, $h(i,j)$, and zero holds. If $h(i,j)$ is equal to $h(i,k)$, choose the node with the smallest load level.

(b) Allocate task i to node j. If task i is a root task, send a release message to its successor tasks.

(c) Remove task i from the *GST*.

Phase 3 : Update

(a) Add each of task i's successor tasks to the *GST* if the successor task is not already in the table.

(b) Set the number of holds on each task added to the *GST* equal to the number of predecessor tasks the task has.

(c) If any task k added to the *GST* has a precedence level, $prec(k)$, less than the minimum active precedence t_p, let $t_p = prec(k)$.

(d) Update the load level for processing node j.

$$L(j) = L(j) + \frac{l_i}{s_j}$$

(e) If t_p has changed, recalculate the precedence component C_p for each (task, node) pair in the GST. If t_p has not been changed, only update the precedence component for the tasks added to the *GST*.

(f) If the average computational load per node, P, has been changed, recalculate the computational load component, C_l, for each (task, node) pair in the *GST*. If P has not been changed, only recalculate the computational load component for each task in the *GST* at node j.

(g) Update the communications load component, C_C, for each task k, a child of task i, in the *GST* for each node l in the system.

$$C_c(k,l) = C_c(k,l) - d_{j,l} c_{i,k}$$

(h) If there are any tasks in the *GST* with zero holds, return to Phase 2(a).

Event 1 : Message transmission completed

(a) When a task receives a data message, it checks the state of its blocking flag. If the flag is not set, a release is sent to each of its successors in the *GST*. The task then sets its blocking flag. A release message decrements the number of holds on the receiving task.

(b) If any task in the *GST* receives a release that causes the number of holds on the task to be reduced to zero, the Pseudo-Dynamic Load Balancing algorithm enters the allocation phase, Phase 2(a).

Event 2 : Terminating task *i* on node *j*

(a) Update the load level, $L(j)$, for node j.

(b) Update the average computational load per node, P.

(c) If the average computational load per node, P, has been changed, recalculate the computational load component, C_l, for each (task, node) pair in the *GST*. If P has not been changed, only recalculate the computational load component for each task in the *GST* at node j.

6.0 Experimental Evaluation of Pseudo-Dynamic Load Balancing

In order to evaluate the Pseudo-Dynamic Load Balancing algorithm a program was developed to simulate the action of the algorithm allocating a set of programs onto a parallel or distributed computing system (Appendix C). Each program can be submitted to the parallel or distributed computing system at different points in time. A program is characterised by an application descriptor file which contains a directed acyclic program graph. This program graph describes the program in terms of a set of inter-communicating tasks, and specifies the computational load of each task and the communications load between tasks. The application descriptor file has the following format.

```
1   1  0  1.0   24.0    (2,1.0) (3,1.0) (4,1.0) (5,1.0)
```

Each task is specified by a line of data, and each field in a line is separated by white space characters. The first entry in the line is the task ID number. This is followed by the type of the task (1 = root task, 2 = normal task, 3 = terminal task). Next is the number of predecessors of the current task. The fourth item is the computational load of the task, which is followed by its precedence level. Finally, the line contains a successor list. Each entry in the successor list contains the ID of the task to which the current task will transmit data, and the communications load between these two tasks.

The parallel or distributed computing system is characterised by a network topology file which contains a network graph in the following format. The first line specifies the number of processing nodes m in the network. The next m lines contain the relative speed of each processing node. Finally, the last m lines of the network topology file contains the communications distance matrix D. The ith row (line) of the communications matrix specifies the communications distance from ith node to the other processing nodes. Each field in the final m lines is separated by white space characters. Processing nodes which are not connected by a path in the network graph have an infinite (or an arbitrarily large) communications distance. The numbering of the processing nodes is important, because if the allocation heuristic for a task is the same for k processing nodes, and each of these processing nodes has the same load level L, then the task is assigned to the processing node with the lowest cardinal node ID. If this scenario was true for two tasks which have a common successor task, to which both tasks communicate, then the successor will not be allocated to the same processing node as one of its parents. In order to minimise the resulting communications time the two parents should be allocated to two processing nodes which are close to each other in terms of communications distance. This can be accomplished by numbering processing nodes such that processing node i is closer, in terms of communications distance, to processing nodes $(i-k),\ldots,(i+k)$, for some arbitrary value of k.

The simulation software uses the Pseudo-Dynamic Load Balancing algorithm to allocate tasks to processing nodes, and then simulates the execution of these tasks on the parallel or distributed computing

network. The simulation software, using the allocation as it is generated, calculates when each task begins execution, receives and transmits its data, preempts other tasks, enters and leaves a waiting state and when the task terminates. From these calculations a profile of the processing node activity is generated and the program completion time is calculated.

If two program graphs start at the same, then the root tasks, and the successors of the root tasks, from each program graph are included into the *GST* and the algorithm would run as normal. If program graph P2 starts at time t_2 and program graph P1 starts at time t_1, where $t_2 > t_1$, then in order to assign the appropriate precedence to the tasks in P2, the precedence load factors for the tasks in P1 are modified by:

$$(t_2 + mp_{P2} - r)$$

and r is updated to

$$r = t2 + mp_{P2}$$

where mp_{Px} is the maximum precedence level in program graph Px and r is initialised to zero.

This equalises the precedence load factors of the tasks of P1 in the *GST* at time t_2 with the root tasks of P2. If there are more than two programs, this technique of equalising the precedence levels is applied to all the programs which still have tasks in the *GST*.

6.1 Experimental Test Results

6.1.1 Pseudo-Dynamic Load Balancing vs. Relaxed Non-Symmetric Mathematical Formulation

The relaxed non-symmetric mathematical formulation of the task allocation problem developed in chapter 4 generates an upper bound on the execution time of a program on a homogeneous parallel or distributed computing system arranged in a bus topology. These upper bounds generated by the relaxed non-symmetric mathematical formulation will be compared with the allocations produced by Simulated Annealing [25], a standard combinatorial optimisation technique, in order to verify the strength of the upper bounds. Then the allocations produced by the Pseudo-Dynamic Load Balancing algorithm will be compared with the upper bounds to test the allocations produced by the Pseudo-Dynamic Load Balancing algorithm.

Since the non-symmetric formulation is still an integer linear program, the solution time for the optimal task allocation can still suffer the effects of a combinatorial expansion of the solution space. To trade excessive solution time against allocation optimality, two Sciconic run streams were written. Run stream 1 terminates the branch and bound search when then reduction in the objective function between two successive valid integer solutions, found by the branch and bound procedure, was less than 1%. Run stream 2 terminates the branch and bound search if the improvement in the objective function between two successive valid integer solutions, found by the branch and bound procedure, is less than 1% on two

occasions. Appendix B.5 contains the source for both run stream 1 and run stream 2.

In order to compare the results of the Pseudo-Dynamic Load Balancing algorithm with the upper bound determined by the relaxed non-symmetric mathematical formulation, a module from the *Spatial Coherence Method* (18 tasks), used in atmospheric analysis [24], was chosen as a problem instance. Figure 8 shows the program graph for this module.

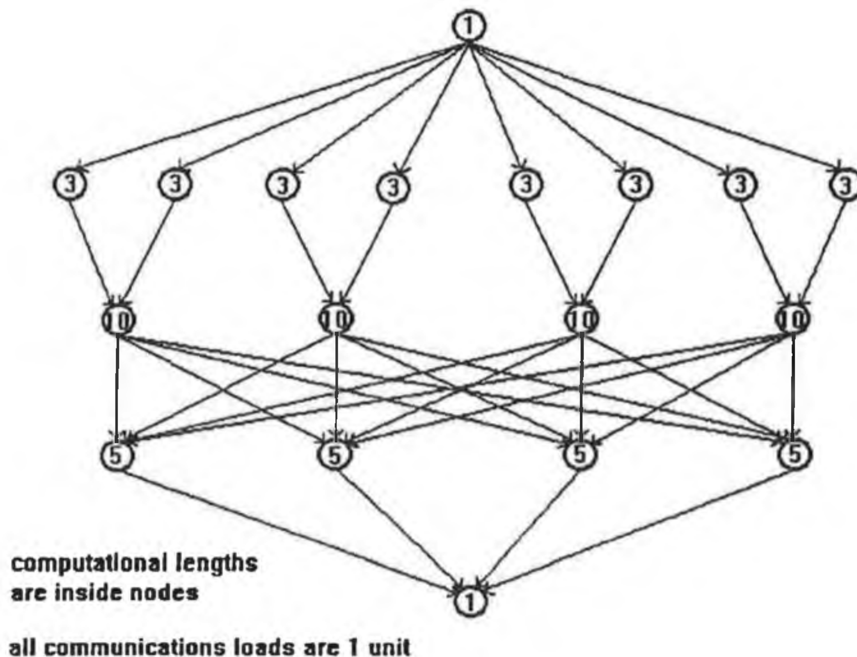


Figure 8 : Program Graph for Atmospheric Analysis Module

Appendix B.4 contains the problem data file for this instance which the matrix generator, MG, used to generate the standard MPS file for the Sciconic program.

Table 2 and Table 3 contains the results produced, in addition to the solution time (hours:minutes:seconds) on a VAXSTATION 3100 , for run stream 1 and run stream 2 respectively.

	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
program run time	110	64	51	42	42	41	42
solution time	0:35:21	1:35:13	0:47:51	1:19:46	0:25:42	0:24:12	0:38:41

Table 2: Run stream 1 results

	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
program run time	73	67	42	41	43	38	40
solution time	1:16:00	4:01:32	(a)	2:00:51	3:07:25	1:43:49	1:43:49

Table 3: Run stream 2 results

Note (a): Sciconic program terminated after a limit of 12 hours of CPU time was reached.

Run stream 2 generally produces better allocations at the expense of solution time. The Pseudo-Dynamic Load Balancing algorithm was applied to the same problem instance. Its results, and those generated by applying Simulated Annealing to the same problem instance, are combined with the results of Tables 2 and 3 in order to:

(a) Compare the performance of the Pseudo-Dynamic Load Balancing algorithm against the upper bounds set by the relaxed non-symmetric mathematical formulation.

(b) Measure the quality of the upper bounds set by the relaxed non-symmetric mathematical formulation.

The values contained in Table 4 and 5 are program run times.

The results in Tables 4 and 5 show that the relaxed non-symmetric mathematical formulation produces strong upper bounds on the execution time of a program graph allocated to a homogeneous processing network. Tables 4 and 5 also shows that the Pseudo-Dynamic Load Balancing algorithm, which is an on-line algorithm, produces allocations which are superior to those generated by the relaxed non-symmetric mathematical formulation.

	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
relaxed non-symmetric formulation	73	64	42	41	42	38	40
pseudo-dynamic load balancing	63	57	38	40	40	40	37
simulated annealing	59	50	38	37	37	37	37

Table 4: Comparison of relaxed non-symmetric mathematical formulation, Pseudo-Dynamic Load Balancing and simulated annealing.

	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
relaxed non-symmetric formulation	24%	28%	11%	11%	14%	3%	8%
pseudo-dynamic load balancing	7%	14%	0%	8%	8%	8%	0%

Table 5: Relaxed Non-Symmetric Formulation and Pseudo-Dynamic Load Balancing Allocations as a percentage of Simulated Annealing Allocations

6.1.2 Pseudo-Dynamic Load Balancing vs. Simulated Annealing and Tabu Search

The Pseudo-Dynamic Load Balancing algorithm was evaluated against two common methods for combinatorial optimisation, Simulated Annealing [25] and Tabu Search [26], using the simulation software. Two test cases were used, whose program graphs have different structures. The program graphs are shown in figures 9 and 10. Figure 9 is the program graph of a computer vision program (31 tasks) initially presented by Kunii, Nishimura and Noma [19] which was used to evaluate the Maximum (k-1) Sum algorithm in section 3.5.3. Figure 10 is a program graph for Gaussian elimination (55 tasks) presented by Darte [13]. The Gaussian elimination program graph exhibits regular repetitive sub-structures.

