1

**PM-1 3 1/4"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

MICRO

1930 COUNTY ROAD 8, BURNING WOOD, OH 44027 USA
TEL 612 438 7887 FAX 612 438 7887 TTX 612 438 7887

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Static Task Allocation for Distributed Computers

by

Fred Joseph Kaudel, B.A.Sc.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering,
Carleton University

January, 1992

© 1992, Fred Joseph Kaudel

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-75933-X

Canada

The undersigned hereby recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

*"Static Task Allocation for
Distributed Computers"*

submitted by Fred Joseph Kaudel, B.A.Sc., in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Thesis Supervisor

External Examiner

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
November, 1991

Abstract

This thesis considers the static allocation of tasks with a precedence graph structure allowing synchronization and conditional execution to a set of processors with a specific communications architecture. Branch-and-bound optimization is used to maximize the throughput capacity of the system. Several novel types of bounds are derived and used to evaluate partial allocations and to guide the optimization. These bounding functions are upper bounds on the attainable throughput after the remaining allocations are made. Various strategies, consisting of combinations of a choice of bounding function and various other search parameters (breadth-first or depth-first for example) are investigated. In both the search techniques and in the evaluation techniques, heuristics are developed that drastically reduce the time required to find a suitable task allocation from those of many alternative techniques. Applying these heuristics is shown to normally incur only a small penalty in application performance. The resulting optimizer can handle a wide range of static allocation problems. Its practical limit in problem size is on the order of about twenty tasks, so it is augmented with several clustering techniques to handle larger problems.

Acknowledgements

I would like to thank my thesis supervisor, Murray Woodside, for all of his guidance, encouragement and patience over the past several years. The support of the National Science and Engineering Research Council of Canada in the form of a post-graduate scholarship and other support is also appreciated. Lastly, I would like to thank my family and friends for their understanding.

Contents

1	Introduction	1
1.1	Background and Motivation	1
1.2	Problem Definition	3
1.3	Thesis Overview	6
2	The Static Task Allocation Problem	9
2.1	Task-Graph and Processor-Graph Notation	11
2.1.1	The Task-Graph Execution Model	15
2.1.2	Extending the Processor-Graph	16
2.2	Example Transaction Processing Application	17
2.3	Nested Task-Graphs	19
2.3.1	Graphical Representation of Nested Task-Graphs	20
2.3.2	Deriving General Task-Graphs from Nested Task-Graphs	26
2.3.3	Comparison of Nested Task-Graphs with the Literature	39
2.4	Multi-Class Task Allocation	30
2.5	Task Allocation as an Optimisation Problem	31
3	Review of Previous Static Task Allocation Work	33
3.1	Historical Development	34
3.2	System Models in the Literature	41
3.2.1	System Hardware Models	41
3.2.2	System Software Models	43
3.3	Cost Functions	46
3.4	Solution Methods	48
3.4.1	Graph-Theoretic Methods	49
3.4.2	Mathematical Programming Methods	50
3.4.3	Heuristic Methods	51
3.5	Related Allocation Problems	52
3.6	Other Recent Work	54
3.7	Summary	58

4	Allocation Search Techniques	59
4.1	The Task Allocation Search-Space	59
4.2	The Branch-and-Bound Task Allocation Method	63
4.2.1	Determining the Order of Task Allocation	69
4.3	Search-Space Size	70
4.3.1	Comparing Search-Space Size Increases	71
4.4	Hardware Symmetry-Based Search-Space Reduction	72
4.4.1	Processor Equivalence Groups	73
4.4.2	Dynamic T^{PREF} Modification	74
4.4.3	Static T^{PREF} Modification	75
4.5	Task Co-Allocation-Based Search-Space Reduction	76
4.6	Task-Graph Reduction	79
4.7	Summary	88
5	Allocation Evaluation Techniques	90
5.1	Performance Models for Complete Allocations	91
5.1.1	Modelling Task-Graphs with Closed Product-Form Queueing Networks	92
5.1.2	Approximate Mean Value Analysis	100
5.1.3	Interconnection Networks for Product-Form Queueing Models	101
5.2	Throughput Capacity Models	102
5.2.1	Bottleneck Bounds	103
5.2.2	Mean Path-Length Bounds	110
5.3	'Consistent' Lower-Bound Costs for Allocations	117
5.3.1	'Allocated' Work Only Lower-Bound Cost	117
5.3.2	'Load-Balancing' Lower-Bound Cost	118
5.3.3	'Communication Minimisation' Lower-Bound Cost	120
5.3.4	'Bottleneck' Lower-Bound Cost	120
5.3.5	'Mean Path-Length and Bottleneck' Lower-Bound Cost	121
5.4	'Inconsistent' Lower-Bound Costs for Allocations	121
5.4.1	'Ignore Partial Allocations' Lower-Bound Cost	122
5.4.2	'Bottleneck' Lower-Bound Cost	122
5.4.3	'Mean Path-Length and Bottleneck' Lower-Bound Cost	122
5.5	'Multi-Class' Lower-Bound Costs	122
5.6	Summary	123
6	Experience with Task Allocation Strategies	124
6.1	Experience Using the General Task-Graph Approach	124
6.1.1	The Stochastic Loop Task Allocation Problem	125
6.1.2	Notation	129
6.1.3	Comparison of Results	133

6.1.4	Task Allocation Ordering	140
6.1.5	Search Optimality Bracket	142
6.1.6	Hardware Symmetry-Based Search-Space Reduction	143
6.1.7	Task Co-Allocation-Based Search-Space Reduction	146
6.1.8	Task-Graph Reduction	148
6.1.9	Summary of Recommendations	148
6.2	Experience Using Nested Task-Graphs	150
6.2.1	Simulation Example	150
6.2.2	DSP Example	153
6.3	The Effects of Problem Size	156
6.3.1	Problem Size: Stochastic Loop	157
6.3.2	Problem Size: A Family of Nested Task-Graphs	157
6.4	Comparison with Stone's Method	162
6.5	Experience with Ma's Example	165
6.6	Summary	172
7	Conclusions	173
7.1	Summary	173
7.2	Areas for Further Research	176
	References	180
A	Detailed Task Allocation Experiment Results	188
A.1	Comparison of Results	189
A.2	Search Optimality Bracket	189
A.3	Problem Size	197
B	Nested Task-Graph Algorithms	201
B.1	Finding $\pi_{ii'}$ for Nested Task-Graphs	201
B.1.1	Algorithm to Find $\pi^{NIL}(G)$	204
B.1.2	Algorithm to Find $\pi^{RET}(G)$	205
B.1.3	Algorithm to Find $\pi^{SOP}(G, T_i)$	207
B.1.4	Algorithm to Find $\pi^{EOP}(T_i, G)$	208
B.1.5	Algorithm to Find $\pi_{ii'}$	209
B.2	Finding $b_{ii'}^L$ for Nested Task-Graphs	211
B.2.1	Finding $\pi_{ii'}^{NOT}$ for Nested Task-Graphs	212
B.3	Finding $\bar{t}$ and δ in Nested Task-Graphs	214
B.3.1	Algorithm to Find $t(G)$ and $\delta(G)$ for $\overline{SJ}$ Nested Task-Graphs	216
B.3.2	Algorithm to Find $\bar{t}(G)$ and $\delta(G)$ for SJ Nested Task-Graphs	217

List of Figures

1.1	General Distributed Computer Hardware Architecture	5
1.2	Task Allocation Problem Structure	5
1.3	Application Execution Framework	6
2.1	Example Task-Graph (a) and Processor-Graph (b)	10
2.2	Example Task-Graph Containing a Split-Join	16
2.3	Two-Level Transaction Processing Queueing-Network Model	18
2.4	Example TASK Task-Graph	20
2.5	Example SEQ Task-Graph	21
2.6	Example SEQ Task-Graph Containing OR Task-Graphs	21
2.7	Example SJ Task-Graph Corresponding to Figure 2.2	22
2.8	Example LOOP Task-Graph	22
2.9	Example Nested Task-Graph Equivalent of Figure 2.1a	22
2.10	Example LOOP Task-Graph Showing the Reference-Task T_1 and Re- Execution Loop	23
2.11	Example Task-Graph Illustrating Task-Free Paths	27
2.12	Nested Task-Graph Repeatedly Executing T_R and G	28
2.13	General Task-Graph Derived from Figure 2.6	29
2.14	General Task-Graph Equivalent of Figure 2.10	29
3.1	Structure of the STAP	33
4.1	Example Task Allocation Search-Tree	60
4.2	Example Task Allocation Search-Space	62
4.3	Example Simplified Task Allocation Search-Tree	63
4.4	The Simplified Task Allocation Search-Tree	64
4.5	Expected Cost-Bounds vs. Number of Nodes Expanded	67
4.6	Lower-Bound Cost vs. Number of Tasks Allocated	68
4.7	Example DM Search-Tree	75
4.8	Example Task-Graph	81
4.9	Reduced Task-Graph for Figure 4.8	81

5.1	Example Processor-Graph (a) and Corresponding Queuing Network (b)	93
5.2	Example SEQ Task-Graph	112
5.3	Example SEQ Task-Graph Containing OR Task-Graphs	113
5.4	Example SJ Task-Graph	114
5.5	Processor Workload Equalization Example	118
6.1	Example GTG Notation	126
6.2	General Stochastic Loop Task-Graph	126
6.3	L1P3 Processor-Graph (a), Hardware Architecture (b), and l matrix (c)	127
6.4	L2P2 Processor-Graph (a), Hardware Architecture (b), and l matrix (c)	127
6.5	L2P3 Processor-Graph (a), Hardware Architecture (b), and l matrix (c)	127
6.6	Nodes Created vs. L^s for L1P2T6 using Ll	137
6.7	Time to Solution vs. L^s for L1P2T6 using Ll	137
6.8	Nodes Created vs. L^s for L1P3T10 using Ll	138
6.9	Time to Solution vs. L^s for L1P3T10 using Ll	138
6.10	Nodes Created vs. L^s for L2P3T10 using Ll	139
6.11	Time to Solution vs. L^s for L2P3T10 using Ll	139
6.12	Example SJ NTG for Simulation	151
6.13	DSP Example SJ Nested Task-Graph	154
6.14	L2P4 Processor-Graph (a), Hardware Architecture (b), and l matrix (c)	155
6.15	Nodes Created vs. $ T $ for L1P3 using Ll	158
6.16	Time to Solution vs. $ T $ for L1P3 using Ll	158
6.17	Mean Nodes Created vs. $ T $ for the Family Examples	160
6.18	Mean Time to Solution vs. $ T $ for the Family Examples	160
6.19	Std. Dev. Nodes Created vs. $ T $ for the Family Examples	161
6.20	Std. Dev. Time to Solution vs. $ T $ for the Family Examples	161
6.21	Stone Example Task-Graph	163
6.22	Ma Example Task-Graph	165
B.1	Relationships between Task-Graphs G , $\overline{G}$, G' and $\overline{G}'$	218

List of Tables

2.1	Example Allocation Workloads for Figures 2.1a and 2.1b	11
5.1	Allocation Independent Parameters for Figure 2.1a	96
5.2	Allocation Dependent Parameters for Figures 2.1a and 2.1b using the Allocation of Table 2.1	97
5.3	Modified Allocation Dependent Parameters for Table 5.2	97
6.1	Effects of T^{ORDER} for L1P3T8 using <i>tall Ll</i>	140
6.2	Effects of T^{ORDER} for L1P3T8 using <i>tadd Ll</i>	141
6.3	Search-Space Reduction for L1P4T6 using increasing T^{ORDER}	144
6.4	Search-Space Reduction for L1P4T10 using increasing T^{ORDER}	144
6.5	Search-Space Reduction for L1P4T6 using decreasing T^{ORDER}	145
6.6	Search-Space Reduction for L1P4T10 using decreasing T^{ORDER}	145
6.7	Task Clustering Results for L2P3T10, Lf and <i>tsb</i>	147
6.8	Task-Graph Reduction Results for L2P3T10 and <i>tsb Lf</i>	149
6.9	Simulation Results and $\bar{\lambda}(a)$ for Different Workloads and N	152
6.10	Comparison with Allocator Results	152
6.11	DSP Example Task Clustering Results for <i>tspb Ll</i>	156
6.12	Original and Modified Stone Example Processing Workloads	162
6.13	Correspondence between Ma's Task Numbers and Our Task Numbers	165
6.14	Modified Ma Results for $ P = 2$ and <i>tsb Ll</i>	167
6.15	Modified Ma Results for $ P = 3$ and <i>tsb Ll</i>	168
6.16	Modified Ma Results for $ P = 4$ and <i>tsb Ll</i>	169
6.17	Modified Ma Results for Ma's allocations, using $ P = 3$, $L^s = 1$ and <i>tsb</i>	171
A.1	Results for L1P2T6 using <i>Ll</i>	190
A.2	Results for L2P2T6, L1P3T6, L2P3T6 and L1P4T6 using <i>Ll</i>	191
A.3	Results for L1P3T10 using <i>Ll</i>	192
A.4	Results for L2P3T10 using <i>Ll</i>	193
A.5	Results for L1P2T10, L2P2T10 and L1P5T10 using <i>Ll</i>	194
A.6	Results for L2P3T10 using <i>c</i> , <i>tspob</i> and <i>tadd</i>	195

A.7	Search Bracket Comparison for L1P2T10 using <i>tadd Lf</i>	195
A.8	Search Bracket Comparison for L1P2T10 using <i>tadd Ll</i>	196
A.9	Search Bracket Comparison for L2P3T10 using <i>tadd Lf</i>	196
A.10	Search Bracket Comparison for L2P3T10 using <i>tadd Ll</i>	197
A.11	Results for L1P3 Stochastic Loops with 6, 8, 10 and 12 Tasks	198
A.12	Results for L1P3 Stochastic Loops with 14, 16, 18 and 20 Tasks . . .	199
A.13	Results for L1P3 Stochastic Loops with <i>tsec</i> and <i>tsecb</i>	200

List of Terms

The following terms are used in several chapters of the thesis. For convenience, we summarize their definitions here:

allocation - an assignment of tasks to processors. There are two kinds of allocation, complete allocations and partial allocations. A complete allocation assigns each task to a processor, and a partial allocation may assign some, but does not assign all of the tasks to the processors. Each expansion of a partial allocation removes one task from the set of unassigned tasks and assigns it to a processor.

AMVA - Approximate MVA (mean value analysis).

branch-and-bound search - the search process described in Chapter 4 that is used to find a suitable (not necessarily optimal) complete allocation. Partial allocations are evaluated to find a lower-bound on their cost; based on this bound, the search chooses, or branches to, another allocation to evaluate.

cost - a performance metric of the allocation state at the search-node. The cost of complete allocations is determined using a cost function. For search-nodes representing partial allocations, the cost is a lower-bound on the cost of each leaf-node in the node's subtree. If a heuristic cost function is used, then the cost is only an estimate of the lower-bound cost.

EOP - end of path. A point-node in a NTG.

general task-graph (GTG) - a directed graph whose nodes represent the tasks and whose arcs represent the transition and communication between tasks. Section 2.1 details the meaning of these graphs.

GTG - a general task-graph. It expresses no precedence constraints.

job - a task being executed by a token. Any number of tokens (up to N , the multiprogramming level) may execute any task simultaneously.

join-task - the last task in a split-join. The join-task receives communication from all the branches of the split-join but only receives tokens from one branch.

link - A communication link, connecting two or more processors, and sending messages between tasks allocated to separate processors.

MVA - mean value analysis: used to evaluate the performance of product-form queueing networks.

nested task-graph (NTG) - a directed graph having a nesting property governed by the recursive expansion rules in Subsection 2.3.1. There are six types of expansions (called LOOP, NIL, OR, SEQ, SJ and TASK task-graphs); of these six types, only the NIL and TASK task-graphs don't contain other nested task-graphs.

NTG - nested task-graph

path-length - the time needed to execute a task-graph once, starting and ending at the beginning of the reference-task. In calculating the path-length, we assume that only one token exists in the task-graph when the reference-task is executed. Since workloads and transitions may be random, the path-length may be a random variable, and we consider either the mean path-length or lower-bounds on the mean path-length. Path-lengths for NTGs are derived in Appendix B.

point - a point-node on a NTG (appearing as a large black dot). There are eight kinds of point-nodes (points), as distinguished by their labels, which are one of the following: OR, $\overline{\text{OR}}$, SJ, $\overline{\text{SJ}}$, LOOP, $\overline{\text{LOOP}}$, SOP and EC. all of which are described in Subsection 2.3.1.

primary subgraph - used for NTGs. For brevity, the outermost NTGs within a NTG are called primary subgraphs.

processor - a hardware device servicing tokens' requests for task execution. Attached links allow communication between processors.

processor-graph - a directed graph introduced in Chapter 2 that represents the hardware available to execute the tasks. Each node is labelled with the processor that it represents and the directed edges are labelled with the link that carries communication from the processor at its tail to the processor at its head.

reference-task - each closed task-graph has one reference-task defining the end of the previous response and the start of the new response. See Chapter 2 for details.

response - tokens execute responses as defined by the task-graph. Each time a token begins executing the reference-task, the application has completed one response and begins another.

search-node - a node in a search-tree traversed by the branch-and-bound search process. Each node has two parts; the node's state, and the node's cost. Leaf-nodes are search-nodes that specify complete allocations and non-leaf-nodes specify partial allocations.

SJ - a split-join structure in a task-graph, consisting of a split-task, a join-task, and several branches.

SJ NTG - a NTG either being a SJ task-graph or containing any SJ task-graphs.

$\overline{\text{SJ}}$ NTG - a NTG not being a SJ task-graph and not containing any SJ task-graphs.

SOP - start of path. A point-node in a NTG.

split-task - the first task in a SJ. The split-task communicates to and sends a token to each branch of the SJ.

STAP - the Static Task Allocation Problem, on which this work focusses.

state - the part of a search-node describing which tasks have been allocated and to which processors they have been allocated.

subtree - the portion of the branch-and-bound search-tree below the search-node being considered.

task - an independent software unit with average input and output communication workloads with other tasks, and an average processor workload. These average workloads are assumed to be from exponential distributions, and to be uncorrelated between task executions. Communication workloads occur only between pairs of communicating tasks allocated onto separate processors. Tasks are re-entrant and are invoked by tokens: several of which can execute a task simultaneously (being multi-tasked by the processor to which the task is allocated).

task-free path - a path in a NTG that leads from one task to another via a series of edge traversals connected only by points or NIL task-graphs. This is a kind of "short-circuit" in the NTG.

task-graph - a directed graph representing the interactions between the tasks composing the application. There are two kinds of task-graphs: NTGs and GTGs.

- termination notification** - used in SJ task-graphs to represent the synchronisation between split-join branches at a join-task. The upper branch sends the token that will execute the join-task, while the other branches send termination notifications to the join-task when they have completed execution.
- throughput** - the frequency with which tokens complete a stochastically prescribed set of task executions describing an application response.
- token** - an active agent (representing a user) executing tasks. The number of tokens equals the multiprogramming level (N). Except for SJ task-graphs, there are a fixed number of tokens in the system.
- visit-ratio** - the average number of times a task in the task-graph or a node in the processor-graphs queueing network is executed per response.

List of Symbols

For quick reference, we will now summarize in two parts the important notation that has been developed in this thesis. The first part describes notation whose values remain the same for all allocations, while the notation in the second part varies between allocations. Note that in the following notation, superscripts *never* indicate a power, but rather only indicate an identifier.

The following symbols remain fixed for all the nodes or allocations in any task allocation problem and are used in several sections:

$b_{ii'}^L$ is the mean communications workload from T_i to $T_{i'}$ in communication units per execution of T_i ,

$\bar{b}_{ii'}^L$ is the mean communications workload from T_i to $T_{i'}$ in communication units every time the communication occurs,

b_i^P is the mean processing workload of T_i in processing units per execution of T_i ,

c_l^L is the speed of L_l in communication units per second,

c_j^P is the speed of P_j in processing units per second,

G is the NTG representing the application being modelled,

$\bar{G}$ is the GTG representing the application being modelled,

$J(l)$ is the set of processor identifier pairs (j, j') using L_l to send messages from P_j to $P_{j'}$ (i.e., $J(l) = \{(j, j') \mid l_{jj'} = l\}$),

L the set of communication links; individual links are referred to as L_l , where l is the unique link identifier,

$l_{jj'}$ is the identifier of the link to use in sending messages from P_j to $P_{j'}$, or $\emptyset$ if P_j can't communicate to $P_{j'}$,

N is the multiprogramming level in the task-graph (i.e., the number of tokens executing responses in the task-graph),

P is the set of processors; individual processors are referred to as P_j , where j is the unique processor identifier,

$\pi_{ii'}$ the probability that $T_{i'}$ is an immediate successor of T_i (i.e., that a token leaving T_i will execute $T_{i'}$ next),

T is the set of tasks to allocate onto P ; individual tasks are referred to as T_i , where i is the unique task identifier,

$T_{ii'}^{COAL} = 0$ if T_i can't be co-allocated with $T_{i'}$, or is 1 if they must be co-allocated, or any other value otherwise,

T_i^{ORDER} the order in which T_i is allocated; if $T_i^{ORDER} = u$, then T_i is the task allocated in the u^{th} layer of the search tree,

T_{ij}^{PREF} describes the processor preferences of T_i ; normally, $T_{ij}^{PREF} = 1$, but for each T_i that can't be allocated to P_j , $T_{ij}^{PREF} = 0$,

$w_{ii'}^L$ is the mean communications workload from T_i to $T_{i'}$ in communication units per response, $y_i b_{ii'}^L$,

w_i^P is the mean processing workload of T_i in processing units per response, $y_i b_i^P$, and

y_i is the average number of times T_i is executed per response.

The following symbols are used in several sections and have values that depend on the allocation (a) being considered:

a is the task allocation: $a_{ij} = 1$ iff T_i is allocated to P_j , and 0 otherwise,

A is the set of allocated tasks,

$\bar{A} \equiv T - A$, is the set of unallocated tasks,

$\delta_i = 1$ if T_i participates in the $\bar{t}$ calculation for an allocation of a NTG, and is 0 otherwise,

x_i^L is the workload of L_i in seconds of communication per response,

x_j^P is the workload of P_j in seconds of processing per response, and

$\bar{t}$ is the path-length bound for the allocation of a NTG.

Chapter 1

Introduction

Over the past two decades, computer system researchers and manufacturers have begun combining several processors into distributed computer systems, with the goal of providing economical power and reliability. Distributed computers present opportunities (e.g., better flexibility) as well as challenges (such as distributing the operating system and the user's application programs across the system). With the recent growth of interest in and availability of distributed computers (e.g., the Amoeba system [71]) comes the need to solve the problem of distributing the execution of a user's applications. This thesis addresses a specific, but widespread version of the application distribution problem, the static task allocation problem.

1.1 Background and Motivation

To execute an application in a distributed computer, one must first partition or decompose the application into a set of cooperating tasks¹ and then allocate these tasks to the processors for execution. We assume that the application has been partitioned², and that the resulting set of tasks have known execution and communication parameters. Such representations have long been used (e.g., [37, 67]).

¹tasks are often referred to in the literature as processes or modules.

²Formal aids to partitioning would be a useful addition, but are outside the scope of this thesis.

Task allocation selects a processor for each task (or sometimes also whether a task should be allocated to several processors³). This allocation must respect the constraints of the application and the processors: for example, two tasks that access data structures in shared memory cannot be allocated onto processors that do not share memory. Task allocation can be either static or dynamic.

An allocation is static when the allocation of tasks to processors is determined before the first task begins execution and remains fixed during the tasks' execution. Static task allocation uses information about the application to predict and distribute the load that its tasks will create onto the processors and communication links.

Statically allocated tasks stay on their processors until the application has finished executing. If the hardware configuration changes (e.g., due to a failure) and the application can still use the new configuration, then a contingency allocation (either pre-planned or computed as required) allows the revised static task allocation to be used⁴. Static task allocation can also predict the effects of changing hardware or software parameters on the application's performance.

In contrast, under dynamic task allocation, tasks are allocated when they are invoked, and may even be re-allocated during execution. Dynamic allocations must be calculated more quickly than static allocations since each task invocation allocates a task.

Both static and dynamic task allocation techniques are often rudimentary, with criterion functions only indirectly related to the designer's goals. Normally, the designer wants to maximise the application's performance (e.g., throughput or response time), subject to certain constraints. When using indirect criterion functions, however, the designer won't know how close the performance of the allocation returned by the allocator is to the performance of an optimal allocation. The designer should have, or be able to impose, an upper bound on this degradation.

³in [73], Towsley called these the restricted and unrestricted allocation problems, respectively. This work considers only the restricted problem where each task is allocated to one processor. Towsley himself converted unrestricted problems into restricted problems.

⁴Although contingency allocations extend the range of static allocation, questions of configuration change detection and recovery are outside the scope of this work.

Although many of the same methods apply to both static and dynamic allocation, this thesis considers only the Static Task Allocation Problem (STAP). The goal of this thesis is to develop and demonstrate techniques for statically allocating applications with less than roughly fifty tasks onto distributed computers having up to about ten processors. A criterion of optimality related to performance is used, and heuristics are sought to overcome the complexity problem. Progress towards this goal is illustrated by the results described in Chapter 6. The rest of this chapter informally defines the STAP that this thesis concentrates on, overviews the following chapters, and summarises our contributions.

1.2 Problem Definition

To evaluate the performance of a task allocation we must first characterise the demand for the application it represents. The applications we will model are all cyclically re-executed in a fixed number of replications that we will associate with “users,” whether these are separate human users or not; a replication may as easily be associated with a particular timer or sensor, or be completely self-contained. Whenever one user completes the application, another starts executing it. An optimal allocation will simultaneously minimise the response time of the application for a user and maximise the throughput of the application (the user response rate). We will concentrate on evaluating the throughput of the application and its throughput capacity, a reasonable and more easily evaluated measure of an allocation’s value.

Our static task allocation techniques optimize a single application’s performance by minimising the distributed computer’s throughput or throughput capacity for that application. It is assumed that, using some technique (e.g., [52, 77, 1, 76]), the application has been partitioned into a cooperating set of tasks (T) for allocation onto the distributed computer. To allow for data-dependent problems (where the task execution and communication workloads depend on the input data) both processing and communication workloads are modeled stochastically.

The three basic elements of the task allocation problem are:

- the set of tasks to be executed (T),
- the set of processors on which to execute them (P), and
- the set of interprocessor communications links (L).

Task allocation determines a static allocation for T onto P having optimal (or good enough) performance. $|P|$ (the number of processors) non-overlapping elements of the power set of T are selected. The elements' union must be T , their intersection must be the empty set, and their performance must be (at least close to) optimal. The allocation problem is then a search for the best $|P|$ disjoint subsets of T , each subset being executed on a different processor.

Figure 1.1 shows the general distributed computer hardware architecture chosen for our allocation problems. Each processor has local memory only and communicates with the other processors via messaging over the intercommunication network. The processors may have different execution speeds. Similarly, each communication link may have a different speed and may not connect all the processors. Where several links share a capacity, we consider them as a single, merged link. Messages from one processor to another are routed directly via the communication interfaces. For any processor A and processor B, the same communication link is always used to send messages from processor A to processor B. Another link may be used to send messages from processor B to processor A. No automatic forwarding of messages is allowed.

Together, P and L form the hardware architecture of Figures 1.1 and 1.2. Task allocation establishes the relationship shown on the left of Figure 1.2 as a diagonal box. The interprocessor communication message routing is represented by the lower connection relationship and is part of the fixed hardware architecture definition. Since at most one communication link sends all messages from one processor to another, the dashed relationship on the right follows from the other two relationships (if no link sends messages from one processor to another, the first processor cannot communicate with the second processor). Task allocation thus unambiguously determines both the communication link and processor workloads.

Figure 1.1: General Distributed Computer Hardware Architecture

Figure 1.2: Task Allocation Problem Structure

Figure 1.3 shows how this thesis fits into the overall application execution framework. Each application is assumed to have been decomposed into two components. The first component is the tasks themselves (T) and their execution requirements (such as their processing and communication workloads, and whether they can be co-allocated with each other or allocated to a processor). The second component is the hardware execution model characterising the demand for the application and containing the details of Figure 1.1 (such as the number of processors and links, their speeds, and their connectivity). Given an allocation, processor and link workloads are scheduled using simple rules, and several techniques are developed to evaluate the allocations. To find an optimal allocation, the allocator creates a sequence of allocations, guided by the evaluation of their performance.

Figure 1.3: Application Execution Framework

1.3 Thesis Overview

Chapter 2 defines the formal static task allocation problem (STAP), introducing two kinds of graphs to describe the application's execution, and another graph to describe the hardware architecture. Chapter 3 examines the contributions from the literature, relating them to this thesis, while chapters 4 and 5 introduce the allocation search and evaluation techniques we use to solve the STAP.

The branch-and-bound search technique described in Chapter 4 is applied to search for an optimal task allocation. This search is general enough for use even in extensions to the STAP (for example, where we need to find both an appropriate hardware architecture and a task allocation for that architecture). We describe ways to cluster the allocation tasks and to speed the search process. The search technique is also general enough to use any of many different allocation evaluation techniques (possibly redefining the performance metric to be maximised as well as the way to calculate it).

Evaluation techniques for potential task allocations are discussed in Chapter 5. These techniques estimate a partial or complete task allocation's performance for a distributed computer. The optimality of the allocation depends partly on the assumptions made by the evaluation technique. Chapter 5 also defines the range of hardware and software architectures that we model, and introduces several performance evaluation bounds and heuristics that can speed the search process.

Chapter 6 examines the performance of these modelling and search techniques on several task allocation problems. Chapter 7 concludes and describes some directions for future investigation.

The main contributions of this thesis are the bound analysis of application models and the search and evaluation techniques for automating the STAP. More specifically, the contributions of this thesis are the following:

- some aspects of the nested task-graph, and its analysis for bounds (Section 2.3, Subsection 5.2.2 and Appendix B),
- the method for exploiting symmetry (Section 4.4),
- the task clustering techniques and experience with their use in task allocation (sections 4.5 and 4.6, and Chapter 6),
- the use of performance bounds as costs in branch-and-bound search, especially mean path-length bounds⁵ and including bottleneck bounds, approximate mean value analysis, and load-balancing cost bounds (Chapter 5),

⁵throughput bounds were original at the time this work began in 1984.

- the set of heuristics used to guide the optimisation search (Section 6.1),
- the comparison and evaluation of the heuristics (Section 6.1), and
- the comparative evaluation of allocations on substantial problems (sections 6.2 through 6.5).

In the course of this research, the methods described in chapters 4 and 5 have been implemented in a mixture of Lisp, C and Assembly codes and are documented in [40]. The examples and experimentation support routines used to produce the results of Chapter 6 are also described in [40].

Chapter 2

The Static Task Allocation Problem

As was discussed in the last chapter, we assume that the application has been broken down into its basic subunits or software modules that we call tasks, and that some requirements or constraints have been placed on their co-allocation and allocation to the processors. Our Static Task Allocation Problem (STAP) is to allocate the application's tasks onto the available processors to maximise the application's performance, or with reduced effort, to find an allocation that is acceptably close to this maximum. The measures by which an application's performance is gauged will be introduced later in this chapter. The application is represented by a task-graph and the distributed computer is represented by a processor-graph. The STAP involves assigning the nodes in the task-graph to the processors in the processor-graph. Similarly, the arcs in the task-graph represent inter-task communication and are assigned to the links (arcs) in the processor-graph. Constraints that must be considered by allowable task allocations are described in Section 2.1.

In a task-graph (e.g., Figure 2.1a) each task is represented by a node and the directed arcs symbolise the flow of all interactions between tasks (i.e., for inter-task communication, synchronisation and control). The labels on the arcs are the probability that the task at its head will be executed next (i.e., be an *immediate successor*) when the execution of the task at its tail has completed. Similarly, a

processor-graph (e.g., Figure 2.1b) represents each processor by a node and each interprocessor communication a connection via a directed arc labelled with the link used for the communication. Here, two unidirectional links connect the two processors. The hardware architecture described by Figure 2.1b is a special case of that in Figure 1.1 (in particular, $|P| = 2$ and $|L| = 2$).

Figure 2.1: Example Task-Graph (a) and Processor-Graph (b)

Figures 2.1a and 2.1b show an example in which three tasks are to be allocated onto two processors. If two tasks are allocated onto separate processors, then any arcs in the task-graph connecting the two tasks are implicitly allocated onto the link of the arc in the processor-graph directed from the originating processor to the destination processor. Thus, task allocation also implies communications allocation. Consider the allocation of tasks to processors described by Table 2.1. Tasks 1 and 2 (T_1 and T_2 , respectively) are allocated to processor 1 (P_1) while T_3 is allocated to P_2 . The processor workload has thus been determined; the corresponding link workloads are also given in Table 2.1. Link 1 (L_1) communicates from T_1 to T_3 and from T_2 to T_3 , and L_2 communicates from T_3 to T_1 . Note that communication from T_1 to T_2 has not been assigned to any link since T_1 and T_2 have been allocated to the same processor.

Corresponding to our task-graphs, Karp and Miller [37] developed an early data-flow graph model to represent and analyze parallel computations. Nodes represented data operations and directed arcs represented data communications. Repeated computations appeared as loops on the graph. Karp and Miller didn't con-

P_1	P_2	L_1	L_2
T_1	T_3	T_1 to T_3	T_3 to T_1
T_2		T_2 to T_3	

Table 2.1: Example Allocation Workloads for Figures 2.1a and 2.1b

sider the decomposition of computations into computation graphs, but rather found several conditions necessary for the parallel computations to terminate and for the data queues to remain bounded in size. Data flows were predetermined and static, differing from our model which allows execution alternatives to be chosen dynamically and probabilistically.