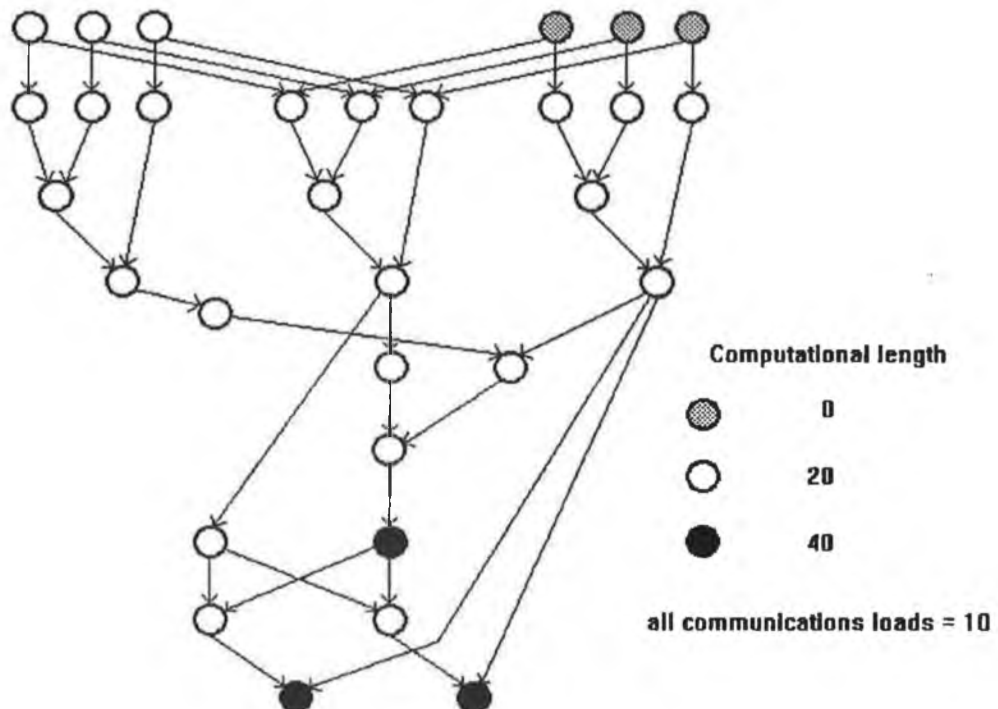


Figure 9: Program Graph for Computer Vision Problem

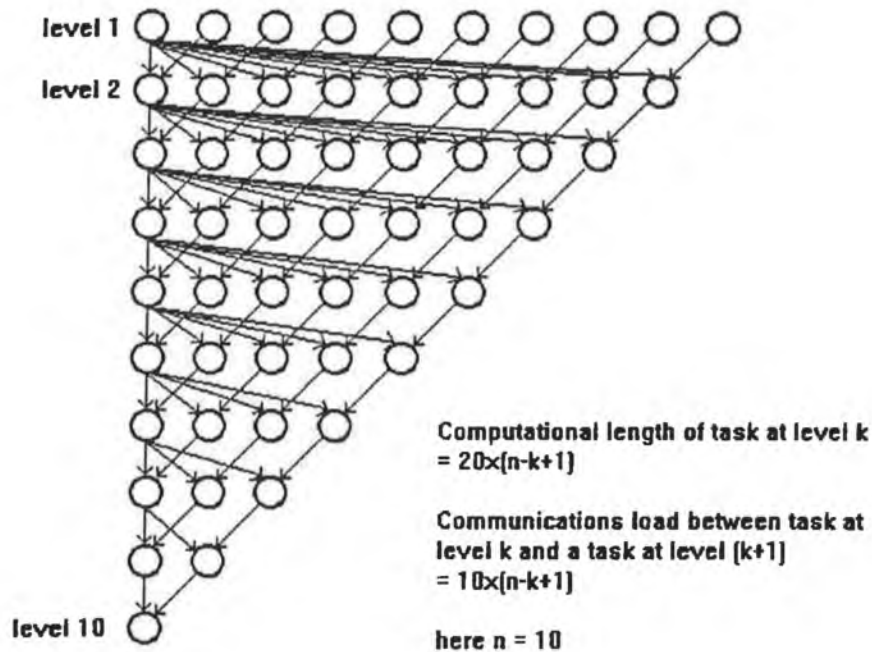


Figure 10: Program Graph for Gaussian Elimination

The tests using Simulated Annealing had a starting temperature of 0.9 and a cooling schedule of 0.8. These values were chosen to produce allocations with lower execution times at the expense of the time taken for the experiment to complete. At each temperature, up to (25 * number of nodes * number of tasks) allocations were generated, with each allocation being a single task move perturbation of the previous allocation. The annealing process at the current temperature was terminated when (25 * number of nodes * number of tasks) allocations were tested, or when (10 * number of tasks) allocations were found which reduced the execution time of the test program at the current temperature.

The size of the tabu list in experiments using the Tabu Search technique was set to (number of tasks / 3). Other experiments with the size of the tabu list depending on the number of processing

nodes, as well as the number of tasks, did not produce any significantly better results. The Tabu Search used an aspiration condition that allowed an allocation with a lower execution time despite the fact that the perturbation that produced the allocation was in the tabu list. The Tabu Search chose the best of 20 randomly generated single task moves as the perturbation, and performed up to (20 * number of tasks * number of nodes) perturbations.

These program graphs were allocated onto a homogeneous bus network with 2 to 8 processing nodes. Tables 6 and 7 summarises the results produced by Simulated Annealing, Tabu Search and Pseudo-Dynamic Load Balancing. Each entry in tables 6 and 7 is the run-time for the program allocated to the specified number of processing nodes.

	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
Pseudo-Dynamic	490	380	370	320	320	320	320
Simulated Annealing	430	380	350	330	320	320	320
Tabu Search	510	410	370	370	370	360	360

Table 6 : Experimental Results for Computer Vision Problem

	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
Pseudo-Dynamic	6010	4960	4210	4069	4089	3669	3360
Simulated Annealing	6540	5010	4090	3690	3550	3550	3480
Tabu Search	6360	5100	4530	4130	3880	3520	3550

Table 7 : Experimental Results for Gaussian Elimination Problem

From the data in Tables 6 and 7, we can see the Pseudo-Dynamic Load Balancing algorithm produces allocations that are comparable to those generated by Simulated Annealing and Tabu Search. Both of these techniques have received considerable academic scrutiny and are considered excellent methods for combinatorial optimisation. Simulated Annealing and Tabu Search are static approaches to load balancing which require several hours processing to generate a near-optimal allocation. The Pseudo-Dynamic Load Balancing algorithm is an on-line algorithm that produces similar quality allocations in real time.

7.0 Worst Case Analysis of Pseudo-Dynamic Load Balancing

Algorithms can be analysed in terms of their average performance, based on mathematical analysis or empirical studies, or in terms of their worst case performance. The worst case ratio, *R*, of load balancing technique *H* is defined as:

$$R^{H} = \lim\sup\left\{\frac{C^{H}(P)}{C*(P)} \quad : \quad P \in Set\ of\ possible\ programs\right\}$$

where $C^{H}(P)$ denotes the makespan of the allocation of program *P* produced by technique *H*, and $C*(P)$ denotes the corresponding makespan in some optimal schedule.

This chapter presents an analysis of the Pseudo-Dynamic Load Balancing technique to determine its worst case ratio, R^{PD}. The structure of the program graph with the worst case allocation under Pseudo-Dynamic Load Balancing is derived, in addition to its optimal allocation. The optimal allocation of the worst case structure will be shown to have a lower makespan than any other structure with similar computational and communication loads, and therefore no other program graph structure can give rise to a higher worst case ratio.

7.1 Worst Case Program Graph Structure

The Pseudo-Dynamic Load Balancing algorithm chooses task to processing node allocations based on the value of the heuristic function of the tasks available for allocation. This function has three components, a computational load component, a communications load component and a precedence component. When the heuristic function value is the same for all the tasks available for allocation, then the first available task is assigned to the processing node with the lowest load level. If more than one processing node have the same load level, then the task is allocated to one of these processing nodes, usually the processing node with the lowest cardinal number.

The worst case program graph structure occurs in Pseudo-Dynamic Load Balancing when at each stage in allocating a program graph of n tasks to a set of m processing nodes, the three components of the heuristic function value are equal for all the tasks available for allocation. When this occurs task n is allocated to processing node (n mod m). If, in addition, each one of the tasks allocated to a different processing node communicates with the same task, then the inter-node communications are maximised. It is under these conditions that the makespan is maximised, since:

(i) If predecessor tasks of the tasks available for allocation, except root tasks, had equal heuristic components and the available tasks do not have equal heuristic components, then some of the successor tasks will have different heuristic

function values and the tasks will be allocated in such a way as to reduce the inter-node communications or computational load level, or both. This will result in some tasks having a lower completion time and hence reduce the makespan.

(ii) If the tasks available for allocation are all root tasks, then for successor tasks to have the same heuristic function values, it is only necessary for the heuristic function values and the precedence components to be equal to ensure that the successor tasks have the same precedence component. However, in a non-homogeneous communications environment, where the time to transmit a unit message between all processing nodes is not the same, the root tasks with larger communications load components will be allocated to the same processing node as their successor task. This will reduce the completion time of these root tasks and may reduce the completion time of the successor task, leading to a reduced makespan.

Therefore the worst case program graph structure for Pseudo-Dynamic Load Balancing is the graph structure shown in figure 11.

The worst case program graph structure consists of n tasks allocated among m processing nodes, and has k levels were:

$$k = \left\{ k \in N \,\middle|\, \frac{1 - m^{\lceil k \rceil}}{1 - m} \geq n \right\}$$

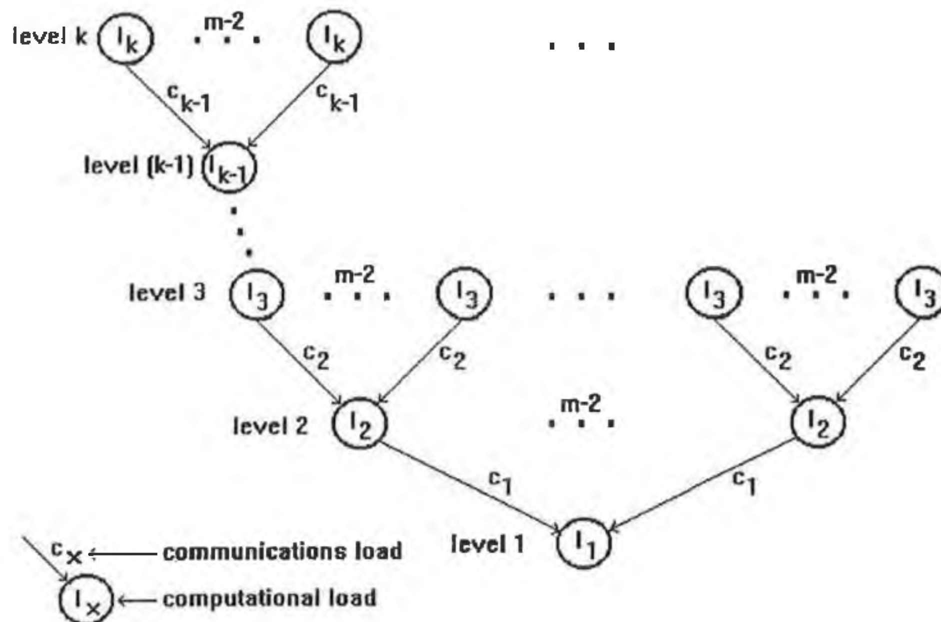


Figure 11 : Worst Case Program Graph Structure

Level i has $m^{(i-1)}$ tasks of computational load l_i. The communications load between a task on level $(i+1)$ and a task on level i is c_i. In each level i, there are $m^{(i-2)}$ groupings of m tasks that communicate with the same task on level $(i-1)$.