This chapter formally describes the present STAP; Section 2.1 further defines the notation used in the above and other task-graphs and processor-graphs, and examines the application's execution model. Section 2.2 introduces our model for an example transaction processing application, and Section 2.3 introduces a restricted type of task-graph, the nested task-graph (in contrast with non-nested or general task-graphs, such as that of Figure 2.1a). Nested task-graphs incorporate extra task synchronisation and precedence information that is used to evaluate the time to complete a computation or response of the task-graph. Section 2.4 discusses allocating several applications to run on the hardware simultaneously, and Section 2.5 then formalises task allocation as an optimisation problem for use in later chapters. Chapter 3 surveys related allocation problems in the literature.

2.1 Task-Graph and Processor-Graph Notation

This section introduces the notation used in the task-graph and processor-graph of Figure 2.1 as well as the auxiliary application information that determines legal task allocations and allows us to evaluate such legal allocations. Subsection 2.1.1 will introduce the execution model for our task-graphs and Subsection 2.1.2 will discuss extensions to the processor-graph notation needed to model some applications.

The three basic STAP elements are the tasks to be allocated, the processors

to allocate the tasks to and the links that execute any interprocessor communication the task allocations require. All tasks, processors and links are members of one of the following three sets: T , P , and L . Set T contains all the tasks an application has been decomposed into, while sets P and L contain all the processors and communication links, respectively. To simplify our notation, we use the set size operator extensively (i.e., there are $|x|$ members of set x).

There are $|T|$ tasks in T , each having a unique identifier i , and being identified as $T_i \forall i \mid T_i \in T$. Similarly, each of the $|P|$ processors in P has a unique identifier j , and each of the $|L|$ links in L has a unique identifier l . To avoid introducing sets of identifiers or complicating the formulae in the following sections and chapters, we use $i \in T$ as shorthand for $i \mid T_i \in T$ (and do likewise for $j \in P$ and $l \in L$).

The following notation represents the task attributes, the relationships between tasks, and the constraints on the allocation of tasks to processors:

π : task successor probabilities. $\pi_{ii'}$ is the probability that $T_{i'}$ is an *immediate successor* of T_i , conditional on T_i terminating. If $T_{i'}$ is an immediate successor of T_i , then $T_{i'}$ is executed immediately after T_i with probability $\pi_{ii'}$. $\sum_{i' \in T} \pi_{ii'} = 1 \forall i \in T$ except for the split-task and the last tasks in all but one split-join branch (these terms are defined in Subsection 2.1.1 and Section 2.3). If the task-graph has no split-tasks, then π , the immediate successor probabilities matrix must be a regular stochastic matrix (irreducible) and can thus be used to generate a continuous time Markov chain that represents the flow of any token through the task-graph. Otherwise, if the task-graph has split-tasks, then π does not even form a stochastic matrix. When $\pi_{ii'} \neq 0$, an arrow from T_i to $T_{i'}$ appears in the task-graph¹.

b^P : task processing work. T_i executes for an average of b_i^P processing units per invocation; these units are defined later (e.g., as machine instructions) and are independent of which $P_j \in P$ that T_i is allocated onto. Note that different processors can have different processing speeds.

¹arrows to join-tasks often appear having $\pi_{ii'} = 0$, as will be explained in Section 2.3.

b^L and $\bar{b}^L$: inter-task communication needs. T_i communicates to $T_{i'}$ an average of $b_{ii'}^L$ communication units per invocation of T_i . When $b_{ii'}^L \neq 0$, an arrow from T_i to $T_{i'}$ appears in the task-graph. We also define the average communication workloads between T_i and $T_{i'}$ as $\bar{b}_{ii'}^L$. These workloads can be found from $b_{ii'}^L$ and $\pi_{ii'}$ using the following relationship:

$$\bar{b}_{ii'}^L \equiv \begin{cases} b_{ii'}^L / \pi_{ii'} & \text{if } \pi_{ii'} > 0 \\ b_{ii'}^L & \text{otherwise} \end{cases}$$

Each time communication occurs between T_i and $T_{i'}$, an average of $\bar{b}_{ii'}^L$ communication units are sent². For the general (non-nested) task-graphs we are now considering, it is reasonable to specify b^L and derive $\bar{b}^L$ from it; however, it will become clear in Subsection 2.3.2 that for nested task-graphs, $\bar{b}^L$ must be specified and b^L derived from it.

T^{PREF} : task processor preferences. These preferences are used to force each task to be allocated only onto those processors that can execute it. The entries in T^{PREF} , T_{ij}^{PREF} , are either 0 or 1, as described next:

if $T_{ij}^{PREF} = 0$ then T_i must not be allocated to P_j
 else T_i may but need not be allocated to P_j

T^{COAL} : task co-allocation constraints. These constraints can force or prevent tasks from being allocated to the same processor (i.e., *co-allocated*). Where several tasks require access to the same changing data – such as state data – we can force their co-allocation. Where several tasks require the exclusive use of a processor resource, we can prevent or exclude their co-allocation. The entries in T^{COAL} , $T_{ii'}^{COAL}$, are either 0, 1, or -1; the meanings of these entry values are defined as follows:

if $T_{ii'}^{COAL} = 0$ then T_i and $T_{i'}$ must not be co-allocated
 elseif $T_{ii'}^{COAL} = 1$ then T_i and $T_{i'}$ must be co-allocated
 else T_i and $T_{i'}$ may but need not be co-allocated

²whenever $\pi_{ii'} > 0$, the last communication unit sent from T_i to $T_{i'}$ carries a token that causes execution of $T_{i'}$ to start.

Note that $T_{ii'}^{COAL} = T_{i'i}^{COAL}$ and $T_{ii}^{COAL} \neq 0$.

The processor to link connections are illustrated by the processor-graph (e.g., Figure 2.1b). To consider all links uniformly, even a bidirectional link between two processors is shown as two separate directed arcs with the same label. We assume that the:

- processors are all uniprocessors communicating only via message passing, there being no shared memory, and that the
- communication links are all uni-server links (i.e., have a single channel).

Note that the changes required to model infinite-server processors and links (where users don't affect each others' service times) are discussed in Subsection 2.1.2. In addition to the above assumptions, the following notation describes the processor and link speeds:

c^P : processor speeds. P_j can execute c_j^P processing units per second.

c^L : link speeds. L_l can send c_l^L communication units per second.

The one relationship connecting the sets P and L is the interprocessor communication link matrix l , described by the processor-graph (e.g., Figure 2.1b), which determines if any link connects one processor to another, and if so specifies that link as follows:

if $l_{jj'} \neq \emptyset$

then use link $l_{jj'}$ to communicate from P_j to $P_{j'}$

(an arc labelled $L_{l_{jj'}}$ is directed from P_j to $P_{j'}$ on the processor-graph)

else P_j can't send any messages directly to $P_{j'}$.

Note that $l_{jj} \equiv \emptyset \forall j \in P$ because processors needn't send messages to themselves and that the definition of $l_{jj'}$ supposes that at most one link can send messages from one processor to another. Multiple connections from P_j to $P_{j'}$ can be modelled by a single link L_l with a speed c_l^L being the sum of their individual speeds.

This task-graph and processor-graph notation can model many realistic applications. After the next subsection further discusses the execution of our task-graphs, and Subsection 2.1.2 extends the processor-graph, Section 2.2 will describe an example of the use of this notation. Later chapters will allow us to find optimal

allocations based on models described using this notation.

2.1.1 The Task-Graph Execution Model

The general task-graph, as illustrated by Figure 2.1a has no starting task or finishing task, but is rather executed repeatedly, representing an application that cycles indefinitely. This subsection discusses the model for the execution of the task-graphs defined above in Section 2.1 and introduces the idea of a response to an application user (defining this response in terms of the task-graph corresponding to the application).

Define one task, say T_i , to be the *reference-task* of an application. Each time T_i is executed, we say that the task-graph has completed one *response* to an application user. We assume that each task is reachable from the reference-task by following arcs having positive π values on the task-graph.

A response of the task-graph begins with the execution of T_i and ends with the start of the next execution of T_i . Let us say that in Figure 2.1a, T_1 is the reference-task. Then only two kinds of responses occur when executing this simple task-graph: one being the following sequence of task executions: T_1, T_2, T_3 , and the other being: T_1, T_3 . Between the execution of T_i and T_i , b_{ii}^L communication units must be transferred via a link if the tasks are not co-allocated. The response includes, but ends with, communication from its last task to T_i .

A fixed number (N) of task-graph responses are always underway, representing a fixed multiprogramming level (of N responses); when one response completes, another begins. The execution of each response is modelled with a token (representing a user). The tasks that a token executes during a response are determined randomly by the path taken as the response is being executed; after a task has been executed, its immediate successor task is chosen randomly according to the immediate successor probabilities labelled on the directed arcs in the task-graph. When the token returns to T_i , its old response has finished and a new response starts. Token throughput at the reference-task then measures the system response throughput (i.e., its performance).

Task-graphs show only the immediate successor probabilities of each task,

and not their inter-relationships. For example, in Figure 2.2, T_S is a split-task that communicates to tasks T_1 , T_2 and T_J (the join-task). T_J executes immediately after it receives its token and communication from T_1 , and so is not a synchronising join-task. Communication from T_2 and T_3 may not even arrive before T_J finishes execution (late communication could be used for the next response). Because $\pi_{ii'}$ is misleading for join-tasks, and for split-join branches going directly from the split-task to the join-task, we use $\sum_{i' \in T} \frac{b_{ii'}^L}{b_{ii'}^L} > 1$ to indicate that T_i is a split-task, and $\sum_{i \in T} \frac{b_{ii'}^L}{b_{ii'}^L} - \pi_{ii'} > 0$ to indicate that $T_{i'}$ is a join-task.

Figure 2.2: Example Task-Graph Containing a Split-Join

Such an approach is not able to unambiguously represent a split-join that executes T_1 and T_2 50% of the time, but executes T_3 and T_4 the other 50% of the time, except by introducing dummy co-ordination tasks and task co-allocation constraints. Section 2.3 will formalise a restricted form of task-graph, nested task-graphs, in which synchronisation and precedence of tasks is more clearly expressed and in which the structure that we require for analysis is enforced. The next subsection describes extending the processor-graph definition to allow more applications to be represented via task-graphs.

2.1.2 Extending the Processor-Graph

Many applications take user input at some point during each response. We make up an input “task” to represent the user’s activity and delay. Since each user has his own terminal, we say that the processor for these input “tasks” is an

infinite-server, whose delay is independent of the number of active tasks (users). Where many application responses occur per terminal interaction or where there is insignificant waiting time per response, we can ignore the effects of these input "tasks" and use the notation of Section 2.1. Section 2.2 gives an example where this waiting time has been considered at a higher level model and doesn't appear in the STAP. This subsection discusses changes to the processor-graph notation of Section 2.1 needed to model user waiting times.

Terminals with identical communication connections are represented on the processor-graph by a single node whose processor is of the infinite-server type. The communication link connections are also amalgamated into a single link of infinite-server type that is labelled on both the arc coming to and leaving from the terminal processor node.

Tasks waiting for the user are preallocated to a terminal processor via T^{PREF} constraints and the communication to and from these tasks is thus implicitly allocated to the communication link dedicated to the terminals (if all terminals share a single communication link, then that link isn't set to being of the infinite-server type).

These extensions to the processor-graph allow more applications to be modelled. For simplicity, these extensions of the present work have not been developed further. The next section describes an example application using the notation and execution model of this section.

2.2 Example Transaction Processing Application

This section shows how the processor-graph and task-graph notation of Section 2.1 were used to model a transaction processing application. The application has a fixed number of users, each represented by a token. Each user may generate only one transaction at a time, whose flow through the system is represented by the user's token. Tokens repeatedly cycle through the following three phases:

- waiting for the user to generate a transaction,

- waiting in the transaction queue until it can be processed, and
- being processed, returning the response to the user.

Figure 2.3: Two-Level Transaction Processing Queueing-Network Model

Figure 2.3 illustrates the two-level application model used. The upper-level includes the generation of transactions and the task-graph's access controller, while the lower-level models the application task-graph that responds to, or executes, these transactions. We assume that there are always at least some fixed positive number, N , of user transactions outstanding and that the access controller can thereby maintain a fixed multiprogramming level of N tokens executing responses in the application task-graph (this assumption matches the execution model of Subsection 2.1.1). Increasing this number N may increase the system's throughput by more fully utilising the processors and links that the application task-graph executes on.

The tasks in the task-graph represent the various stages of transaction processing. Figure 2.1a shows a simple example where transactions require either two or three stages of processing, depending on their type; this type is randomly determined by the user, with 80% of the transactions requiring only two stages of processing (i.e., first T_1 and then T_3). Each response begins with a transaction entering the task-graph (beginning execution at the reference-task) and ends with the results of that transaction leaving the task-graph and reaching the user. At multiprogramming level N , we assume that there is negligible inter-response interference (i.e., interference between tokens) such as that caused by two responses attempting to write to a database simultaneously. Because N tokens are always executing the task-graph, we include both the transaction entry and result exit workloads in the

processing workload for the reference-task, T_1 for Figure 2.1a. For simplicity, we assume that this T_1 workload is the only upper-level model effect on the processors and links to which the task-graph will be allocated.

Clearly, to minimise the response time to the users' transactions at a given N , we must maximise the throughput of the lower-level (inner) model by choosing an appropriate task allocation (via the methods to be introduced in later chapters). The next section introduces a nested type of task-graph that can represent synchronisation as well as parallel execution, and shows how to transform this nested task-graph into the same type of task-graph described in Section 2.1.

2.3 Nested Task-Graphs

Nested task-graphs (NTGs) represent the structured execution, synchronisation, precedence and intercommunication of the tasks of well-structured applications. NTGs are developed in the present work because they can incorporate additional task synchronisation information beyond that contained in the *general* or non-nested task-graphs (GTGs) described earlier in this chapter. This additional information is used in Chapter 5 to develop bounds that improve the allocation quality.

NTGs can be recursively divided into simpler NTGs. Only the following five NTG patterns are required to represent the control, execution and communication flows (i.e., the structure) of the applications NTGs represent:

- execution of one task,
- sequential execution of each of several NTGs,
- probabilistic execution of one of several NTGs,
- parallel execution of each of several NTG split-join branches, coordinated by a split-task and a join-task, and
- probabilistic repeated execution of a NTG one or more times.

These NTGs are called TASK, SEQ, OR, SJ, and LOOP task-graphs, respectively. In addition to these five types, a NTG with no tasks, called a NIL task-graph,

represents the possibility of bypassing all processing work in an OR task-graph, and also represents a split-task communicating directly to its corresponding join-task in a SJ task-graph with a NIL task-graph branch. Figures and descriptions illustrating each of these task-graph types appear in Subsection 2.3.1.

As with GTGs, NTGs are repeatedly executed by tokens to generate responses for the users. Let G represent the user specified NTG. When G begins with an OR task-graph, we implicitly prepend a reference-task, say T_R , to G by putting T_R as the first and G as the only other NTG in a SEQ task-graph. To represent the repeated execution of G , the implied SEQ task-graph is put in an implied LOOP task-graph with no probability of stopping execution. These issues are clarified in the next subsection, where we detail NTGs and their graphical representation. So that allocation can be based on a common workload model, Subsection 2.3.2 then discusses how to derive the GTG model described in Section 2.1 from the NTGs of this section. Subsection 2.3.3 compares NTGs with three similar graphs from the literature.

2.3.1 Graphical Representation of Nested Task-Graphs

To represent these six NTG types, a graphical notation consisting of arrows, circles, dashed-boxes and points is used. An NTG is a directed graph having two types of nodes: NTGs and control-points. NTGs begin at an SOP control-point and end at an EOP control-point. Figures 2.4 to 2.10 are examples of this NTG notation. This subsection defines the meaning of these symbols after first describing the execution patterns specified by these figures.

Figure 2.4: Example TASK Task-Graph

Figure 2.5: Example SEQ Task-Graph

Figure 2.6: Example SEQ Task-Graph Containing OR Task-Graphs

Figure 2.4 shows a NTG that executes task T_1 , a simple NTG that is embedded in several of the other figures. Figure 2.5 specifies that NTGs G_A and G_B , each of arbitrary complexity and *sub-graphs* in the overall NTG shown, are executed in sequence. Figure 2.6 may represent the same NTG as Figure 2.5, but with graphs G_A and G_B expanded to be OR task-graphs, each executing one of the two tasks within it.

Figure 2.7 represents a SJ task-graph where the branches containing T_1 and T_2 are executed in parallel with the NIL task-graph branch that represents direct communication from T_S , the split-task, to T_J , the join-task. Each token arriving at T_S is split into three tokens, one being sent out onto each branch. Only one of the three arrowheads attached to T_J – the upper one – has a non-zero probability annotated to its arrow; this arrow carries the only tokens reaching T_J . T_J must wait for all three branches to finish processing and communication before executing

Figure 2.7: Example SJ Task-Graph Corresponding to Figure 2.2

its own processing workload and sending a token out; the other two branches send termination notifications, rather than tokens, when they have executed all their workloads. These details are discussed further below.

Figure 2.8: Example LOOP Task-Graph

Figure 2.8 represents a LOOP task-graph where NTG G_C is always executed at least once, and has a ten percent chance of being executed again immediately after each execution.

Figure 2.9: Example Nested Task-Graph Equivalent of Figure 2.1a

Figure 2.10: Example LOOP Task-Graph Showing the Reference-Task T_1 and Re-Execution Loop

Figures 2.9 and 2.10 are the only example NTGs we have included that explicitly show the implicit reference-task and re-execution loop. Figure 2.9 shows a NTG equivalent to Figure 2.1a, and Figure 2.10 shows an application where a response may execute just the reference-task T_1 , or T_1 and any number of executions of T_2 . Normally in this thesis, the re-execution loop is not shown, but it is always implied.

Each NTG has one starting point³ (labelled SOP) where its execution begins and one ending point (labelled EOP) where its execution ends. Where several NTGs are shown in the same figure (as in all the above example NTG examples but that of Figure 2.4) only the SOP and the EOP of the outermost NTG in the overall NTG are labelled. To simplify the figures, the other SOP and EOP points (i.e., all but the first SOP and the last EOP in the overall NTG) are implied but not labelled.

Neither the points nor the arrows of the graphical notation for NTGs are themselves executed; they merely extract the common elements of the control and execution flows of the tasks representing the structured application. Control and execution flows occur directly between tasks. The following description details the syntactic rules for NTGs and their corresponding meanings. After a description of arrows, we define four types of circles, dashed boxes and points:

- each arrow represents the flow of either or both tokens and communications

³a point is a point-node on the NTG, appearing as a black dot.

workloads leaving from wherever the arrow-tail is attached and directed to wherever the arrowhead is attached. Arrows are never attached to other arrows, but are rather attached to the objects in the NTG (i.e., points, circles and dashed-boxes). An arrow's head and tail are never attached to the same object. Annotated to each arrow is the probability that a token arriving at whatever object the arrow-tail is attached to causes a token to follow this arrow. Except at a split-task (to be discussed later) only one token can leave any task when one token arrives. At most one arrow leaving any object may have an annotated probability of zero. The probability that a token follows this arrow is zero, but the probability that a token arriving at the object the arrow-tail is attached to becomes a termination notification is one minus the sum of the probabilities annotated to any other arrows leaving that object. This unusual convention is used because in SJ task-graphs, only one branch sends a token to the join-task (i.e., all but one arrow reaching each join-task has zero annotated to it).

- each circle represents the *task* labelled therein, which receives a token, with a possible communications workload, via the single attached arrowhead, executes its processing workload and then sends the token and a possible communications workload onto the single attached arrow-tail. Split-tasks and join-tasks differ from "normal" tasks both in their operation and in the symbols (modified circles) used to represent them. The next two items elaborate on these differences.
- each $\square$, such as that of T_S in Figure 2.7, denotes a *split-task*, which receives one token, executes its processing and communications workloads, and then sends one token onto each of the arrows attached to its right wall (each has an annotated probability of one).
- each $\bigcirc$, such as that of T_J in Figure 2.7, denotes a *join-task*, which receives a token from a predetermined arrowhead (all our figures show this arrowhead as coming from the upper branch in the SJ task-graph); the other attached arrowheads represent branches having no immediate successor tasks (i.e., their attached arrowheads have annotated probabilities of zero; when such a branch finishes all its execution and communication workloads, it sends a termination notification, rather than a token, to the join-task). The join-task waits until it receives one token and all the required branch termination notices before executing its processing workload and sending a token and a possible communication workload onto the attached arrow-tail.
- each dashed circle, as shown in Figure 2.10, represents a NIL task-graph, which passes any input token and communication directly to its output. This may

probabilistically bypass any choice in an OR task-graph or may send communication directly from a split-task to a join-task with no intermediary tasks (via a branch consisting only of a NIL task-graph). In the latter application, unless the branch is the one chosen to send tokens to the join-task, the annotated probability of the arrow from the NIL task-graph to the join-task is zero and the token is interpreted as a termination notification at the join-task.

- each dashed box (identified as graph C and labelled G_C in Figure 2.8) represents an arbitrary NTG. Only one entering and one leaving arrow is permitted.
- the two points labelled SOP and EOP define the “start of path” and “end of path” points, respectively. Each NTG response starts with a token leaving SOP, and ends with the token reaching EOP.
- each point labelled OR defines the start of an OR task-graph. There must be only one entering arrow and at least two leaving arrows. It receives one token and randomly passes it to one of its leaving arrows. The sum of the annotated probabilities on the leaving arrows must be one.
- each point labelled $\overline{\text{OR}}$ defines the end of an OR task-graph. There must be as many entering arrows as there were leaving arrows on the matching OR point and there must be only one leaving arrow.
- each point labelled LOOP is part of a LOOP task-graph. Two arrowheads and one arrow-tail are attached to each LOOP point. One arrowhead comes from the start of the LOOP task-graph and the other comes from the matching $\overline{\text{LOOP}}$ point within the LOOP task-graph.
- each point labelled $\overline{\text{LOOP}}$ is part of a LOOP task-graph. The attached arrow-tail leading to the matching LOOP point is annotated with the probability of the NTG between the LOOP and $\overline{\text{LOOP}}$ points being executed again immediately. The LOOP task-graph is thus a semi-Markov process, with the decision to execute the NTG within again being independent of past decisions.

Because SJ task-graphs alter the analysis methods used for NTGs, any NTG containing at least one SJ task-graph is called a SJ NTG; all other NTGs are called $\overline{\text{SJ}}$ NTGs. Split-tasks and join-tasks must appear as matched pairs in SJ NTGs; with their branch NTGs, a matching split-task and join-task pair form a NTG with all external communication entering via the split-task and leaving via the join-task. At most one SJ task-graph branch may have a path through it that only executes NIL task-graphs (otherwise, it would be unclear which path represented the direct

communication from the split-task to the join-task, since this communication occurs at most once). To allocate a NTG, we first convert it into a GTG, as described in the next subsection.

2.3.2 Deriving General Task-Graphs from Nested Task-Graphs

To allocate the tasks in a nested task-graph (NTG), we first derive a general task-graph (GTG) version of it, thereby finding the required workload parameters. The derived GTG is equivalent to the NTG in its workloads, but may have a simplified control flow, because task synchronisation delays represented at join-tasks in the NTG can't be represented in the GTG model of Section 2.1; however, by using the NTG from which a GTG was derived, these delays can be approximated by the path-length bounds in some of the performance models of Chapter 5. This subsection discusses the conversion process and gives four examples of GTGs derived from NTGs (three of which are equivalent in both workloads and control flows).

GTGs have reference-tasks and explicitly show that they repeatedly execute responses, while the NTGs they were derived from may neither have such a reference-task nor show their repeated execution. If these are missing, the conversion process will make them explicit.

Before discussing the conversion process, we introduce a concept it uses, that of a *task-free path* between two tasks; a task-free path from T_i to $T_{i'}$ in NTG G is any sequence of arrows, connected only by points or by NIL task-graphs, that leads from T_i to $T_{i'}$. Note that several task-free paths may lead from T_i to $T_{i'}$, and that a task-free path from T_i to itself is a self-loop. A task-free path between T_i and $T_{i'}$ means that during the execution of G , there is a non-zero probability of communication being sent from T_i to $T_{i'}$; when this communication occurs, $T_{i'}$ either also receives a token from T_i or also receives a termination notification from T_i . Standard path-finding algorithms must be modified to find the existence of a task-free path from T_i to $T_{i'}$ in G because there are two kinds of nodes: tasks and non-tasks (i.e., NIL task-graphs and points). Only arrows attached to non-task nodes can be traversed in reaching $T_{i'}$ from T_i via a task-free path.

Figure 2.11: Example Task-Graph Illustrating Task-Free Paths

Figure 2.11 illustrates this notion of task-free paths. Task-free paths from T_2 to T_1 include $T_2, \overline{OR_2}, \overline{LOOP}, \overline{LOOP}, OR_1, T_1$, and $T_2, \overline{OR_2}, \overline{LOOP}, \overline{LOOP}, OR_1, NIL, \overline{OR_1}, OR_2, NIL, \overline{OR_2}, \overline{LOOP}, \overline{LOOP}, T_1$. Similarly, task-free paths from T_1 to T_2 include $T_1, \overline{OR_1}, OR_2, T_2$, and $T_1, \overline{OR_1}, OR_2, NIL, \overline{OR_2}, \overline{LOOP}, \overline{LOOP}, OR_1, NIL, \overline{OR_1}, OR_2, T_2$.

To convert NTG G to GTG $\overline{G}$ we follow these four steps:

1. if G doesn't show an initial reference-task and its own repeated re-execution, we make these explicit as Figure 2.12 suggests, by adding a reference-task (T_R in Figure 2.12) before G and nesting both in a LOOP task-graph with an exit probability of zero⁴. The implicit reference-task so created is given zero communications and processing workloads. Further references to G refer to a NTG with a reference-task that shows its repeated execution.
2. recursively replace all the dashed boxes in G by their equivalent NTGs (removing their SOP and EOP points) so that all tasks and arrows are shown.
3. for each task in G , generate a corresponding task in $\overline{G}$ with the same identifier. Note that tasks in $\overline{G}$ are all the same type, split-tasks and join-tasks

⁴if G is a TASK task-graph, a SJ task-graph, or a SEQ task-graph beginning with either of these, then this task is the new reference-task and no other task need be added.

being indistinguishable from other tasks. NIL task-graphs are used in the next step to find task-free paths in G , but do not appear in $\bar{G}$.

4. whenever there is a task-free path from T_i to $T_{i'}$ in G , put an arrow from T_i to $T_{i'}$ in $\bar{G}$ and annotate a probability of $\pi_{ii'}$ to it, as found using the algorithms in Section B.1. If there is a unique task-free path from T_i to $T_{i'}$ in G , then $\pi_{ii'}$ is simply the product of the probabilities annotated to those arrows. There may however be an infinite number of task-free paths from T_i to $T_{i'}$ in G (because of one or more LOOP task-graphs along these paths), a case handled in Section B.1.

Figure 2.12: Nested Task-Graph Repeatedly Executing T_R and G

Applying this conversion procedure to Figure 2.6 yields Figure 2.13, where T_S is the reference-task initiating responses but having insignificant communication and processing workloads itself. Similarly, deriving a GTG from Figure 2.7 yields Figure 2.2, with T_S as the reference-task. The explicit task synchronisation at T_J in Figure 2.7 is however lost in Figure 2.2 because the GTG does not represent it.

Figures 2.9 and 2.10 already show the reference-task and re-execution loop, meaning that they needn't be added in step 1. Converting Figure 2.9 to a GTG yields Figure 2.1a, the same GTG from which it was derived.

Consider the conversion of the graph in Figure 2.9. Because a token leaving both tasks on Figure 2.10 can reach that task again via a task-free path through the NIL task-graph, Figure 2.14 has the two self-loops that this implies. Unlike the three conversion examples above, the immediate successor probabilities π_{11} , π_{12} , π_{21} and π_{22} , labelled on Figure 2.14 are not obvious; each depends on both of the probabilities u and v annotated to the arrows on Figure 2.10 and are calculated using the methods of Section B.1.

Figure 2.13: General Task-Graph Derived from Figure 2.6

Figure 2.14: General Task-Graph Equivalent of Figure 2.10

For larger NTGs, the $\pi_{ii'}$ probabilities can be even less obvious. Because the modeller cannot be expected to determine these $\pi_{ii'}$ values, we propose that the modeller specify $\bar{b}_{ii'}^L$, the average communication workload from T_i to $T_{i'}$ each time the communication occurs. To convert from $\bar{b}_{ii'}^L$ to $b_{ii'}^L$, the task successor probabilities, $\pi_{ii'}$, must first be found using the method of Section B.1, and then used in Section B.2 ⁵.

Contrasting with the $b_{ii'}^L$ values discussed above, the processor-graph, task-constraints and b_i^P values are unaffected by changes between nested and general task-graphs.

⁵Recall that $b_{ii'}^L$ was specified in Section 2.1 as being the average communication workload between T_i and $T_{i'}$ per execution of T_i . Since this communication only occurs when any task-free path from T_i to $T_{i'}$ is taken, and this probability is often tedious to find (especially where there are an infinite number of such paths, as often results when NIL task-graphs are used), it is unreasonable to expect the modeller to specify $b_{ii'}^L$ for NTGs.

2.3.3 Comparison of Nested Task-Graphs with the Literature

NTGs are similar to Chu *et al*'s control flow graphs [13], Alford's requirements networks (R-nets) [2] and Peng and Shin's task flow graphs [54]. NTGs differ from Alford's R-nets in that we have only one output from the NTG and that we allow NIL task-graphs. Also, we allow concurrent execution of NTGs by several tokens, whereas only one R-net is active at any time. Chu *et al*'s control flow graphs are not structured and also allow no NIL task-graphs. Control flow graphs can have multiple entry and termination points, and represent loops as conditional backwards branches. Data flow is also more independent of execution flow than in NTGs. Peng and Shin use the same building blocks as NTGs, other than their lack of NIL task-graphs and different semantics for SJ task-graphs, and their application for modelling modules within tasks rather than for modelling the sequencing of the tasks themselves.

Chapter 5 will use the extra task synchronisation and precedence information in NTGs to derive several performance bounds. The next section discusses the effects on our notation of allocating several applications (task-graphs) onto the same processor-graph for simultaneous execution.

2.4 Multi-Class Task Allocation

For simplicity, we have previously assumed that each token or user executes the same task-graph. Where several applications need to be run on the same hardware, each application is characterised by its own task-graph. The users, or tokens, of each application task don't execute the same task-graph because they use different tasks, or the same tasks differently. To model the execution of these different applications by tokens, we partition the tasks into task classes (all the tasks of an application belong to the same class) and we partition the tokens into corresponding token classes. Tokens of each class repeatedly execute the task-graph representing their application. Each task is executed only by tokens of its class and communicates only with tasks of the same class. This section discusses the effects of task

classes on the notation developed in Section 2.1.

Let K be the set of all task class identifiers k , and let each task belong to one of these task classes. Then K partitions T into $|K|$ non-intersecting subsets. Extending the previous notation for the tasks and the set of tasks:

- T^k is the set of class k tasks, and
- T_i^k is T_i in class k .

Each class of tasks appears on a separate task-graph. Tokens of each class execute the task-graph of that class to complete responses, and hence no communication can occur between tasks of different classes. The other task-graph notation of Section 2.1 must also be modified to consider only tasks within one task class by defining $\pi_{ii'k}$, $b_{ii'k}^L$ and $T_{ii'k}^{COAL}$ for all task pairs, T_i and $T_{i'}$, within each task class k , and by changing the single fixed multiprogramming level N to $|K|$ multiprogramming levels, N_k , one for each class k .

Using these multiple task classes, more complex applications can be decomposed into tasks and multiple applications running on the same hardware simultaneously can be modelled. For simplicity of notation, this thesis mostly considers single classes, but the methods easily generalise to multiple classes.

2.5 Task Allocation as an Optimisation Problem

This section restates the single class STAP of Section 2.1 as an optimisation problem, transforming it into a minimisation problem suitable for use by chapters 4 and 5, and finally revisits the multi-class case. Chapter 5 will describe several techniques to evaluate the performance of task allocations.

To phrase task allocation as an optimisation problem, we need to formalise the allocation of tasks to processors. Task allocation produces the allocation matrix a , wherein:

$$a_{ij} \equiv \begin{cases} 1, & \text{if } T_i \text{ is allocated to } P_j; \\ 0, & \text{otherwise,} \end{cases}$$

Note that $\sum_{j \in P} a_{ij} = 1, \forall i \in T$ since each task must be allocated.