In this worst case structure task j will be allocated to node (j mod m) since each task on level i :

(i) Has the same computational load, communications load and precedence level.

(ii) Communicates with the same task on level *(i-1)* as *(m-1)* other tasks on level *i*.

(iii) Each level *i*, except level 1, has a multiple of *m* tasks, and therefore an equal number of off-node and on-node task to task data transfers. Hence no processing node will be available to process any task on level *(i-1)* before the other *(m-1)* processing nodes.

The time taken to execute and transmit data for all the tasks on level *k* is the time taken until the last task on level *k* is completed. Since all the tasks are similar, this is the time for any one processing node to complete all the tasks allocated to it. A processing node will have $m^{(k-2)}$ tasks allocated to it and every *m-th* one of these tasks will communicate with a task assigned to the same node. The remaining tasks will communicate with tasks allocated to different processing nodes.

The time taken to complete level *k* is:

$$\begin{aligned} & m^{(k-3)} l_k + (m-1) m^{(k-3)} (l_k + c_{k-1}) \\ = \; & m^{(k-3)} \{ l_k + (m-1)(l_k + c_{k-1}) \} \end{aligned}$$

The time taken to complete the tasks on level *(k-1)* is the time taken for a processing node to complete all the $m^{(k-3)}$ tasks from level *(k-1)* allocated it. The time taken to receive data from the tasks on level *k*, execute the tasks on level *(k-1)* and transmit data to the tasks on level *(k-2)* is:

$$m^{(k-4)}\{(m-1)c_{k-1}+l_{k-1}\}+(m-1)m^{(k-4)}\{(m-1)c_{k-1}+l_{k-1}+c_{k-2}\}$$
$$= m^{(k-3)}\{(m-1)c_{k-1}+l_{k-1}\}+(m-1)m^{(k-4)}c_{k-2}$$

In general the time taken to complete the tasks on level *i* is:

$$m^{(i-2)}\{(m-1)c_i+l_i\}+(m-1)m^{(i-3)}c_{i-1}$$

Therefore, the total time taken to execute a program graph exhibiting the worst case structure is:

$$C^{PD}=m^{(k-3)}\{l_k+(m-1)(l_k+c_{k-1})\}+\sum_{i=3}^{k-1}\{m^{(i-2)}\{(m-1)c_i+l_i\}+(m-1)m^{(i-3)}c_{i-1}\}$$
$$+\{(m-1)c_2+l_2\}+\{(m-1)c_1+l_1\}$$

7.2 Optimal Allocation for Worst Case Structure

To reduce the makespan, a multiple of *m* collections must be found which reduce the inter-node communications time by more than the corresponding maximum increase in computation time for any of the *m* processing nodes.

Consider allocating tasks $\{(t-1)m + 1, \ldots , tm\}$ to processing node $(t \bmod m)$.

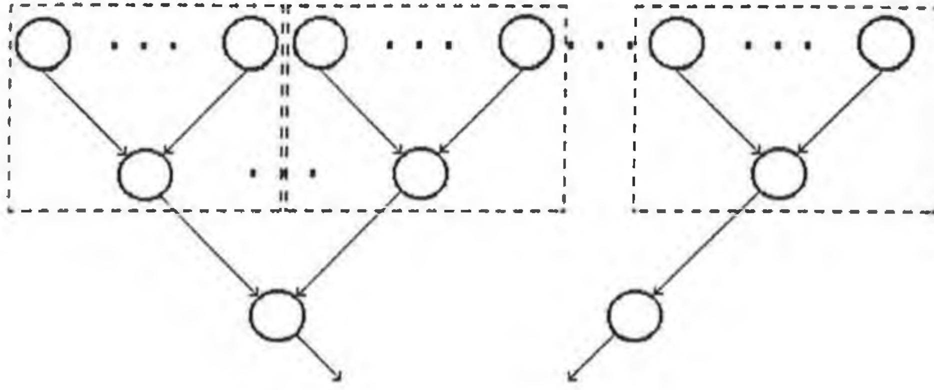


Figure 12: One Stage in the Transformation to an Optimal Allocation

This effectively removes level k and transforms level $(k-1)$ into a level with $m^{(k-2)}$ tasks of computational load $(ml_k + l_{k-1})$. The makespan for the transformed graph, with $k-1$ levels, is:

$$C_{k-1}^{PD} = m^{(k-4)}\{(ml_k + l_{k-1}) + (m-1)((ml_k + l_{k-1}) + c_{k-2})\}$$
$$+\sum_{i=3}^{k-2}\{m^{(i-2)}\{(m-1)c_i + l_i\} + (m-1)m^{(i-3)}c_{i-1}\} + \{(m-1)c_2 + l_2\} + \{(m-1)c_1 + l_1\}$$

and the reduction in the makespan over C_k^{PD} is:

$$C_k^{PD} - C_{k-1}^{PD} = m^{(k-3)}\{l_k + (m-1)(l_k + c_{k-1})\} + m^{(k-3)}\{(m-1)c_{k-1} + l_{k-1}\} + (m-1)m^{(k-4)}c_{k-2}$$
$$- m^{(k-4)}\{(ml_k + l_{k-1}) + (m-1)((ml_k + l_{k-1}) + c_{k-2})\}$$

$$= l_k\left(m^{(k-3)} + m^{(k-2)} - m^{(k-3)} - m^{(k-3)} - m^{(k-2)} + m^{(k-3)}\right)$$
$$+ l_{k-1}\left(m^{(k-3)} - m^{(k-4)} - m^{(k-3)} + m^{(k-4)}\right)$$
$$+ c_{k-1}\left(m^{(k-2)} - m^{(k-3)} + m^{(k-2)} - m^{(k-3)}\right)$$
$$+ c_{k-2}\left(m^{(k-3)} - m^{(k-4)} - m^{(k-4)} + m^{(k-4)}\right)$$

$$= 2\left(m^{(k-2)} - m^{(k-3)}\right)c_{k-1} + \left(m^{(k-3)} - m^{(k-4)}\right)c_{k-2}$$
$$= (m-1)m^{(k-4)}(2mc_{k-1} + c_{k-2})$$

Therefore the reduction in makespan is due to a reduction in inter-node communications time alone. There is no counteracting increase in computation time. Therefore repeated applications of this transform will continually reduce the makespan until the maximum level in the transformed program graph is 2. The transform can not be applied again because level 1 does not have a multiple of *m* tasks (it has only one task).

Therefore, the program graph can be transformed to:

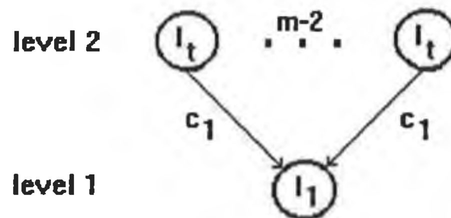


Figure 13: Level 2 Program Graph

where $l_t = \sum_{i=2}^{k} \{m^{(i-2)} l_i\}$

The processing node to which the level 1 task is allocated can start to receive data immediately after it has finished executing its level 2 tasks. Because of this, another transform can be performed, assigning all the tasks to the same processing node, if:

$$l_t + (m-1)c_1 + l_1 \geq ml_t + l_1$$
$$\Rightarrow \quad (m-1)c_1 \geq (m-1)l_t$$
$$\Rightarrow \quad c_1 \geq l_t$$

The minimum makespan for the worst case structure is:

(a) If $c_1 \geq l_t$, then

$$C^* = \sum_{i=1}^{k} \{ m^{(i-1)} l_i \}$$

(b) If $c_1 < l_t$, then

$$C^* = \sum_{i=2}^{k} \{ m^{(i-2)} l_i \} + (m-1)c_1 + l_1$$

Program graphs, with similar computational and communication loads, can not have a lower minimum makespan. This is because the optimal allocation of Pseudo-Dynamic Load Balancing's worst case structure initially divided *(n-1)* tasks equally among the *m* processing nodes minimising the largest computational load of any processing node, while, at the same time, minimising communications to *m* messages of length c_1. Testing if $c_1 < l_t$ determines whether or not the makespan can be reduced if all tasks are allocated to the same node.

7.3 Worst Case Ratio, *R*, for Pseudo-Dynamic Load Balancing

Given a worst case program graph structure, shown in figure 11, with n tasks, to be allocated among m processing nodes, with the n tasks divided into k levels, $k = \left\{ k \in N \,\middle|\, \frac{1 - m^{\lceil k \rceil}}{1 - m} \geq n \right\}$, in which the computational load of a task on level i is l_i and the communications load between a task on level $(i+1)$ and a task on level i is c_i, then the worst case scheduling ratio, R^{PD}, is:

Case (a): If $c_1 \geq \sum_{i=2}^{k} \{ m^{(i-2)} l_i \}$, then

$$R^{PD} = \frac{C^{PD}}{C^*} = \frac{m^{(k-3)}\{l_k + (m-1)(l_k + c_{k-1})\} + \sum_{i=3}^{k-1}\{m^{(i-2)}\{(m-1)c_i + l_i\} + (m-1)m^{(i-3)}c_{i-1}\} + \{(m-1)c_2 + l_2\} + \{(m-1)c_1 + l_1\}}{\sum_{i=1}^{k}\{m^{(i-1)} l_i\}}$$

Case (b): If $c_1 < \sum_{i=2}^{k} \{ m^{(i-2)} l_i \}$, then

$$R^{PD} = \frac{C^{PD}}{C^*} = \frac{m^{(k-3)}\{l_k + (m-1)(l_k + c_{k-1})\} + \sum_{i=3}^{k-1}\{m^{(i-2)}\{(m-1)c_i + l_i\} + (m-1)m^{(i-3)}c_{i-1}\} + \{(m-1)c_2 + l_2\} + \{(m-1)c_1 + l_1\}}{\sum_{i=2}^{k}\{m^{(i-2)} l_i\} + (m-1)c_1 + l_1}$$

7.4 Estimating the Worst Case Ration, R^{PD}, for a Given Graph

Given a program graph with n tasks to be allocated among the m processing nodes of a parallel or distributed computing system, let:

$l_i = \max_{1 \le j \le n}(l_j)$ = maximum computational load

$c_i = \max_{1 \le j,k \le n}(c_{j,k})$ = maximum communications load

$$k = \left\{ k \in N \,\middle|\, \frac{1 - m^{\lceil k \rceil}}{1 - m} \ge n \right\}$$

Then the worst case makespan can be written as:

$$C^{PD} = m^{(k-3)}\{l + (m-1)(l+c)\} + \sum_{i=3}^{k-1}\{m^{(i-2)}\{(m-1)c + l\} + (m-1)m^{(i-3)}c\} + \{(m-1)c + l\} + \{(m-1)c + l\}$$

$$= m^{(k-3)}\{l + (m-1)(l+c)\} + \sum_{j=0}^{k-4}\{m^{(j+1)}\{(m-1)c + l\} + (m-1)m^{j}c\} + 2\{(m-1)c + l\}$$

$$= m^{(k-3)}\{l + (m-1)(l+c)\} + \{(m-1)c + l\}\sum_{j=0}^{k-4} m^{(j+1)} + (m-1)c\sum_{j=0}^{k-4} m^{j} + 2\{(m-1)c + l\}$$

$$= m^{(k-2)}l + (m-1)cm^{(k-3)} + \{(m-1)c + l\}\sum_{j=1}^{k-3} m^{j} + (m-1)c\sum_{j=0}^{k-4} m^{j} + 2\{(m-1)c + l\}$$

$$= m^{(k-2)}l + \{(m-1)c + l\}\sum_{j=1}^{k-3} m^{j} + (m-1)c\sum_{j=0}^{k-3} m^{j} + 2\{(m-1)c + l\}$$

$$= m^{(k-2)}l + \{(m-1)c + l\}\sum_{j=0}^{k-3} m^{j} + (m-1)c\sum_{j=0}^{k-3} m^{j} + \{(m-1)c + l\}$$

$$= m^{(k-2)}l + \{2(m-1)c\}\sum_{j=0}^{k-3} m^j + l\sum_{j=0}^{k-3} m^j + \{(m-1)c + l\}$$

$$= m^{(k-2)}l + \{2(m-1)c + l\}\sum_{j=0}^{k-3} m^j + \{(m-1)c + l\}$$

But since $\sum_{i=p}^{N} a^i = \frac{a^p - a^{(N+1)}}{1-a}$ then,

$$C^{PD} = m^{(k-2)}l + \{2(m-1)c + l\}\left\{\frac{1-m^{(k-2)}}{1-m}\right\} + \{(m-1)c + l\}$$

$$= \frac{(1-m)m^{(k-2)} + \{2(m-1)c + l\}\{1-m^{(k-2)}\} + (1-m)\{(m-1)c + l\}}{1-m}$$

$$= \frac{l\left(-m^{(k-1)} - m + 2\right) + c\left(-2m^{(k-1)} + 2m^{(k-2)} - m^2 + 4m - 3\right)}{1-m}$$

$$= \frac{l\left(-m^{(k-1)} - m + 2\right) + c\left(-2(m-1)m^{(k-2)} - m^2 + 4m - 3\right)}{1-m}$$