System throughput (λ), is the number of task-graph responses per second that the system completes; clearly, λ is a function of the allocation a . Chapter 5

describes several techniques used to find the steady-state value of $\lambda(a)$. The task allocation optimisation problem may be stated as the following nonlinear integer programming problem:

$$\max_a \lambda(a).$$

subject to the following constraint sets:

$$a_{ij} \leq T_{ij}^{PREF}, \forall i \in T, j \in P$$

$$a_{ij} \in \{0, 1\}, \forall i \in T, j \in P$$

$$\sum_{j \in P} a_{ij} = 1, \forall i \in T$$

$$a_{ij} a_{i'j} = 0, \forall i, i' \in T, j \in P \mid T_{ii'}^{COAL} = 0$$

$$a_{ij} - a_{i'j} = 0, \forall i, i' \in T, j \in P \mid T_{ii'}^{COAL} = 1$$

$$a_{ij} a_{i'j'} = 0, \forall i, i' \in T, j, j' \in P \mid b_{ii'}^L \neq 0, l_{jj'} = \emptyset$$

To obtain a minimisation problem, the above objective function may be restated as:

$$\min_a \lambda^{-1}(a),$$

where $\lambda^{-1}(a)$ represents an allocation's cost. It is this minimisation form of the optimisation problem that we will use throughout this thesis.

Multi-class STAPs, as described in Section 2.4, require only straightforward modifications to the constraints and to the objective function (we will maximise some function of the throughputs for each class). Section 5.5 will further consider the effects of multi-class STAPs, but most of our work deals with single class STAPs.

Chapter 4 will examine ways of searching for such an optimal allocation a , given a function $\lambda^{-1}(a)$; Chapter 5 will rephrase and expand this minimisation problem in several ways to speed the evaluation of each possible allocation in the search for an optimal one. The next chapter discusses some relevant allocation literature.

Chapter 3

Review of Previous Static Task Allocation Work

Many authors have considered problems related closely to the Static Task Allocation Problem (STAP) of this thesis. Often these authors use different terminology to describe the same concepts. For example, task allocation is often called module allocation, and the entire allocation problem is sometimes termed a “graph-matching problem.” This chapter consolidates this terminology and examines the literature, first from a historical perspective and then according to certain aspects of the problem.

Figure 3.1: Structure of the STAP

Figure 3.1 illustrates a framework for the STAP. Applications to be allocated are first decomposed into hardware and software models; the software model is then

partitioned into tasks. A cost function applied to allocations of the tasks to the hardware allows allocations to be compared and is used by the solution technique to find an optimal or satisfactory allocation.

Section 3.1 describes the historical development of the literature based on a few key papers. Section 3.2 examines the assumptions made by various authors and the system models they used. Section 3.3 discusses the literature on the basis of the cost functions used, and Section 3.4 then categorises the solution techniques. This substantially covers the literature most relevant to our immediate problem (although some papers have been omitted for reasons of space). Section 3.5 briefly surveys two other allocation problems related to the STAP, namely: file allocation and dynamic task allocation. Several other recent works related to task allocation are then introduced in Section 3.6.

3.1 Historical Development

Gyls and Edwards [30] presented some of the earliest static task allocation work. They compared a mathematical programming method and a heuristic clustering method, both trying to minimise the total communications workload while respecting processor memory and speed limitations. No consideration was given to potential concurrency in the communication media. The mathematical programming method was time-consuming for large problems, while the heuristic clustering method gave sub-optimal, but still good, allocations. (An example of this sub-optimality is that they ignored the queueing component of response times).

At about the same time, Jenny and Kümmerle looked at designing an integrated communications switching node having up to forty homogeneous processors interconnected by a high-speed bus [35]. To delay throughput saturation caused by communications overhead as processors were added, they designed a symmetrical distributed algorithm. Allocation of the tasks was manual and intended to balance the processor workloads while using the bus as little as possible. Soon after this work, Jenny developed a formal method for characterising such systems and for automatically allocating the tasks to the processors [34]. Given a fully connected

homogeneous network of processors with custom peripherals optionally attached to each processor, and a set of relations between tasks, such as inter-task communication and task co-allocation requirements, Jenny found an allocation of the tasks to the processors that respected the relations and minimised the bus usage. Since parallelism within the application was not modelled, Jenny recognised that undesirable allocations could occur. To combat these, he suggested two steps; (1) estimating the value of parallelism between two tasks and assigning them a reduced or negative communication cost, and (2) manually recognising the unbalanced processing loads and assigning reduced communication costs to move tasks around. Although Jenny's method only applied exactly to sequential programs, Jenny clearly believed they could be applied to parallel applications, especially those similar to that of Jenny and Kümmerle.

Haessig and Jenny [31] extended Jenny's earlier work [34], by allowing much larger processor networks to be solved approximately. A fast heuristic algorithm minimised the total communications traffic among identical processors fully connected by identical communications links. This was a less general problem than Gylys and Edwards', allowing solutions to be found more quickly. As in [34], since neither application performance nor processor loading was considered, allocation restrictions that were neither natural nor automatic were needed to force the use of more than one processor. Haessig and Jenny's algorithm thus worked well only when communications was the system bottleneck (i.e., minimising communications maximises performance). Even then, they did not consider the way communications was distributed among the links. Negative communication was again proposed to promote parallelism. Heuristic post-processing of the allocation allowed resource limits such as memory to be respected.

Stone extended [31] by using a graph-theoretic min-cut algorithm to minimise the sum of all the processor and communication loads for a network of communicating tasks allocated to $|P|$ processors [69]. He assumed that no parallelism existed within the application, allowing the sum of all task execution and communication times to approximate the completion time of the application (and to be used as the allocation's cost). The processing load for each task depended on the

processor it was allocated to; communication loads between tasks, however, needed to be independent of which processors the tasks were allocated to. Infinite-weight communications arcs could be applied between tasks to force their co-allocation.

Stone then proposed a $|P|$ -dimensional min-cut algorithm to find an optimal allocation¹. Only one processor, the fastest, was used, unless some tasks were manually allocated to other processors. These manual preassignments were not obvious, were not optimal and limited the usefulness of this approach. It was awkward to prevent the allocation of a task to a group of processors that could not execute it. Stone's model could not consider the effects of multiple communication links, an important factor in many realistic task allocation problems. Note that the processor loads of Stone's tasks on different processors were not related by a processor speed factor, and the allocations were heuristic.

In [70], Stone and Bokhari extended Stone's earlier results by considering allocation to more than three processors and the allocation of special tree-like task-graphs. For these special task-graphs (which limited communications but implied no precedence relations among the tasks), dynamic programming found a minimum cost allocation. Stone and Bokhari also considered the application of their cost function to applications containing parallelism. Chapter 6 compares our allocations with those of Stone's method for a problem based on an example in both [69] and [70].

Ma *et al* [48, 47] used a branch-and-bound search technique in a deterministic environment, where real-time responses to external requests needed to be guaranteed. The total communications workload was minimised, while constraints preventing task co-allocation, satisfying task allocation preferences, response time needs, and memory capacities, were satisfied. Ma suggested manually tuning the problem parameters to try to satisfy other requirements (such as queueing delays and task execution times). The application was driven by periodic external stimuli, and hence had a fixed throughput. Response time could be optimised, but this wasn't

¹this min-cut algorithm becomes intractable for $|P|$ greater than three [70]. Since parallelism in the application was forbidden, it was unlikely that more than three processors would be used anyway. Even for three processors, a heuristic solution algorithm was proposed.

looked at. A flexible hardware architecture would allow cost savings by reducing the hardware cost while still meeting the constraints. Ma didn't consider this. The search method used in the present thesis is similar, but is based on performance derived criteria, and avoids hand-tuning parameters.

Price [56] discussed a quick suboptimal extension of Stone's work to the case of more than three processors. A heuristic initial allocation was made and then improved step-wise by reallocating one task at a time, always reducing the sum of processing and communication costs. In addition to the limitations of Stone's method, Price showed that optimality can only be guaranteed for her method when communications is negligible (a strong condition). Price considered reallocating groups of more than one task to guarantee finding an optimal solution, but acknowledged that this approach exceeds the complexity of a complete search-space evaluation, and is thus usually impractical.

Price and Krishnaprasad [57] examined several apparently polynomial-time heuristics (all still ignoring queueing delays) for the same problem. An iterative heuristic doing a repeated nearest neighbour search to minimise an artificial cost criterion was found to be computationally expensive. A two phase clustering algorithm was slightly quicker. Inter-task communications matrices "usually" exhibited a narrow banded structure, which a quick quadratic program solved. The quicker heuristics provided worse solutions, with the quadratic program solutions having twice the criterion cost of the nearest neighbour search solutions.

Stone's method was heuristically extended to the general many processor case by Lo [45], who compared several artificial criterion functions penalizing both intertask communication and task co-allocation interference. When evaluating an allocation, costs arising from the tasks were totaled; two tasks assigned to the same processor incurred an interference cost, as did communicating tasks assigned to different processors. These costs depended on the system architecture and the allocation of other tasks; execution times and system overheads were considered. A fast heuristic algorithm found sub-optimal solutions to this cost minimisation problem. Under restricted cases, her heuristics generated optimal solutions (in the sense of minimising the criterion function, rather than maximising system performance).

Being an indirect approach, the designer needed to adjust costs until an acceptable solution was obtained.

In [66], Sips discussed several heuristic static task allocation methods for "recurrent task systems," where the task-graph consisted of tightly coupled "systems" of tasks with no internal feedback and one coordinating task, and was executed cyclically. Each task could have arbitrary predecessors, but only within its own task system. The heuristics all considered the execution-only path-lengths of the acyclic task systems. Sips' methods allocated the task systems to clusters of homogeneous processors where they were dynamically executed. Communication costs within a cluster were ignored and the global network interconnecting the clusters was only used by the coordinating task that communicated between the task systems at the end of each execution cycle. The heuristics were compared with dynamic system-wide allocation (i.e., where one processor cluster was used rather than two to fourteen smaller processor clusters) and were found to give at most 15% longer cycle-times for a series of random task-graphs. Allocating systems of tasks is known as the job allocation problem and is a coarser kind of allocation problem than the STAP.

As did Ma *et al*, Shen and Tsai [63] used the branch-and-bound search method on task-graphs similar to Stone's. They minimised the total communications plus processing time for that processor with the largest total, a measure they claimed was novel (but that appears to originate in Doty *et al* [19]). A heuristic bound was developed to speed the branch-and-bound search method used for the minimisation. They called their approach "graph-matching," because their allocations matched the nodes and edges of their undirected task-graph to the nodes and edges of their undirected processor-graph. No precedence considerations or execution model for the tasks were used. The current work overcomes these shortcomings of the graph-matching approach: directed graphs are used, precedence constraints are allowed, and an execution model is required.

Towsley published solution algorithms for *restricted* and *unrestricted* task allocation problems [73]. The restricted problem differed from the unrestricted problem in that it separately co-allocated all instances of any task. Both problems

attempted to find an allocation minimising the total communication and execution costs, but the solution techniques used a sequence of graph operations, all ignoring execution costs. The unrestricted case was allocated by "unravelling" loops and branches, giving a much larger equivalent restricted allocation problem. For many unrestricted task-graphs, this restricted equivalent grew exponentially in the number of tasks, making this an *NP*-complete problem. The restricted algorithm was found to have $O(|T||P|^3)$ complexity. The present author pointed out and corrected some errors in Towsley's algorithm [39].

Chu and Leung have developed and validated a heuristic algorithm for the more general problem of allocating a directed task-graph of variable multiplicity to a set of processors [15]. The task-graph represented the control-flow of execution, including split-joins, alternatives and loops. Response times calculated for each task considered queueing effects at the processors and in the communication network. The system for which tasks were being allocated had open arrivals and real-time limits on the response times for several of its threads (threads are sequences of tasks). Random initial allocations were improved by heuristically and iteratively: (1) moving tasks to less loaded processors, (2) replicating tasks (where allowed) and allocating the replicate to a lightly loaded processor, and (3) deleting replicated tasks that were not needed to meet response time needs. The best such locally optimal allocation from many random starting points was used. Allocating a 12 task system onto 6 processors required several hours execution time on a VAX 11/780.

Sinclair studied the use of branch-and-bound search on task-graphs with the same quadratic cost function Ma used [64]. Two heuristic lower-bound cost functions were discussed to restrict the amount of the search-space actually evaluated. A heuristic based on non-communicating tasks allowed the number of levels in the search-space to be reduced, provided these tasks were allocated last. Random task-graphs of from 3 to 10 tasks were allocated onto three processors. Based on the number of nodes in the search-spaces for each cost function, Sinclair recommended the more complex cost function even though he acknowledged not being able to compare all the relevant factors.

In [8], Bokhari extended Stone and Bokhari's earlier results [70] from pro-

viding polynomial-time solutions for the two processor case to the general case of any number of processors. To reach this result, strong conditions were added to the task-graph and processor-graph. First, tasks only communicated in pipelines (in the forward direction). Second, processors only sent messages to their successors in the processor pipeline (no bus was allowed). These conditions allowed Bokhari to add a rudimentary path-length constraint to Doty *et al*, and Shen and Tsai's cost function (note that this thesis applies a similar path-length constraint to more general task-graphs). These conditions were most realistic for systems like the digital signal processing pipeline and host to satellite pipeline cases Bokhari described. Unfortunately, the user of this algorithm will have no way of bounding the reduction in system performance caused by not allowing jumping in the processor pipeline (e.g., executing a task on P_4 , then P_5 , then P_4 , then P_5) to better use the processors.

Montforton applied and extended the Doty *et al*, and Shen and Tsai cost function (minimising the workload of the most heavily loaded processor) differently than Bokhari did [49]. Bokhari's restricted path-length constraint was not used; rather, routing of communications for non-adjacent processors was allowed and linear processor overhead for communication was represented². Montforton allocated general undirected task-graphs (even multi-class or multi-threaded) to general undirected processor-graphs using a heuristic bin-packing algorithm. The results of his algorithm were compared with exhaustive enumeration for a variety of problems and it was found that the heuristic produced nearly optimal allocations and was quick enough for allocation problems having 120 tasks and 30 processors.

Summarising this historical review, much ingenuity has been expended in trying to make artificial criterion functions work in realistic task allocation problems. The essence of the new contribution of this thesis is the development of ways to incorporate performance measures into the search methods while retaining efficiency, and particularly to exploit special partial performance measures at intermediate steps to the solution. The next three sections discuss STAP system models, cost functions and solution techniques, respectively, in more detail.

²The non-adjacent communication allocation problem appears to have first been considered, heuristically, by Bokhari in [6].

3.2 System Models in the Literature

Basic to any STAP work are the assumptions it makes and the models it uses for the application and for its execution on the hardware. We find that realism in the models must be traded off against the tractability of the solution techniques: the more real the models, the less tractable the solution. This section details the concepts underlying the system models used for the STAP work described in the literature by looking first at the assumptions used by the hardware model and then looking at the issues surrounding the application models that drive the hardware. After reviewing the cost functions typically associated with these system models in Section 3.3, Section 3.4 shows how closely the solution methods for STAPs link to the system model and to the system cost function.

3.2.1 System Hardware Models

STAP hardware models describe the system hardware capabilities. That is, the use of the hardware resources, such as the processors (including each processor's memory and scheduling discipline) and communication links, and any restrictions applying to the use of or capabilities of these resources. Typical restrictions include fully connected communication paths (possibly implied by the use of a single link or bus) and identical processors and links. These restrictions simplify both the system model and the solution techniques, at the expense of realism and applicability. This subsection explores and compares the set of hardware models used in the STAP literature, based on the references in Section 3.1.

The following are assumed in most of the STAP literature:

- loosely coupled processors with local memory, where execution of P_j only interferes with the execution of another processor via P_j 's communication link usage, and
- zero delay for communication when the source and destination tasks are co-allocated³.

³Note that this zero delay local messaging assumption is not accurate for some systems, wherein local messaging takes a noticeable time also. In these cases, the local communications delay can be considered to be part of the task execution time, and only the incremental delays would be

Two additional considerations concern the hardware homogeneity and interconnectivity:

1. Processor homogeneity: The simplest situation is identical processors [34]; next is workload conserving processors that execute their processing workloads at known but different speeds⁴ [30]; then there are heterogeneous processors [69]. In the last case, task processing times must be specified separately for every task - processor pair. The first two cases conserve the total processor workload across any allocation of tasks.
2. Communication homogeneity: involves characteristics of each communication link and the processor interconnection topology and connectivity. Communication link topology usually fits one of three situations. The simplest situation is the full interconnection of the processors via a single bus [34]. The next situation is the full interconnection of the processors via any number of identical, bi-directional links. Finally, there is a heterogeneous case, where a mixture of uni-directional and bi-directional links partially interconnect the processors. Expressiveness in the more complicated situations comes at the price of extra notation. The notation developed in Chapter 2 allows partial connectivity and uni-directional links, but requires workload conservation, whereby communication time relates linearly to communication volume (a universal assumption in the STAP literature).

For remote communications, which usually involves both processors and a communication link executing simultaneously, both the literature and the present work

attributed to the communication link. Incorporating local messaging delays into task execution, while preserving the total system workload, does not accurately represent the link utilisation and requires the assignment of one communication workload independent of the processors communicating. At the expense of more cumbersome notation, these co-allocation and heterogeneous resources problems could be avoided in the present work by specifying communication workloads as a function of both the communicating tasks and processors, rather than just the communicating tasks now used.

⁴using the notation of Chapter 2, known execution speeds means that each execution of T_i on P_j takes $\frac{b_i^P}{c_j^P}$ seconds, with differences in c_j^P reflecting the instruction execution rates of each P_j . The present work assumes known execution speeds, but does not require them except for notational simplicity and for some heuristics in Chapter 5.

ignore any execution overlap, by assuming sequential execution of the component workloads, if an execution model is used at all.

Various other constraints are sometimes included, such as the availability of specialised peripherals [31], memory size limits [48], forwarding of messages [49], and real-time limits [30].

3.2.2 System Software Models

Software models capture the relationships between an application's tasks. First we examine the representation techniques for the software models in the literature and then we look at what is required to have a comprehensive execution model for an application's behaviour.

Virtually all the STAP literature uses a graph to represent the system software. Usually, additional constraints, such as memory requirements or real-time response time needs are expressed via additional relationships. A few authors, such as Stone [69] extend the software graph model to include the processors as nodes and the processing workloads as arc weights. For consistency and because such a graph cannot represent many constraints (e.g., it assumes that the processors are fully interconnected via homogeneous links), the present work considers Stone's graph to be a solution mechanism and not a software model. Graphs having both processors and tasks become overly complex for more than two processors.

Software Structural and Execution Models

Some STAP work considers (Bokhari [8], Ma *et al* [48, 47], Sips [66], and Chu and Leung [15]), as the present work does, the order in which task processing and communication workloads will be executed (i.e., their precedence relationships) while other work considers these workloads only in aggregate, ignoring any precedence or other execution constraints. We call models containing the precedence relationships "execution models," while those models relying only on aggregate workloads are called "structural models." Execution models describe the flow of execution control between tasks during the execution of a response of the application⁵.

⁵analogous to these software execution models, hardware execution models detail any overlaps

Most allocation techniques have either no execution model (i.e., they assume that all models having the same total processor and link workloads are equivalent) or have an unstated model (e.g., Stone's technique [70], which is easily interpreted as being intended for multiprocessing applications, but which was originally intended for a sequential application spread across several processors⁶). Execution models may be either open or closed, in the sense of having requests for the system arriving from outside (open) rather than having a fixed number of requests in the system (closed). Multiple simultaneous invocations of the same task may not be allowed. Many models serialise task invocations by assuming a serial execution model. Another factor affecting the time to complete a response is whether the workloads are stochastic or deterministic⁷. This work considers closed systems having stochastic workloads in which tasks are re-entrant, allowing their overlapped execution by any number of tokens.

In addition to this execution model, constraints of various types are often included in the software model. For example, task co-allocation or exclusion constraints can be used to force or inhibit the allocation of any two tasks to the same processor (this allows quite complex details of the software to be represented, such as the need to share memory).

Most of the present work, as indeed most of the STAP literature, considers only the allocation of a single application, composed of a single class of tasks. Exceptions to this limitation include Montforton [49], which is not complicated by the existence of multiple classes since the workloads of tasks in all classes are aggregated and no precedence-based performance measures are developed. Ma *et al* [48, 47] also allows multiple classes and evaluates deterministic port-to-port times for allocations. Section 5.5 shows how the present work extends to cover multiple classes of tasks, all competing for execution and communication.

in execution (e.g., the sender and receiver processors for a message may need to execute part of their workload for the message simultaneously).

⁶Stone's technique follows from those of Lowe [46], and Gylys and Edwards [30], both of which assumed that at most one invocation of the software system was underway at any point in time.

⁷deterministic workloads allow scheduling or bin-packing techniques to be applied.

Based on their complexity and the degree to which these system software models embody execution and structural models, the following four categories of STAP software models are apparent (listed with the pertinent examples from the literature, and explained below):

1. Structural Model: Gylys and Edwards [30], Stone [69], Sinclair [64], Jenny [34], Haessig and Jenny [31], Price [56], and Price and Krishnaprasad [57],
2. Heuristic Structural Model⁸: Jenny [34], Haessig and Jenny [31], Lo [45],
3. Undirected Execution Model: Doty *et al* [19], Shen and Tsai [63], and Montforton [49], and
4. Directed Execution Model: Ma *et al* [48, 47], Chu and Leung [15], Bokhari [8]⁹ and the present work.

Structural models consider a global cost only, as if the application were being executed serially. Heuristic structural models also make use of an overall sum, but the weights on the graph are now heuristically adjusted to enhance the parallelism or other goodness factors of the resulting allocation. Undirected execution models use the notion of a response on a lightly loaded parallel system, taking the cost of an allocation to be the workload of the most heavily used resource. Directed execution models have additional information on the execution of the application (such as precedence constraints between the tasks) that can allow more accurate performance-directed allocations to be determined.

The hardware and software models of this section define the STAP, but provide no way to compare allocations satisfying the STAP requirements. The next section describes the techniques the literature proposes for comparing allocations.

⁸note that the repetition of references from the last category reflects these papers' suggestion of both approaches.

⁹recall that Ma *et al* had response or port-to-port time limit constraints and deterministic workloads, Chu and Leung used non-linear cost functions to penalise lengthy allocations and Bokhari had a rudimentary path-length constraint.

3.3 Cost Functions

Cost functions are used to compare different allocations. Allocations having a lower cost are said to be better or more efficient than those having a higher cost. There are two types of cost functions: those which closely represent the real system but (commonly) are hard to solve, and those which may be further from the real system but are easier to solve.

Since we assume that the STAP specifies a fixed hardware configuration, an allocation's costs are associated only with using the hardware. These costs can be one or any combination of: time, monetary, or heuristic. Time-based costs are the most realistic, since static task allocation is normally intended to minimise the application's response time or maximise its throughput rate, both time-based. Monetary-based costs are not realistic, since monetary costs for STAPs are normally fixed items such as leased lines, and the time cost for using these items depends on their utilisation. Heuristic-based costs include those such as the minimisation of the total volume of inter-processor communication or the sum of processor execution times. These heuristic costs relate to time costs only under restrictive hardware capability assumptions such as homogeneous processors and software assumptions such as serial execution. For example, Stone [69] minimised the sum of processor and link execution times, a measure related to the application's execution time only if all communication links have the same speed and the application executes serially (only one link or processor executing at any time).

There is a close coupling between STAP cost functions, allocation constraints (part of the system model) and solution methods. While constraints form part of the system model and have been discussed in Section 3.2, this section discusses the cost functions separately, allowing for clearer comparisons between them.

Cost functions are often non-linear and may be combined with a system model having non-linear constraints (e.g., Ma uses a quadratic cost function¹⁰). For use in an allocator, these cost functions are associated with optimisation problems.

¹⁰the quadratic terms in Ma's cost function represent the communication workloads, each of which depends on the allocation of both of its communicating tasks: the product of two allocation variables.

Unfortunately, such optimisation problems are notoriously difficult to solve, and to scale up for large STAPs, leading to the lack of success in applying realistic (performance-based) cost functions to STAPs. This thesis emphasises using cost functions based on performance and on linear constraints to speed the solution process.

Although they have not yet been widely applied to task allocation due to their complexity, a range of queueing-theoretic costs have been proposed. For example, Baccelli and Liu developed steady-state and transient performance measures for acyclic split-join task-graphs [3]. Chu and Leung have used a simpler approximate response time cost function for task-graphs that allow cycles (loops), split-joins and conditional execution [15].

Typical cost functions in the STAP literature include the:

- sum of all communication times [31, 34], which assumes serial execution and homogeneous processors,
- sum of all communication times and computation times (exact for a sequential model of execution, which is highly unrealistic, since only one processor is in use at any one time: Stone and Bokhari justify it though [69]),
- artificial interference costs that differ from execution or communication time costs when tasks are co-allocated or allocated separately (this approach was suggested by Jenny [34] and used by Lo [45]),
- maximum processor or link load. This cost function balances the resource utilisations by minimising the load on the maximally loaded resource (e.g., [63, 14]),
- response time (this cost function is difficult to compute and requires more detailed software and hardware models than the above cost functions. Its use in (a modified form of) static task allocation has only been reported in Chu and Leung [15]), and
- reciprocal or negative of the system throughput¹¹.

¹¹Since throughput is the benefit derived from the system, its negative or reciprocal can be used as the system's cost.

These cost functions fall into two categories: those close to the real system (i.e., related to its performance) and those simple to solve (but not necessarily performance-related). The first four cost functions above are simple to solve, but normally are not close to the real system (e.g., task precedence requirements are ignored). The present work emphasises using a cost function – the reciprocal of the system's throughput capacity, a measure defined by Chapter 5 – as an approximation to the system's throughput. Throughput capacity is, in many cases, both close to the real system and easy to solve.

Often the cost function incorporates additional, or “under-the-table,” sets of constraints (e.g., Stone's infinite processing and communication costs [69]). The present models separate out the constraint sets from the cost functions to leave the meaning of an allocation's cost unambiguous. Other models sometimes use several cost functions (although there is little or no guidance about how to weight their relative importance). For example, Ma's cost function supposedly minimises the sum of processing and communication workloads and balances the processor utilisation.

Most cost functions are partly heuristic, especially those for use in systems whose execution models are not considered by the cost function. Only performance-related cost functions for deterministic systems or certain stochastic systems correspond closely to the real system (e.g., the maximum throughput capacity we introduce in Chapter 5) and thus can be used to find optimal allocations.

3.4 Solution Methods

After having chosen a system model and a compatible cost function, a solution method is needed to choose an appropriate task allocation. In many papers, the allocations resulting from using several candidate solution methods are compared, showing the tradeoffs in both allocation quality and solution time between them. However, solution methods depend on the system model and cost function chosen. Some solution methods require only limited detail about the system model, while others require a detailed model. Either way, a mismatch in detail can easily occur,

requiring either a compression of detail with the attendant possible loss of solution quality or an "extrapolation" of detail required to supply missing information. Thus, the selection of the solution method is closely tied to the characteristics of the system model and the cost function.

The solution methods minimise the cost function either exactly or heuristically, independently of whether the cost function is heuristic. Following Chu *et al* [13], the approaches for solving the STAP can be grouped into the following three categories or methods: graph-theoretic, mathematical programming and heuristic. The next three subsections detail these methods, in the above order. Various papers have considered using other methods, such as Chou and Abraham [12], who examined a STAP with real-time delays, constructed a Markov-reward model and solved it iteratively. The present work does not elaborate on these other methods since they seem to have had little impact.

3.4.1 Graph-Theoretic Methods

Graph-theoretic methods¹² are based on representing the STAP, both hardware and software, on a graph and using graph-based methods – such as the min-cut max-flow algorithm – to cut this graph into pieces, one piece allocated to each processor.

Usually, the system software graph is augmented by extra nodes representing the processors and extra arcs representing the processing workloads associated with executing the tasks on any processor. For the case of equal execution times on the processors, Jenny [34, 31] ignored the processor workloads and cut the software graph onto the processors to minimise the total communications workload (pre-allocation of tasks to separate processors were required to prevent degenerate allocations of all tasks to one processor). Stone and Bokhari considered tasks having different workloads on each processor. As shown in Subsection 3.2.2, this scheme can not model partly connected processors or the restriction of tasks to some of the processors.

¹²a term apparently first coined in [13] and denoting their basis in graph theory, which is typically their use of the Ford-Fulkerson max-cut min-flow theorem [25].

For more than three processors, the graph cutting problem is, in general, *NP*-complete [70]. Accordingly, various authors have looked at applying heuristics to this problem. Haessig and Jenny heuristically solved the min-cut, even allowing memory limits on each processor to be respected [31]. Even for the three processor case, Stone and Bokhari proposed a heuristic solution. Sinclair [64] used mathematical programming techniques to solve his allocation problems of min-cutting a Stone-like graph, reducing, but not eliminating the inherent complexity.

Note that since no execution model is incorporated into the graph, this method is implicitly intended for allocating serial applications onto a multiprocessor. However, both Jenny and Stone have suggested using this representation for parallel applications (Jenny suggested using negative communication to enhance parallelism [34]). Lo uses both heuristic solution techniques and a heuristic system model that tries to consider parallelism in the application [45].

The above examples of graph-theoretic methods illustrate the difficulties of: (1) complexity explosion with added processors, (2) restriction to uniform communications costs, (3) no general way to prevent co-allocation of tasks, and (4) communications and processing workloads are mixed together, ignoring performance.

Bokhari extended these algorithms for the special case of mapping pipelines of tasks to pipelines of processors. For this special case, a polynomial-time algorithm using doubly weighted arcs was presented in [8]. Rather than involving one cut per processor, this method was based on finding a minimum cost path between nodes in a special graph. It is not clear how practical this algorithm is to real problems or the degree to which enforcing pipeline usage of the processors artificially reduces system performance.

3.4.2 Mathematical Programming Methods

Mathematical programming is a standard set of solution techniques for problems that can be phrased as a cost function and a set of constraints, all based on a set of real or integer valued variables that represent the state of the allocation. Usually, the STAP is represented using a quadratic cost function and a set of linear constraints that can be stated as a 0-1 integer linear programming problem (ILP).

To solve such ILPs, a branch-and-bound search is normally applied because it allows easy tradeoffs between execution time and solution optimality¹³. Branch-and-bound is the solution method that will be described in Chapter 4, and that we use for all our STAPs. It allows performance estimates for incomplete allocations to be used to evaluate only those complete allocations close to being optimal in cost. As a further advantage over graph-theoretic methods, extra constraints, such as memory limitations or response time limits for deterministic systems, can easily be represented using branch-and-bound.

Examples of the two general ways mathematical programming has been used include,

1. Gylys and Edwards [30], Ma *et al* [48, 47] and Sinclair [64] who minimised Stone's cost function along with respecting several new constraints, and
2. Doty *et al* [19] and Shen and Tsai [63], who minimised the load of the most heavily utilised processor¹⁴.

3.4.3 Heuristic Methods

Heuristic methods for solving STAPs tend to be related to the above methods, but to provide solutions more quickly, at the expense of their optimality (optimality is often questionable anyway, since the accuracy of the model's quantities is usually low early in the design process). Heuristic searches are normally reserved for either the less realistic cost functions, since these make the solution questionable anyway, or for the larger problems, where an exact search is infeasible.

For large STAPs, both the graph-theoretic and mathematical programming methods require heuristic solution techniques (e.g., Haessig and Jenny [31] described a heuristic solution technique for problems defined using the graph-theoretic method,

¹³Other methods, functionally equivalent to branch-and-bound for this problem are cited in the literature (e.g., spatial dynamic programming [19], A* search [63] and branch-and-bound-with-underestimates [64]).

¹⁴minimisation of the maximal processor load is clearly not a quadratic or linear function, but it can be easily specified using branch-and-bound or via a set of quadratic or linear functions.

and the effort in mathematical programming could be limited in branch-and-bound, at the cost of solution quality). Heuristic methods differ from graph-theoretic and mathematical programming methods in the sense of not necessarily minimising their cost function, even for small problems.

Often, a clustering technique is followed by an iterative improvement process that transfers a single task at a time to improve the cost function (e.g., Price [56]). Extensions of this to transfer more than one task at a time are called simulated annealing. At first, the allocator models the system as having a high temperature and allows multiple transfers. After several such transfers, the temperature is reduced and only smaller transfers are allowed. Although expensive, this heuristic allows local minima to be avoided, probabilistically leading to a globally optimal solution. Gaudiot *et al* have applied this technique to allocations for the Hughes data-flow multiprocessor [26]. They only reallocated a single task at a time and used the probability of accepting an increase in total system workload as their temperature. Their results showed that simulated annealing produced marginally better allocations (with about a 1% improvement in cost) than single transfers with improvements only (in the two examples they reported).

Chu and Leung used multiple random allocations followed by local searches to try to achieve a global optimum [15]. Another example of heuristic solution is Montforton [49] who used an adapted bin-packing algorithm to rapidly minimize his Doty-like cost function.

3.5 Related Allocation Problems

This section considers two problems related to the STAP. Following a description of the file allocation problem, the dynamic task allocation problem is discussed. As the names suggest, file allocation distributes files onto disks and processors, and dynamic allocation allows tasks to be placed on different processors for different invocations.

A counterpart to static task allocation is file allocation, which involves statically allocating files to disks and processors in a distributed system. Eswaran [23]

has shown the unconstrained file allocation problem to be isomorphic to the traveling salesperson and other *NP*-complete problems; thus optimal solutions quickly become too expensive to obtain and heuristic search techniques or problem redefinition become necessary. Similar work for the STAP has shown it also to be, in general, *NP*-complete [7, 58]. Dowdy and Foster [20] have surveyed this field of file allocation, and we refer the reader there for further information.