Therefore the worst case ratio, R^{PD}, is given by:

Case (a): If $c \geq \sum_{i=2}^{k}\{m^{(i-2)}l\} = l\left\{\frac{1-m^{(k-1)}}{1-m}\right\}$, then

$$C^* = \sum_{i=1}^{k}\{m^{(i-1)}l\} = \sum_{j=0}^{k-1}\{m^j l\} = l\left\{\frac{1-m^k}{1-m}\right\}$$

$$\Rightarrow R^{PD} = \frac{C^{PD}}{C^*} = \frac{l\left(m^{(k-1)} + m - 2\right) + c\left(2(m-1)m^{(k-2)} + m^2 - 4m + 3\right)}{m^k - 1}$$

Case (b): If $c < \sum_{i=2}^{k}\{m^{(i-2)}l\} = l\left\{\frac{1-m^{(k-1)}}{1-m}\right\}$, then

$$C^* = \sum_{i=2}^{k}\{m^{(i-2)}l\} + (m-1)c + l$$

$$= \sum_{j=0}^{k-2}\{m^{j}l\} + (m-1)c + l$$

$$= l\left\{\frac{1-m^{(k-1)}}{1-m}\right\} + (m-1)c + l$$

$$= \frac{l\left(1-m^{(k-1)}\right) + (1-m)(m-1)c + (1-m)l}{1-m}$$

$$= \frac{l\left(-m^{(k-1)} - m + 2\right) + c\left(-m^2 + 2m - 1\right)}{1-m}$$

Therefore,

$$\Rightarrow R^{PD} = \frac{C^{PD}}{C^*} = \frac{l\left(m^{(k-1)} + m - 2\right) + c\left(2(m-1)m^{(k-2)} + m^2 - 4m + 3\right)}{l\left(m^{(k-1)} + m - 2\right) + c\left(m^2 - 2m + 1\right)}$$

7.5 Worst Case Ratio, R^{PD}, as a Function of the Communications to Computations Ratio, a

If the maximum communications load is c and the maximum computational load is l, then let $c=al$. The worst case ratio, in terms of a is then:

Case (a): If $a \geq \dfrac{1-m^{(k-1)}}{1-m}$, then

$$R^{PD} = \frac{m^{(k-1)}(1+2a) - 2am^{(k-2)} + am^2 + m(1-4a) + (3a-2)}{m^k - 1}$$

Case (b): If $a < \dfrac{1-m^{(k-1)}}{1-m}$, then

$$R^{PD} = \frac{m^{(k-1)}(1+2a) - 2am^{(k-2)} + am^2 + m(1-4a) + (3a-2)}{m^{(k-1)} + am^2 + m(1-2a) + (a-2)}$$

7.6 Asymptotic Bounds on Worst Case Ratio, R^{PD}

(a) Let $m \to \infty$. Since the worst case program graph structure is a balanced tree with branching factor *m*, then as $m \to \infty$, $k \to 2$, since *k* is the depth of the tree.

If $a \geq \dfrac{1-m^{(k-1)}}{1-m} = \dfrac{1-m}{1-m} = 1$, then:

$$R^{PD} = \frac{am^2 + m(2-2a) + (a-2)}{m^2 - 1}$$

Therefore as $m \to \infty$, $R^{PD} \to \dfrac{am^2}{m^2} = a$.

If $a < \frac{1-m^{(k-1)}}{1-m} = \frac{1-m}{1-m} = 1$, then:

$$R^{PD} = \frac{m^{-1}(1+2a) - 2a + am^2 + m(1-4a) + (3a-2)}{m^{-1} + am^2 + m(1-2a) + (a-2)}$$

Therefore as $m \to \infty$, $R^{PD} \to \frac{am^2}{am^2} = 1$.

(b) As $n \to \infty$, $k \to \infty$ since more tasks must be accommodated in the worst case program graph, which is a balanced tree with branching factor m. Therefore $a < \frac{1-m^{(k-1)}}{1-m}$ and

$$R^{PD} = \frac{m^{(k-1)}(1+2a) - 2am^{(k-2)} + am^2 + m(1-4a) + (3a-2)}{m^{(k-2)} + am^2 + m(1-2a) + (a-2)}$$

$$\Rightarrow R^{PD} \approx (1+2a) - \frac{2a}{m}$$

$$\Rightarrow R^{PD} \approx 1 + 2a\frac{(m-1)}{m}$$

Therefore as $n \to \infty$, $(n-m) >> 1$ $, R^{PD} \to 1 + 2a$.

8.0 Conclusions

8.1 Goals

Parallel and distributed processing form one of the corner stones of the future of computing. A popular paradigm of parallel and distributed processing is concurrent processing, in which a set of inter-communicating sequential tasks cooperatively solve a problem. Two fundamental problems prevent parallel and distributed processing, particularly concurrent processing, from entering the mainstream of computing:

(a) A language that expresses the parallelism inherent in the problem without reference to the computing environment in which the program will be executed.

(b) A method, which is independent of the user, of effectively exploiting the expressed parallelism by distributing the components of the program among the available processing nodes.

This thesis examined the second problem, the load balancing problem. In order to solve the load balancing problem the aims of this thesis were to:

(a) Analysis the fundamental structure of the load balancing problem.

(b) Design algorithms which effectively allocate the n inter-communicating tasks, that form the program, among the available m processing nodes.

8.2 Results and Achievements

The Elastic Force algorithm, presented in chapter 3, is a static load balancing algorithm for a program of n inter-communicating tasks with equal computational loads. It draws an analogy with a physical system of object connected by elastic forces. Such a physical system reaches equilibrium when the energy stored in the system is minimised. In the Elastic Force algorithm the inter-task communications were represented as distant-dependent elastic forces. The Elastic Force algorithm performs a direct search of the solution space seeking a solution with minimum inter-task communications. Since this minimises the time the processing nodes spend processing tasks and communicating between tasks, this maximises the throughput of tasks through the parallel or distributed computing system.

The Maximum (k-1) Sum algorithm is a heuristic static load balancing technique that was derived from the Elastic Force algorithm. It also seeks to maximise the throughput of the parallel or distributed computing system. The Maximum (k-1) Sum algorithm is computationally efficient, $O(mn^2)$, and produces high quality allocations when the program graph does not contain repetitive

sub-structures with chains of communicating tasks. When applied to a real world computer vision program, and the number of processing nodes was greater than 5 (Table 1), the Maximum (k-1) Sum algorithm produced better allocations than those produced by Darte's or Sarkar's algorithms.

The load balancing problem was analysed mathematically in order to understand the fundamental structure of the problem, with a view to developing an on-line algorithm for load balancing. The mathematical formulation presented in this thesis is a non-symmetric formulation of the load balancing problem that minimises the makespan. For every s non-empty subsets of the n tasks to be allocated to m processing nodes, this formulation reduces the solution space by removing the $\frac{m!}{(m-s)!}$ identical allocations since the processing nodes can be numbered arbitrarily. Removing such symmetries can significantly reduce the size of the solution space. The non-symmetric mathematical formulation of the load balancing problem shows that the structure of the problem is not only discrete, but also quadratic. A new technique for relaxing the formulation into a mixed integer-linear programming problem was developed. This relaxed non-symmetric formulation of the problem can be used to generate upper bounds on the load balancing problem. When applied to the program graph of an atmospheric analysis module, and compared against the allocations produced by a standard combinatorial optimisation technique, Simulated Annealing, the upper bounds generated by the relaxed non-symmetric formulation were shown to be strong (table 5).

Based on the non-symmetric formulation of the load balancing problem, the Pseudo--Dynamic Load Balancing algorithm was developed. This algorithm is an on-line heuristic algorithm that seeks to allocate the tasks of a program among the available processing nodes in order to minimise the makespan of the allocated program. Key elements of the mathematical formulation were analysed and used to generate a heuristic function that determines the on-line allocation of tasks to processing nodes. This algorithm was tested by developing a simulation package that uses the Pseudo-Dynamic Load Balancing algorithm to allocate tasks to processing nodes. The behaviour of the allocated tasks was recorded and compared with the relaxed non-symmetric mathematical formulation and two standard static combinatorial optimisation techniques, Simulated Annealing and Tabu Search, which were optimised for producing near-optimal allocations at the expense of solution time. When tested against the upper bounds set by the non-symmetric mathematical formulation for the atmospheric analysis module, the allocations generated by Pseudo-Dynamic Load Balancing were lower than the upper bounds, with one exception. On two other test cases the allocations generated by Pseudo-Dynamic Load Balancing were better than those generated by Tabu Search, and very close to those generated by Simulated Annealing. The allocations generated by the Pseudo-Dynamic Load Balancing algorithm were less than 115% of the Simulated Annealing allocation and generally less than 103%. The major advantages of Pseudo-Dynamic Load Balancing is that it is an on-line technique, generating allocations in real-time, and capable of reacting to changes in the parallel or distributed computing system.

The Pseudo-Dynamic Load Balancing algorithm was then analysed to determine its worst case performance. The structure of the program graph which gives rise to this worst case performance was derived and its optimal allocation was shown to be the optimal for any similar set tasks. From this the worst case scheduling ratio, R^{PD}, was derived. If the ratio of the maximum communications load to maximum computational load is given as a, then the upper bound on R^{PD} is $(1+2a)$. Unlike previous worst case scheduling ratio analysis [21] in which the effects of inter-task communications were ignored, when inter-task communications are included in the model, the ratio of the maximum communications load to maximum computational load is the significant factor that influences on the worst case performance. The worst case program graph structure is very symmetrical in structure and can be easily detected by the compiler module/preprocessor which generates the program graph. If small irregularities are introduced to the program graph by this module or preprocessor then the scheduling ratio can be significantly reduced.

8.3 Topics for further Research

8.3.1 Enhancements to the Pseudo-Dynamic Load Balancing Algorithm

The Pseudo-Dynamic Load Balancing algorithm generates near-optimum task allocation in a real time. However, the speed of the algorithm can be increased by only allowing phases 2 and 3 to occur at fixed intervals in time. This way multiple events which

affect the state of the *GST* are amalgamated into one update phase. This will increase the speed of the algorithm at the expense of the run-time of the program being balanced across the parallel or distributed computing system.

Another enhancement to the algorithm deals with tasks that require resources which are not available at every processing node in the system. The description of each task can be extended to include a list of specific resources required by the task. Phase 2(a) can be modified to exclude processing nodes that do not have the specified resources from the allocation phase.

8.3.2 Distributing the Pseudo-Dynamic Load Balancing Algorithm

The term global scheduling table (*GST*) was used to differentiate it from any data structure required at each processing node by the local schedulers. However, the algorithm can be distributed across the parallel or distributed computing system by partitioning the system into a set of inter-connected regions [27]. In the Pseudo-Dynamic Load Balancing algorithm this is accomplished by confining the global scheduling table to a neighbourhood of processing nodes. In each neighbourhood there would be some nodes from which programs can be launched, and some nodes specifically for computational and/or resource usage. Each node, in the neighbourhood, from which a program could be launched would have its own global scheduling table for that neighbourhood. These *GSTs* would contain the

neighbourhood values of the minimum active precedence, the average computational load per node and the actual load levels of each processing node. All other information in each *GST* would be local to each launch node.