Another problem related to the STAP is the dynamic task allocation problem, wherein tasks are allocated to processors as execution of the application proceeds, and the invocation of a task is requested by another task¹⁵. Stone and Bokhari [70] considered a static type of dynamic task allocation in which the application's execution is broken into phases, with the possibility of re-allocation, and its costs included on a "dynamic assignment graph;" applying a cutset algorithm to this graph then yields a static allocation for each phase and re-allocation of tasks only between phases. Dynamic allocation is most appropriate when accurate global knowledge about the system state is not available and thus allocation may need to proceed heuristically, based on local information. Even where reasonably accurate information is available, allocation decisions must be made quickly, balancing the potential extra system efficiency gained from better allocations with the extra run-time overhead they cause.

To reduce run-time overheads, multiple static allocations can be precalculated (either directly, or in groups). Two examples of these precalculated allocations are given by de Souza e Silva and Gerla [18], and Tantawi and Towsley [72], wherein task requests arrive at processors and are either processed locally or at a remote processor, depending on the expected utilisation of the processors and the speed of communication (and not on the actual state of any of the processors). Eager *et al* [21] found that only limited benefits over dynamic allocation at task invocation are gained by allowing task reallocation, or migration, during their execution. Philp recently suggested that this is inherent due to the problem of delayed information

¹⁵This definition is extended by Stankovic and Sidhu [68] who allow tasks to be moved (during execution) from processor to processor, depending on the distributed computer's state. For example, tasks may be moved from heavily loaded processors to lightly loaded processors.

[55]. This helps to explain why few distributed operating systems allow tasks to be moved from one processor to another while it is running (the V kernel is one that does, via a dynamic logical-to-physical host address mapping scheme [11]).

The dynamic task allocation problem covers the spectrum from the STAP to this task migration problem. In order of increasing dynamism, we have the following three intermediate dynamic problems:

1. task allocation is determined using state-independent probabilities, as described above [18, 72],
2. as in 1, but the allocation probabilities are updated periodically to reflect the recent system conditions (noted in [65, 56]) and
3. as in 2, but some updating is done for each task to be allocated [68].

Both problems 2 and 3 require the periodic exchange of state information between the system components. For many systems, the overheads associated with any of these dynamic problems are not justified, and STAP solutions are more appropriate.

3.6 Other Recent Work

This section reviews some recent work not described earlier in this Chapter. These references are grouped into two categories: task-graphs for extensions to the present task-graphs, and optimization problems, for allocation approaches related to the present STAP.

Task-Graphs

In [36], Kant introduces a class of generalized task-graphs having nodes and directed arcs. These task-graphs can represent many acyclic dependencies between tasks (no communication workloads exist between the tasks). Split-tasks and join-tasks are present, as are modified OR task-graphs (where the OR and $\overline{\text{OR}}$ points of the present nested task-graph notation are replaced by explicit tasks, an important difference should communications ever be added to Kant's task-graphs). An additional node type, the max (operator) task picks the first of several tokens arriving at

its incoming arcs and releases a token on the corresponding outgoing arc. Max tasks allow deadlines and timeouts to be represented (this concept could be extended for multiple replications of a task-graph). A task algebra for these task-graphs, similar in some ways to the nested task-graphs in the present work is developed and used to find completion time distributions for such task-graphs in the absence of any contention. Two deficiencies in this work are that the flow rate assignment analysis incorrectly makes max tasks flow conserving and that the algebraic representation of task-graphs doesn't model the selection property of max tasks.

Extending Chu and Leung's earlier work [15], Chu, Sit and Leung develop approximate analytical models for the response time of task-graphs in which additional resources, such as files, are shared between tasks [16]. Tasks contend for these additional resources, whose use is annotated onto Chu and Leung's task-graphs. Task precedence relations and resource contention appear in extended queueing networks which are analysed for approximate response times, validated via simulation. This technique could be extended for use as a cost function in the present STAP.

This resource contention problem has also recently been studied by Page, Jacob and Chern [53], who take the extreme example of arbitrary allocation of a resource usage graph (corresponding to a task-graph) to a processor-graph. The correctness of two new algorithms for management of contention for resources in this environment is established. The average performance of these algorithms is then compared with earlier methods, via simulation, showing reduced waiting times for access to the resources. This type of problem is analogous to the dynamic task allocation problem.

Optimization Problems

Lee, Srinivasan and Yano [44] look at optimally allocating work onto a closed queueing network in which processing nodes can be multi-processors. Standard product-form assumptions are made, and two solution algorithms, both of which quickly obtain globally optimal solutions for multiple uni-processors, are compared. This problem corresponds approximately to a STAP allocating a large task-graph having low communication needs. The processor-graph can have multi-processors

at some nodes, in which case, the authors conjecture pseudoconvexity and contend that their algorithms still provide optimal solutions.

Dallery and Stecke [17] look at a complementary problem, that of finding an optimal configuration for the processor-graph, given a closed queueing network model. As with [44], both single and multi-processor nodes are considered. Although the authors apply these results to flexible manufacturing systems, they may also be useful for selecting among multiple processor-graphs for the STAP.

Lavoie [43] focusses on task-to-processor assignment methods for digital signal processing problems to minimise input-to-output processing delays. Deterministic execution times and communication needs for tasks are modelled, along with the memory limits of processors. Communication delays between tasks on separate processors are considered (communication occurs via at most two shared memory transfers). The task-graph assumes identical speeds and memory capacities for each processor. A variety of list scheduling techniques are reviewed. Such techniques are intended for deterministic systems, where the delay performance of an allocation can be determined more easily than in STAPs having stochastic workloads, loops and multiple invocations (queueing). In comparison with Kasahara and Narita's work [38] on list scheduling and branch-and-bound search, [43] uses a new list scheduling heuristic, considering pipelines of tasks and uses a fixed task allocation order in the branch-and-bound search. Task-graphs containing loops are transformed into acyclic task-graphs, from which a critical path analysis determines a list schedule for task ordering. The branch-and-bound search using this task ordering is compared with heuristic approaches for several task-graphs ranging in size up to 25 tasks. Solution using this optimal branch-and-bound search takes up to two days on a SUN SPARC station 1, while heuristic solutions require less than a minute for the same problems.

It is interesting that [43] states that other allocation problems consider only a ' "one-time run" of a large computational task (divided into *modules*) on a set of interconnected identical processors.' As the references earlier in this chapter have shown, this is not accurate. Similarly, [43] states that only Fernandez and Ngo [24, 51] consider interprocessor communication times. This is clearly not so (e.g.,

[8, 73, 75]). This work also does not recognise any of the graph-theoretic work since Stone and Bohkari in 1978 [70] (e.g., [8, 45]), leading to an excessively negative assessment of this approach.

Murata, Bhatia and Shatz [50] recently looked at applying Markov chain reduction and analysis to the STAP. Tasks complete all their communication before they begin execution, a rather restrictive requirement. Task execution and communication times are stochastic, having exponential distributions (as in Lavoie [43], no contention is modelled for communication resources). Their cost function is the maximum of the mean completion times of the processors for a response (this is not the mean completion time of the system that the authors claim, and leads the authors to simplify their analysis by avoiding general cyclic task-graphs). No consideration is given to partial allocations, leading understandably to enumeration of a small example of two processors and four tasks. Further work with extended versions of this work may allow more general STAPs to be solved.

A recent report by Resnick [62] described the application of genetic algorithms - a type of heuristic search - to the STAP. Her work was not thoroughly tested, but looks promising. Further work, in which additional genetic algorithm techniques [27] are mapped to the STAP, may produce good allocations quickly.

In [42], Krueger, Lai and Radiya compare the job scheduling and processor allocation problems on hypercube computers. Job scheduling picks which jobs to run together, while processor allocation dedicated processors to a job, possibly migrating another job to make room. For a stream of jobs, the authors conclude that job scheduling via a new heuristic - scan scheduling - has a more important effect on system performance than processor allocation, for which they propose a simple strategy. The authors' job scheduling heuristic uses dynamic job allocation to subcubes of the hypercube, corresponding to dynamic task allocation and dynamic multi-class task allocation. Applying the authors' conclusion about job scheduling vs. processor allocation to the STAP, large multi-class STAPs may in some cases best be handled via analogous simple dynamic allocation of tasks or task-graphs to groups of processors.

In Chapter 9 of [29], a good overview of and literature survey on distributed

systems. Goscinski introduces a taxonomy of resource management systems. Emphasis is placed on dynamic load balancing, and when to migrate a task from one processor to another. Unfortunately, Goscinski is biased towards dynamic allocation. He groups all STAP techniques that make use of task-graph information together and states that they use "unrealistic or impractical assumptions."

3.7 Summary

Most previous allocation work has used artificial criterion functions to simplify the problem. This is justified in special cases where the artificial criterion function has some intuitive resemblance to the actual criterion function. Various graph-theoretic or mathematical programming solution algorithms and heuristics are then used to solve this artificial problem quickly. Stone and Bokhari [70] looked at the two or three processor case. Lo [45] and Price [57] extended this for many processors. Bokhari has found a polynomial-time algorithm to allocated pipelines of tasks to pipelines of processors [8]. These graph-theoretic heuristics are faster than the branch-and-bound solution method used by the current work, but they can't be adapted to consider the present allocation problem.

To properly apply branch-and-bound to our allocation problem, efficient bounding calculations must be found. Chapter 4 describes the branch-and-bound search algorithm we developed and used, and Chapter 5 describes a series of bounds that we developed for use with this search algorithm. Chapter 5 also discusses how our task execution model of Chapter 2 can improve the performance of our allocator.

The main difference between this work and the existing literature is that this work examines direct performance optimisation for the underlying application, as opposed to minimising an artificial criterion function. Performance is the main goal of task allocation, and hence, a more realistic system performance criterion gives more generally applicable solutions. For example, a system with a communications bottleneck will cause our optimal allocation to reduce (but likely not to minimise) communications. Observations on the kinds of allocations we generate and a limited comparison with those of the literature will be given in Chapter 6.

Chapter 4

Allocation Search Techniques

The task allocation problem is to choose – search for – a complete allocation with the lowest “cost” or within a certain range of that optimal cost (the cost of an allocation was defined in Section 2.5 and will be discussed in Chapter 5). This chapter states the discrete optimisation problem used for task allocation and discusses several techniques that speed the search for a suitable task allocation. Section 4.1 shows how the search arises naturally in task allocation problems and provides two ways of displaying the state-space used in the search, while Section 4.2 introduces our basic task allocation search method. After the size of the allocation problem is discussed in Section 4.3, the following techniques to reduce the difficulty of some allocation problems are discussed:

- hardware symmetry-based search-space reduction in Section 4.4,
- task co-allocation-based search-space reduction in Section 4.5, and
- task-graph reduction in Section 4.6.

The ideas behind our search techniques have been derived from standard discrete optimisation search methods (e.g., [41]). The use of hardware symmetry and task-graph reduction is novel and important for solving large problems.

4.1 The Task Allocation Search-Space

To find an optimal task allocation among all the possible task allocations,

a search in the space of task allocations is used. To structure the task allocations in this space (we call it a search-space) and to better illustrate the search method described below in Section 4.2, we introduce several hierarchical task allocation search-spaces. The general structure is a tree; after pointing out the duplicate allocations therein, we refine this search-tree into a hierarchy with no duplicates. By simplifying this hierarchy, we develop the search-tree that we will use later.

An Example Task Allocation Search-Tree

Figure 4.1 shows an example search-tree for the problem of allocating two tasks to two processors. Nodes, represented by circles, describe the allocation of one or both of the tasks to the processors. Each node represents an allocation; for example, the node marked d represents the allocation of task T_1 to processor P_2 . The node marked e has allocated T_2 to P_1 in addition to allocating T_1 to P_2 .

Figure 4.1: Example Task Allocation Search-Tree

The dots on the search-tree are not nodes, but rather symbolise the two stage decision that leads from the node immediately above the dot to the nodes immediately below the dot. This two stage decision first selects the task to allocate and then selects the processor to allocate the task to; the arrow entering the dot is

labeled with the task to be allocated and each arrow leaving the dot is labeled with the processor to which the task is allocated in the node at the leaving arrow's head.

The nodes at the lowest level of Figure 4.1 represent complete allocations and are called leaf-nodes, while all the other nodes are non-leaf-nodes wherein one or more tasks remain unallocated. The node at the top is the root-node or null allocation wherein no task is allocated. Nodes on the same level of the search-tree all have the same number of tasks allocated. All the possible allocations of any single task are on the first level of the tree below the root-node. Each further level in the tree binds another task to a processor. Hence, including the root-node level, there are $|T| + 1$ (here 3) levels of nodes in the tree.

Each leaf-node can be reached from the root-node via a unique path of arrows; the arrows traversed in reaching it are labeled with the task to processor allocations of the leaf-node. Because any unallocated task may be chosen for allocation by the arrows leading into the dots, several leaf-nodes may represent the same allocation¹. These duplicate allocations are labeled by the arcs on the leaf-nodes connecting the duplicated allocations. Connected leaf-nodes represent the same task to processor allocations, but with the two tasks allocated in the reverse order (e.g., the leaf-node marked *f* on Figure 4.1 represents the same allocation at that represented by the leaf-node marked *e*). With more tasks to allocate, several non-leaf-nodes would also represent the same partial allocations; all permutations of the order to the task to processor allocations exist as leaf-nodes in this search-tree, increasing both the size of the search-tree and the complexity of the search.

Recall that Section 1.2 limited all communication from P_j to $P_{j'}$ to using the same link. Allocating the tasks to the processors then determines both the communications and processing workloads. If more than one link could be allocated for communications from P_j to $P_{j'}$, then allocating each task would also involve deciding which links to allocated the communications to. These extra decisions would complicate the search-space and so we use link $l_{jj'}$ to carry all communication from P_j to $P_{j'}$.

¹for problems having more than two tasks, there will be duplicated non-leaf-nodes as well as duplicated leaf-nodes.

Simplified Task Allocation Search-Spaces

By eliminating the duplicate nodes in Figure 4.1, we obtain the simplified search-space shown Figure 4.2. Note that the search-space is no longer a tree, but is rather just a directed acyclic graph, and that more than one path can be used to reach some nodes.

Figure 4.2: Example Task Allocation Search-Space

A Simplified Task Allocation Search-Tree

If the "choice of task to allocate" decision is eliminated from the above simplified search-space by deciding the order of task allocation in advance (specifying which task each level in the search-space will allocate) or by making the choice a function of the node, a simpler search-space is obtained. For the example of Figures 4.1 and 4.2, we eliminate the dots and the arrows that are labeled with tasks and get the tree shown in Figure 4.3. For simplicity, we predetermine the task that each level of the search-tree allocates.

Figure 4.4 illustrates this simplified search-tree notation for the general case. As before, the root-node represents the null allocation. All the possible allocations of the first task to be allocated are represented by the nodes on the first level of the

Figure 4.3: Example Simplified Task Allocation Search-Tree

tree below the root-node. Each leaf-node represents a unique complete allocation and is reachable from the root-node by a single path of task allocations.

The search method we describe in the next section chooses a complete allocation by starting at the root-node and executing an algorithm that evaluates some non-leaf-nodes and some leaf-nodes as a quick alternative to evaluating all the leaf-nodes and selecting an optimal one.

4.2 The Branch-and-Bound Task Allocation Method

This section describes the branch-and-bound search method as it is applied to our task allocation problem. First we review branch-and-bound and the search-tree. The elements of the search method, as well as the steps followed during the search, are then discussed. Finally, in Subsection 4.2.1, we introduce the factors involved in choosing the tasks that each level of the search-tree will allocate (we call this the “order” of task allocation).

Branch-and-bound is a standard and general search method not requiring

Figure 4.4: The Simplified Task Allocation Search-Tree

any special search-space properties. To find an optimal allocation, we need only select the appropriate elements for the search method and provide a node evaluation technique able to find a lower-bound of the cost of the partial or complete allocation the node represents. Together, these allow branch-and-bound to evaluate only part of the search-space and yet still to find an optimal allocation. This section elaborates on the structure of each node in the search-tree and discusses the branch-and-bound task allocation search method and its elements.

Kohler and Steiglitz [41] thoroughly examined using branch-and-bound for a permutation problem (like task scheduling on one processor) where only the order in which the tasks were executed was important. For our task allocation problem, the allocation order is unimportant: only the allocation is. To create a branch-and-bound task allocation method we adapted Kohler and Steiglitz's branch-and-bound algorithm, and included some concepts from Ma *et al* [48]. As in [48], task processing and intertask communication needs are represented (via the notation of Chapter 2). Redundancy requirements can be met by using an appropriate T^{COAL} , and pre-determined task allocations are represented in T^{PREF} , both defined in Chapter 2.

Each node in the branch-and-bound search-tree corresponds to a partial or

complete allocation of the tasks T and has the following two attributes, a state and a lower-bound cost. The state defines the task to processor allocations represented by the node, and a lower-bound cost is an estimate of the lowest cost leaf-nodes in the search-tree that can be reached from the node.

We normally use the lower-bound of the reciprocal of the system's throughput bound as it's lower-bound cost. By finding a node with the lowest lower-bound cost, we have also found an allocation that maximises the system's throughput bound. Non-leaf-nodes whose lower-bound costs are undetermined (possibly due to not being able to evaluate the node's lower-bound cost) can be assigned a zero lower-bound cost (representing a possibly infinite throughput).

The way branch-and-bound is applied in this research may be described by the following eight elements of the search method [41], presented in the order that they are used in the search method:

- a selection rule to pick the next branching node from an active set of non-leaf-nodes (the active set will be described later in this section),
- a branching rule that, given any non-leaf-node, generates a set of nodes having one more task allocated,
- elimination rules using the dominance relation and the upper-bound cost to deactivate or eliminate newly generated nodes and active set nodes,
- a characteristic function eliminating newly generated nodes that violate any of the T^{PREF} , T^{COAL} or communication constraints of Section 2.5,
- a lower-bound function used to find the lower-bound cost of non-leaf-nodes and the cost of leaf-nodes. These functions are discussed in Chapter 5,
- an upper-bound cost for the "best" known complete allocation (determined using the lower-bound function),
- a bracket of from zero to one specifying the degree of optimality required in the allocation (if any). A bracket of zero specifies that the difference between the upper-bound cost and the lowest lower-bound cost must be zero while a bracket of one allows any complete allocation to stop the search, and
- a limit on the number of nodes expanded during the search for an appropriate allocation. If this limit is reached, the search stops, even though further searching may have found an appropriate allocation.

Kohler and Steiglitz used an additional element - a dominance relation - to compare non-leaf-nodes and eliminate those for which no node in its subtree could be optimal. Kohler and Steiglitz were able to find a useful dominance relation for their flow-shop problems; however, a useful dominance relationship for partial allocations in STAPs is not known to the author. Even for the simple communications minimisation lower-bound cost of Subsection 5.4.3, the dominance relation for partial allocations is recursively defined in terms of the complete allocations that they lead to. Our task allocation method uses no dominance relation because its complexity would approach that of the branch-and-bound search it is intended to speed up.

The branch-and-bound search method is initialised by setting the active set to be the root-node (representing the null allocation); the search is then a cycle consisting of the following six steps:

1. selecting a node from the active set, terminating unsuccessfully if no non-leaf-nodes exist therein,
2. generating the immediate descendant nodes from the selected non-leaf-node by allocating the task to be allocated to each of the processors (generating $|P|$ descendent nodes),
3. eliminating any infeasible nodes (ones that lead to no legal leaf-nodes because they violate T^{PREF} or T^{COAL} constraints or require communication that no link can provide),
4. evaluating the lower-bound cost of any nodes not eliminated,
5. inserting the remaining newly generated non-leaf-nodes into the active set, and
6. either terminating successfully if the optimality condition is met, or terminating unsuccessfully if the node expansion limit is exceeded or else going back to the selection step (1).

Together, we say that the second, third and fourth steps "expand" the node selected by the first step. The number of nodes expanded during a search, as well

as the time it takes to expand them will be used in Chapter 6 to describe the effectiveness of the various elements of the search method chosen.

Figure 4.5: Expected Cost-Bounds vs. Number of Nodes Expanded

Throughout the search, both the best leaf-node (i.e., the leaf-node with the lowest lower-bound cost; called the upper-bound node) and its cost (the upper-bound cost) are updated to check for the search termination condition. Figure 4.5 shows how the upper-bound cost may start at infinity because no leaf-node has been evaluated or supplied and then decrease occasionally as better leaf-nodes are evaluated. The best selection rules will be those that select the nodes having the lowest cost first. The lowest lower-bound cost of all nodes not yet expanded is then approximated via a smooth curve on Figure 4.5. After d nodes are expanded, the lowest lower-bound cost has risen to equal the upper-bound cost, and the search terminates with an optimal solution of cost $f(d)$. Depending on the bracket, the search may have terminated much earlier, possibly after d' expansions, with a nearly optimal solution of cost $f(d')$ (rather than the optimal $f(d)$). The exact shape of the curves on Figure 4.5 depend on the problem, but are expected to be similar to those shown. Subsection 6.1.5 will verify this general property of diminishing returns from further node expansions.

To avoid sorting or searching the active set of non-leaf-nodes for each node

expanded (when using the lowest lower-bound selection rules), we sort the active set and maintain its order when inserting new non-leaf-nodes into it.

Given a technique to evaluate complete allocations, a performance bound for any partial allocation can be found by recursively fixing each allocation of the first task to be allocated and applying the branch-and-bound technique to the sub-problems of determining an optimal allocation of the remaining tasks and selecting a leaf-node with the lowest cost. This method would evaluate each node in the subtree of the partial allocation, a slow process even for nodes near the leaf-nodes.

Depth-first or breadth-first search methods are quicker than this recursive method because they may not evaluate all the nodes, but they are still not quick enough for large problems (because neither the depth-first nor the breadth-first method uses the lower-bound costs of partial allocations to select the next node to expand). Still quicker lower-bound cost methods, producing looser bounds than the recursive method, but reducing the amount of the search-space evaluated, will be introduced in the next chapter and used in the examples of Chapter 6.

Figure 4.6: Lower-Bound Cost vs. Number of Tasks Allocated

Figure 4.6 compares the accuracy of the recursive lower-bound cost with the quicker lower-bound costs to be introduced in the next chapter. The recursive cost is the same for partial allocations as for optimal complete allocations. The quicker

lower-bound costs, on the other hand, provide cost bounds that are progressively more optimistic (lower) as fewer tasks are allocated. The next subsection discusses how to determine the order of task allocation for branch-and-bound so as to decrease the gap between the recursive lower-bound cost and the quicker lower-bound costs.

4.2.1 Determining the Order of Task Allocation

As was discussed above, predetermining the “order” of task allocation simplifies both the search-space and the branch-and-bound search technique. This subsection describes the selection of this task allocation order.

In our task allocation problem, the order in which tasks are allocated is irrelevant, only the task to processor allocations themselves are important. The actual task allocation order chosen (allocating the tasks to levels in the search-tree) can either be supplied as part of the problem definition or determined via a heuristic applied to it. The task allocation order is represented using a T^{ORDER} vector with $|T|$ elements. Any T^{ORDER} must meet the following two constraints:

$$T_i^{ORDER} \in T, \forall i \in T$$

$$T_i^{ORDER} \neq T_{i'}^{ORDER}, \forall i, i' \in T \mid i \neq i'$$

which require that T^{ORDER} be a vector whose elements form the set T .

The T^{ORDER} chosen for a problem should be one that allows the lower-bound cost to discriminate between poor and good allocations as early as possible in the search-tree², thereby reducing the gap in Figure 4.6. For example, in processor bound problems, allocating the tasks in order of decreasing assigned processing workload per response reduces the number of infeasible nodes expanded, since it allows the performance modelling and bounding tools of Chapter 5 to discriminate between allocations of similar performance earlier. The effectiveness of various T^{ORDER} generation heuristics in easing task allocation is examined in Subsection 6.1.4. These heuristics will include a greedy heuristic that selects the tasks with the largest processing requirements for allocation first. No rules exist, in general, for generating satisfactory T^{ORDER} vectors. To enable solutions to be found as quickly

²by early in the tree, we mean close to the root, or top of the tree

as possible, the generated T^{ORDER} should consider the effects of T^{COAL} and T^{PREF} restrictions by allocating tasks with tight restrictions earliest (for simplicity, none of our heuristics below consider T^{COAL} and T^{PREF} restrictions because they are not present in most of the examples we study in Chapter 6).

We provide the following four heuristics to allow the user to determine T^{ORDER} :

1. T_i is allocated before $T_{i'}$ if $i < i'$.
2. T_i is allocated after $T_{i'}$ if $i < i'$.
3. T_i is allocated after $T_{i'}$ if $b_i^P y_i < b_{i'}^P y_{i'}$, and
4. T_i is allocated after $T_{i'}$ if $\sum_{g \in T} b_{ig}^L y_i < \sum_{g \in T} b_{i'g}^L y_{i'}$.

Heuristic 1 (2) allocates tasks in order of increasing (decreasing) task numbers. Heuristics 3 and 4 allocate tasks in order of decreasing processing and communications workload per response, respectively. In addition to the above four heuristics, the user may explicitly set T^{ORDER} for his allocation problem: manually applying any number of other heuristics³ to guide the search.

Both eliminating multiple paths to a single state and generating a discriminatory T^{ORDER} increase the size of solvable task allocation problems. The allocation evaluation techniques of Chapter 5 all generate lower-bound costs and will be used as the branch-and-bound lower-bound cost functions in the examples of Chapter 6.

4.3 Search-Space Size

This section examines the effects of two techniques for constructing the allocation search-space on the search-space size (and presumably on the effort needed to find a solution). We also compare the effects of changing $|P|$ and $|T|$ on the search-space size.

³Other task allocation orders, such as allocating tasks on a “critical path” of the task-graph first, are expected to be effective for some allocation problems. The present work does not evaluate such orderings.

We now show how more restrictive search techniques reduce the search-space size. Allowing any unallocated task to be allocated at any search-tree node gives a search-space size of:

$$(|P||T|)(|P|(|T| - 1)) \cdots (|P|) = |P|^{|T|}|T|!$$

nodes, since any of the unallocated tasks at each level of the search-tree could be allocated to any processor. As in Figure 4.1, this size assumes that duplicate nodes can be evaluated multiple times (eliminating multiple evaluations while maintaining multiple approaches to any node complicates the search and won't be considered). Limiting the number of paths reaching each node to one reduces the search-space size to $|P|^{|T|}$, but adds the problem of deciding the order to allocate the tasks in the search-tree. The simplest way to choose the order of task allocation is to restrict each level in the search-tree to allocating the same task, as we have done above with T^{ORDER} .

Allowing any unallocated task to be allocated on any level of the search-tree increases the allocation search-space (and the effort needed to find a solution) by a factor of $|T|!$. With $|T| \geq 10$, this difference exceeds a factor of a million. Although this dramatic increase in the search-space size need not affect the number of nodes actually evaluated, they complicate the search and so we use a heuristically predetermined allocation order, as described above.

4.3.1 Comparing Search-Space Size Increases

We will now compare the increase in the search-space size caused by adding a processor with that caused by adding a task. We know that the search-tree of Figure 4.2 has $|P|^{|T|}$ possible allocations. Adding a processor or task changes this number to $(|P| + 1)^{|T|}$ or $|P|^{|T|+1}$, respectively. For $|P| \ll |T|$, adding a processor clearly increases the search-space size more than adding a task does.

Consider, for fixed $|P|$, the problem of finding the minimum $|T| \in \mathcal{N}$ for which adding a processor increases the search-space size more than adding a task, i.e., where:

$$(|P| + 1)^{|T|} > |P|^{|T|+1} \quad (4.1)$$

Taking the logarithms of both sides of (4.1) and rearranging the terms (assuming only that $|T| > 0$ and that $|P| > 1$) gives:

$$|T| > \frac{1}{\frac{\ln(|P|+1)}{\ln(|P|)} - 1} \quad (4.2)$$

For all $|T|$ satisfying (4.2), adding a processor increases the search-space size more than adding a task.

Since $|P|^{|T|}$ changes rapidly for changes in either $|P|$ or $|T|$, a simple but close approximation to this result can be found by assuming that $|P|$ and $|T|$ are continuous (which they aren't) and finding where the derivative of $|P|^{|T|}$ with respect to $|P|$ equals that of $|P|^{|T|}$ with respect to $|T|$. Where these derivatives are equal, adding a processor has about the same effect on the search-space size as adding a task. As the following equation shows, $d(|P|^{|T|})/d(|P|) > d(|P|^{|T|})/d(|T|)$, for $|T| > |P| \ln(|P|)$:

$$\frac{d(|P|^{|T|})}{d(|P|)} = |T||P|^{|T|-1} > \frac{d(|P|^{|T|})}{d(|T|)} = |P|^{|T|} \ln(|P|) = |P| \ln(|P|) |P|^{|T|-1}$$

This result is close to that of (4.2) and is easier to calculate. Examples in Section A.1 will support these general conclusions. These results can also be used in reverse to compare the reduction in allocation cost caused by eliminating one task (possibly via task clustering) with that caused by eliminating a processor (i.e., predetermining that no tasks will be allocated to the processor).

4.4 Hardware Symmetry-Based Search-Space Reduction

This section discusses how the search-space size can be substantially reduced, independently of the task-graph, by considering the symmetry that many processor-graphs have. This processor-graph, or hardware symmetry will be used to modify T^{PREF} , so only one of a set of symmetrical alternative allocations is allowed. This modification requires that P be partitioned into one or more processor equivalence groups (PEGs). Each PEG contains only “similarly” connected processors of the same speed, as explained below. Hardware symmetry-based search-space reduction

doesn't affect the cost of the solution obtained, or the existence of a solution: it only reduces the size of the search-space, and thereby makes it easier to find a solution or to be sure that there is no solution.

4.4.1 Processor Equivalence Groups

The processor equivalence group (PEG) is a general concept indicating that allocating a task to any PEG member (i.e., a processor in a PEG) is equivalent to allocating the task to any other member of that PEG, and thus only one of these equivalent allocations need be evaluated.

For a group of processors to constitute a PEG, they must all have the same speed (equal c_j^P values) and be able to communicate to the same processors the others in the PEG can, using the same link or a similar one. Two examples of PEGs are, a fully connected homogeneous processor group and a common bus architecture. Similar links have the same speed (i.e., equal c_l^L values) and the same workloads. The following algorithm describes how to find the PEGs for an application:

```

foreach  $P_j \in P$ 
  foreach existing  $PEG_k \in$  the set of PEGs
    try to add  $P_j$  to  $PEG_k$ , as described below
    if could add then insert  $P_j$  into  $PEG_k$ 
  endfor
  if couldn't add  $P_j$  to any existing PEG, then
    make a new PEG with  $P_j$  as its only member
  end

```

To add P_j to PEG_k , we must first check the following three constraints:

1. P_j has the same speed as any member of PEG_k ,
2. P_j communicates using the same link to the same processors as any member of PEG_k , and
3. each task that can be allocated to P_j can also be allocated to any member

of PEG_k and vice-versa⁴.

By induction, checking the processor speeds, link connections, and T^{PREF} for any PEG member is the same as checking it for all the PEG members. Because of the above conditions, it is clear that P_j could be added to at most one PEG.

As processors in a PEG have tasks allocated onto them, they are no longer equivalent⁵; rather, when a task is allocated to PEG member P_j , P_j leaves the PEG.

This reduction in PEG size as tasks are allocated is represented in the search by modifying T^{PREF} via one of two methods: the dynamic T^{PREF} modification (DM) method of Subsection 4.4.2 and the static T^{PREF} modification (SM) method of Subsection 4.4.3. DM avoids evaluating more than one symmetrical node by considering the symmetry of each node separately. SM, on the other hand, generates one T^{PREF} for an entire level of the search-tree. SM thereby allows quicker node characteristic function evaluations, but makes worst case symmetry reduction assumptions, so allowing some redundant nodes to be evaluated.

4.4.2 Dynamic T^{PREF} Modification

The dynamic T^{PREF} modification (DM) method for exploiting PEGs may have a different T^{PREF} for each node in the search-tree and thus may reduce the number of nodes evaluated at the cost of extra T^{PREF} modifications. The changes to T^{PREF} are easy to compute, and a node eliminated near the top of the search-tree can eliminate the redundant evaluation of hundreds of nodes below it.

Consider the example of Figure 4.7, showing the search-space required to allocate three tasks to three processors ($T^{ORDER} = [T_1, T_2, T_3]$ for this example). The nine nodes of Figure 4.7 contrast with the 28 nodes⁶ for the search-tree not using symmetry. Beside each allocation decision point in Figure 4.7 is the modified

⁴Note that this constraint simplifies the application of the PEGs found and guarantees that any of the PEG members could be substituted for another when the application is run

⁵We will not consider sets of processors having equivalent tasks allocated onto them as composing a PEG.

⁶ $(3 \times 3 \times 3) + 1$

T^{PREF} for that decision. The zeroes in T^{PREF} represent the redundant symmetrical allocations that have been eliminated.

Figure 4.7: Example DM Search-Tree

4.4.3 Static T^{PREF} Modification

The static T^{PREF} modification (SM) method for exploiting PEGs uses the same T^{PREF} for all the nodes in the tree and thus makes worst-case assumptions about the reductions in PEG sizes as nodes lower in the tree are evaluated. In particular, each row of the static T^{PREF} is the row with the fewest zeroes from the same level in the DM search-tree⁷.

The SM search-tree corresponding to that of Figure 4.7, has one extra node (i.e., 10 nodes) since the last row of T^{PREF} is $[1, 1, 1]$ for both T_3 decision points. Although both the DM and SM search-trees showed comparable improvements over the 28 node search-tree that results from not employing symmetry, DM normally provides a significantly smaller search-space than that of SM.

⁷ The method implemented does somewhat worse than this because it does not consider the effects of missing communication links or T^{TOTAL} constraints, whereas the DM method does.

When each processor has a different speed, or when no group of processors is "similarly" connected, no hardware symmetry exists and hence the DM approach would waste search time by adding to the characteristic function evaluation time, while never eliminating any nodes. Similarly, for two processors⁸, DM provides no advantages over SM (any symmetry that existed would be lost after the first allocation, for which DM is equivalent to SM anyway).

4.5 Task Co-Allocation-Based Search-Space Reduction

To speed up the allocation method for systems with many tasks, task clustering and task scattering can be used. Task clustering forces the co-allocation of several tasks while task scattering prohibits it. While both clustering and scattering are heuristics that may produce suboptimal allocations, they can also find acceptable allocations much quicker than a search not using them. First, we will examine task clustering.

Task Clustering

This technique forces the co-allocation (via T^{COAL} modifications) of tasks having such large inter-task communication requirements, that they will be allocated together anyway. By making this decision explicitly rather than as an implicit result of a much larger search, the search-tree is greatly pruned (Section 4.6 will show how to simplify the search for a clustered task-graph by transforming it into a reduced or smaller task-graph). Task clustering has been used for similar allocation problems (e.g., in [45, 57, 68]) and prunes the search-tree more for systems having more processors.

Traditional clustering methods would use common characteristics of all tasks (e.g., the amount of communications and the amount of work it involves) to position tasks in a multidimensional space and then find several groups of tasks that are

⁸For a single processor, clearly, neither DM nor SM can reduce the search-space size from one complete allocation

most alike, according to these characteristics. Unfortunately, these methods and hence much of the literature is unsuited for our needs, since we aren't interested in grouping similar tasks, but rather in grouping tasks that will likely be grouped together anyway by the branch-and-bound allocation method.

Stone type clustering is communication only based, ignoring the processing loads, which are often at least as important. Optimal min-cuts become very unnatural (due to the required preallocations) and hard to find where the desired number of clusters is high. Quicker algorithms, for example those which repeatedly break the network into two parts using the Ford-Fulkerson algorithm, still have problems in deciding how many clusters to make and in connecting minimum communication based clustering with the allocation method's clusters.

Proper task clustering reduces the time spent in the allocation method by reducing the search-space size while still allowing a good allocation to be found. The allocation method tries to minimise the cost of the application (discussed earlier) by co-allocating tightly communicating groups of tasks to processors while approximately balancing the processors' utilisations.

The amount of communication between a group of tasks that would force their co-allocation depends both on the task-graph and on the processor-graph. We will now describe three of our task clustering heuristics, the last of which uses a threshold on inter-task communication. The first heuristic has one parameter, p_1 , to tailor it to the family of problems it will cluster, while the second and third heuristics each have two parameters.

The first heuristic - `clust_sp` - clusters all the tasks having $w_i^P < p_1 \max_{i' \in T} w_{i'}^P$, where $w_i^P \equiv y_i b_i^P$ is the average processing workload of T_i per response. Clustering a group of tasks sets $T_{i'}^{COAL} = 1$ for each pair of tasks, T_i and $T_{i'}$, in the group. Note that no limit is placed on the processing workload (size) of the resulting clustered task: it contains all tasks smaller than p_1 times the largest task.

The second heuristic - `csp` - is similar to `clust_sp` except for the following two differences:

1. each clustered task, T_i , must satisfy: $w_i^P < p_1 \frac{\sum_{i' \in T} w_{i'}^P}{|T|}$; i.e., be no larger

than p_1 times the average task size, and

2. the clustered tasks that result must be no larger than p_2 times the average task size.

The third heuristic – chc – clusters those task pairs having the largest average per response communication requirements, as long as:

- the total size of the clustered task is less than p_2 times the average task size ($\frac{\sum_{i \in T} w_i^P}{|T|}$), and
- the communication requirements between the component tasks is at least p_1 times the average non-zero communication requirements (i.e., $w_{ii'}^L \geq \frac{\sum_{k, k' \in T} w_{kk'}^L}{\sum_{k, k' \in T | w_{kk'}^L > 0} 1}$, for each requirement that causes T_i and $T_{i'}$ to be clustered).

Task clustering based on communication is weak for a heterogeneous network because a threshold suiting the weakly connected processors would be too high for the tightly connected processors and many tasks could be allocated to either group.

After clustering the tasks, unless the task-graph will be reduced using the technique described in Section 4.6, we normally rearrange T^{ORDER} to make the best use of each allocation decision. Whenever a group of tasks is clustered, the tasks appear together in the rearranged T^{ORDER} , call it T^{ORDER^*} . The order of each pair of clustered tasks appearing in T^{ORDER^*} is the same as their order in T^{ORDER} ; however, any non-clustered tasks between each pair are moved, as described by the following algorithm:

```

 $T_i^{ORDER^*} = \emptyset \ \forall i \in T$ 
 $curpos = 0$ 
for  $i = 1$  to  $|T|$  do
    if  $T_i^{ORDER} \notin T^{ORDER^*}$  then do
        let  $T^Q$  be the tasks in  $T^S(i)$  in the order they are
            allocated using  $T^{ORDER}$ 
        for  $i' = 1$  to  $|T^Q|$  do
             $curpos = curpos + 1$ 
             $T_{curpos}^{ORDER^*} = T_{i'}^Q$ 

```

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

1980 COUNTY ROAD 3, BURNVILLE, MN 55337, USA
TEL: 612 436 7087 FAX: 612 436 7087 TUA: 516880408

```

        end
    end
end

```

Where $T^S(i)$ is the task set in T^S containing T_i (Section 4.6 describes how T^S is derived). To speed up the search method, we can also modify the branching rule of Section 4.2 to allocate all clustered tasks to the same processor whenever any one of the tasks is allocated.

Task Scattering

The task scattering search-space reduction technique forces tasks that will most likely be allocated onto separate processors (i.e., scattered) to not be co-allocated. By scattering a group of tasks, we mean setting $T_{ii'}^{COAL} = 0$ for each pair of tasks T_i and $T_{i'}$ in the group. Clearly we can never scatter more than $|P|$ tasks and often even scattering $|P|$ tasks leaves no complete allocations in the search-tree because of T^{PREF} constraints or $l_{jj'}$ communication constraints.

Two task scattering techniques are implemented. The first - `scat_bp` - scatters the biggest $\min(|T|, |P|)$ tasks, regardless of their sizes (where we define T_i to be bigger than $T_{i'}$ if $w_i^P > w_{i'}^P$). The second - `sbp` - accepts two parameters, p_1 and p_2 ; the biggest $\min(p_1, |T|, |P|)$ tasks are then scattered, as long as each has $w_i^P > p_2 \frac{\sum_{i' \in T} w_{i'}^P}{|T|}$.

4.6 Task-Graph Reduction

After a task allocation problem has been clustered, or just from user-defined co-allocation constraints, the solution via branch-and-bound can be speeded by transforming (reducing) the clustered problem to a smaller equivalent problem, and then transforming the resulting allocation back to the original problem⁹.

First, the tasks to cluster are extracted from T^{COAL} . Second, infeasibilities due to either T^{PREF} clashes, or to conflicting clustering and scattering constraints

⁹The techniques of this section focus on reducing GTGs. NTGs are first converted into GTGs as described in Subsection 2.3.2.

in T^{COAL} are uncovered. Then, the T^{ORDER} for the reduced problem is found. The parameters of the reduced model, in which each task represents a cluster of one or more of the original tasks, are then derived. The result is a complete allocation model for the reduced problem, together with a mapping back of solutions for this reduced problem to solutions of the original problem.

Task-graph reduction is based on the task co-allocations forced by T^{COAL} (these forced co-allocations arise both as a result of problem requirements and as a result of task clustering). We will describe how, based on T^{COAL} , we derive the new set of tasks (T'), and how the b^P , b^L , $\bar{b}^L$, T^{ORDER} , T^{PREF} and T^{COAL} values for T' are found from the old values (the new values will all have a prime superscript)¹⁰. Note that, for completeness, we will also describe how the new task visit ratios y' , to be defined and used in Chapter 5, are found from the old visit ratios y . The transformation is developed only for a single class, although it can be extended for multiple classes by considering each class independently and applying these reduction techniques to it.

Throughout this section, we will be following the reduction of the task-graph of Figure 4.8 into the task-graph of Figure 4.9. Tasks 1 and 8 from Figure 4.8 map into T_1 of Figure 4.9. Similarly, tasks 2, 4, 5 and 6 map to T_2 , and T_3 and T_7 map to T_3 . First, we describe how to find which tasks in T belong in each task in T' (this information is contained in T^S).

Finding T^S

From T^{COAL} , we derive T^S , a set of $|T'|$ task-sets denoted by $T_i^S, i \in T'$. All the tasks in task-set T_i^S are combined to form T_i' . T^S satisfies the following constraints:

$$\begin{aligned} \bigcup_{i \in T'} T_i^S &= T \\ T_i^S &\neq \emptyset \forall i \in T' \\ T_i^S \cap T_{i'}^S &= \emptyset \forall i, i' \in T', i \neq i' \end{aligned}$$

¹⁰Since the new clustered tasks have the old tasks as a substructure, we will not evaluate any π information for T' .

Figure 4.8: Example Task-Graph

Figure 4.9: Reduced Task-Graph for Figure 4.8

To select T^S , we first define a set of $|T|$ task-sets, T^R , initialised as follows:

$$T_i^R = \{i' \mid T_{ii'}^{COAL} = 1\} \forall i \in T$$

For Figure 4.8, where

$$T^{COAL} = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & 1 & -1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

T^R is then found to be:

$$T^R = \{\{8\}, \{5,6\}, \{7\}, \{5\}, \{2,4\}, \{2\}, \{3\}, \{1\}\}$$

T_i^R is part¹¹ or all of the set of tasks that T_i must be co-allocated with. We then define the following recursive function based on T^R to find all the tasks in T that must be co-allocated:

$$peruse(T^Q, i) = T^Q \cup \bigcup_{h \in T_i^R - (T^Q \cap T_i^R)} peruse(T^Q \cup T_i^R, h)$$

where T^Q is a set of tasks that must be co-allocated, and i is a task identifier. *peruse* works by adding to a known set of co-allocated tasks, T^Q , those extra tasks that must be co-allocated with T_i . Using the above *peruse* function, we now modify T^R as follows to ensure that T_i^R contains all the tasks that must be co-allocated with T_i :

$$T_i^R = peruse(\emptyset, i) \forall i \in T$$

Using this T_i^R definition, we find, for Figure 4.8:

$$T^R = \{\{1,8\}, \{2,4,5,6\}, \{3,7\}, \{2,4,5,6\}, \{2,4,5,6\}, \{2,4,5,6\}, \{3,7\}, \{1,8\}\}$$

The following procedure then creates T^S by eliminating all but the first copy of each task-set from the T^R that *peruse* created.

```

h = 0
for i = 1 to |T| do
     $T_{i'}^R = \emptyset \forall i' \in T_i^R, i' \neq i$ 
    if  $T_i^R \neq \emptyset$  then do
        h = h + 1
         $T_h^S = T_i^R$ 
    end
end
|T^S| = h

```

For Figure 4.8, the above procedure determines that $T^S = \{\{1,8\}, \{2,4,5,6\}, \{3,7\}\}$.

¹¹The reason that T_i^R may be only part of this set of tasks is that T^{COAL} may not have all the co-allocation dependencies expressed. For example, if $T_{12}^{COAL} = 1$ and $T_{23}^{COAL} = 1$, then T_1 , T_2 and T_3 must be co-allocated, but T_{13}^{COAL} may be -1 and so T_i^R , as initialised above, would not include T_3

Note that T^S may be infeasible as a result of several conditions, all reflecting infeasibility in the original problem. For example, T_1 and T_2 must be co-allocated, as must T_2 and T_3 , but T_1 and T_3 must be separated. This is a simple example of a cyclic dependence causing infeasibility. This condition is detected later as $T^{COAL'}$ is generated. Another infeasibility condition, not detected until the search, is where tasks must be separated, but where dependencies prevent their allocation onto communicating processors.

Finding $T^{PREF'}$

Before we find $T^{PREF'}$, we introduce both a matrix R , mapping from T to T' , and novel addition and multiplication operators ($\oplus$ and $\otimes$ respectively).

R has $|T|$ rows, has $|T'|$ columns and is initialised using T^S as follows:

$$R_{ii'} = \begin{cases} 1 & \forall i \mid T_i \in T_{i'}^S \\ 0 & \forall i \mid T_i \notin T_{i'}^S \end{cases}$$

For Figure 4.8.

$$R^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$\otimes$ is a non-associative multiplication operator that obeys the following rules:

$$0 \otimes u = Z$$

$$1 \otimes u = u$$

where u is any value and Z is a distinct nil (zero) indicator. Note that $v \otimes u$ is only defined for $v \in \{0, 1\}$.

$\oplus$ is an associative addition operator operating on symbols in the set $\{Z, 0, 1\}$, as defined by the following table:

$\oplus$	Z	0	1
Z	Z	0	1
0	0	0	0
1	1	0	1

Using $\oplus$ and $\otimes$ as defined above, we can now define matrix "multiplication," $\odot$, analogously to normal matrix multiplication, i.e.,

$$d \odot e = f$$

wherein:

$$f_{ij} = (d_{i1} \otimes e_{1j}) \oplus (d_{i2} \otimes e_{2j}) \oplus \dots$$

$T^{PREF'}$ can now be found as:

$$T^{PREF'} = R^T \odot T^{PREF}$$

where R^T is the transpose of R . $T^{PREF'}$ can not contain any Z entries, since that would imply that R has a row of all 0 entries (which is impossible, by its construction above).

For Figure 4.8, when

$$T^{PREF} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, \text{ then } T^{PREF'} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}.$$

Finding $T^{COAL'}$

Finding $T^{COAL'}$ is very much like finding $T^{PREF'}$, as has just been described. We make use of the same R matrix and the same $\otimes$ and $\odot$ operators. However, we must redefine the $\oplus$ operator to understand -1, the don't care indicator, and to trap the error conditions resulting from clustering tasks that must be separated. We let $\oplus$ be an associative addition operator operating on symbols in the set $\{Z, 0, -1, 1, E\}$, as defined by the following table:

$\oplus$	Z	0	-1	1	E
Z	Z	0	-1	1	E
0	0	0	0	E	E
-1	-1	0	-1	1	E
1	1	E	1	1	E
E	E	E	E	E	E

where E is an error value that results from $1 \oplus 0$ and $0 \oplus 1$ and is propagated through all further $\oplus$ operations.

Using this new $\oplus$ operator, $T^{COAL'}$ is found as:

$$T^{COAL'} = R^T \odot (R^T \odot T^{COAL})^T$$

As above, from the construction of R , $T^{COAL'}$ can not contain any Z entries. If any E entry appears in $T^{COAL'}$, then the reduction failed because the original problem was infeasible. Using T^{COAL} and R^T for Figure 4.8, we find that

$$T^{COAL'} = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}.$$

Unless an E appears in $T^{COAL'}$, we can search for a solution to the reduced problem. If the original problem has a feasible solution, then the reduced problem also has a feasible solution. Note that $T^{COAL'}$ contains 1 entries only on the diagonal, since all tasks that needed to be co-allocated were already clustered using T^S .

Finding $T^{ORDER'}$

To find $T^{ORDER'}$, two general methods can be used; (1), a heuristic can be applied to the new problem parameters and (2), $T^{ORDER'}$ can be derived from T^{ORDER} . Method (1) will not be discussed until Chapter 6, where various heuristic T^{ORDER} values are compared. Method (2) will however be examined further now¹².

¹²Method (2) is essential where T^{ORDER} was supplied by the user and not generated by a heuristic. Method (1) would suffice if T^{ORDER} were always generated heuristically.

In method (2), each T'_i will be allocated by $T^{ORDER'}$ as soon as the first of its T_i component tasks was allocated by T^{ORDER} . T_i^O is the index of which T^{ORDER} entry is T_i :

$$T_{T_i^O}^{ORDER} = T_i \quad \forall i \in T$$

Say $T^{ORDER} = [3, 5, 8, 2, 1, 4, 6, 7]$ for Figure 4.8, then $T^O = [5, 4, 1, 6, 2, 7, 8, 3]$.

Now let $T_i^{O'}$ be the $T_i \in T_i^S$ that T^{ORDER} allocates earliest:

$$T_i^{O'} = T_{\min(T_i^O \quad \forall i' \in T_i^S)}^{ORDER} \quad \forall i \in T'$$

For Figure 4.8, then $T^{O'} = [8, 5, 3]$.

Since $1 \leq T_i^{O'} \leq |T|$, we generate $T^{ORDER'}$, $1 \leq T_i^{ORDER'} \leq |T'| \quad \forall i \in T'$, via the following algorithm:

```

h = 0
for i = 1 to |T| do
  for i' = 1 to |T'| do
    if ( $T_i^{ORDER} = T_{i'}^{O'}$ ) then do13
      h = h + 1
       $T_{i'}^{ORDER'} = T_h'$ 
    end
  end
end
end

```

This algorithm ranks the $|T'|$ $T_i^{O'}$ entries in terms of which appears first in T^{ORDER} , placing the resulting order into $T^{ORDER'}$. For Figure 4.8, then $T^{ORDER'} = [3, 2, 1]$.

Finding y' , $b^{P'}$, $b^{L'}$ and $\bar{b}^{L'}$

Having found the sets of tasks to put in each new task (T^S) as above, y' , the new visit-ratios to the tasks in T' , can be found by summing the visit-ratios to the component tasks in T ,

$$y'_i = \sum_{i' \in T_i^S} y_{i'} \quad \forall i \in T'$$

¹³Since T_i is in only one of the T^S task sets, $T^{ORDER'}$ is set once for each $i' \in T'$.

The new reference-task is denoted $T'_{i'}$, where $T_{i'} \in T_{i'}^S$. Note that the visit ratio to this new reference-task is $y'_{i'}$, and can be greater than one (iff $|T_{i'}^S| > 1$)¹⁴. For Figure 4.9, where $y = [1, .6, .4, .3, .3, .6, .4, 1]$, $y' = [2, 1.8, .8]$. The π information in Figure 4.9 was derived from that in Figure 4.8 and can be used to solve for y' . Normally, since the π information for T' is not needed, it is not derived.

Using y' , $b_i^{P'}$, the new average processing workload per visit to $T'_{i'}$, is also found quite easily as the following weighted sum of the old b_i^P values:

$$b_i^{P'} = \frac{\sum_{i' \in T_{i'}^S} b_{i'}^P y_{i'}}{y'_{i'}} \quad \forall i \in T'$$

Similarly, for $b_{ii'}^{L'}$ and $\bar{b}_{ii'}^{L'}$, the new average per execution of $T'_{i'}$ and per communication from T'_i to $T'_{i'}$ communication workloads, respectively are:

$$b_{ii'}^{L'} = \frac{\sum_{h \in T_{i'}^S} \sum_{h' \in T_{i'}^S} b_{hh'}^L y_h}{y'_{i'}} \quad \forall i, i' \in T'$$

$$\bar{b}_{ii'}^{L'} = \frac{\sum_{h \in T_{i'}^S} \sum_{h' \in T_{i'}^S} \bar{b}_{hh'}^L y_h}{y'_{i'}} \quad \forall i, i' \in T'$$

where we condition the values for both $b_{ii'}^{L'}$ and $\bar{b}_{ii'}^{L'}$ by dividing the average total communication from T'_i to $T'_{i'}$ by $y'_{i'}$, the visit ratio of $T'_{i'}$.

Mapping the Solution for the Reduced Problem to the Original Problem

Any solution a' to the reduced problem can be mapped onto a solution a to the original problem using the following relation:

$$a = Ra'$$

For example, if $a' = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$ were an optimal allocation for Figure 4.9, then $Ra' =$

¹⁴To avoid re-normalising and de-normalising our visit ratios, we allow the reduced task-graph to have $y_{i'} > 1$. This relaxation simplifies the later analysis.

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ would be an optimal allocation for Figure 4.8.}$$

The cost of the reduced allocation is the same as the cost of the corresponding allocation in the original problem. In Chapter 6, task-graph reduction will be seen to speed the search for optimal allocations.

4.7 Summary

To avoid near-complete enumeration of the present task allocation search-space (whose size grows exponentially in the number of tasks), good lower-bound estimates must be made about the cost of the best complete allocation reachable from any node in the search-tree. These lower-bounds are used by the branch-and-bound task allocation method; to get good lower-bounds, we represent the search-space hierarchically, with complete allocations under partial allocations, and we provide bounds on the cost of any subtree within the search-tree.

Chapter 5 examines these bounds. Heuristic lower-bound costs will cause the search to prune subtrees of the search-tree (leaving them unevaluated). Is it better to incorrectly prune some worthwhile subtrees because of poor heuristic lower-bound costs than to prune no subtrees at all? As more subtrees are incorrectly pruned, the problem is "solved" quicker, although there is then no assurance of finding a nearly optimal allocation. To allocate sizable problems, however, parts of the search-tree must be pruned, either via such a heuristic cost function or via the search-space reduction techniques already described.

Chapter 6 will show that if the powerful bounds required to predict the performance of partial allocations are not used, task allocation quickly becomes im-

practical to solve as the search-space grows. The better these bounds are and the better the search-space reduction techniques are, the fewer nodes need to be evaluated. For small problems, simpler techniques will be shown to verify the powerful heuristic bounds, while larger problems overwhelm the simpler techniques, leaving only solutions found using the bounds.

The branch-and-bound implementation allows arbitrary cost and characteristic functions to be used, tailoring the search to the problem. System throughput is the branch-and-bound algorithm's usual cost function; modelling and bounding, as discussed in the next chapter, provide these performance estimates. Eventually, an expanded branch-and-bound search may automate the entire allocation problem solution process of Figure 1.3.

Chapter 5

Allocation Evaluation Techniques

The effect of having a search-space that grows exponentially in the number of tasks is reduced by using bounding techniques at each node in the solution process and pruning some subtrees (although the problem is still *NP*-complete, Chapter 6 will show that we can solve systems with large search-space sizes using reasonable effort). The branch-and-bound search uses allocation (node) evaluation techniques to generate lower-bound costs. This chapter discusses several of these techniques in sections 5.3 and 5.4, while Section 5.1 introduces performance modelling techniques for $\overline{ST}$ nested and general task-graphs (NTGs and GTGs, respectively), and Section 5.2 introduces the throughput capacity cost criterion as an alternative to throughput and uses it to develop several lower-bound costing techniques.

Not all levels of the allocation tree need to be modeled identically. Different performance bounds can be used at each level, with the quick inaccurate estimates being used near the root-node, where less information is available, while the slower more accurate estimates being used near or at the leaf-nodes, hopefully only a small fraction of which will need to be evaluated. We call these inconsistent lower-bound costing techniques.

Consistent lower-bound costing techniques (see Section 5.3) use the same method for all allocations, while the inconsistent methods examined in Section 5.4 use one method for partial allocations and approximate mean-value analysis (see Subsection 5.1.2) for complete allocations. Examples of the lower-bound costing

techniques that Chapter 6 will use include:

- set all non-leaf-node costs to zero, meaning that they need not be evaluated (see Subsection 5.4.1),
- evaluate partial allocations ignoring unallocated communications and processing workloads (see Subsection 5.3.1),
- evaluate partial allocations ignoring unallocated communications but equalising processor utilisation as much as possible with the unallocated tasks' processing workloads (see Subsection 5.4.2), and
- find throughput capacity bounds by solving a linear program (see subsections 5.3.4, 5.3.5, 5.4.2 and 5.4.3).

These allocation evaluation techniques are defined below in sections 5.3 and 5.4. None of these techniques can guarantee finding an allocation with the maximum throughput, but the consistent techniques find a solution quicker than the inconsistent techniques and the consistent throughput capacity techniques find an allocation with the maximum throughput capacity. The impact of multi-class STAPs on the allocation evaluation techniques of this chapter is described in Section 5.5.

5.1 Performance Models for Complete Allocations

This section discusses performance models for complete allocations, as measured by their steady-state throughput of responses per second. Subsection 5.1.1 shows how to approximately model any task allocation using closed product-form queueing networks. Closed product-form queueing networks always reach steady-state if they execute long enough. For the expected long-running applications, the startup and shutdown transient effects on an allocation's throughput can be neglected, and the steady-state throughput used.

Subsection 5.1.1 also discusses product-form and shows how its assumptions are violated for many allocation problems. Subsection 5.1.2 introduces a performance analysis technique for closed product-form queueing networks, and Subsection 5.1.3 then discusses the set of interconnection networks that can be modelled

using such networks.

5.1.1 Modelling Task-Graphs with Closed Product-Form Queueing Networks

This subsection shows how to model any complete allocation of a $\overline{SJ}$ NTG or GTG using closed product-form queueing networks that are derived from and represent the processor-graph. Each server in the queueing network is a processor or a communication link. Allocating the task-graph to the processor-graph then allocates the task workloads to the servers in the queueing network.

We first discuss how to find the closed queueing network corresponding to a task allocation problem. The workloads and other parameters associated with this network are then derived for any allocation and the assumptions needed to find a product-form solution (see Baskett *et al* [5]) are discussed. Subsection 5.1.2 describes a technique that analyzes these closed product-form queueing networks.

Finding the Closed Queueing Network

We will see how to find a closed product-form queueing network corresponding to a processor-graph¹. The nodes or servers in the network are the processors and links, and the directed arcs in the network show the possible transitions tokens can make from server to server. Both the workloads at the servers and the probabilities of the transitions from server to server depend on the allocation of the task-graph to the processor-graph.

Figure 5.1b shows such a queueing network, corresponding to the processor-graph in Figure 2.1b (repeated here as Figure 5.1a for convenience). The nodes on Figure 5.1b represent the processors and links, and the directed arcs show the possible paths of tokens representing users as they travel from one service request to another. Return arcs, such as from L_1 to P_1 , represent split-tasks and tasks that send several messages to another task (the last message, if any, carries the token

¹since this queueing network does not depend on the allocation being considered, it need be done only once per STAP

Figure 5.1: Example Processor-Graph (a) and Corresponding Queueing Network (b)

that causes the destination task to begin execution). Split-tasks send tokens to each of their branches; after communication to all but the last branch has finished, the token returns to the split-task (the original token is sent to the last branch). Since Figure 2.1a has no split-task, the return arcs on Figure 5.1b represent the splitting of communication between two tasks into a series of small messages. Tokens executing a task allocated to the other processor next (i.e., executing T_i now and having $T_{i'}$ as an immediate successor) must use the appropriate communication link first, sending $\bar{b}_{ii'}^L$ communication units. Tokens follow the self-loop on the processor nodes if they will execute either the same task next or will execute another task allocated to the same processor next.

Where several links can be used to communicate from one processor to another, we select one and model the link's server at that speed. Combining the link speeds gives an accurate link capacity, but an optimistic response time.

More formally, the queueing network representing the processor-graph is constructed in which:

- nodes labelled $P_j \forall j \in P$ are servers representing the processors,
- nodes labelled $L_l \forall l \in L$ are servers representing the links, and
- arcs representing possible communication paths are directed from node $P_{j'}$ to node L_l , from L_l to P_j , and from L_l to $P_{j'} \forall j, j' \in J(l)$.

Tokens from the processor-graph become tokens in the queueing network. The parameters for the queueing network defined above are derived below.

Finding the Queueing Network Parameters

In contrast to the finding of a queueing network described above, which is done once per STAP, the parameters for this queueing network depend on the allocation being considered, and hence must be done for each allocation, as described below.

The STAP modelled in Chapter 2 consists of three parts: the task-graph, the processor-graph and extra solution constraints. The product-form network analysis methods to be discussed below need only the visit-ratios of tokens to each node in the queueing network during a response and the service times at each node per visit; from these, they find the token throughput rate of the allocation. After showing how to find these workloads, we will discuss the assumptions needed to ensure the queueing network has the product-form that will be assumed.

To determine the node workloads, we first find the visit-ratios for each task in the task-graph. Aggregating the task visit-ratios for the workloads allocated to each node in the queueing network will then give us the node visit-ratios for the queueing network. A weighted average of the task and intertask workloads allocated to each node gives the average workload per visit and the average service time for each visit to a node.

The *visit-ratio* of task T_i (y_i) is the average number of times T_i is executed per response. As in Chapter 2, T_{i^*} is the *reference-task*, and is executed once per response. To find y_i , $\forall i \in T$, we solve the following system of $|T|+1$ linear equations with $|T|$ unknowns:

$$y_i - \sum_{i' \in T} \pi_{i',i} y_{i'} = 0, \forall i \in T$$

$$y_{i^*} = 1$$

The first $|T|$ equations are called the local flow balance equations because they balance the token inflow rates at each task T_i with the token outflow rate at that task (i.e., y_i). To simplify this system, we fix y_{i^*} as 1, removing it from the variables to be solved for and creating the following system of $|T|$ linear equations with $|T|-1$ unknowns (i.e., $y_i \forall i \neq i^*$):

$$y_i - \sum_{i' \neq i^*} \pi_{i'i} y_{i'} = \pi_{i^*i}, \quad \forall i \neq i^*$$

$$- \sum_{i'=i^*} \pi_{i'i} y_{i'} = \pi_{i^*i^*} - 1$$

If this system is consistent² and if it has a row rank of $|T| - 1$, then the visit-ratios for each task are uniquely determined. We consider only task-graphs where this system uniquely defines the visit-ratios.

The allocation matrix, a , determines both the visit-ratios to each server in the queueing network and the average service times there, as follows. The server visit-ratios, y_j^P for P_j and y_l^L for L_l , are given by:

$$y_j^P = \sum_{i \in A} y_i a_{ij}$$

$$y_l^L = \sum_{i \in A} \sum_{i' \in A} \sum_{j, j' \in J(l)} y_i \frac{b_{ii'}^L}{\bar{b}_{ii'}^L} a_{ij} a_{i'j'}$$

where $J(l) = \{(j, j') \mid l_{jj'} = l\}$ is the set of all processor identifier pairs using L_l to send messages from P_j to $P_{j'}$, and where A is the set of allocated tasks in a (those T_i with $\sum_{j \in P} a_{ij} = 1$). Both y_j^P and y_l^L depend only on the allocated tasks, A , and not at all on the unallocated tasks $\bar{A}$. While this subsection considers only the case of complete allocations (i.e., where $A = T$), later work will use the same notation we develop here for partial allocations, thereby ignoring the unallocated workload. Note also that in the calculation of y_l^L , $b_{ii'}^L / \bar{b}_{ii'}^L$ is the probability that the communication workload of $\bar{b}_{ii'}^L$ from T_i to $T_{i'}$ occurs each time T_i is executed³.

Before we find the average server workloads per response, we use the workload per task execution (b_i^P for T_i) and the task visit-ratios per response (y_i for T_i) to define the processing workload per response associated with T_i as:

$$w_i^P = y_i b_i^P$$

Similarly, the communications workload per response caused by communication from T_i to $T_{i'}$ is:

$$w_{ii'}^L = y_i b_{ii'}^L$$

²the system can be inconsistent if, for instance, $\pi = 0$.

³ $\pi_{ii'}$ is not used here because it is misleading for join-tasks, which appear in SJ GTGs (i.e., those GTGs in which $\sum_{i' \in T} \frac{b_{ii'}^L}{\bar{b}_{ii'}^L} \neq 1$, for some T_i).

Using these per response workload definitions, we will now find the processor and link workloads per response. Each processor and link was modeled in Chapter 2 as a server whose workload is determined by a , the allocation matrix. Server P_j has a workload of x_j^P seconds of processing per response, where:

$$x_j^P = \sum_{i \in A} a_{ij} \frac{w_i^P}{c_j^P}$$

Similarly, server L_l has a workload of x_l^L seconds of communication per response, where:

$$x_l^L = \sum_{j, j' \in J(l)} \sum_{i \in A} \sum_{i' \in A} a_{ij} a_{i'j'} \frac{w_{ii'}^L}{c_l^L}$$

Using this notation, we now find the average service times per token arrival at or visit to the servers – z_j^P for arrivals to P_j and z_l^L for arrivals to L_l – as the total workload per response divided by the server's visit-ratio:

$$z_j^P = x_j^P / y_j^P$$

$$z_l^L = x_l^L / y_l^L$$

As an example of the above process of finding the queueing network parameters, consider figures 2.1a and 2.1b, a processor-graph and a task-graph, respectively. The workloads as specified for Figure 2.1a in Chapter 2 are given in the following table:

i	1	2	3	i, i'	1,2	1,3	2,3	3,1
b_i^P	5	35	15	$b_{ii'}^L$	2	4	6	8
w_i^P	5	7	15	$\bar{b}_{ii'}^L$	10	5	6	8
				$w_{ii'}^L$	2	4	1.2	8

Table 5.1: Allocation Independent Parameters for Figure 2.1a

Using the above process for the allocation of Table 2.1, the following allocation dependent parameters are found:

Because only the product of z and y at each server is important in the analysis technique to follow, we simplify the above values of z and y so that the new z values

j	1	2	l	1	2
y_j^P	1.2	1	y_l^L	1	1
c_j^P	10	3	c_l^L	2	5
x_j^P	12	5	x_l^L	2.6	1.6
z_j^P	10	5	z_l^L	2.6	1.6

Table 5.2: Allocation Dependent Parameters for Figures 2.1a and 2.1b using the Allocation of Table 2.1

– $\tilde{z}$ – are constant⁴, and only the new y values – $\tilde{y}$ – vary as a function of the allocation⁵. This change speeds and simplifies the algorithms for lower-bound cost functions that are developed later in this chapter and is illustrated for the above example by the following table:

j	1	2	l	1	2
y_j^P	1.2	1	y_l^L	1	1
c_j^P	10	3	c_l^L	2	5
z_j^P	10	5	z_l^L	2.6	1.6
$\tilde{z}_j^P$	.1	$\frac{1}{3}$	$\tilde{z}_l^L$	.5	.2
$\tilde{y}_j^P$	120	15	$\tilde{y}_l^L$	5.2	8

Table 5.3: Modified Allocation Dependent Parameters for Table 5.2

With the above workload parameters, the throughput for an allocation can be found by solving the corresponding product-form queueing network. Reiterating, the complete allocation a has been reduced to the following queueing network parameters:

- the visit-ratios to each of the servers (y_j^P for P_j and y_l^L for L_l),
- the average service times per visit at each of these resources (z_j^P for P_j and z_l^L for L_l),

⁴for example, we let $\tilde{z}_j^P = \frac{1}{c_j^P}$ for P_j and $\tilde{z}_l^L = \frac{1}{c_l^L}$ for L_l .