Adjacent neighbourhood can transfer tasks between them by the use of a **gateway node**. A gateway node is a processing node that presents a summary of the state of the adjacent neighbourhood through its load level. The load level of a gateway node is the average load level of the processing nodes in the adjacent neighbourhood, modified upwards to account for the extra communications distances in the adjacent neighbourhood. This load level could also be modified downwards to account for any particularly lightly loaded processing nodes in the adjacent neighbourhood. Tasks allocated to a gateway node are entered into the gateway node's *GST* for the adjacent neighbourhood, and are allocated accordingly in that neighbourhood. A neighbourhood may have several gateway nodes to other neighbourhoods. While gateway nodes do not fully distribute the task allocation problem across the parallel or distributed computing system, recent research has shown that a program will tend to limit itself to a set of processors called the **processor working set** [28]. The processor working set is a direct result of the effects of inter-task communications and inter-node communication distances. A program with a processor working set that is greater than the neighbourhood may have tasks exported to adjacent neighbourhoods for execution, depending on the current state of adjacent neighbourhoods.

8.4 Concluding Remarks

The load balancing problem is one of the fundamental problems which has prevented concurrent computing from entering the mainstream of computing. While the load balancing problem, in the general sense, belongs to the class of NP-complete problems, and is therefore believed to be intractable, many approaches, static and dynamic, have been tried to either generate an optimal solution to a restricted set of program graphs and computing environments, or generate near-optimal solutions to the general problem by using graph theory, mathematical programming, queuing theory or heuristics.

This thesis proposes two efficient heuristic algorithms for generating near-optimal load balancing allocations. The Maximum (k-1) Sum algorithm is a static algorithm that seeks to maximise the throughput of programs through the parallel or distributed computing system. This algorithm is derived from a directed search algorithm that uses an analogy with a minimum energy physical system to direct a search of the solution space.

The Pseudo-Dynamic Load Balancing algorithm is on-line heuristic algorithm that represents an important extension to the existing approaches to load balancing. It is unique in that it combines the advantages of both static and dynamic load balancing methodologies in order to minimise the makespan of an allocated program. The algorithm is developed from a mathematical analysis of the load balancing problem. It takes two inputs, a program graph describing the program, generated when the program is compiled, and a

network graph that describes the current configuration of the parallel or distributed computing system. These two inputs are combined at run time, allowing the algorithm to exploit knowledge about the structure of the program and the current state of the computing environment, to produce a near-optimal allocation of tasks to processing nodes as the program is being executed.

Since Pseudo-Dynamic Load Balancing algorithm is an on-line algorithm, the information relating to the state of the computing environment is only required when the task is being allocated. Therefore the network graph, which represents the computing environment, can be re-configured 'on the fly'. This may be necessary due to a failure, or addition, of a processing node or communications channel.

A major issue with any load balancing algorithm is the load that it places on the computing system. In the Pseudo-Dynamic Load Balancing algorithm this is kept to a minimum, since the Global Scheduling Tables (*GST*) are only kept on processing nodes that are capable of launching applications. In addition, events which require that the *GST* be updated are generally infrequent compared to the computational and communications loads of the tasks. The *GST* was designed so that only a few entries need to be updated when an event occurs.

One of the inputs into the Pseudo-Dynamic Load Balancing algorithm is the program graph, which contains the computational and communications loads of each task in the program. How does the

algorithm cope when these are non-deterministic? The values of the computational and communications loads of each task are used to compare tasks with one another. They need not be absolutely accurate, only relatively accurate. Even if there are relative errors in these values, the Pseudo-Dynamic Load Balancing algorithm compensates for these errors since the tasks with the 'incorrect' computational and communications loads will remain in the *GST* as active tasks, and maintain the associated processing node load levels at their current value until the tasks actually terminated.

The Pseudo-Dynamic Load Balancing algorithm is a significant stepping stone to the day when the average computer user will have a parallel computer or distributed computer terminal on his or her desk, and will be using commercially developed parallel or distributed software packages. It does this because:

(a) Pseudo-dynamic load balancing provides a very efficient on-line method for exploiting the parallelism expressed in a program.

(b) It can be embedded in the operating system of a parallel or distributed computing system and not be visible to the user (User Independence).

(c) It only requires the software developer to produce a description of the parallel or distributed program. This description of the program is independent of the parallel

or distributed computing system on which the program will be eventually executed (Developer Independence).

REFERENCES

[1] M.R. Garey and D.S. Johnson. "Computers and Intractability : a guide to the theory of NP-completeness.", W.H. Freeman, New York, 1979.

[2] J. Du and J.Y.-T. Leung, "Complexity of scheduling parallel task systems", SIAM J. Discrete Math., no. 2, pp. 473-487, 1989.

[3] T.L. Casavant and J.G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", IEEE Trans. Software Eng., vol. 14, pp. 141-154, Feb. 1988.

[4] M.J. Gonzalez, "Deterministic Processor Scheduling", ACM Comput. Surveys, vol. 9, pp. 173-204, Sept. 1977.

[5] J.A. Stankovic et al., "A Review of Current Research and Critical Issues id Distributed System Software", IEEE Comput. Soc. Distributed Processing Tech. Committee Newslett., vol. 7, pp. 14-47, Mar. 1982.

[6] T. Hu, "Parallel Sequencing and Assembly Line Problems", Operations Research, vol. 9, pp. 841-848, 1961.

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

[8] S.H. Bokhari, "Assignment Problems in Parallel and Distributed Computing", Kluwer Academic Press, 1987.

[9] S. Epstein, Y. Wilamowsky and B. Dickman, "Deterministic Microprocessor Scheduling with Multiple Objectives", Computers Ops. Res., vol. 19, no. 8, pp. 743-749, 1992.

[10] M. Gaudioso and P. Legato, "Linear Programming Models for Load Balancing", Computers Ops. Res., vol. 18, no. 1, pp. 59-64, 1991.

[11] E. Lawler and J. Labetoulle, "On Preemptive Scheduling of Unrelated Parallel Processors by Linear Programming", Journal of the Association for Computing Machinery, vol. 25, no. 4, pp. 612-619, October 1978.

[12] A. Billionnet, M.C. Costa and A. Sutter, "An Efficient Algorithm for a Task Allocation Problem", Journal of the Association for Computing Machinery, vol. 39, no. 3, pp. 502-518, July 1992.

[13] E.R. Barnes, A. Vannelli and J.Q. Walker, "A New Heuristic for Partitioning the Nodes of a Graph", SIAM Journal of Discrete Mathematics, vol. 1, no. 3, pp. 299-305, Aug.. 1988.

[14] S. Holm and M.M. Sørensen, "The Optimal Graph Partitioning Problem: Solution Method Based on Reducing Symmetric Nature and Combinatorial Cuts", Research Report from Department of Management Science, The Aarhus School of Business, Denmark, Sept. 1992.

[15] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", Bell Systems Technical Journal, vol. 49, no. 2, pp. 291-307, Feb. 1970.

[16] L.R. Ford and D.R. Fulkerson, "Flows in Networks", Princeton University Press, 1962.

[17] A. Darte, "Two Heuristics for Task Scheduling", Ecole Normale Supérieure de Lyon, May 1991.

[18] N.S. Bowen, C.N. Nikolaou, and A. Ghafoor, "On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computing Systems", IEEE Trans. Computers, vol. 41, no. 3, pp. 257-273, Mar. 1992.

[19] T.L. Kunii, S. Nishimura and T. Noma. The Design of a Parallel Processing System for Computer Graphics. P.M. Dew et al., editor, Parallel Processing for Computer Vision and Display, chap. 26, pp. 353-377, Addison-Wesley, 1989.

[20] V Sarkar. "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors". The MIT Press, 1989.

[21] R.L. Graham, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey", Annals of Discrete Mathematics, vol. 5, pp. 287-326, 1979.

[22] W.P. Helgeson and D.P. Birnie, "Assembly Line Balancing Using the Ranked Positional Weight Technique", Journal of Industrial Engineering, no. 6, 1961.

[23] S. Gotto and T. Matsuda, "Partitioning, Assignment and Placement", Advances in CAD for VLSI, vol. 4, pp. 68-82, 1986.

[24] D. Judge and W. Rudd, "A Test Case for the Parallel Programming Support Environment: Parallelizing the Analysis of Satellite Imagery Data", Tech. Report 89-90-2, Department of Computer Science, Oregon State University, 1989.

[25] S. Kirkpatrick, C.D. Gelatt and M.P. Vechi, "Optimization by Simulated Annealing", Science 220 (May 13, 1983), No. 4598.

[26] F. Glover, "Tabu Search", Centre for Applied Artificial Intelligence, Graduate School of Business, University of Colorado, Boulder, 1988.

[27] I. Ahmad and A. Ghafoor, "Semi-distributed Load Balancing For Massively Parallel Multicomputer Systems", IEEE Trans. Software Eng., vol. 17, pp. 987-1004, Oct. 1991.

[28] D. Ghosal, G. Serazzi and S.K. Tripathi, "The Processor Working Set and Its Use in Scheduling Multiprocessor Systems", IEEE Trans. Software Eng., vol. 17, pp. 443-453, May 1991.

APPENDIX A: Maximum (k-1) Sum Algorithm Source Code

The 'C' source code for the Maximum (k-1) Sum algorithm is on the enclosed PC-formatted disk in the "\static\slc" subdirectory. This directory contains the following files:

slc.exe	Executable Maximum (k-1) Sum program.
slc.c	Source code for Maximum (k-1) Sum algorithm.
task.h	Definition file for Maximum (k-1) Sum algorithm.

Also in this directory is a subdirectory "\static\slc\cvis" which contains the following files:

The input file "cvin.dat" for the computer vision program.

The files "cvis*X*.dat" that contain the results of using the Maximum (k-1) Sum algorithm to allocate the computer vision program to a bus network of *X* processors.

The file "cvis4r12.dat" that contains the results of using the Maximum (k-1) Sum algorithm to allocate the computer vision program to a bus network of 4 processors, but relaxing the algorithm to allow up to 12 tasks to be allocated to a processor.

APPENDIX B: Sciconics Implementation of the Relaxed Non-Symmetric Formulation of the Task Allocation Problem

B.1 Introduction

When specifying the problem the user is restricted to MGG's syntax. The MGG syntax requires the user to specify the problem in a series of sections.

B.1.1 Suffices

This section defines the suffices to be used in the problem formulation. For the non-symmetric model four suffices are required. A dummy index D is needed to allow the user supplied problem data to be formatted with one data item on each line of the problem data file. The need for the suffices I, J, and K are obvious from equations [4.22] to [4.36].

B.1.2 External Values

This section defines the data to be supplied by the user for a given instance of the problem. The non-symmetric model requires

external values to hold the computational load of each task (CLOAD), the precedence level of each task (PREC), whether two tasks communicate (COMMW), the communications load between each task (CLOAD), the bounding coefficient (BCOEFF) and the number of processing nodes (NODES).

B.1.3 Internal Values

This section defines any additional data storage area required by the matrix generator (MG) and the report writer (RW), as well as a subroutine to initialise these internal values. This section is optional.

This section could have used to calculate the precedence level for each task, but it was decided to simplify the formulation and read the precedence levels directly from the problem data file.

B.1.4 Declarations

This section is used to modify the definition of external and internal values. This section is optional.

B.1.5 Variables

This section defines the variables in the mathematical formulation of the problem, and specifies upper and lower limits on their values. From equations [4.22] to [4.36] the non-symmetric model requires the following variables.

Y(i):	An n element array containing the relaxed earliest start time for task *i*.
Z(i,j), W(i,j):	Two n x n element arrays of relaxation variables.
N(i):	An n element array specifying if a task is the lowest cardinal numbered task allocated to its processing node.
A(i,j):	An n x n element adjacency array.