⁵for example, we let $\tilde{y}_j^P = y_j^P \tilde{z}_j^P = y_j^P z_j^P c_j^P$ for P_j and $\tilde{y}_l^L = y_l^L \tilde{z}_l^L = y_l^L z_l^L c_l^L$ for L_l .

- the resource service disciplines (processor sharing for each processor, and first-come-first-served (FCFS) for the links), and
- the number of tokens executing the task-graph (N).

Different allocations change both the server visit ratios and the average service times. The number of tokens and the server disciplines are both fixed parts of the problem definition. The next subsection describes the assumptions necessary to give the above network a product-form solution, so that the solution technique of Subsection 5.1.2 can be applied.

Product-Form Considerations

To find the allocation's throughput from the product-form queueing network, we need no execution patterns for the servers; only the service times and visit-ratios to the servers parameters found above are required.

Product-form allows the fixed routing of communication specified in Section 2.1 to be generalised to be random, but dependent only on the source and destination processors. We restrict ourselves to fixed routing.

The execution of any GTG or NTG with no split-joins (a $\overline{SJ}$ GTG or $\overline{SJ}$ GTG, respectively) can be represented by a Markov chain (this section assumes that NTGs have been converted into GTGs via the process of Subsection 2.3.2, and that this GTG only is considered). By aggregating, or allocating, several workloads onto one server node in the queueing network, we hide state information that may affect both the transition probabilities and the average node workloads in the queueing network.

This hidden state information may violate product-form requirements since the real transition probabilities and workloads may then depend both on the task a token leaves from and the task the token goes to. Product-form requires that the node to node transition probabilities (the probabilities of taking the arrows on Figure 5.1b) be constant, and apply for all tokens of any class, regardless of where the tokens came from.

To avoid this violation of product-form, all tasks allocated to each processor must have the same transition probabilities π_{ij} to all the processors and the same

workloads (this condition is not likely for many allocations). Many task-graphs exist that can be allocated to processor-graphs and not violate these product-form requirements, but only a few of the allocations generated for evaluation by the search method in Chapter 4 would meet these requirements. To make the correspondence between the resource queueing network, and the task-graph and processor-graph as close as possible, the approximate analysis will make these assumptions, which may or may not be true:

- there is no synchronisation between tokens (i.e., task-graphs must not have any split-joins),
- the communication costs between tasks executing on the same processor are negligible,
- links serve their requests in the order of their arrival (i.e., they have FCFS service disciplines),
- the intertask communication requirements $\bar{b}_{ij}^L$ are the means of independent random variables with exponential distributions,
- processors serve all requests "simultaneously" (i.e., they have processor sharing service disciplines),
- the processing work requirements b_i^P are the means of independent random variables of general distribution, and
- processor and communication work requests are independent and uncorrelated with other requests.

Even the above assumptions are insufficient to make the queueing network product-form because of the communication links. Being FCFS, product-form requires that all tokens using a link have exponential service time distributions with identical means. Tokens using the link to transfer from T_i to T_j send b_{ij}^L communication units. Depending on the allocation, a link may connect many pairs of tasks; product-form for each FCFS link requires that they all have equal $\bar{b}_{ij}^L$ values. Adhering to this requirement would usually restrict all $\bar{b}_{ij}^L$ values to be the same: hardly realistic for most task allocation problems. Summarising, product-form is usually violated for each link because non-equal $\bar{b}_{ij}^L$ values for the tasks connected

by a link create hyper-exponential service demands for that link, and because hyper-exponential service demands are not product-form.

Under the assumptions above that the generated queueing network model corresponding to the allocation has product-form, any of the available product-form network analysis methods to be used. For example, Buzen's convolution method [10] finds the system performance metrics only after it computes a normalisation constant, and mean value analysis (MVA), as introduced by Reiser [61, 59], is an alternative way to find the same results. MVA uses simple formulae dealing only with mean values of queue lengths, queueing times and service times. This thesis uses an iterative approximate version of Reiser's MVA, as is described in the next subsection for some of our evaluation techniques.

5.1.2 Approximate Mean Value Analysis

This subsection introduces a tool used in this thesis to evaluate the performance of an allocation: approximate Mean Value Analysis (MVA). The last subsection showed how to find an approximately product-form queueing network representation of a task allocation; both processors and communication links were modeled using uni-servers.

Although MVA itself could've been used for the networks that the method of the last subsection generates, approximate MVA is more easily extended for approximate analysis of priority queueing networks and non-integer numbers of tokens (e.g., used to model critical-regions [33]).

As discussed above, since processor workloads are in general different for each task, allocating several tasks to the same processor creates a hyper-exponential service time demand that violates product-form requirements. The method this subsection describes assumes that all service demands are exponentially distributed. Product-form networks also allow no synchronisation between tokens, limiting this method to $\overline{SJ}$ task-graphs.

Even though the examples in this thesis are single class, the approximate MVA technique we present allows multiple task classes. This technique is similar to that of Bard [4], and consists of the following two steps:

1. estimate the initial values of the token distributions and service times at each resource for each class. The token distribution for each class is initialised to zeroes while the service time per visit at each resource is initialised to the z_j^P or z_j^L value of that resource for each class. The initial distribution chosen affects the rate of convergence but does not significantly affect the solution.
2. repeat the following actions until a convergence criterion is met:
 - find the total times spent at each resource for each class based on data from the previous iteration or the initial estimates,
 - find the new class throughputs using Little's result, and
 - find the new class token distributions using Little's result and the new class throughputs.

As discussed in Subsection 5.1.1, approximate MVA assumes that it is analyzing product-form queueing networks. We have approximated the allocation of the task-graph onto the processor-graph by a product-form queueing network whose performance is found using approximate MVA. The performance of the optimal allocation obtained using these approximations is expected to be close to the performance of any real optimal allocation.

The approximate MVA technique is quite flexible. Reiser has described a change to MVA that approximately analyzes hyper-exponential service demands [60] such as those created in the last subsection by co-allocating exponential workloads of different means. Heidelberger and Trivedi [32] have also considered approximations for the case of SJ task-graphs.

5.1.3 Interconnection Networks for Product-Form Queueing Models

This subsection examines several interprocessor communication link structures, and the corresponding hardware architectures that can be modelled using the above product-form queueing networks. The problem notation of Section 2.1 can

model a variety of interconnection networks, of which the following two communication link structures will be considered:

- a single common bus with no shared memory, and
- multiple dedicated point to point links.

The following three kinds of link interfaces (intelligent, DMA and plain) are considered for the above link structures:

- Intelligent links use dual-ported memories and have enough buffer space not to block or interfere with the processor.
- DMA links cause load dependent interference when they access the processor's memory, meaning, in general, that the processor components (i.e., the memory, the central processing unit and the processor bus) must be modelled separately, an area we don't consider.
- Plain links interrupt the processor more frequently than either DMA or intelligent links, and can be accommodated by viewing link operation as operational overhead (like the operating system overhead) that reduces the processor's speed (c_j^P) according to the level of link traffic, again an option we don't consider.

If the link speeds (the c_j^L values) are much smaller than the processor bus speeds, we can realistically approximate both DMA and plain links by intelligent links of the same speed (we make this approximation).

Multiple-server links (i.e., several links operating in parallel) approximately describe redundant interconnection structures, in which each link circuit would be dynamically allocated. Unfortunately, such servers violate product-form. We can approximate them optimistically by combining their individual link speeds into a faster single link with the same overall capacity, or for low utilisation links, we can use the highest single link speed.

5.2 Throughput Capacity Models

The throughput capacity ($\bar{\lambda}$) of an allocation is the largest throughput possible in the system, subject to that allocation. We have used zero-order throughput

bounds [22] to define $\bar{\lambda}$, because they are quickly found, even for complicated task-graphs and processor-graphs. Higher order throughput bounds, or throughputs are much harder to obtain and were found to be impractical to use in this allocation problem.

This section discusses finding bounds on an allocation's throughput capacity. Non-leaf-nodes in the search-tree of Chapter 4 represent partial allocations: only some of the tasks have been allocated, meaning that not all the eventual workloads are yet determined. We will examine several bounds on the performance of partial allocations⁶. These bounds also apply for complete allocations, determining an upper bound on their throughput (hence throughput capacity).

Using the processor and link workloads per response (x_j^P and x_l^L) derived by Subsection 5.1.1, $\bar{\lambda}$ is defined as the largest throughput satisfying the following three constraint sets:

- no processor exceeds 100% utilisation, giving:

$$x_j^P \bar{\lambda} \leq 1, \forall j \in P \quad (5.1)$$

- the corresponding link utilisation constraints on $\bar{\lambda}$ are:

$$x_l^L \bar{\lambda} \leq 1, \forall l \in L \quad (5.2)$$

- the mean path-length constraint on $\bar{\lambda}$ discussed in Subsection 5.2.2, but that is shown to only constrain $\bar{\lambda}$ for systems with small enough N .

Because tokens interact, the actual system throughput will be less than $\bar{\lambda}$, as constrained above. The next subsection develops the resource bottleneck bounds as a series of programming problems and Subsection 5.2.2 then discusses adding the mean path-length constraint to these programming problems. While both subsections consider only single class problems, the method can be extended to multi-class problems.

5.2.1 Bottleneck Bounds

This subsection develops bottleneck bounds (resource utilisation bounds) as a series of programming problems (first using quadratic constraints and then using

⁶An earlier version of some of these bounds was reported in [75].

linear constraints). We will first develop bottleneck bounds for complete allocations and then generalise these bounds to partial allocations.

A partial allocation cost bound is obtained from the fact that all processor and link utilisations are constrained to be at most one. When restated in terms of the reciprocal of system throughput ($\bar{\lambda}^{-1}$, from Section 4.2), these constraints conveniently form a linear convex constraint set⁷.

Rather than maximising $\bar{\lambda}$, subject to the above constraint sets (5.1) and (5.2), we consider the equivalent problem of minimising $\bar{\mu}$, where $\bar{\mu} \equiv \bar{\lambda}^{-1}$. This simplifies constraint sets (5.1) and (5.2) by replacing the $x_j^P \bar{\lambda}$ and $x_l^L \bar{\lambda}$ product terms with a simpler sum. For example, transforming constraint set (5.1) to use $\bar{\mu}$ gives the linear constraint set:

$$x_j^P - \bar{\mu} \leq 0, \forall j \in P \quad (5.1b)$$

Similarly transforming constraint set (5.2) gives the quadratic constraint set:

$$x_l^L - \bar{\mu} \leq 0, \forall l \in L \quad (5.2b)$$

Constraint sets (5.1b) and (5.2b) are linear and quadratic, respectively, in a because of the definitions of x_j^P and x_l^L given in Subsection 5.1.1.

The restrictions on task allocations because of T_{ij}^{PREF} and $T_{ii'}^{COAL}$ can also be expressed as the following linear and quadratic constraint sets:

$$a_{ij} = 0, \forall i \in A, j \in P \mid T_{ij}^{PREF} = 0 \quad (5.4)$$

$$a_{ij}a_{i'j} = 0, \forall i, i' \in A, j \in P \mid T_{ii'}^{COAL} = 0 \quad (5.5)$$

$$a_{ij} - a_{i'j} = 0, \forall i, i' \in A, j \in P \mid T_{ii'}^{COAL} = 1 \quad (5.6)$$

Since the processor graph is not in general fully connected, the allocation of communicating tasks is restricted to communicating processors by using the following quadratic constraint set:

$$a_{ij}a_{i'j'} = 0, \forall i, i' \in A, j, j' \in P \mid b_{ii'}^L \neq 0, j \neq j', l(j, j') = \emptyset \quad (5.7)$$

⁷quadratic terms arising from allocating communications are dropped with no loss in accuracy to simplify the problem. The linear terms then automatically form a convex constraint set that restricts $\bar{\lambda}^{-1}$ from below.

The allocation problem of this paper, maximising $\bar{\lambda}$, can now be stated as the following integer program with quadratic constraints:

$$F_{IP1} = \min_{\alpha, \bar{\mu}} \bar{\mu} \quad (IP1)$$

subject to: (using $A = T$)

constraint sets (5.1b), (5.2b), (5.4) - (5.7)

$$\sum_{j \in P} a_{ij} = 1, \forall i \in A \quad (5.8)$$

$$a_{ij} \in \{0, 1\}, \forall i \in A, j \in P \quad (5.9)$$

We restricted A to equal T so that constraint sets (5.8) and (5.9) could be reused later when $A \subseteq T$.

There are no standard solution methods other than branch-and-bound for problems like IP1, because of the quadratic constraint sets (5.5) and (5.7). If the problem is feasible, we find a solution by solving several simpler linear programming problems⁸. Infeasibility can only be caused by conflicts between T^{PREF} , T^{COAL} and the interconnection structure of the processor graph.

Problem IP1 implies the evaluation of the bottleneck bounds of all feasible complete allocations. As was discussed in Chapter 4, lower-bound costs for partial allocations don't require that all the complete allocations be evaluated to ensure that an optimal complete allocation has been found. To find such lower-bound costs for partial allocations, we will modify problem IP1 for use in the branch-and-bound search technique of Chapter 4. Before discussing how to modify IP1 for branch-and-bound, we define some branch-and-bound terms for our $\bar{\mu}$ cost criterion.

Each node in the branch-and-bound search-tree consists of a state k and a lower-bound cost (here we define $\tilde{\mu}$ as the lower-bound cost on $\bar{\mu}$). IP1 defines $\bar{\mu}$ only for complete allocations. We also define $\hat{\mu}(k)$ as the best $\bar{\mu}$ reachable from k , and $\tilde{\mu}(k)$ our corresponding lower-bound cost. For partial allocations, $\tilde{\mu}(k) \leq \hat{\mu}(k)$, while for complete allocations $\tilde{\mu}(k) = \hat{\mu}(k) = \bar{\mu}(k)$ is required. We will now find some bounds on the throughput capacity of partial allocations.

⁸each of which has feasible $\bar{\mu}$ values and is convex.

Bounds on Throughput Capacity

In order to use the branch-and-bound method of Chapter 4, we need some quick, but tight bounds on the throughput capacity of any state k . We will show how these can be obtained from problem IP1, which will be restated for an arbitrary state k to find its $\tilde{\mu}$ and $\hat{\mu}$ values. Several of its constraints will then be relaxed, giving $\tilde{\mu}$ values which can be found easily, but which still allow branch-and-bound to find an optimal solution to IP1. Since many effects of $\bar{A}$ on $\hat{\mu}$ are considered in $\tilde{\mu}$, $\tilde{\mu}$ is close to $\hat{\mu}$ and the branch-and-bound search is efficient, as section VI will demonstrate. These bounds are related to the zero-order bounds of [22], but are conditional on partial allocations.

Problem IP1 above uses $A = T$ (i.e., complete allocations). We define $\bar{a}$ as the analog of a , applied to set $\bar{A}$. To find $\tilde{\mu}$ for a legal state k (where constraint sets (5.4) - (5.9) are already satisfied), we rewrite IP1 to optimise over $\bar{a}$ and $\tilde{\mu}$. Notice that $\tilde{\mu}$ is simply an optimisation variable here: given as a function of $\bar{a}$ etc. only by the solution of the following problem IP2:

$$F_{IP2}(k) = \min_{\bar{a}, \tilde{\mu}} \tilde{\mu} \quad (IP2)$$

subject to:

$$\sum_{i \in \bar{A}} \bar{a}_{ij} \frac{w_i^P}{c_j^P} - \tilde{\mu} \leq -x_j^P, \quad \forall j \in P \quad (5.1c), \text{ from (5.1b)}$$

$$\begin{aligned} & \sum_{j, j' \in J(l)} \sum_{i \in \bar{A}} \sum_{i' \in \bar{A}} \bar{a}_{ij} \bar{a}_{i'j'} \frac{w_{ii'}^L}{c_l^L} + \sum_{j, j' \in J(l)} \sum_{i \in A} \sum_{i' \in \bar{A}} \left(\bar{a}_{ij} \bar{a}_{i'j'} \frac{w_{ii'}^L}{c_l^L} + \bar{a}_{i'j} a_{ij'} \frac{w_{i'i}^L}{c_l^L} \right) - \\ & \tilde{\mu} \leq -x_l^L, \quad \forall l \in L \quad (5.2c), \text{ from (5.2b)} \end{aligned}$$

$$\bar{a}_{ij} = 0, \quad \forall i \in \bar{A}, j \in P \mid T_{ij}^{PREF} = 0 \quad (5.4b), \text{ from (5.4)}$$

$$a_{ij} \bar{a}_{i'j} = 0, \quad \forall i, i' \in \bar{A}, j \in P \mid T_{ii'}^{COAL} = 0 \quad (5.5b), \text{ from (5.5)}$$

$$\bar{a}_{ij} a_{i'j} = 0, \quad \forall i \in \bar{A}, i' \in A, j \in P \mid T_{ii'}^{COAL} = 0 \quad (5.5c), \text{ from (5.5)}$$

$$\bar{a}_{ij} - a_{i'j} = 0, \quad \forall i, i' \in \bar{A}, j \in P \mid T_{ii'}^{COAL} = 1 \quad (5.6b), \text{ from (5.6)}$$

$$a_{ij} = a_{i'j}, \quad \forall i \in \bar{A}, i' \in A, j \in P \mid T_{ii'}^{COAL} = 1 \quad (5.6c), \text{ from (5.6)}$$

$$\begin{aligned} & a_{ij} \bar{a}_{i'j'} = 0, \quad \forall i, i' \in \bar{A}, j, j' \in P \\ & \mid b_{ii'}^L \neq 0, j \neq j', l(j, j') = \emptyset \quad (5.7b), \text{ from (5.7)} \end{aligned}$$

$$\bar{a}_{ij}a_{i'j'} = 0, \forall i \in \bar{A}, i' \in A, j, j' \in P$$

$$|b_{ii'}^L \neq 0, j \neq j', l(j, j') = \emptyset \quad (5.7c), \text{ from (5.7)}$$

$$\sum_{j \in P} \bar{a}_{ij} = 1, \forall i \in \bar{A} \quad (5.8b), \text{ from (5.8)}$$

$$\bar{a}_{ij} \in \{0, 1\}, \forall i \in \bar{A}, j \in P \quad (5.9b), \text{ from (5.9)}$$

Since all the IP1 constraints are re-expressed, $F_{IP2}(k) = \tilde{\mu} = \hat{\mu}$ for all legal feasible states k (i.e., where IP2 is feasible for k). The first sum in (5.2c) represents communications between unallocated tasks, while the second represents communications between allocated and unallocated tasks. Both sums vanish for complete allocations (leaf-nodes).

Problem IP2 is an integer program (in $\tilde{\mu}$ and the components of $\bar{a}$), with quadratic constraints. We will now simplify IP2 to give an ordinary (non integer) linear program LP1 by doing the following:

- relaxing the integrality constraint set (5.9b), replacing it with:

$$\bar{a}_{ij} \geq 0, \forall i \in \bar{A}, j \in P \quad (5.9c)$$

- deleting quadratic constraint sets (5.5b) and (5.7b), and
- ignoring the communications between unallocated tasks in (5.2c).

All these relaxations of IP2's constraints can only lower LP1's $\tilde{\mu}$ values. Since no constraints are relaxed for leaf-nodes, $\tilde{\mu} = \hat{\mu}$ for them, as required to allow branch-and-bound to find an optimal IP1 solution.

LP1 can be substantially simplified, at the expense of having even looser non-leaf-node bounds. We first simplify LP1, giving LP2, by:

- ignoring the communications between allocated and unallocated tasks in (5.2c), and
- deleting (5.5c), (5.6b), (5.6c) and (5.7c).

Let $X = [\bar{a}_{11}, \dots, \bar{a}_{|\bar{A}||P|}, \tilde{\mu}]^T$ be the LP variables, and $C = [0, \dots, 0, 1]$. We state LP2 as follows:

$$F_{LP2}(k) = \min CX \quad (LP2)$$

subject to:

$$\begin{aligned} & (5.1c), (5.8b), (5.9c) \\ & -\tilde{\mu} \leq -x_l^L, \forall l \in L \end{aligned} \quad (5.10)$$

As before, these relaxations can only lower $\tilde{\mu}$ for non-leaf-nodes, but at leaf-nodes, $\hat{\mu} = \tilde{\mu}$.

Problem LP2 has many solutions for states with more than one unallocated task. This can be seen later when another LP (LP4) with fewer variables will be defined, from which an LP2 solution can always be constructed. First, we add the "uniform allocation" constraint set:

$$\bar{a}_{ij} = \bar{a}_{i'j}, \forall i, i' \in \bar{A}, j \in P \quad (5.11)$$

to LP2, getting a more constrained problem (LP3):

$$F_{LP3}(k) = \min CX \quad (LP3)$$

subject to: (5.1c), (5.8b), (5.9c), (5.10), (5.11)

Proposition: LP2 and LP3 both give the same $\tilde{\mu}$ value.

Proof: It is easily verified that LP2 is always feasible, provided $w_i^P, w_{ii'}^L < \infty, \forall i, i' \in T, c_j^P > 0, \forall j \in P$, and $c_l^L > 0, \forall l \in L$, and hence has a solution, X_{LP2}^* . Define the vector X' in terms of X_{LP2}^* as follows, component by component:

$$\bar{a}'_{ij} = \frac{1}{w^P} \sum_{i' \in \bar{A}} \bar{a}_{i'j} w_{i'}^P, \forall i \in \bar{A}, j \in P$$

$$\tilde{\mu}' = \tilde{\mu}$$

where $w^P = \sum_{i \in \bar{A}} w_i^P$. We will show that X' is optimal to both LP2 and LP3, by showing that it is feasible to both and that it has the lowest feasible cost. Note the following two identities:

$$\begin{aligned} \sum_{i \in \bar{A}} \bar{a}'_{ij} \frac{w_i^P}{c_j^P} &= \sum_{i \in \bar{A}} \frac{1}{w^P} \sum_{i' \in \bar{A}} \bar{a}_{i'j} w_{i'}^P \frac{w_i^P}{c_j^P} = \frac{1}{w^P} \sum_{i \in \bar{A}} w_i^P \sum_{i' \in \bar{A}} \bar{a}_{i'j} \frac{w_{i'}^P}{c_j^P} = \sum_{i' \in \bar{A}} \bar{a}_{i'j} \frac{w_{i'}^P}{c_j^P} \\ \sum_{j \in P} \bar{a}'_{ij} &= \frac{1}{w^P} \sum_{j \in P} \sum_{i' \in \bar{A}} \bar{a}_{i'j} w_{i'}^P = \frac{1}{w^P} \sum_{i' \in \bar{A}} \sum_{j \in P} \bar{a}_{i'j} w_{i'}^P = \frac{1}{w^P} \sum_{i' \in \bar{A}} w_{i'}^P \sum_{j \in P} \bar{a}_{i'j} = \sum_{j \in P} \bar{a}_{i'j} \end{aligned}$$

By inspection, using the above identities and the feasibility of X_{LP2}^* , X' then satisfies the constraints for both LP2 and LP3. By construction, X' has the same cost as X_{LP2}^* and hence is also optimal to LP2. Define X_{LP3}^* as an LP3 solution. Since LP3 is more constrained than LP2, $CX_{LP3} \geq CX_{LP2}^*$. Now, X' is feasible to LP3 and also has the lowest possible cost ($CX' = CX_{LP2}^*$); hence, X' is optimal to LP3 also. This completes the proof.

Hence $\tilde{\mu}'$ can be used as a lower-bound on $\hat{\mu}$. This is intuitively reasonable since constraint sets (5.9c) and (5.8b) enforced the allocation of the unallocated workload w^P , while (5.11) merely changed the way it was allocated, dividing each unallocated task uniformly among the processors.

We will now show that LP2 can be reduced to a problem with fewer variables. Since (5.11) constrains many $\bar{a}_{ij}$ variables to be equal, we can simplify the expression and solution of LP3. Let $\alpha_j = \bar{a}_{ij}$, $\forall i \in \bar{A}, j \in P$ be the new uniform allocation variables. Let $\bar{X} = [\alpha_1, \dots, \alpha_{|P|}, \tilde{\mu}]^T$ represent these new variables, and $\bar{C} = [0, \dots, 0, 1]$ the new cost function. We can now re-express LP3 more simply as:

$$F_{LP4}(k) = \min \bar{C} \bar{X} \quad (\text{LP4})$$

subject to:

$$\alpha_j \frac{w^P}{c_j^P} - \tilde{\mu} \leq -x_j^P, \quad \forall j \in P \quad (\text{from (5.1c)})$$

$$-\tilde{\mu} \leq -x_l^L, \quad \forall l \in L$$

$$\sum_{j \in P} \alpha_j = 1 \quad (\text{from (5.8b)})$$

$$\alpha_j \geq 0, \quad \forall j \in P \quad (\text{from (5.9c)})$$

Note that X_{LP3}^* can be trivially found from $\bar{X}_{LP4}^*$, and $CX_{LP2}^* = \bar{C} \bar{X}_{LP4}^*$. LP4 uses $|P| + 1$ variables and $|P| + |L| + 1$ constraints, while LP2 uses $|\bar{A}||P| + 1$ variables and $|L| + |P| + |\bar{A}|$ constraints. For $|\bar{A}| > 1$, LP4 is smaller than LP2, but gives the same $\tilde{\mu}$.

This subsection has developed the processor and link utilisation bottleneck bounds as a sequence of programming problems; LP1 produces the tighter $\hat{\mu}$ bounds

and LP4 gives the smaller LP. The next subsection describes how to add mean path-length bounds to the programming problems developed in this subsection and when this can tighten the $\bar{\lambda}$ values found.

5.2.2 Mean Path-Length Bounds

This subsection describes how to derive several mean path-length bounds for nested task-graphs (NTGs). We limit ourselves to NTGs because general task-graphs (GTGs) are not as restrictive as NTGs, complicating their analysis.

First, the concept of the path-length of a NTG is introduced, and then we discuss the mean path-length, comparing it with the mean response time of the NTG. Further sections give examples of how to find a lower-bound on the mean path-length for all NTGs, either fully or partially allocated.

A *path* through a NTG is any sequence of arc traversals and possible task executions that takes a token from the point SOP to the point EOP (if the NTG shows a reference-task and re-execution loop, then any path "through" it starts with the execution of the reference-task and ends with the token returning to the reference-task). The path-length of NTGs is defined to be the time required by a single token, in an otherwise empty task-graph, starting at SOP, to reach EOP, via any path. For most task-graphs, the path-length is non-deterministic both since the token may use any path through the NTG, and since the tasks may have stochastic workloads.

Path-lengths are easier to calculate in $\overline{SJ}$ NTGs than in SJ NTGs since with only one token, there is no resource contention. For SJ NTGs there may be no way to avoid contention between the execution of the parallel branches and the resulting scheduling problems in calculating the path-length. $\overline{SJ}$ NTGs have no such scheduling problems since the single token never finds any contention.

[13] has examined the path-length distribution for similar task-graphs, exposing severe computational problems in manipulating the cumulative distribution functions for the processing and communications workloads. To avoid these computational problems, we will derive several quick lower-bounds on the mean path-lengths of NTGs.

Next, we define when mean path-length bounds should, and should not, be used in conjunction with the bottleneck bound programming problems already defined in Subsection 5.2.1. Then we describe how to calculate mean path-lengths for fully allocated $\overline{\text{SJ}}$ task-graphs, the simplest case, before extending this method to SJ task-graphs and partial allocations (both of which generate a lower-bound on the mean path-length).

When to use Mean Path-Length Bounds

The mean path-length (t) is the mean value of the path-length defined above. t is a lower-bound on the mean response time (t^R) because only one token executes the task-graph while we calculate t , but $N \geq 1$ tokens execute the task-graph for the calculation of t^R (possibly causing extra contention). We will find a throughput bound based on t .

From Little's formula, we know that $\lambda = N/t^R$. Since $0 < t \leq t^R$, as just shown, $\lambda \leq N/t$, and we can always find an upper bound on λ ($\bar{\lambda} \geq \lambda$) satisfying the following mean path-length constraint:

$$\bar{\lambda} \leq \frac{N}{t} \quad (5.12)$$

We will now prove that this mean path-length constraint is never active for all allocations of some SJ and $\overline{\text{SJ}}$ NTGs. The server workloads (x_j^P and x_l^L) developed in the last subsection also apply to NTGs since, using the conversion process of Subsection 2.3.2, we can derive a GTG from any NTG. Adding the following first-order hardware utilisation bounds of the last subsection:

$$x_j^P \bar{\lambda} \leq 1, \forall j \in P$$

$$x_l^L \bar{\lambda} \leq 1, \forall l \in L$$

we see that:

$$\bar{\lambda} \left(\sum_{j \in P} x_j^P + \sum_{l \in L} x_l^L \right) \leq |P| + |L|$$

but, since $\bar{\lambda} t \leq N$:

$$\bar{\lambda} \leq \frac{|P| + |L|}{t}$$

and clearly for $N \geq |P| + |L|$, the hardware utilisation bounds are at least as restrictive on $\bar{\lambda}$ as Equation (5.12) could be, and hence Equation (5.12) can be

dropped. Note that this applies for both SJ and $\overline{\text{SJ}}$ NTGs since there are always at least N tokens in SJ NTGs.

Chapter 6 examines the effects of including Equation (5.12) in integer and linear programming formulations for finding $\bar{\lambda}$ when it can be active. The rest of this subsection elaborates on the methods we have developed to find path-length bounds on partial and complete allocations.

Calculating Mean Path-Lengths for Fully Allocated $\overline{\text{SJ}}$ Task-Graphs

With $\overline{\text{SJ}}$ NTGs, the mean path-length t can be found exactly. We consider only fully allocated NTGs now; however, the next subsection will consider both fully and partially allocated NTGs⁹. Define t_i^P to be the mean time needed to complete T_i on its allocated processor (i.e., $t_i^P = b_i^P/c_j^P$, $a_{ij} = 1$). Similarly, let $t_{ii'}^L$ be the mean time needed to complete communication from T_i to $T_{i'}$ for each execution of T_i (if the tasks are co-allocated, then this is assumed to be zero; otherwise, $t_{ii'}^L = b_{ii'}^L/c_l^L$, $a_{ij} = 1$, $a_{i'j'} = 1$, $j \neq j'$, $l_{jj'} = l$). Where NTG G doesn't show the implicit reference-task and repeated execution loop, we ignore them for simplicity, their workloads being insignificant, as was discussed in the conversion process of Subsection 2.3.2.

Figure 5.2: Example SEQ Task-Graph

Consider Figure 5.2, in which t is the sum of the mean workloads along the one possible path:

$$t = t_A^P + t_{A,B}^L + t_B^P$$

Similarly, for Figure 5.3, we add the weighted mean path-lengths for each of the four possible paths through the NTG, giving:

$$t = .4(t_1^P + t_{1,3}^L + t_3^P) + .1(t_1^P + t_{1,4}^L + t_4^P) + .4(t_2^P + t_{2,3}^L + t_3^P) + .1(t_2^P + t_{2,4}^L + t_4^P)$$

⁹and can thus be used to calculate mean path-length bounds for partially allocated $\overline{\text{SJ}}$ task-graphs.

Figure 5.3: Example SEQ Task-Graph Containing OR Task-Graphs

In general, for arbitrary $\overline{SJ}$ NTGs, clearly:

$$t = \sum_{i \in \text{All Paths}} \text{prob}(\text{Path}_i) t_{\text{Path}_i}, \quad (5.13)$$

where $\text{prob}(\text{Path}_i)$ is the probability that a token will follow Path_i from SOP to EOP, and t_{Path_i} is the mean path-length of Path_i . Note that because of the common tasks in many paths, and the special structure of NTGs, t will be calculated more efficiently than (5.13) suggests.

Realising that all communication in (5.13) occurs over links unless t_u^L is zero (i.e., the tasks don't communicate, or are co-allocated) we find that t for $\overline{SJ}$ NTGs is given by:

$$t = \sum_{j \in P} x_j^P + \sum_{l \in L} x_l^L \quad (5.14)$$

For a $\overline{SJ}$ NTG with a single token, no resource contention occurs, and t^R is just the sum of the mean times spent at each of the system resources (i.e., processors and links), meaning that $t = t^R$ for $N = 1$. We have already shown that this bound can only be active when $N < |P| + |L|$.

Calculating a Lower-Bound on the Mean Path-Lengths for SJ Nested Task-Graphs

We will now find a lower-bound on t for SJ NTGs; we define $0 < \bar{t} \leq t$, where $\bar{t}$ is the lower-bound on t . As discussed before, t is more difficult to find for SJ NTGs because of the possible conflict between parallel paths, even though only one token entered the NTG and only one will leave it. For example, several

parallel branches executing concurrently may have member tasks allocated to the same processor, which could change the t for each branch from that expected for executing each branch alone.

Ignoring this possible contention between branches allows us to find the mean path-length of a corresponding SJ NTG without contention (call this t_{SJ}); the cumulative distribution functions for all the workloads is required ([13] covers the details). For the example of Figure 5.4, the necessary synchronisation at the join-task requires that:

$$t_{SJ} \geq t_{split}^P + t_{join}^P + \max_{i=1}^3 (t_{split,i}^L + t_i^P + t_{i,join}^L)$$

Figure 5.4: Example SJ Task-Graph

To extend this definition of t_{SJ} for arbitrary SJ task-graphs, we first define B_h as the NTG showing the h^{th} branch of the SJ task-graph and any communication from the split-task and to the join-task, but with the split-task and the join-task having no processing workloads. $\bar{t}(B_h)$ is a lower-bound on the mean path-length of B_h . In Figure 5.4, each B_h was a sequence of three tasks; in general, B_h may be a SJ NTG. In addition, we define $\bar{t}_{SJ}$ as a lower-bound on t_{SJ} , finding this formula for $\bar{t}_{SJ}$:

$$t_{SJ} \geq \bar{t}_{SJ} \equiv t_{split}^P + t_{join}^P + \max_{i \in \text{Branches}} \bar{t}(B_{\text{Branch}_i}) \quad (5.15)$$

where B is the set of all B_h and where $\bar{t}(B_h)$ ignores contention between branches and is calculated as in the examples above. Since resource contention can only increase some or all of the $t(B_h)$ values, $t_{SJ} \geq \bar{t}_{SJ}$.

Let A be the subset of tasks T which are allocated, and $\bar{A}$ its complement. When $T_i \in \bar{A}$, t_i^P may be undefined, since we may not be sure of the speed of the processor on which its workload w_i^P will be executed. Similarly, for $T_i, T_{i'} \in \bar{A}$, $t_{ii'}^L$ is also undefined since we may neither be sure that T_i and $T_{i'}$ won't be co-allocated nor of the speed of the link that may need to execute the workload $w_{ii'}^L$. t_{SJ} may thus not be calculable when $A \neq T$, since it may depend on the allocation of $\bar{A}$. To aid in the branch-and-bound process, we will establish lower-bounds on t_{SJ} , even when $A \neq T$.

As described above, the calculation of t_{SJ} can be difficult and (5.15) usually just gives us a lower-bound on it because of the possible resource contention caused by concurrent branches and the possibly stochastic workloads. Eliminating branches from the max in (5.15) can only decrease $\bar{t}_{SJ}$. Let us eliminate all the branches except one, say B_{h^*} . We can then say that:

$$\bar{t}_{SJ^*} \equiv t_{split}^P + \bar{t}(B_{h^*}) + t_{join}^P \leq \bar{t}_{SJ} \quad (5.16)$$

By choosing B_{h^*} to be a branch with the biggest $\bar{t}(B_{h^*})$, we get the tightest such bound and are effectively representing a new task-graph SJ^* which is a bounding NTG.

$\bar{t}$ is a lower-bound on t . Combining inequalities (5.15) and (5.16) shows that $\bar{t}_{SJ^*} \leq t_{SJ}$. Let $\bar{t}$ for SJ NTGs be $\bar{t}_{SJ^*}$ as defined by (5.16). Clearly then, all λ satisfying (5.12) also satisfy:

$$\bar{\lambda} \leq \frac{N}{\bar{t}} \quad (5.12b)$$

and hence $\bar{\lambda}$ can only be increased by using (5.12b) in place of (5.12) in a programming problem: meaning that $\bar{\lambda} \geq \lambda$ as required.

All tasks (and communication link usages) contribute to $\bar{t}$ except those that are part of any eliminated (i.e., non- h^*) branch of any SJ. Define the indicator of whether T_i contributes to $\bar{t}$ as follows:

$$\delta_i \equiv \begin{cases} 1, & \text{if } T_i \text{ contributes to } \bar{t}; \\ 0, & \text{otherwise.} \end{cases}$$

If there is more than one choice for B_{h^*} , any eligible one is chosen; all other branches having $\delta_i = 0$ for all their member tasks. For complete allocations, δ thus depends on T and the w_i^P , $w_{ii'}^L$, c_i^P and $c_{ii'}^L$ values; for partial allocations, δ also depends on

A. Appendix B.3 explains how δ can be found for any allocation. We can now use δ to define $\bar{x}_j^P$ and $\bar{x}_l^L$ as the analogues of x_j^P and x_l^L , respectively, for SJ NTGs as follows:

$$\bar{x}_j^P = \sum_{i \in A} a_{ij} \delta_i \frac{w_i^P}{c_j^P}$$

$$\bar{x}_l^L = \sum_{j, j' \in J(l)} \sum_{i \in A} \sum_{i' \in A} a_{ij} a_{i'j'} \delta_i \delta_{i'} \frac{w_{ii'}^L}{c_l^L}$$

Note that $\bar{x}_j^P \leq x_j^P$ and $\bar{x}_l^L \leq x_l^L$ always, since some work may be ignored (processing work for tasks with $\delta_i = 0$ is ignored, as is communication work between task pairs, either of which having $\delta_i = 0$). We can now define $\bar{t}$ (as we already did for the t of $\overline{\text{SJ}}$ NTGs) in terms of the mean workloads as:

$$\bar{t} = \sum_{j \in P} \bar{x}_j^P + \sum_{l \in L} \bar{x}_l^L$$

We are now able to repeat the steps already done to use the path-length constraint in $\overline{\text{SJ}}$ NTGs, for these SJ NTGs. Transforming (5.12b) to use $\bar{\mu}$ ($\equiv \bar{\lambda}^{-1}$) gives the following constraint that can be added to Problem IP1 in Section 5.2.1:

$$\sum_{j \in P} \bar{x}_j^P + \sum_{l \in L} \bar{x}_l^L - N\bar{\mu} \leq 0$$

Recall that the mean path-length constraint can only be active in IP1 when $N < |P| + |L|$. For a particular branch-and-bound state k with allocation a , this gives the following added constraint for IP2 in Subsection 5.2.1:

$$\sum_{j \in P} \sum_{i \in \bar{A}} \bar{a}_{ij} \delta_i \frac{w_i^P}{c_j^P} + \sum_{l \in L} \sum_{j, j' \in J(l)} \sum_{i \in \bar{A}} \sum_{i' \in \bar{A}} \bar{a}_{ij} \bar{a}_{i'j'} \delta_i \delta_{i'} \frac{w_{ii'}^L}{c_l^L} +$$

$$\sum_{l \in L} \sum_{j, j' \in J(l)} \sum_{i \in A} \sum_{i' \in \bar{A}} \delta_i \delta_{i'} (a_{ij} \bar{a}_{i'j'} \frac{w_{ii'}^L}{c_l^L} + \bar{a}_{i'j} a_{ij'} \frac{w_{i'i}^L}{c_l^L}) - N\bar{\mu} \leq - \sum_{j \in P} \bar{x}_j^P - \sum_{l \in L} \bar{x}_l^L \quad (5.17)$$

Unfortunately, when $A \neq T$, δ depends on $\bar{A}$ and hence $\bar{a}$ in a complex way (as described in Appendix B.3).

By ignoring part of the unallocated work¹⁰, we can simplify the derivation of δ and also simplify (5.17) tremendously. Ignoring the second and third terms in (5.17) leaves us with the following path-length constraint:

¹⁰those terms in (5.17) that combine allocated and unallocated tasks (i.e., the $a_{ij} \bar{a}_{i'j'}$ and $\bar{a}_{i'j} a_{ij'}$ terms).

$$\sum_{j \in P} \sum_{i \in \bar{A}} \bar{a}_{ij} \delta_i \frac{w_i^P}{c_j^P} - N\tilde{\mu} \leq - \sum_{j \in P} \bar{x}_j^P - \sum_{l \in L} \bar{x}_l^L \quad (5.18)$$

Since $\sum_{j \in P} \bar{a}_{ij} = 1$ for each $i \in \bar{A}$, and $c_j^P \leq \max_{j' \in P} c_{j'}^P$, we can rewrite the first term on the RHS and obtain the following:

$$-N\tilde{\mu} \leq - \sum_{j \in P} \bar{x}_j^P - \sum_{l \in L} \bar{x}_l^L - \sum_{i \in \bar{A}} \delta_i \frac{w_i^P}{\max_{j \in P} c_j^P} \quad (5.19)$$

This can be a looser bound than (5.17), but it is much simpler to use and fits into the LP formulations of the problem as a single extra constraint. Chapter 6 will show the effects of using this bound on several SJ and SJ NTGs.

5.3 'Consistent' Lower-Bound Costs for Allocations

This section discusses consistent techniques used to find lower-bounds on the cost of all the complete allocations reachable from a given partial allocation. The methods are consistent because they use the same techniques to evaluate complete allocations as they do for partial allocations.

The first two subsections use approximate MVA for all search-nodes, making different approximations to the workloads that non-leaf-nodes represent. Subsection 5.3.3 describes how we can duplicate the communication minimisation results of previous researchers using our branch-and-bound search technique. Two of the linear program based bounds introduced in Section 5.2 are used in Subsections 5.3.4 and 5.3.5.

5.3.1 'Allocated' Work Only Lower-Bound Cost

This technique estimates the system workloads using only the allocated tasks. The effects of unallocated tasks are ignored, as are interactions between allocated and unallocated tasks. To find the system throughput associated with this artificial problem, approximate MVA is used. Since unallocated tasks are ignored, this technique produces a strictly optimistic bound for a partial allocation's throughput and hence also for its state cost. Gordon and Dowdy have shown this in [28].

This being a simple method, the bounds at each non-leaf-node are not expected to be tight, leading to more node evaluation than some other methods (see Chapter 6 for more details).

5.3.2 'Load-Balancing' Lower-Bound Cost

This heuristic bounding technique tries to get a more realistic lower-bound cost by assigning the unallocated processing workload to the processors to equalise processor utilisation as much as possible. This technique differs from that of the last subsection in that the unallocated processing workload is considered (the unallocated communications workload is still ignored, however). An example illustrating the technique will be followed by a justification for its use and an algorithm describing how the technique is implemented.

Figure 5.5 shows an example of how the processor workloads are modified for a partly allocated three processor example. The solid bars represent the old x_i^P values and the dashed bars show the changes made to nearly equalise the processor workloads by adding the unallocated processing workload. As in Subsection 5.3.1, the undetermined communications load is given an optimistic lower-bound of zero (eventually, some of this communication could be considered, as it has been above, in the bottleneck bounds).

Figure 5.5: Processor Workload Equalization Example

Buzen has shown [9] that for processors of equal speeds, the best processing work allocation for maximum throughput equalises processor utilisations. Unfortunately, this result doesn't generalise to our case of processors that may have different speeds. For the case of two processors, Buzen found that the faster processor should

be given a higher utilisation. We expect that for multiple processors of different speeds, the slower ones should have smaller (but usually non-zero) workloads than the faster ones. The optimal workload distribution for this multiple processor case has not yet been determined.

This technique assumes that, even for processors of different speeds, assigning unallocated processor work to equalise processor utilisation as much as possible will allow us to find close to optimal allocations. Chapter 6 will present results showing that this assumption is valid. We will now elaborate on the load-balancing technique.

Given a partial allocation, the processing work already allocated to each processor is known. The unallocated processing work is totaled and distributed among the processors to fill the deepest valleys in a processor work to processor speed ratio chart like Figure 5.5 first. The algorithm proceeds in stages, with each set of lowest valley processors having work added until either the next lowest group's x^P value is reached or no unallocated work remains to be allocated.

We will present the algorithm as an iterative procedure (of at most $|P|$ iterations) using the following definitions:

$\bar{w} = \sum_{i \in \bar{A}} w_i^P$, the total unallocated processor workload

R^M = the set of processors, P_j , all having the lowest relative utilisations, x_j^P

R^N = the set of processors, $P_{j'}$, all having the next-to-lowest relative utilisations.

Note that R^M always has at least one member (i.e., $|P| \geq 1$), and that R^N has no members iff all $|P|$ processors have the same relative utilisation. Two additional properties of the above three definitions are that:

1. if $R^M \neq P$ then $x_j^P < x_{j'}^P, \forall j \in R^M, j' \in R^N$;
2. if $R^M + R^N \neq P$ then $x_j^P < x_{j''}^P, \forall j' \in R^N, j'' \in P - R^M - R^N$.

The algorithm initialises $\bar{w}$ to the above value and then decreases it to zero, in stages, modifying $x_j^P \forall P_j \in R^M$ at each stage; refining this:


```

 $\bar{w} = \sum_{i \in \bar{A}} w_i^P$ 
do while  $\bar{w} > 0$ 
    if  $|R^N| = 0$  then ( $R^M = P$ )
         $\Delta R = \frac{\bar{w}}{\sum_{j \in R^M} c_j^P}$ 
    else
         $\Delta R^{\max} = x_j^P - x_{j'}^P, j \in R^N, j' \in R^M$ 
         $\Delta R = \min(\Delta R^{\max}, \frac{\bar{w}}{\sum_{j \in R^M} c_j^P})$ 
    endif
     $\bar{w} = \bar{w} - \Delta R \sum_{j \in R^M} c_j^P$ 
     $x_j^P = x_j^P + \Delta R \quad \forall j \in R^M$ 
endwhile

```

In Figure 5.5, the while loop is repeated twice, first with $R^M = \{P_2\}$ and then with $R^M = \{P_2, P_3\}$. After the second iteration, $\bar{w} = 0$ and the loop stops. The revised x_j^P values are then used to find the lower-bound cost of partial allocations. Since $\bar{w} = 0$ for complete allocations, the above algorithm leaves them unaffected.

5.3.3 'Communication Minimisation' Lower-Bound Cost

This heuristic respects all the allocation problem constraints (i.e., T_{ii}^{COAL} and T_{ij}^{PREF}) but ignores processor execution and only minimises the sum of the communication workloads x_i^L . Communication minimisation was previously used by Stone [70], and Haessig and Jenny [31], among others. Constraints were added to the problem to try to allow communications minimisation to correspond to performance maximisation. As was discussed in Section 3.1, this correspondence is not possible for many realistic problems where both processor and communication workloads affect performance.

5.3.4 'Bottleneck' Lower-Bound Cost

These two heuristics use the bottleneck bounds of Subsection 5.2.1 for all nodes. A search using these heuristics doesn't necessarily give an optimal allocation

since they deal only in throughput bounds. The throughput bound is maximised and typically the actual throughput is also nearly maximised.

For inherently processing bound problems, the size of the linear program can be reduced, without increasing the number of nodes inspected, by ignoring link utilisations.

The tightness of these bottleneck bounds should increase with the number of tokens in the system, since they cause operation close to whatever bottlenecks the allocation exposes.

Chapter 6 shows some results of applying these two techniques. Solutions are generally found far quicker using these techniques than using the methods of Subsection 5.3.3, since a single bound is used throughout the search, whereas the methods in Subsection 5.3.3 use a heterogeneous modelling solution cost for leaf-node bounds (causing many extra leaf-node evaluations to be made in the solution search).

5.3.5 'Mean Path-Length and Bottleneck' Lower-Bound Cost

This technique finds throughput capacity bounds using both the bottleneck bounds of the last subsection and the mean path-length bounds of Subsection 5.2.2.

5.4 'Inconsistent' Lower-Bound Costs for Allocations

All three of the methods described in this section differ from those of Section 5.3 in that they use approximate MVA to evaluate leaf-nodes (complete allocations) and some other method to evaluate non-leaf-nodes. This inconsistency, or gap, in which leaf-nodes often have higher cost, caused by using approximate MVA, makes an optimal solution much harder to find, as Chapter 6 will verify.

5.4.1 'Ignore Partial Allocations' Lower-Bound Cost

This lower-bound cost assigns all partial allocations lower-bound costs of zero (i.e., representing unbounded throughputs), reducing the cost of creating non-leaf-nodes. Leaf-nodes are evaluated more slowly, using the queueing network analysis tool of Section 5.1. Since a lower-bound cost of zero allows no differentiation of non-leaf-nodes, the entire search-space must be evaluated to find an optimal allocation. Evaluation can't even be cut short via a non-zero bracket (see Section 4.2) because the lowest lower-bound cost is always zero until all non-leaf-nodes have been expanded.

5.4.2 'Bottleneck' Lower-Bound Cost

This method differs from the bottleneck lower-bound cost method of Subsection 5.3.4 only in using approximate MVA for complete allocations (leaf-nodes). The processor induced bounds are either considered alone in the linear program's constraints or together with the communication link bounds. Results found using both of these methods are described in Chapter 6.

5.4.3 'Mean Path-Length and Bottleneck' Lower-Bound Cost

This technique finds throughput capacity bounds for non-leaf-nodes using both the bottleneck bounds of the last subsection and the mean path-length bounds of Subsection 5.2.2. For leaf-nodes, approximate MVA finds the throughput, as was done in Subsection 5.3.1.

5.5 'Multi-Class' Lower-Bound Costs

Section 2.4 introduced the extensions to the notation to allow multi-class STAPs. These extensions impact both the linear program and mean-value analysis based evaluation techniques, as sketched below.

Linear Programs

The previous linear programs can be extended to allow multiple classes to be considered. The bottleneck lower-bounds can be extended most easily to add the work from each class and to weight the throughput bounds for each class to arrive at the cost of an allocation, and estimates for partial allocations. The path-length bound for each class depends on the throughputs of the other classes as well. This can be considered in much the same way as described in Subsection 5.2.2.

Mean-Value Analysis

To find the allocation's throughput via approximate MVA from the product-form queueing network, we only need the service times and visit-ratios of each class to the servers. In product-form queueing networks, synchronization effects in SJ NTGs cannot be modelled.

The workloads for FCFS servers (such as links) should be exponentially distributed and have the same mean for each class. Product-form is usually violated for each link for multi-class STAPs both because non-equal $\bar{b}_{ij}^L$ values for the tasks connected by a link create hyper-exponential service demands for that link within a class and because of different $\bar{b}_{ij}^L$ values for each class.

5.6 Summary

This chapter has discussed several evaluation techniques to find lower-bound costs for both partial and complete allocations of both nested and general task-graphs. These evaluation techniques are used by the branch-and-bound search technique of Chapter 4. The next chapter examines and compares the results of using these evaluation techniques on a range of examples.

Chapter 6

Experience with Task Allocation Strategies

This chapter compares the task allocation strategies introduced in the last two chapters by applying them to several static task allocation problems (STAPs). Section 6.1 discusses experiences with a particular family of general task-graph (GTG) examples; while these examples are specific, they give insights which are applied more generally in sections 6.2 and 6.3. Section 6.2 considers several nested task-graph (NTG) examples, using mean path-length bounds, and Section 6.3 discusses the effects of problem size on task allocation. Section 6.4 compares several allocations found by the present approach with that of Stone's method for a version of an allocation problem that he published. Section 6.5 then describes experiences with an example related to Ma's air-defense example. The GTG experiences of Section 6.5 include a study of hardware symmetry and task co-allocation based search-space reduction, and task-graph reduction. Section 6.6 summarises our experiences and extracts the common parts of our recommendations.

6.1 Experience Using the General Task-Graph Approach

The general task-graph (GTG), defined in Section 2.1, has an unrestricted,

or non-nested, successor-predecessor structure. This section discusses a family of GTG STAPs with a loop structure, which allows us to examine the effects of several solution search strategies (these STAPs actually have a nested structure but the algorithms used in this section will not exploit it). By solving the same STAPs in several ways and comparing the results, we determine appropriate strategies and show that sizable STAPs can be solved using these strategies.

Subsection 6.1.1 introduces this family of stochastic loop examples, and Subsection 6.1.2 summarises the notation for reporting and comparing the resulting allocations. Subsections 6.1.3 through 6.1.8 compare the performance of the search and bounding techniques discussed in chapters 4 and 5 for several of these examples. The task allocation order and search optimality bracket are found to have dramatic effects on the time needed to find a solution. Detailed results supporting these examples are contained in Appendix A. Subsection 6.1.9 makes recommendations as to which parameter selections should be used for these STAPs, and STAPs in general.

6.1.1 The Stochastic Loop Task Allocation Problem

This set of examples is constructed so as to easily be scaled from small graphs which are easily allocated, to large graphs, which are not. A variety of STAP sizes are used to test the cost bounds of Chapter 5 and the search strategies of Chapter 4. We start by describing the task-graphs and the processor-graphs for these STAPs and continue with the relations constraining their allocation.

The notation used to describe the stochastic loop GTGs is given in Figure 6.1, which illustrates the simplest non-trivial example¹. Each node is labelled both with the task it represents and with the average processing workload for that task (in processing units per response). Similarly, the arcs are labelled with the average communication workload from the task at the arc's tail to the task at the arc's head (in communication units per response).

¹this notation is an extension of that developed in Chapter 2 to describe STAPs in general.

Figure 6.1: Example GTG Notation

Figure 6.2 shows the general stochastic loop task-graph, wherein tasks are executed sequentially in a loop by a set of tokens. The tokens cause processing work as they execute tasks and communications work as they pass from one processor to another². The mean intertask communications workload is labelled on each directed arc ($b_{ii}^L = i$ here) while the mean processing workload is labelled below each node ($b_i^P = |T| + 1 - i$ here). Each link is modeled as a smart asynchronous first-come-first-served bus, as described in Subsection 5.1.3.

Figure 6.2: General Stochastic Loop Task-Graph

Figure 6.3a shows the processor-graph for the case of one link (i.e., a common bus) and three processors; we call it the L1P3 processor-graph. Figure 6.3b shows the hardware architecture that the processor-graph implies; each processor uses the link to communicate with the other processors. The l matrix of Figure 6.3c represents this single link processor-graph with three rows and three columns (when non-zero, $l_{j,j'}$ is the communication link used for communication from P_j to $P_{j'}$). Diagonal entries in the l matrix must be zero since self-communication (i.e.,

²unless otherwise stated, all our stochastic loop examples use 10 tokens.

Figure 6.3: L1P3 Processor-Graph (a), Hardware Architecture (b), and l matrix (c)

communication between any two tasks on the same processor) never uses a link (recall that when $l_{jj'}$ is zero, no communication can occur from P_j to $P_{j'}$). For the more general case of $|P|$ processors and one link, the processor-graph is represented by the array l , having $|P|$ rows and $|P|$ columns defined as follows:

$$l_{jj'} = \begin{cases} 1, & \text{if } j \neq j' \\ 0, & \text{otherwise} \end{cases} \quad \forall j, j' \in P$$

Figure 6.4: L2P2 Processor-Graph (a), Hardware Architecture (b), and l matrix (c)

Figure 6.5: L2P3 Processor-Graph (a), Hardware Architecture (b), and l matrix (c)

Figures 6.4a and 6.5a are examples of two link processor-graphs (L2P2 and L2P3³, respectively). Communication in both directions between any two of the processors in Figure 6.5a uses the same link. Figures 6.4b and 6.5b illustrate the corresponding hardware architectures. The l matrices of figures 6.4c and 6.5c represent the processor-graphs of figures 6.4a and 6.5a, respectively.

For simplicity, to create l matrices for processor-graphs with two links and more than three processors, we will have all communications from P_j to $P_{j'}$ use L_2 if $j < j'$ and use L_1 if $j > j'$. This rule creates the following more regular l matrix for the four processor case:

$$l = \begin{bmatrix} 0 & 2 & 2 & 2 \\ 1 & 0 & 2 & 2 \\ 1 & 1 & 0 & 2 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

A potential disadvantage of using this rule for the three processor case is that all three processors may then need the hardware to use both links, whereas in Figure 6.5, P_1 only uses L_1 . The processor-graphs of figures 6.3, 6.4 and 6.5 will be used in the stochastic loop examples of subsections 6.1.3, 6.1.4 and 6.1.5.

The communications workloads of stochastic loop STAPs are represented by b^L matrices of the following form:

$$b^L = \begin{bmatrix} 0 & 1 & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & 0 & 2 & 0 & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot & 0 & 0 & (|L| - 1) \\ |L| & 0 & \cdot & \cdot & \cdot & \cdot & 0 \end{bmatrix}$$

Since we want these STAPs to challenge our allocator as much as possible, all tasks can be allocated onto all the processors (T^{REF} is then a $|T|$ by $|P|$ matrix

³Note that although this is one of 2^6 , or 64, different architectures using two links to fully connect three processors: whenever we specify L2P3, we refer to the hardware architecture and processor-graph of Figure 6.5.

of ones) and each task can but need not be co-allocated with any other task (i.e., T^{COAL} is a $|T|$ by $|T|$ matrix with its entries set to any value other than zero or one). b_i^P is $|T| + 1 - i$. Unless otherwise stated, we allocate the tasks in order of decreasing b^P (i.e., $T_i^{ORDER} = i$).

This subsection has introduced stochastic loop STAPs as an easily scalable family of STAPs that allow us to test allocation strategies. Later subsections do this testing and compare the results.

6.1.2 Notation

This subsection presents the notation used to describe the stochastic loop STAP examples. The terms used in the tables and figures describing the results of this chapter and Appendix A are now defined⁴:

problem: a code of the form $L\alpha P\beta T\gamma$ for the static task allocation problem (STAP) being solved (e.g., L1P2T6): where α , β and γ are the number of communication links ($|L|$), processors ($|P|$) and tasks ($|T|$), respectively, in the STAP being studied (e.g., L1P2T6 means the STAP has one link, two processors, and six tasks). The processor-graph and the task-graph are determined uniquely from these parameters as described in the previous subsection.

L^s : the link speed scaling factor by which each c_l^L value has been multiplied (this allows a family of related STAPs to be easily created by changing this factor and leaving the rest of the STAP definition unchanged). It will be seen that increasing L^s increases the difficulty of the STAP because more allocations have similar cost bounds (the processors becoming the bottlenecks). As L^s is decreased, on the other hand, the links become bottlenecks and the allocations with the lowest cost bounds communicate across the links less and less, forcing pairs of tasks that communicate even moderately onto the

⁴only the problem term is not applicable to the NTG and other than stochastic loop GTG examples later in the chapter.

same processor. Thus fewer allocations (both partial and complete) have cost bounds similar to the optimal value for lower L^s values, reducing the difficulty of the search.

mode: a two word code (i.e., "cost search-mode") describing how to solve the STAP; the first word describes the lower-bound cost (function) to minimise and the second word determines how to search by specifying a search-mode. Cost and search-mode are defined next.

cost: the lower-bound cost function to minimise. Function names beginning with the letter t mean that the search will minimise either λ^{-1} or $\bar{\lambda}^{-1}$ (depending on the particular cost). The function name is one of the following:

c: the sum of the mean communications load using the links, per response; hence, this cost minimises communication link loads regardless of processor loads.

tadd: (Subsection 5.3.2) tries to equalise processor utilisations by allocating the unallocated processor work (ignoring unallocated communication link work) amongst the processors before using the approximate MVA (AMVA) technique of Chapter 5 to evaluate the cost.

talk: (Subsection 5.3.1) ignores unallocated work but generates and solves an AMVA for each node.

tno: ignores non-leaf-nodes, but evaluates all leaf-nodes using AMVA.

ts: uses linear program generated lower-bounds (considers both processor and link bottlenecks) for non-leaf-nodes and AMVA for leaf-nodes.

tsb: (Subsection 5.3.4) uses linear program generated lower-bounds directly as the cost metric (even for leaf-nodes, re-

sulting in non-optimal throughputs, but optimal throughput bounds).

tsp: uses linear program generated lower-bounds (accounts for both processor and link bottlenecks and mean path-length constraints) for non-leaf-nodes and AMVA for leaf-nodes.

tspb: (Subsection 5.3.5) uses linear program generated lower-bounds directly as the cost metric (even for leaf-nodes, resulting in non-optimal throughputs).

tspo: uses linear program generated lower-bounds (accounts for only processor bottlenecks) for non-leaf-nodes and AMVA for leaf-nodes.

tspob: ignores communications link bottlenecks, but otherwise is like *tsb*.

tslo: uses linear program generated lower-bounds (accounts for only communications link bottlenecks) for non-leaf-nodes and AMVA for leaf-nodes.

tslob: (Subsection 5.3.3) ignores processor bottlenecks, but otherwise is like *tsb*.

tsec: uses linear program generated lower-bounds (accounts for extra communications) for non-leaf-nodes and AMVA for leaf-nodes.

tsecb: (LP2 in Subsection 5.2.1) considers extra communications link bottlenecks, but otherwise is like *tsb*.

These cost functions were introduced in Chapter 5.

search-mode: describes the way the search is to proceed. The following search methods are allowed:

b: breadth first,

d: depth first,

Lf: least lower-bound (first in first out), and

Ll: least lower-bound (last in first out).

These methods are used to select a node from the active set and insert created nodes into the active set, as discussed in Section 4.2.

time: seconds of processor computation time (not wall-clock time) needed to search for a solution to a STAP using compiled (via Liszt: version 8.39a) Franz Lisp (version 38.92) with Sun Unix, Version 4.2 on Sun 3/60 workstations. Since garbage collection depends on the Lisp system's past calculations, this time doesn't include the Lisp system's garbage collection time (which typically ranges from 5% to 20% of the time). Even after excluding garbage collection time, time depends somewhat on the Lisp system's past calculations and on the workload of the workstation. In determining the same allocation several times, differences in time are small (usually less than 5%, and always less than 10%)⁵.

nodes: the number of nodes created in reaching the solution. Creating a node includes evaluating its lower-bound cost.

a : the solution allocation to the STAP. a is a sequence of $|T|$ digits, wherein if the i^{th} digit is j , then T_i is allocated to P_j . This notation for a is thus limited to STAPs with less than ten processors. All our examples fit this category⁶.

$\lambda(a)$: the throughput of the solution allocation, a , in tokens per second along the task loop.

$\bar{\lambda}(a)$: the throughput capacity of the solution allocation, a , in tokens per second along the task loop. $\bar{\lambda}(a)$ is an upper bound on $\lambda(a)$.

⁵Most allocations were only determined once. Where allocations were determined several times, only the average times are reported in this chapter and in Appendix A.

⁶This convention for a is a shorthand for the notation developed in Chapter 5, where a_{ij} was one if T_i was allocated to P_j . This convention for a is used only in this chapter and in Appendix A.

These terms will be used throughout the rest of this chapter and in Appendix A.

6.1.3 Comparison of Results

This subsection compares the allocations found using the various strategies described above. The results reported in tables A.1 through A.6 of Appendix A will be discussed here; the tables uncover trends in both the search-modes and in the lower-bound cost functions. The effectiveness of each of the search-modes will be considered first, followed by a discussion and comparison of the cost functions. It is seen that an additional processor has a much more dramatic effect on the solution cost than adding an extra task. This is just what Section 4.3 predicted.

The Effect of Search-Modes

Because of their small housekeeping overhead, both the breadth-first and depth-first search-modes perform acceptably for small STAPs having a search-space of only a few hundred nodes⁷. However, these brute-force methods aren't useful for larger STAPs since they ignore each node's cost information in choosing the next node to expand. This leads to unacceptably long solution times. The depth-first method could be used as a primer for another method, since it can be used to find an initial solution allocation by setting the solution bracket to one. Of these two methods, the depth-first method generally is more efficient, since it examines leaf-nodes first (and hence may find a guaranteed optimum earlier).

Lf and *Ll* are the only viable search-modes for larger STAPs, since only these use the non-leaf-node (i.e., partial allocation) cost values in determining the node to expand next. Both methods create almost the same number of nodes, but *Ll* manipulates the active set more efficiently (typically saving 2% of the time that *Lf* would use⁸).

⁷Nodes are only added to the head or tail of the active set. This contrasts with the *Lf* and *Ll* modes, which keep the active set sorted in order of decreasing lower-bound cost.

⁸comparing tables A.9 and A.10 shows that *Ll* is about 1.5% quicker than *Lf*.

The Effect of Cost Functions

As the results in Appendix A show, lower-bound cost functions greatly affect both the solution time and the solution optimality. Each of the cost functions for GTGs will now be examined:

- *c*, communications only:

This method, which chooses an allocation that minimises the total amount of communication, can yield arbitrarily bad results since the processors are often the system bottleneck, and are ignored while minimising the *c* cost function. This method is quicker, but less accurate, than the *tskob* method, since it does not consider link bottlenecks. Table A.6 illustrates the poor performance, in terms of solution λ as compared with the *tadd* method. Note that as L^* decreases, the λ of *c*'s allocation approaches that of *tadd* (optimal).

- *tadd*, equalise utilisation, AMVA:

Figures 6.6 to 6.11 show that the *tadd* heuristic is relatively more effective for the ten task STAPs than for the six task STAPs. Even though *tadd* is heuristic, it has always produced the correct solutions (where this was verifiable). This may be because stochastic loop STAPs use only processors of the same speed. It is known in another context that optimal allocations utilise faster processors more heavily than slower processors [9].

- *tall*, allocated work only, AMVA:

Although this method always finds an optimal allocation, it is neither an efficient nor an affordable method in many cases (as shown by figures 6.6 through 6.11). Ignoring both unallocated computational and communications work, as this method does, makes it the second worst method shown on the figures, both in terms of nodes and time.