B.1.6 Problem

This section defines the problem in terms of an objective function subject to constraints. The definition of the problem in the syntax of MGG requires that:

(a) Constants and coefficients must have names with 3 characters

Constants and coefficients must have exactly 3 character names; no more and no less.

(b) Only constants are allowed on the right hand side.

Remember that a constant in a given instance of a problem is a variable in the formulation of the problem.

(c) The objective function and left hand sides of the constraints have a specific format.

The objective function and the left hand side of the constraints must be the sum of terms, in which each term has the following format.

SUM (Suffices) (Coefficient) * (Generic Variable Name)

The suffices are required in the summation if the suffices differ from those in the generic constraint. However, it is always a good idea to include the suffices.

The coefficient must have a 3 character name. Therefore, using BCOEFF as a coefficient is illegal. The solution is to define a 3 character name, say OB1, as equal to BCOEFF in the elements section of the formulation.

The generic variable name must have its suffices in the correct order.

(d) The first character of a coefficient determines its type.

Because MGG generates FORTRAN source code coefficient names that start with I, J, K, L, M or N are implicitly integers. A coefficient called JLS is an integer and will cause an error when MGCL is used to compile and link the matrix generator MG, if JLS is used as a floating point value.

(e) Specific elements of generic variable arrays can not be referenced.

If Y(i) is an n element array and you wish to reference the first element in a constraint, i.e. Y(1) > 0.0, you must sum Y(r) times a coefficient over all r. The coefficient is defined in the elements section to be 1.0 for r = 1, and 0.0 otherwise. Similarly, to reference Y(j), you must sum Y(r) times a coefficient over all r

and define the coefficient to be 1.0 if $r = j$, and 0.0 otherwise. But r is not a valid suffix for Y, so you must define it as being square, e.g.

```
SUM (I1) C10 * Y(I1) - Y(I) + CIJ*A(I,J) .GE. TMP

FOR ALL I, J

FOR I1 = SQUARE
```

B.1.7 Elements

This section defines any 3 character elements that have been used as coefficients in the objective function and constraints in the problem section, or in the upper and lower bounds when variables are declare in the variables section. The values of the coefficients can be conditional on the suffices and external data using FORTRAN constructs. The inclusion of this section is optional in the MGG syntax, but essential in practice.

B.1.8 Functions

This section contains any FORTRAN functions used in the problem and elements section. Typically, these functions are used to specify when a constraint exists, and to calculate complex coefficients in the objective function and the constraints. Symbol names defined in

the suffices section will be global in the matrix generator and the report writer, and therefore should be avoided in user supplied functions.

B.2 MGG Problem Formulation

```
C
C                         Parallel Critical Path Problem
C
C       DATE: 5-3-93
C
C
OPTIONS EIGHT
C
NOTATION
C
SUFFICES
C                                        Dummy index from 1 to 1
 D     MAXA          1
C
 I     MAXI         60
C
 J     MAXJ         60
C
 K     MAXK         60
C
C
C
EXTERNAL VALUES
C
 LOAD(I,D)          F5.0           Computational load of task I
C
 COMMW(I,J)         (1X, I1)       Set to 1 if task I communicates with
task J
C
 CLOAD(I,J)         (1X, F5.0)     Communications from task I to task J
C
 PREC(I,D)          F5.0           Precedence Level of task I
C
 BCOEFF             F10.0          Bounding Coefficient
C
 NODES              I3
C
C
C
VARIABLES
C
 Y(I)         '* II'
BOUND LO 0.0
C
 Z(I,J)       '* II JJ'
BOUND LO 0.0
C
 W(I,J)       '* II JJ'
BOUND LO 0.0
C
 N(I)         '* II'
BOUND BV
```

```
C
 A(I,J)      '* II JJ'
BOUND BV
C
C
C
PROBLEM
C
MINIMISE
C
* OBJ        '***'
C
C
 SUM (I) OB1*Y(I) - SUM (I,J) W(I,J)
C
C
SUBJECT TO
C
C
C
* FIRST      '******'
C
C Y(1) .EQ. 0.0
C
 SUM (I) C50*Y(I) .EQ. 0.0
C
C                                    Early Start Constraint
C
* ES         '** II JJ'      NOT IF (COMMW(I,J) .EQ. 0)
C
C Y(J) - Y(I) + CIJ*A(I,J) .GE. TMP
C
 SUM (I1) C10*Y(I1) - Y(I) + CIJ*A(I,J) .GE. TMP
FOR ALL I, J
FOR I1 = SQUARE
C
C                                    One Task at a Time Constraint
C
* OT         '** II JJ'      NOT IF (EXISTS() .EQ. 0.0)
C
C A(I,J)*Y(I) - A(I,J)*Y(I) - LOAD(I,1)*A(I,J) .GE. 0.0
C
 Z(I,J) - Z(J,I) - CPI*A(I,J) .GE. 0.0
FOR ALL I, J
C
C
C
* ZB         '** II JJ'
C
 Z(I,J) - SUM (I7) C60*Y(I7) .LE. 0.0
FOR ALL I, J
FOR I7 = SQUARE
C
C
C
* WA         '** II JJ'
C
```

```
 W(I,J) - Z(I,J) .LE. 0.0
FOR ALL I, J
C
C
* WB         '** II JJ'
C
 W(I,J) - SUM (I5) C40*Y(I5) + Z(I,J) .LE. 0.0
FOR ALL I, J
FOR I5 = SQUARE
C
C                                 Associativity Constraint
C
* ASS        '**IIJJKK'
C
C A(I,J) + A(I,K) - A(J,K) .LE. 1
C
 A(I,J) + SUM (J1) C20*A(I,J1) - SUM (I2,J2) C21*A(I2,J2) .LE. 1
FOR ALL I, J, K
FOR J1 = SQUARE
FOR I2 = SQUARE
FOR J2 = SQUARE
C
C                                 Symmetry Constraint
C
* SYM        '*** I J'
C
 A(I,J) - A(J,I) .EQ. 0
FOR ALL I, J
C
C                                 Unit Diagonal Constraint
C
* DIAG       '**** I J'     NOT IF (I .NE. J)
C
 A(I,J) .EQ. 1
FOR ALL I, J
C
C                                 Number of Nodes Constraint
C
* N1         '**'
C
 SUM (I) C50*N(I) .EQ. 0
C
* NL         '** II'
C
 N(I) - SUM (J) C70*A(I,J) .LE. 0.0
FOR I .GT. 1 .AND. I .LE. MAXI
C
* NU         '** II JJ'     NOT IF (I .EQ. 1 .OR. I .LE. J)
C
 N(I) - C71*A(I,J) .GE. 0.0
FOR ALL J
C
* SUMN       '****'
C
 SUM (I) N(I) .GE. SMN
C
C
```

```
C
ELEMENTS
C
       CPI = LOAD (I,1)
       CIJ = CLOAD(I,J)
       TMP = LOAD(I,1) + CLOAD(I,J)
C
       OB1 = BCOEFF
C
       C10 = 0.0
      0IF (I1 .EQ. J) C10 = 1.0
C
       C20 = 0.0
      0IF (J1 .EQ. K) C20 = 1.0
C
       C21 = 0.0
      0IF (I2 .EQ. J .AND. J2 .EQ. K) C21 = 1.0
C
       C40 = 0.0
      0IF (I5 .EQ. J) C40 = 1.0
C
       C50 = 0.0
      0IF (I .EQ. 1) C50 = 1.0
C
       C60 = 0.0
      0IF (I7 .EQ. J) C60 = 1.0
C
       C70 = 0.0
      0IF (J .LT. I) C70 = 1.0
C
       C71 = 0.0
      0IF (J .LT. I) C71 = 1.0
C
       SMN = MAXI - NODES
C
C
C
FUNCTIONS
C
       FUNCTION EXISTS ()
C
       INTEGER M
       REAL CLOAD_I, CLOAD_J
C
       CLOAD_I = 0.0
       DO 10 M = 1, MAXI
          CLOAD_I = CLOAD_I + CLOAD (M,I)
   10  CONTINUE
C
       CLOAD_J = 0.0
       DO 20 M = 1, MAXI
          CLOAD_J = CLOAD_J + CLOAD (M,J)
   20  CONTINUE
C
C
       EXISTS = 0.0
C
```

```
       IF (PREC(I,1) .GT. PREC(J,1)) EXISTS = 1.0
C
       IF ((PREC(I,1) .EQ. PREC(J,1)) .AND.
     1     (CLOAD_I .LT. CLOAD_J)
     2    ) EXISTS = 1.0
C
       IF ((PREC(I,1) .EQ. PREC(J,1)) .AND.
     1     (CLOAD_I .EQ. CLOAD_J) .AND.
     2     (I .LT. J)
     3    ) EXISTS = 1.0
C
C
      RETURN
      END
C
C
C
ENDATA
```

B.3: User Supplied Report Routime

```
C=====================================================================
C
C                                  REPORT.FOR
C
C      DATE:    21 JAN 1993
C      AUTHOR: DAVID SINCLAIR
C
C=====================================================================
C
C
      SUBROUTINE REPORT
C     =================
C
C                  Report Writer for Parallel Critical Path Problem
C
C
      INCLUDE 'MGCOMS'
      INCLUDE 'RWCOMS'
C
C                  Declare Local Data
C
      INTEGER ASSIGNED (50), NODE, TASK, NEMPTY
      REAL RUNTIME (50), EXTIME (50), TIME, PROG_TIME
      REAL NDAVTIME (20)
      LOGICAL SUCC (50,50)
C
      LOGICAL ROOT
C
      REAL STACK (50,2)
      INTEGER TOP
C
C
C
      IF (MAXI .GT. 50) THEN
         WRITE (6,5)
    5    FORMAT (1X, 'Too many tasks!! Need to recompile REPORT.FOR')
         GO TO 999
      END IF
C
C                  Setup Successor array
C
      DO 10 IL = 1, MAXI
         DO 20 JL = 1, MAXI
            IF ( COMMW (IL,JL) .NE. 0) THEN
               SUCC (IL, JL) = .TRUE.
            ELSE
               SUCC (IL, JL) = .FALSE.
            END IF
   20    CONTINUE
   10 CONTINUE
C
```

```
C                 Assign Tasks to Nodes
C
      DO 30 IL = 1, MAXI
         ASSIGNED (IL) = 0
   30 CONTINUE
C
      NODE = 1
      I1 = 1
C
   50 IF (ASSIGNED (I1) .NE. 0) GO TO 60
      DO 70 J1 = 1, MAXI
         IF (BA (I1, J1) .EQ. 1.0) ASSIGNED (J1) = NODE
   70 CONTINUE
      NODE = NODE + 1
   60 IF (I1 .EQ. MAXI) GO TO 80
      I1 = I1 + 1
      GO TO 50
   80 CONTINUE
C
C                 Calculate run time for each task
C                 (computational load + receiving time +
C                  transmission time)
C
      DO 220 I1 = 1, MAXI
         RUNTIME (I1) = LOAD (I1, 1)
         DO 230 J1 = 1, MAXI
            IF (BA (I1, J1) .NE. 1.0) THEN
               IF (COMMW (J1, I1) .EQ. 1) THEN
                  RUNTIME (I1) = RUNTIME (I1) + CLOAD (J1,I1)
               END IF
               IF (COMMW (I1, J1) .EQ. 1) THEN
                  RUNTIME (I1) = RUNTIME (I1) + CLOAD (I1,J1)
               END IF
            END IF
  230    CONTINUE
  220 CONTINUE
C
C
C                 Find root tasks and sort them by increasing
C                 precedence
C
      J1 = 1
      CALL INITSTK (TOP)
C
  110 ROOT = .TRUE.
      DO 100 I1 = 1, MAXI
         IF (SUCC (I1, J1) .EQ. .TRUE.) ROOT = .FALSE.
  100 CONTINUE
      IF (ROOT .EQ. .TRUE.) THEN
         CALL PUSH (STACK, TOP, J1, 0.0)
      END IF
      IF (J1 .EQ. MAXI) GOTO 120
      J1 = J1 + 1
      GO TO 110
  120 CONTINUE
C
C
```

```
C
      WRITE (6, 152)
      WRITE (KPRINT, 152)
  152 FORMAT (1X, //, '          Parallel Critical Path Report',/,
     1               '          ==============================',//)
C
C                    Initialise Earliest Node Available Times
C
      DO 180 I1 = 1, NODES
         NDAVTIME (I1) = 0.0
  180 CONTINUE
C
C                    Initialise Task Execution Times
C
      DO 190 I1 = 1, MAXI
         EXTIME (I1) = -1.0
  190 CONTINUE
C
C                    Update Task Execution Times
C
  160 CONTINUE
      NEMPTY = IPOP (STACK, TOP, TASK, TIME)
      IF (NEMPTY .EQ. 1) THEN
         CALL UPDATE (TASK, TIME, STACK, TOP, NDAVTIME, ASSIGNED,
     1                    EXTIME, SUCC, RUNTIME)
      ELSE
         GO TO 170
      END IF
      GO TO 160
  170 CONTINUE
C
C                    Print out Execution Times
C
      DO 200 IL = 1, MAXI
         WRITE (6,201) IL, ASSIGNED (IL), EXTIME (IL)
         WRITE (KPRINT,201) IL, ASSIGNED (IL), EXTIME (IL)
  201    FORMAT (1X,'Task ', I2,' executes on Node ', I2,' at T= ',
     1                F10.5)
  200 CONTINUE
C
      PROG_TIME = 0.0
      DO 240 IL = 1, NODES
         IF (NDAVTIME (IL) .GT. PROG_TIME) PROG_TIME = NDAVTIME (IL)
  240 CONTINUE
C
      WRITE (KPRINT,211) PROG_TIME
      WRITE (6,211) PROG_TIME
  211 FORMAT (1X,//, ' Runtime of Program = ', F10.5,//)
C
C
  999 RETURN
      END
C
C
C
      SUBROUTINE INITSORT (TAIL)
C     ==========================
```

```
C
      INTEGER TAIL
      TAIL = 1
      RETURN
      END
C
C
C
      SUBROUTINE ADDSORT (S, TAIL, TASKNUM)
C     ====================================
C
      INCLUDE 'MGCOMS'
      INCLUDE 'RWCOMS'
C
      INTEGER S (50), TAIL, TASKNUM
C
      IF (TAIL .EQ. 1) THEN
         S (TAIL) = TASKNUM
         TAIL = TAIL + 1
      ELSE IF (PREC (TASKNUM, 1) .GE. PREC (S (TAIL - 1), 1)) THEN
         S (TAIL) = TASKNUM
         TAIL = TAIL + 1
      ELSE
         DO 130 IL = 1, (TAIL - 1)
         IF (PREC (S (IL), 1) .GT. PREC (TASKNUM, 1)) THEN
            DO 140 I2 = 1, (TAIL - IL)
               S (TAIL - I2 + 1) = S (TAIL - I2)
  140       CONTINUE
            S (IL) = TASKNUM
            TAIL = TAIL + 1
         END IF
  130    CONTINUE
      END IF
C
C
      RETURN
      END
C
C
C
      SUBROUTINE INITSTK (TOP)
C     ========================
C
      INTEGER TOP
      TOP = 1
      RETURN
      END
C
C
C
      SUBROUTINE PUSH (S, TOP, TASK, TIME)
C     ==================================
C
      INCLUDE 'MGCOMS'
      INCLUDE 'RWCOMS'
C
C                  Add to Prioritised Stack
```

```
C                    Stack prioritised by Precedence. Duplicate Tasks
C                    are prioritised by TIME.
C
      REAL S (50,2), TIME
      INTEGER TOP, TASK
C
      IF (TOP .EQ. 50) THEN
         WRITE (6, 340)
  340    FORMAT (1X, 'STACK OVERFLOW',//)
         GOTO 399
      END IF
C
C
      DO 310 IP = 1, (TOP - 1)
         IF ((PREC (TASK, 1) .LT. PREC (S (IP, 1), 1))
     1        .OR. ((PREC (TASK, 1) .EQ. PREC (S (IP, 1), 1))
     2              .AND. (TIME .GE. S (IP,2))
     3             )
     4       ) GO TO 315
  310 CONTINUE
C
  315 IF (IP .EQ. TOP) THEN
         S (TOP, 1) = REAL (TASK)
         S (TOP, 2) = TIME
      ELSE
         DO 320 I2 = 1, (TOP - IP)
            S ((TOP - I2 + 1), 1) = S ((TOP - I2), 1)
            S ((TOP - I2 + 1), 2) = S ((TOP - I2), 2)
  320    CONTINUE
         S (IP, 1) = REAL (TASK)
         S (IP, 2) = TIME
      END IF
C
      TOP = TOP + 1
C
C
C
  399 RETURN
      END
C
C
C
      INTEGER FUNCTION IPOP (S, TOP, TASK, TIME)
C     ==========================================
C
      REAL S (50, 2), TIME
      INTEGER TOP, TASK
C
      IF (TOP .EQ. 1) THEN
         IPOP = 0
      ELSE
         TASK = INT (S ((TOP - 1), 1))
         TIME = S ((TOP - 1), 2)
         TOP = TOP - 1
         IPOP = 1
      END IF
C
```

```
C
      RETURN
      END
C
C
C
      SUBROUTINE UPDATE (TASK, TIME, S, TOP, AVAILT, ASGN,
     1                      XTIME, CHILDS, RTIME)
C     ====================================================
C
      INCLUDE 'MGCOMS'
      INCLUDE 'RWCOMS'
C
      INTEGER TASK, TOP, ASGN (50), NODE
      REAL S (50,2), AVAILT (20), XTIME (50), RTIME (50)
      REAL TIME, PSTIME, CSUM
      LOGICAL CHILDS (50,50), MOVED
      INTEGER WAIT
C
      INTEGER SUCCS (50), STAIL
C
C                 Set Task Execution Time
C
      NODE = ASGN (TASK)
C
      IF (XTIME (TASK) .GE. TIME) THEN
         GO TO 599
      END IF
C
C                 Adjust time so that it does not overlap with a
C                 higher precedence task already allocated to this
C                 node.
C
  540 MOVED = .FALSE.
      DO 550 IU = 1, MAXI
         IF ((PREC (IU,1) .GE. PREC (TASK,1)) .AND.
     1        (IU .NE. TASK) .AND.
     2        (XTIME (IU) .NE. -1.0) .AND. (NODE .EQ. ASGN (IU)) .AND.
     3        (TIME .GE. XTIME (IU))
     4       ) THEN
            FINISH = XTIME (IU) + RTIME (IU)
            IF (TIME .LT. FINISH) THEN
               TIME = FINISH
               MOVED = .TRUE.
C
C
C
               ENDTIME = TIME + RTIME (TASK)
               DO 560 JU = 1, MAXI
                  IF ((PREC (JU,1) .GT. PREC (TASK,1)) .AND.
     1                 (ENDTIME .GT. XTIME (JU))
     2                ) THEN
                     FINISH = XTIME (JU) + RTIME (JU)
                     IF (ENDTIME .LT. FINISH) THEN
                        TIME = FINISH
                        ENDTIME = TIME + RTIME (TASK)
C
```

```
C
C
                         END IF
                      END IF
  560              CONTINUE
                   END IF
                END IF
  550 CONTINUE
C
      IF (MOVED .EQ. .TRUE.) GO TO 540
C
C
C
      IF (XTIME (TASK) .EQ. -1.0) THEN
         IF (TIME .GT. AVAILT (NODE)) THEN
            XTIME (TASK) = TIME
         ELSE
            XTIME (TASK) = AVAILT (NODE)
         END IF
      ELSE
         XTIME (TASK) = TIME
      END IF
C
C                 Update the time at which this node will become
C                 available for another task.
C
      AVAILT (NODE) = XTIME (TASK) + RTIME (TASK)
C
C                 Push Sucessors on to Stack
C
      CALL INITSORT (STAIL)
C
      DO 510 JU = 1, MAXI
         IF (CHILDS (TASK, JU) .EQ. .TRUE.) THEN
            CALL ADDSORT (SUCCS, STAIL, JU)
         END IF
  510 CONTINUE
C
      CSUM = 0.0
      DO 520 JU = 1, (STAIL - 1)
         CSUM = CSUM + ((1.0 - BA (TASK, SUCCS (JU))) *
     1         CLOAD (TASK, SUCCS (JU)))
         PSTIME = AVAILT (NODE) - CSUM
         CALL PUSH (S, TOP, SUCCS (JU), PSTIME)
  520 CONTINUE
C
C                 Push tasks with lower precedence on the same node,
C                 which now overlap in time with the current task, on
C                 to the task stack.
C
      DO 570 IU = 1, MAXI
         IF ((PREC (IU,1) .LE. PREC (TASK,1)) .AND.
     1         (IU .NE. TASK) .AND. (NODE .EQ. ASGN (IU)) .AND.
     2         (XTIME (IU) .GE. XTIME (TASK)) .AND.
     3         (XTIME (IU) .LE. AVAILT (NODE)) .AND.
     4         (RTIME (IU) .NE. 0.0)
     5       ) THEN
```

```
                CALL PUSH (S, TOP, IU, AVAILT (NODE))
C
C
C
            END IF
  570 CONTINUE
C
C
  599 CONTINUE
C
C
C
      RETURN
      END
```

B.4: Problem Data File

```
*
* TEST FILE 3: ATMOSPHERIC ANALYSIS
*
* DATE: 4-1-93
*
*
*
EXTERNAL VALUES
*
*
KDNAME 1     DUMMY
*
KINAME 1     TASK.1
KINAME 2     TASK.2
KINAME 3     TASK.3
KINAME 4     TASK.4
KINAME 5     TASK.5
KINAME 6     TASK.6
KINAME 7     TASK.7
KINAME 8     TASK.8
KINAME 9     TASK.9
KINAME 10    TASK.10
KINAME 11    TASK.11
KINAME 12    TASK.12
KINAME 13    TASK.13
KINAME 14    TASK.14
KINAME 15    TASK.15
KINAME 16    TASK.16
KINAME 17    TASK.17
KINAME 18    TASK.18
*
KJNAME 1     JTASK1
KJNAME 2     JTASK2
KJNAME 3     JTASK3
KJNAME 4     JTASK4
KJNAME 5     JTASK5
KJNAME 6     JTASK6
KJNAME 7     JTASK7
KJNAME 8     JTASK8
KJNAME 9     JTASK9
KJNAME 10    JTASK10
KJNAME 11    JTASK11
KJNAME 12    JTASK12
KJNAME 13    JTASK13
KJNAME 14    JTASK14
KJNAME 15    JTASK15
KJNAME 16    JTASL16
KJNAME 17    JTASK17
KJNAME 18    JTASK18
*
KKNAME 1     KTASK1
KKNAME 2     KTASK2
KKNAME 3     KTASK3
KKNAME 4     KTASK4
```

```
KKNAME 5      KTASK5
KKNAME 6      KTASK6
KKNAME 7      KTASK7
KKNAME 8      KTASK8
KKNAME 9      KTASK9
KKNAME 10     KTASK10
KKNAME 11     KTASK11
KKNAME 12     KTASK12
KKNAME 13     KTASK13
KKNAME 14     KTASK14
KKNAME 15     KTASK15
KKNAME 16     KTASK16
KKNAME 17     KTASK17
KKNAME 18     KTASK18
*
*
*
*    Computational Load Data
*
LOAD   TASK.1     1.0
LOAD   TASK.2     3.0
LOAD   TASK.3     3.0
LOAD   TASK.4     3.0
LOAD   TASK.5     3.0
LOAD   TASK.6     3.0
LOAD   TASK.7     3.0
LOAD   TASK.8     3.0
LOAD   TASK.9     3.0
LOAD   TASK.10    10.0
LOAD   TASK.11    10.0
LOAD   TASK.12    10.0
LOAD   TASK.13    10.0
LOAD   TASK.14    5.0
LOAD   TASK.15    5.0
LOAD   TASK.16    5.0
LOAD   TASK.17    5.0
LOAD   TASK.18    1.0
*
*
*
*    Communicates With Data
*
COMMW  TASK.1   1   0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
COMMW  TASK.2   1   0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
COMMW  TASK.3   1   0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
COMMW  TASK.4   1   0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
COMMW  TASK.5   1   0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
COMMW  TASK.6   1   0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
COMMW  TASK.7   1   0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
COMMW  TASK.8   1   0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
COMMW  TASK.9   1   0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
COMMW  TASK.10  1   0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0
COMMW  TASK.11  1   0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0
COMMW  TASK.12  1   0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0
COMMW  TASK.13  1   0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0
COMMW  TASK.14  1   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
COMMW  TASK.15  1   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

```
COMMW  TASK.16  1   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
COMMW  TASK.17  1   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
COMMW  TASK.18  1   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
*
*
*
*    Communications Load Data
*
CLOAD  TASK.1  1   0.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  0.0
CLOAD  TASK.1  2   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
*
CLOAD  TASK.2  1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
CLOAD  TASK.2  2   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
*
CLOAD  TASK.3  1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
CLOAD  TASK.3  2   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
*
CLOAD  TASK.4  1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.4  2   1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
*
CLOAD  TASK.5  1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.5  2   1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
*
CLOAD  TASK.6  1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.6  2   0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
*
CLOAD  TASK.7  1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.7  2   0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
*
CLOAD  TASK.8  1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.8  2   0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
*
CLOAD  TASK.9  1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.9  2   0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
*
CLOAD  TASK.10 1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.10 2   0.0  0.0  0.0  1.0  1.0  1.0  1.0  0.0
*
CLOAD  TASK.11 1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.11 2   0.0  0.0  0.0  1.0  1.0  1.0  1.0  0.0
*
CLOAD  TASK.12 1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.12 2   0.0  0.0  0.0  1.0  1.0  1.0  1.0  0.0
*
CLOAD  TASK.13 1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.13 2   0.0  0.0  0.0  1.0  1.0  1.0  1.0  0.0
*
CLOAD  TASK.14 1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.14 2   0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
*
CLOAD  TASK.15 1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.15 2   0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
*
CLOAD  TASK.16 1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD  TASK.16 2   0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
*
CLOAD  TASK.17 1   0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
CLOAD TASK.17 2  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
*
CLOAD TASK.18 1  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
CLOAD TASK.18 2  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
*
*
*
*   Precedence Levels
*
PREC  TASK.1     24.0
PREC  TASK.2     22.0
PREC  TASK.3     22.0
PREC  TASK.4     22.0
PREC  TASK.5     22.0
PREC  TASK.6     22.0
PREC  TASK.7     22.0
PREC  TASK.8     22.0
PREC  TASK.9     22.0
PREC  TASK.10    18.0
PREC  TASK.11    18.0
PREC  TASK.12    18.0
PREC  TASK.13    18.0
PREC  TASK.14     7.0
PREC  TASK.15     7.0
PREC  TASK.16     7.0
PREC  TASK.17     7.0
PREC  TASK.18     1.0
*
*
*
*   Bounding Coefficient
*
BCOEFF 9.5
*
*   Maximum Number of Nodes
*
NODES 8
*
*
ENDATA
```

B.5: Sciconic Run Streams

Run Stream 1

```
$SET DEF SCRATCH
$CREATE/DIR [.LBAL]
$SET DEF SYS$LOGIN
$ASSIGN "SAT0$DKB100:[SCICONIC]HELPFILE.DAT" HELPLINP
$ASSIGN "SAT2$DKB300:[SCRATCH.LBAL]" TEMPPCP
$RUN SAT0$DKB100:[SCICONIC]SCICONICX.EXE
INT J
DP DIFF, LIMIT
INFILE = '[.LOAD_LP]MATRIX.DAT'
OUTFILE = 'TEMPPCP:TEST.OUT'
SOLNFILE = '[.LOAD_LP]SOLN.DAT'
GLOBSCRATCH='TEMPPCP:GLOBSCRATCH.TMP'
GLOBFILE='TEMPPCP:GLOBFILE.TMP'

CONVERT
SETUP
PRIMAL

J = 0

10 J = J + 1

GLOBAL (USER)

DIFF = CUTOFF - OBJVAL
LIMIT = 0.01 * OBJVAL

IF ((DIFF .LT. LIMIT) .AND. (J .NE. 1)) GOTO 20
IF (FINISHED .EQ. 5) GOTO 20

IF (FINISHED .EQ. 3) CUTOFF = OBJVAL
IF (FINISHED .EQ. 3) OBJTAR = OBJVAL
IF (FINISHED .EQ. 3) GOTO 10

20 PUNCHSOLN
STOP
```

Run Stream 2

```
$SET DEF SCRATCH
$CREATE/DIR [.LBAL]
$SET DEF SYS$LOGIN
$ASSIGN "SAT0$DKB100:[SCICONIC]HELPFILE.DAT" HELPLINP
$ASSIGN "SAT2$DKB300:[SCRATCH.LBAL]" TEMPPCP
$RUN SAT0$DKB100:[SCICONIC]SCICONICX.EXE
INT J, EXIT
DP DIFF, LIMIT
INFILE = '[.LOAD_LP]MATRIX.DAT'
OUTFILE = 'TEMPPCP:TEST.OUT'
SOLNFILE = '[.LOAD_LP]SOLN.DAT'
GLOBSCRATCH='TEMPPCP:GLOBSCRATCH.TMP'
```

```
GLOBFILE='TEMPPCP:GLOBFILE.TMP'

CONVERT
SETUP
PRIMAL

J = 0
EXIT = 0

10 J = J + 1

GLOBAL (USER)

DIFF = CUTOFF - OBJVAL
LIMIT = 0.01 * OBJVAL

IF ((DIFF .LT. LIMIT) .AND. (J .NE. 1) .AND. (EXIT .EQ. 1)) GOTO 20
IF ((DIFF .LT. LIMIT) .AND. (J .NE. 1)) EXIT = 1

IF (FINISHED .EQ. 5) GOTO 20

IF (FINISHED .EQ. 3) CUTOFF = OBJVAL
IF (FINISHED .EQ. 3) OBJTAR = OBJVAL
IF (FINISHED .EQ. 3) GOTO 10

20 PUNCHSOLN
STOP
```

APPENDIX C: Pseudo-Dynamic Load Balancing algorithm Source Code

The source code for the parallel and distributed computing system simulator, which contains the Pseudo-dynamic Load Balancing algorithm, is contained on the enclosed PC-formatted disk in the subdirectory "\dynamic".

The subdirectory "\dynamic\source" contains the executable file, source code files and header files.

dlb_sim.exe	Executable parallel and distributed computing system simulation program.

'C' Source Files:

dlb_sim.c	Source code for main simulation driver module.
data_io.c	Source code for data input/output routines.
gs_table.c	Source code for Global Scheduling Table (*GST*) routines.
events.c	Source code for event handling routines.
utils.c	Source code for general utility routines.

Header Files:

compiler.h	Compiler specific definitions.
dlb_sim.h	Definitions for main simulation driver module.
data_io.h	Definitions for data input/output routines.
events.h	Definitions of event specific data.
gs_table.h	Definitions for *GST* routines.
node.h	Definitions of node specific data.
project.h	Definitions of project specific data.
task.h	Definitions of task specific data.

The subdirectory "\dynamic\adg" contains the application descriptor files for the test data.

atmos.adg	Application descriptor file for the module used in the Spatial Coherence Method of atmospheric analysis.
cvis.adg	Application descriptor file for a computer vision program.
gelim.adg	Application descriptor file for a Gaussian elimination program.

The subdirectory "\dynamic\ntp" contains a set of network topology files.

bus2.ntp	Network topology file for a 2 processor bus network.
bus3.ntp	Network topology file for a 3 processor bus network.

bus4.ntp	Network topology file for a 4 processor bus network.
bus5.ntp	Network topology file for a 5 processor bus network.
bus6.ntp	Network topology file for a 6 processor bus network.
bus7.ntp	Network topology file for a 7 processor bus network.
bus8.ntp	Network topology file for a 8 processor bus network.
bring4.ntp	Network topology file for a 4 processor ring network..
bring6.ntp	Network topology file for a 6 processor ring network..
bring8.ntp	Network topology file for a 8 processor ring network..
h3cube.ntp	Network topology file for a 3-dimensional hyper-cube.
h4cube.ntp	Network topology file for a 4-dimensional hyper-cube.

The subdirectory "\dynamic\atmos" contains the test data generated by allocating the atmospheric analysis module to a bus topology of 2 to 8 processors. The file "atmosb***X***.log" contains the simulation log file generated when the atmospheric analysis module was allocated to ***X*** processors. The file "atmosb***X***.rpt" contains the simulation report file generated when the atmospheric analysis module was allocated to ***X*** processors.

The subdirectory "\dynamic\cvis" contains the test data generated by allocating the computer vision program to a bus topology of 2 to 8 processors. The file "cvisb*X*.log" contains the simulation log file generated when the computer vision program was allocated to *X* processors. The file "cvisb*X*.rpt" contains the simulation report file generated when the computer vision program was allocated to *X* processors.

The subdirectory "\dynamic\gelim" contains the test data generated by allocating the Gaussian elimination program to a bus topology of 2 to 8 processors. The file "gelimb*X*.log" contains the simulation log file generated when the Gaussian elimination program was allocated to *X* processors. The file "gelimb*X*.rpt" contains the simulation report file generated when the Gaussian elimination program was allocated to *X* processors.

APPENDIX D: Simulated Annealing Source Code

The source code for the Simulated Annealing program is contained on the enclosed PC-formatted disk in the subdirectory "\static\anneal". This directory contains:

sanneal.exe	Executable Simulated Annealing program.
sanneal.c	Source code for Simulated Annealing program.
task.h	Definition file for Simulated Annealing program.

This directory also contains a subdirectory "\static\anneal\atmos" which contains the data generated when applied to the module used in the Spatial Coherence Method of atmospheric analysis. The file "atmos.dat" is the input file and the files "atmosb*X*.dat" are the results when "atmos.dat" was allocated to a bus topology of *X* processors.

This directory also contains a subdirectory "\static\anneal\cvis" which contains the data generated when applied to the computer vision program. The file "cvin.dat" is the input file and the files "cvisb*X*.dat" are the results when "cvin.dat" was allocated to a bus topology of *X* processors.

This directory also contains a subdirectory "\static\anneal\gelim" which contains the data generated when applied to the Gaussian

elimination program. The file "gelim.dat" is the input file and the files "gelimb*X*.dat" are the results when "gelim.dat" was allocated to a bus topology of *X* processors.

APPENDIX E: Tabu Search Source Code

The source code for the Tabu Search program is contained on the enclosed PC-formatted disk in the subdirectory "\static\tabu". This directory contains:

tabu.exe	Executable Tabu Search program.
tabu.c	Source code for Tabu Search program.
task.h	Definition file for Tabu Search program.

This directory also contains a subdirectory "\static\tabu\atmos" which contains the data generated when applied to the module used in the Spatial Coherence Method of atmospheric analysis. The file "atmos.dat" is the input file and the files "atmosb***X***.dat" are the results when "atmos.dat" was allocated to a bus topology of ***X*** processors.

This directory also contains a subdirectory "\static\tabu\cvis" which contains the data generated when applied to the computer vision program. The file "cvin.dat" is the input file and the files "cvisb***X***.dat" are the results when "cvin.dat" was allocated to a bus topology of ***X*** processors.

This directory also contains a subdirectory "\static\tabu\gelim" which contains the data generated when applied to the Gaussian elimination program. The file "gelim.dat" is the input file and the

files "gelimb*X*.dat" are the results when "gelim.dat" was allocated to a bus topology of *X* processors.