- *tno*, evaluate leaves only, AMVA:

This method is only usable for the smallest STAPs, since it amounts to a near complete evaluation of the space search; no use being made

of the costs of partial allocations⁹.

– *ts*, LP, AMVA:

tsb is a poor method (see figures 6.8 through 6.11, where it is usually the worst method), because the lower-bound cost methods used for leaf-nodes and non-leaf-nodes are inconsistent (as discussed in Section 5.4).

– *tsb*, LP:

tsb is an outstanding method, with answers usually within 1% of the optimum (see tables A.1 through A.5).

– *tspo*, processing only LP, AMVA:

tspo uses the modelling method to evaluate leaf-nodes, and hence can get optimal solutions, but at nearly the highest cost in nodes and time (tables A.2, A.5, A.11 and A.12), since processing only minimisation has thrown it off track (as happens with all the inconsistent lower-bound cost methods).

– *tspob*, processing only LP:

The *tspob* method is similar to *c* in that it can also yield arbitrarily bad results. Notice that λ drops from 0.04522 to 0.00414 in Table A.6 as L^s decreases, converting the STAP to being communications-bound and forcing optimal allocations to reduce their communications. This shifts *tspob* from being within 1% of the optimal λ at $L^s = 1$ to being even worse than *c* when $L^s = 0.01$.

– *tslo*, links only LP, AMVA:

tslo uses the modelling method to evaluate leaf-nodes, and hence can get optimal solutions, but at high cost in nodes and time (tables A.2 and A.5), since communications only minimisation has thrown it off track.

– *tslob*, links only LP:

The *tslob* method is similar to *c*, but where preallocations force some

⁹the results in this chapter and Appendix A never use the evaluate leaves only (*tno*) method because it is inferior to the other methods, but appears in this list for completeness.

communications, this method tries to limit each link to at most the same limit (not to drive it to zero, as c would).

- *tsec*, extra communications LP, AMVA:

Like *ts*, but with a 50% higher solution time (see tables A.11, A.12 and A.13). Not used in our examples.

- *tsecb*, extra communications LP:

The *tsecb* method is observed to give almost the same solution λ as *tsb* with nearly the same number of nodes created, but at the expense of from two to five times the solution time for *tsb*. Not used in our examples.

Comparison of Cost Functions

The cost functions that use linear program bounds for both partial and complete allocations (i.e., *tsb* and *tspob*) are shown to be effective (often much beyond the *tadd* method). The sub-optimality observed in the linear program bound results occur because the linear program bounds are used even for complete allocations. STAPs that have greater numbers of tokens should reduce these sub-optimality by bringing the actual performance closer to the bound¹⁰.

The c and *tspob* methods are duals. Table A.6 shows this duality in the degradation of allocation throughput as the extremes of L^s are approached. Where L^s is low and communications need to be reduced, then c gives good allocations, whereas when L^s is high and processor bottlenecks are approached in optimal allocations, *tspob* gives better allocations.

Although *tspob* typically saves 10% of the solution time of *tsb*, *tsb* is recommended since it is robust to processing-dominated problems (having high L^s values), which *tspob* is not.

Figures 6.6 through 6.11, which are based on tables A.1, A.3 and A.4 in Appendix A, show the effects of both L^s variations and four different solution methods on two stochastic loop STAPs (L1P2T6 and L2P3T10).

¹⁰increasing N , the number of tokens is not a general option, since it increases response times for the application users.

Figure 6.6: Nodes Created vs. L^s for L1P2T6 using Ll

Figure 6.7: Time to Solution vs. L^s for L1P2T6 using Ll

Figure 6.8: Nodes Created vs. L^s for L1P3T10 using Ll

Figure 6.9: Time to Solution vs. L^s for L1P3T10 using Ll

Figure 6.10: Nodes Created vs. L^s for L2P3T10 using Ll

Figure 6.11: Time to Solution vs. L^s for L2P3T10 using Ll

As discussed in Subsection 6.1.2, increasing L^s , and thus lightening the communication workload, expands the number of allocations that have cost bounds similar to the optimal value (even though the search-space size remains the same). Varying L^s allows the STAP to be tuned in difficulty without redefining its structure. The figures vary L^s over the range from 0.001 to ten because increasing or decreasing L^s beyond this range has no further effect.

The *ts* method is inferior to *tsb*, as both sets of figures show. The *tall* method is worse than *tadd* in all cases. The *tsb* method is best among these four, if its reduced throughput, with respect to optimal allocations can be accepted¹¹.

6.1.4 Task Allocation Ordering

The order of task selection for allocation plays a significant role in the allocation speed (but not in the resulting allocation itself for most bounding methods). This was described in Subsection 4.2.1. Tables 6.1 and 6.2 illustrate this effect for the L1P3T8 STAP using the *tall* and *tadd* cost functions, respectively. Remember that task 8 has the smallest processing time and the heaviest communications load, and task 1 is the opposite in the loop problems (see Figure 6.2).

L^s	T^{ORDER}	nodes	time	$\lambda(a)$	a	comments
1.0	12345678	144	10.2	0.0692	12321333	best
1.0	18273645	413	30.2	0.0692	13231222	intermediate
1.0	87654321	933	69.0	0.0692	23132111	worst
0.01	12345678	86	5.8	0.0307	12111111	worst
0.01	18273645	60	4.1	0.0307	12111111	intermediate
0.01	87654321	24	1.6	0.0307	12111111	best

Table 6.1: Effects of T^{ORDER} for L1P3T8 using *tall* Ll

¹¹throughputs were reduced by less than 10% for these examples.

L^s	T^{ORDER}	nodes	time	$\lambda(a)$	a	comments
1.0	12345678	48	3.5	0.0692	12321333	best
1.0	18273645	146	10.4	0.0692	13231222	intermediate
1.0	87654321	507	37.1	0.0692	23132111	worst
0.01	12345678	86	5.8	0.0307	12111111	worst
0.01	18273645	60	4.2	0.0307	12111111	intermediate
0.01	87654321	24	1.7	0.0307	12111111	best

Table 6.2: Effects of T^{ORDER} for LIP3T8 using *tadd Ll*

Tables 6.1 and 6.2 show that when the total communication workloads dominate the total processing workloads, the tasks with the heaviest communication workloads should be allocated first. Conversely, when the processing workloads dominate, the tasks with heavy processing workloads should be allocated first.

With a L^s of 1.0, the processing workloads dominate, and the processors form a bottleneck (see Figure 6.3). Hence, the T^{ORDER} of 12345678¹², which allocates tasks in order of decreasing w^P , is the most efficient (in both search time and nodes created) at discriminating between partial allocations. Conversely the T^{ORDER} of 87654321, which performs poorly, is the least discriminating possible. The 18273645 T^{ORDER} – a mixture or permutation of the most and least discriminating T^{ORDER} values – has the expected intermediate performance.

If the STAP is known or expected to be communications-bound, then strict communications minimisation is efficient. In most realistic multiprocessor STAPs, communications minimisation (even where preallocations of key tasks to separate processors is used) is insufficient since it ignores the performance differences of equally communicating allocations and when extra communications is worthwhile (link bottlenecks are also ignored). As expected, the total number of nodes created by all three T^{ORDER} values decreases as L^s decreases¹³.

¹²i.e., where $T^{ORDER} = [1, 2, 3, 4, 5, 6, 7, 8]$.

¹³This reduction occurred because the communications-bound STAP has fewer reasonable allocations. Only those allocations with small communications workloads, a fraction of the possible allocations, have costs close to the optimum, and are thus reasonable.

For communications-bound STAPs (in which link usage is more important than processor load balancing, since the communication links are the performance bottlenecks), an ordering based on decreasing task communication workloads is effective. Alternatively, for processor-bound STAPs, an ordering based on decreasing computational work requirements is effective. Subsection 4.2.1 listed the four available T^{ORDER} generation heuristics (lexically increasing and decreasing, and decreasing w^P and total w^L values). Alternatively, T^{ORDER} can be specified manually. When part of the task-graph communicates heavily and another part is processing bound, a conflict occurs in the order in which tasks should appear in T^{ORDER} . To resolve this conflict, a task's communications workload must be combined with its processing workload. At first glance, execution time of the workload components should be used to combine the workloads; however, two factors combine to make this difficult; (1) task co-allocation and preference requirements prevent or enforce some of the allocations, and (2) heterogeneous architectures mean that execution time depends on the allocation, but T^{ORDER} does not.

6.1.5 Search Optimality Bracket

The optimality bracket used for the branch-and-bound search method, introduced in Section 4.2, has a dramatic effect on both the solution cost and on the time required to find a solution. The bracket (β) is in the range $0 \leq \beta \leq 1$, with a bracket of zero meaning that only a leaf-node of optimal cost is acceptable as a solution, while with a bracket of one, the first leaf-node evaluated stops the search. Letting $c(a_{LB})$ be the cost of the least lower-bound node¹⁴, we terminate the search if we have a complete allocation, a , satisfying:

$$c(a)(1 - \beta) \leq c(a_{LB})$$

Having found such an allocation a , we know that optimal solution a^* must have a cost in the range $(1 - \beta)c(a) \leq c(a^*) \leq c(a)$. Guaranteed optimality (a bracket of zero) has its price: solution time, a relationship that is explored in tables A.7 through A.10 of Appendix A.

¹⁴the leaf-node with the least lower-bound cost.

The times and node counts of Table A.7 show that for a ten task system, a bracket of zero (requiring an optimal allocation to be found) needs about five times more solution time and node creations than a bracket of 0.01 (i.e., allowing the search to stop after finding a complete allocation with a cost at most 1% higher than the optimal allocation). All the solutions in Table A.7 are the same, even though the assurance of optimality is missing from those with non-zero brackets.

Tables A.9 and A.10 shows the same trends for a more difficult STAP with three processors. The first complete allocation found is almost optimal ($\lambda(a) = 0.04523$), however to gain the last 0.1% of the optimal throughput ($\lambda(a) = 0.04526$), three times as many nodes must be created.

During the search process, the first complete allocation isn't evaluated until many nodes have been created (e.g., in Table A.9, more than 300 nodes must be created before the first leaf-node is evaluated). A useful property, seen in both tables, is that the bracket can be reduced to near zero without increasing the solution time. This means that the solution is guaranteed to be nearly optimal, but that most of the time needed to guarantee its optimality is avoided. Note that this behavior can be improved by seeding the branch-and-bound solution algorithm with any solution found by a quick method (e.g., a depth-first search with an optimality bracket of zero). By itself, this reduces the number of nodes inserted in the active set, an optimisation yielding a modest reduction in time, that depends on the STAP and a non-zero bracket.

6.1.6 Hardware Symmetry-Based Search-Space Reduction

Hardware symmetry-based search-space reduction techniques were introduced in Section 4.4. The following two tables, 6.3 and 6.4, illustrate the effectiveness of these techniques for the L1P4T6 and L1P4T10 examples. Search-space reductions based on dynamic and static T^{REF} modifications are compared with the "none" entry which corresponds to no search-space reductions. In each case, $L^s = 1$ and L^f are used.

mode	T^{REF} changes	nodes	time	cost	$\lambda(a)$
tadd	dynamic	18	1.4	6.973	0.1434
	static	18	1.4	6.973	0.1434
	none	356	26.0	6.973	0.1434
tsb	dynamic	18	0.9	6.0	0.1397
	static	18	0.9	6.0	0.1397
	none	204	7.4	6.0	0.1397

Table 6.3: Search-Space Reduction for L1P4T6 using increasing T^{ORDER}

mode	T^{REF} changes	nodes	time	cost	$\lambda(a)$
tadd	dynamic	142	13.7	17.994	0.05558
	static	142	13.5	17.994	0.05558
	none	3240	321.6	17.994	0.05558
tsb	dynamic	90	5.5	14.0	0.05540
	static	90	5.3	14.0	0.05540
	none	1848	121.5	14.0	0.05540

Table 6.4: Search-Space Reduction for L1P4T10 using increasing T^{ORDER}

All the allocations produced have the same cost, but an order of magnitude extra time and node creations are required when no T^{PREF} modifications are made. As predicted, the same allocations always resulted from the static and dynamic modifications, but no modification produced shuffled (symmetric) allocations.

Interestingly, no significant differences are evident in the dynamic and static modification results for these examples. This is due solely to having chosen an appropriate, increasing, T^{ORDER} (i.e., $T^{ORDER} = [1, 2, \dots, |T|]$). The next two tables, 6.5 and 6.6, show the results from trying to allocate the same two examples with the opposite, decreasing, T^{ORDER} (i.e., $T^{ORDER} = [|T|, \dots, 2, 1]$).

mode	T^{PREF} changes	nodes	time	cost	$\lambda(a)$	nodes more
tadd	dynamic	96	7.3	6.973	0.1434	433%
	static	189	14.2	6.973	0.1434	950%
	none	2020	153.0	6.973	0.1434	467%
tsb	dynamic	66	2.9	6.0	0.1434	267%
	static	121	5.1	6.0	0.1434	572%
	none	1336	55.5	6.0	0.1434	555%

Table 6.5: Search-Space Reduction for L1P4T6 using decreasing T^{ORDER}

T^{PREF}						nodes
mode	changes	nodes	time	cost	$\lambda(a)$	more
tadd	dynamic	5124	518.0	17.994	0.05558	3508%
	static	>11997	>1226.1	[17.98, 18.01]	>0.05552	>8349%
	none	>12004	>1598.9	[17.94, 19.16]	>0.05220	>270%
tsb	dynamic	6296	473.8	14.0	0.05552	6896%
	static	>11997	>1076.6	[14.0, 16.0]	>0.05461	>13230%
	none	>12004	>1711.0	>13.75	?	>550%

Table 6.6: Search-Space Reduction for L1P4T10 using decreasing T^{ORDER}

The column labelled "nodes more" is the percentage increase in nodes that were

created when using this opposite T^{ORDER} . In these tables, the savings of T^{PREF} modification on allocation time and node creations is quite high (the savings in time is comparable). The size of the L1P4T10 problem prevented this opposite T^{ORDER} from finding the optimal allocation for any method but dynamic modification. Notice that dynamic T^{PREF} modification now clearly outperforms static modification, both in time and node creations¹⁵. Those results having a greater than sign before the number of nodes and time fields denote situations where the default node expansion limit of 3000 was reached. Searching stops at this point, and the best results available are listed in the table. Where an upper-bound node has been evaluated, its $\lambda(a)$ indicates that the optimal λ is greater than this value. If no upper-bound node has been evaluated, then a question mark is in the $\lambda(a)$ field. Similarly, the cost field records the interval bounded by the current upper and lower bounds if both are available; otherwise, the optimal allocation cost is shown as greater than the lower-bound node's cost.

This example of having chosen an opposite T^{ORDER} to the appropriate one for this case is important because it demonstrates the robustness of the dynamic T^{PREF} modification method, especially for those general problems in which it is not clear which T^{ORDER} will be the most effective (e.g., where one part of the processor-graph is communications bound, and another part is processor bound). Larger problems are expected to yield greater savings for both of these hardware symmetry-based search-space reduction methods.

6.1.7 Task Co-Allocation-Based Search-Space Reduction

Table 6.7 illustrates the various task-clustering techniques of Section 4.5 for the L2P3T10 example. In each case, $L^s = 1$, and Lf and tsb are used. Dynamic T^{PREF} modification is used wherever it can be of help; otherwise static T^{PREF} modification is used, where appropriate.

In this table, note that all clustering techniques produced the same cost al-

¹⁵When static modification was able to produce an allocation, naturally, it had the same λ as those allocations produced by dynamic modification.

clustering heuristic	time	nodes	cost	a
sbp(2,1.4)	33.3	644	19	1213233233
sbp(3,1.4)	13.9	269	19	1232313123
csp(0.5,1.5)	31.1	612	19	1213233233
csp(0.6,1.8)	25.6	522	19	2123311333
csp(0.7,2.1)	25.6	522	19	2123311333
cbc(1.3,1.5)	25.6	522	19	2123311333
cbc(1.2,1.8)	25.6	522	19	2123311333
cbc(1.0,2.2)	18.8	402	19	2123331112
cbc(0.8,2.2)	9.4	207	19	2123331112
cbc(0.6,2.2)	9.3	207	19	2123331112
scat_bp	14.0	269	19	1232313123
clust_sp(0.01)	33.5	648	19	1213233233
scat_bp(clust_sp(0.01))	13.9	269	19	1232313123
none	33.5	648	19	1213233233

Table 6.7: Task Clustering Results for L2P3T10, Lf and tsb

locations, but that effective techniques result in a 50% savings in time and node creations. Larger problems are expected to yield greater clustering and hence savings. If the parameters for the heuristics are further liberalised, solutions with higher costs will be generated due to excessive clustering. Clustering techniques are further illustrated in Section 6.5, for Ma's example task-graph.

6.1.8 Task-Graph Reduction

For the same L2P3T10 example and clustering heuristics illustrated in Table 6.7, Table 6.8 illustrates the effectiveness of the task-graph reduction techniques of Section 4.6. In each case, $L^s = 1$, and L_f and tsb are used. Dynamic T^{PREF} modification is used wherever it can be of help; otherwise static T^{PREF} modification is used, where appropriate.

Again, all the clustering techniques produced the same cost allocation, but the most effective clustering techniques now result in an extra order of magnitude reduction in time and node creations. Larger problems are expected to yield greater savings. Task-graph reduction techniques are further illustrated in Section 6.5, for Ma's example task-graph.

6.1.9 Summary of Recommendations

Experience with the above stochastic loop examples allows us to derive several recommendations on the choice of allocation search parameters. These recommendations are summarised below:

- the *tsb* and *tadd* cost functions of subsections 5.3.4 and 5.3.2, respectively, gave the best results in terms of allocation throughput and time and node creations required to reach a solution (*tadd* is more suitable where optimal λ allocations are desired even at the expense of extra search time; *tsb* uses less restrictive assumptions and yet produces good allocations more quickly than *tadd*),

clustering heuristic	time	nodes	cost	savings (%)	
				time	nodes
sbp(2.1.4)	33.3	644	19	0	0
sbp(3.1.4)	13.9	269	19	0	0
csp(0.5,1.5)	29.1	594	19	6.4	2.9
csp(0.6,1.8)	16.5	378	19	35.5	27.6
csp(0.7,2.1)	16.5	378	19	35.5	27.6
cbc(1.3,1.5)	16.5	378	19	35.5	27.6
cbc(1.2,1.8)	16.5	378	19	35.5	27.6
cbc(1.0,2.2)	6.8	169	19	63.8	58.0
cbc(0.8,2.2)	1.4	37	19	85.1	82.1
cbc(0.6,2.2)	1.4	37	19	85.1	82.1
scat_bp	14.0	269	19	0	0
clust_sp(0.01)	33.5	648	19	0	0
scat_bp(clust_sp(0.01))	13.9	269	19	0	0
none	33.5	216	19	0	0

Table 6.8: Task-Graph Reduction Results for L2P3T10 and *tsb Lf*

- dynamic T^{PREP} modification should always be used (the allocator reduces its strength to static or off automatically as the STAP warrants).
- the LL search-mode should always be used.
- if T^{COAL} clustering heuristics are applied to a STAP, task-graph reduction should be applied, and
- if optimal allocations are not required, then setting the bracket to 0.01 reduces the effort required to find a solution, and introduces only an at most 1% increase in allocation cost.

6.2 Experience Using Nested Task-Graphs

This section summarizes our experiences with allocating NTGs (i.e., problems in which path-length bounds are computed). Two example task-graphs are described. The first is a small one, for which our results are compared with simulation. The second is a larger NTG for which clustering techniques were required. An extensive family of NTGs was also studied to see the effects of problem size; these examples are reported in the next section.

6.2.1 Simulation Example

For task-graphs which strongly violates product-form assumptions, such as SJ task-graphs, throughput bounds are the only possible evaluation technique. In this section, a pre-existing simulation study of a SJ task-graph is examined to see how the bound values compare to simulated throughputs.

Figure 6.12 shows a SJ NTG with six tasks. The tasks were allocated heuristically and the resulting system was simulated for one, five and ten tokens in the system. The following parameters describe the workloads of the task-graph:

Figure 6.12: Example SJ NTG for Simulation

$$b^L = \begin{bmatrix} 0 & 0 & 5.0 & 0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.0 & 2.0 & 0.1 \\ 0 & 0 & 0 & 0 & 0 & 4.0 \\ 0 & 0 & 0 & 0 & 0 & 4.0 \\ 0.1 & 0.1 & 0.1 & 0 & 0 & 0 \end{bmatrix}$$

$$b^P = [10 \ 5 \ 20 \ 12 \ 18 \ 5]$$

In addition, T^{COAL} contains no zero or one entries and so does not restrict the allocation search.

Comparison with Simulation

Simulation was conducted for the fixed allocation described by the following T^{PREF} :

$$T^{PREF} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The hardware had $c^L = [2, 5]$ and $c^P = [10, 5, 5]$. Communication between P_1 and P_2 used L_1 and communication between P_2 and P_3 used L_2 but no communication was possible between P_1 and P_3 . Results for 1, 5 and 10 tokens in the system (N) are in Table 6.9, as are $\hat{\lambda}(a)$ values from our *tspb* bound.

$\lambda(a)$ has four columns, representing the workloads used in the simulations. 'Exponential' means that all the workloads in a simulation were the means of expo-

		$\lambda(a)$				$\bar{\lambda}(a)$
		uniform				
N	a	exponential	constant	$\pm 10\%$	$\pm 50\%$	$tspb$
1	132132	0.080	0.083	0.082	0.078	0.105
5	132132	0.161	0.215	0.204	0.199	0.200
10	132132	0.193	0.223	0.201	0.197	0.200

Table 6.9: Simulation Results and $\bar{\lambda}(a)$ for Different Workloads and *N*

nentially distributed random variables. 'Constant' means that deterministic workloads were used. ' $\pm 10\%$ ' and ' $\pm 50\%$ ' means that workload samples were drawn from uniform distributions, having their limits 10% or 50% above and below the mean values, respectively.

Simulation results had 20% confidence intervals [74] and hence sometimes exceeded the *tspb* $\bar{\lambda}(a)$ values. The confidence intervals inherent in simulation clearly make simulation a poor cost function for allocation purposes.

Improvement via Allocation

To see what sort of improvement optimal allocation could provide, these results are compared with our *tspb* path length LP bounds in Table 6.10. The column labelled "improvement over fixed $\bar{\lambda}(a)$ " shows the percent improvement in $\lambda(a)$ over the heuristic manual allocation.

<i>N</i>	allocator results		improvement over fixed $\bar{\lambda}(a)$
	<i>a</i>	$\bar{\lambda}(a)$	
1	211111	0.177	69%
5	112111	0.235	17%
10	112111	0.235	17%

Table 6.10: Comparison with Allocator Results

In this case, selecting allocations via the allocator produced better allocations than those arrived at heuristically. In addition, the allocation search provides assurance that the resulting allocation has an optimal *tspb* cost. Notice that the al-

locator does not produce balanced allocations, but rather only balances loads when a balanced load produces the highest allocation $\bar{\lambda}(a)$ value.

Allocation Sensitivity

As a further example, consider what happens when P_1 fails and is replaced by a processor with $c_1^P = 3$ (vs. the previous $c_1^P = 10$). How should the allocation change? The answer is not intuitively or heuristically obvious. Keeping the same allocation yields lower $\bar{\lambda}$ values (0.088 vs. 0.105 before, for $N = 1$). The allocator finds two allocations ($a = 223332$ and $a = 312132$ for $N = 1$ and $N \geq 5$, respectively) both with $\bar{\lambda}$ values at least 10% higher than those of the fixed allocation (although still lower than the $\bar{\lambda}$ values in Table 6.10). This example of a changed allocation following a hardware change illustrates again the benefits of optimal allocation over hand-tuned or heuristic allocations.

6.2.2 DSP Example

This digital signal processing STAP was adapted from an industrial-scale example digital processing system [49]. The system has 41 tasks and an NTG structure illustrated in Figure 6.13. The processor-graph onto which this task-graph is allocated is given in Figure 6.14, along with the hardware architecture and l matrix.

The number of tasks for this example requires that we present b^P and b^L differently than in our earlier examples. b^P will be presented as four subvectors of length 9 and one of length 5. b^L will be presented as deriving from a set of triples of the form: (from-task to-task $b_{from-task, to-task}^L$).

$$b^P[1 : 9] = [0.0, 0.17024, 0.06272, 0.22784, 0.4096, 0.17024, 0.06272, 0.22784, 0.4096]$$

$$b^P[10 : 18] = [0.17024, 0.06272, 0.22784, 0.4096, 0.0, 0.0, 0.02224, 0.00614, 0.00123]$$

$$b^P[19 : 27] = [0.09968, 0.02128, 0.0112, 0.00307, 0.00031, 0.00028, 0.02128, 0.0112, 0.00307]$$

$$b^P[28 : 36] = [0.00031, 0.00028, 0.02128, 0.0112, 0.00307, 0.00031, 0.00028, 0.02128, 0.0112]$$

$$b^P[37 : 41] = [0.00307, 0.00031, 0.00028, 0.0, 0.0]$$

Figure 6.13: DSP Example SJ Nested Task-Graph

Figure 6.14: L2P4 Processor-Graph (a), Hardware Architecture (b), and l matrix (c)

$b^L \Leftarrow \{(2\ 3\ 2048), (3\ 4\ 2048), (4\ 5\ 512), (5\ 14\ 32768), (6\ 7\ 2048), (7\ 8\ 2048), (8\ 9\ 512), (9\ 14\ 32768), (10\ 11\ 2048), (11\ 12\ 2048), (12\ 13\ 512), (13\ 14\ 32768), (16\ 17\ 16384), (17\ 18\ 16384), (19\ 20\ 16384), (20\ 21\ 32768), (21\ 22\ 16384), (22\ 23\ 1024), (23\ 24\ 1024), (19\ 25\ 16384), (25\ 26\ 32768), (26\ 27\ 16384), (27\ 28\ 1024), (28\ 29\ 1024), (19\ 30\ 16384), (30\ 31\ 32768), (31\ 32\ 16384), (32\ 33\ 1024), (33\ 34\ 1024), (19\ 35\ 16384), (35\ 36\ 32768), (36\ 37\ 16384), (37\ 38\ 1024), (38\ 39\ 1024)\}.$

The hardware has $c^L = [1000000, 500000]$ (bytes/second) and $c^P = [1.5, 1, 1, 1.5]$. This is the largest system the present allocation techniques have been applied on. A range of T^{COAL} heuristics were applied to this problem, to give an idea of the capabilities of the allocator. The results of these trials are presented in Table 6.11. The manual heuristics (manual(5) ... manual(15)) were manually input based on partitioning the task-graph into that number of regions (the branches of the split-joins were targeted for clustering). We see that an optimal allocation has a cost¹⁶ greater than 0.747 and less than 0.807. Several heuristics gave results within 10% of the optimal allocation. The heuristics finding these best allocations required as little as 64.5 seconds and 203 node creations. Some of the heuristics did not succeed in finding an optimal allocation. Where the cbc heuristic had its p_2 , the total size of the clustered task, increased beyond 6, the cost of the solution allocations began to rise, due to excessive clustering. Surprisingly, the manual heuristics each produced worse allocations than several of the cbc allocations.

Using no heuristic ('none' in the table) resulted in the default node expansion limit of 3000 being reached before a solution was found. When the cbc heuristic's

¹⁶recall that for $tspb$, cost is $\bar{\lambda}(a)^{-1}$.

p_2 was set to four or less, it also was not able to find a satisfactory allocation due to the lack of sufficient reduction in the size of the STAP.

clustering heuristic	time	nodes	cost
cbc(.1,6)	64.5	203	.807
cbc(.4,12)	68.0	207	.807
cbc(.4,6)	67.9	211	.807
cbc(.01,6)	73.6	245	.807
cbc(.03,6)	73.8	245	.807
manual(10)	68.3	253	.85
manual(15)	70.9	262	.85
cbc(.1,12)	104.1	340	.87
cbc(.03,12)	129.8	424	.87
cbc(.01,12)	129.9	424	.87
manual(5)	37.2	126	.961
manual(8)	37.4	128	.961
cbc(.1,24)	67.8	193	1.0
cbc(.01,24)	67.4	227	1.002
cbc(.03,24)	67.4	227	1.002
cbc(.2,4)	>1064	>4534	>.747
cbc(.7,2.1)	>1328	>4602	>.747
cbc(.6,2.2)	>1023	>5161	>.747
none	>1184	>6190	>.747

Table 6.11: DSP Example Task Clustering Results for *tspb Ll*

6.3 The Effects of Problem Size

To know what size of STAP these techniques will be able to allocate effectively, two sets of experiments were run: one set for the stochastic loop GTGs and the other set for a family of NTGs. Both of these experiment sets varied the size of

task-graph that was being allocated onto a fixed size processor-graph ($|P| < |T|$)¹⁷. This section describes results from each of these experiment sets.

6.3.1 Problem Size: Stochastic Loop

The same stochastic loop example already examined is now scaled over the range of six to twenty tasks, and the complexity of the resulting allocation problems are compared in terms of the nodes created and the time required. Figures 6.15 and 6.16 show these measures as the L1P3 STAP is scaled in size. Although this example can not be generalized, it appears that two of the costs plotted (*ts* and *tall*) grow exponentially as the problem is scaled (Note the exponential vertical scales in these figures. The levelling off at about 8000 nodes created is caused by the default node expansion limit of 3000). The other two costs appear to grow with "bumps," and a smaller exponential coefficient¹⁸. These bumps are also apparent for the *tspob* results in tables A.11 and A.12. The bumps appear to arise due to a non-linear growth in the number of allocations of similar cost at certain sizes of the task-graph.

6.3.2 Problem Size: A Family of Nested Task-Graphs

To study how the size of nested task-graphs affects the difficulty of their allocation, a family of related graphs of increasing size are developed, and the results of their allocation analysed.

¹⁷The other aspect of problem size is growth in the processor-graph. For at least partly homogeneous systems, the effects of such growth is restricted by the static and dynamic T^{PREF} modification techniques. Subsection 4.3.1 showed that the search-space for these sets of experiments grows quicker as processors are added than when tasks are added. Tables 6.14 through 6.16 show how changing the size of the processor-graph scales the solution time and nodes created for Ma's 18 task example. Although the search-space size may increase quickly, the part of that explored to reach a solution sometimes even shrinks as the problem size increases (because an "easier" part of the search-space is being examined).

¹⁸the *tsb* results have the bumps growing exponentially, while the valleys appear to have a decreasing slope, reflecting a polynomial increase for some of the problems.

Figure 6.15: Nodes Created vs. $|T|$ for L1P3 using Ll

Figure 6.16: Time to Solution vs. $|T|$ for L1P3 using Ll

The Family

The family originated in four generic NTG STAPs, which were manually refined across the range of 6 to 14 tasks, in steps with two additional tasks per step. The refinement was done both by decomposing a task into smaller sub-tasks with a nested structure, and then clustering sequentially in various ways, as a partitioning front-end to the allocator may do.

These four generic NTGs and the five sizes of each gives twenty basic allocation experiments. Each of these twenty was converted into twelve related allocation experiments to ensure that a wider range of task-graph structures and parameters were exercised.

The twelve related STAPs for each of the twenty arise from the use of all combinations of two alternate L^s factors and three alternative sets of parameters found by mirroring¹⁹. Each of these STAPs was allocated onto the same processor-graph as was used for the DSP example in Subsection 6.2.2. No T^{COAL} or T^{PREF} constraints were introduced, and no clustering was applied.

The Results

A total of 240 STAPs were allocated as twenty related groups of twelve problems. The mean number of nodes created for each of these twelve problem groups is shown in Figure 6.17. The standard deviation of this number for the problems within the group is shown in Figure 6.19. Similarly, the mean and standard deviation of the time required for this allocation process for a group is shown in figures 6.18 and 6.20. In all cases, time appears to be proportional to the number of nodes created.

¹⁹When a STAP is mirrored, the non-zero w^P and w^L values for that STAP are mirrored either individually or in combination about the mean non-zero values of those parameters. Workload parameters above the mean non-zero values are geometrically reflected below the mean, while parameters below the mean are arithmetically reflected above the mean.

Figure 6.17: Mean Nodes Created vs. $|T|$ for the Family Examples

Figure 6.18: Mean Time to Solution vs. $|T|$ for the Family Examples

Figure 6.19: Std. Dev. Nodes Created vs. $|T|$ for the Family Examples

Figure 6.20: Std. Dev. Time to Solution vs. $|T|$ for the Family Examples

As can be expected due to the mirroring process and the different L^s values, the standard deviations for each of the problem groups is quite high. The growth of mean time and nodes appears to depend heavily on the task-graph structure, with problem group 2 experiencing apparently super-exponential growth at first and then leveling off. The other groups appear to follow an exponential growth pattern for both solution time and nodes created.

6.4 Comparison with Stone's Method

As a comparison with a well-known allocation method in the literature, we took the STAP defined by Figure 1 in [70], modified it to fit the format of our STAPs²⁰, and solved it using both Stone's method and our method. Stone's method was discussed in Section 3.1.

This STAP allocates six tasks onto two processors. In its original form, some of the tasks executed quicker on P_1 than on P_2 while other tasks executed quicker on P_2 than on P_1 . Both Stone's original processing workload definitions, and a version modified so that each processor could complete the tasks at the same speed²¹, are indicated in Table 6.12.

task	Original		Modified
	P_1 time	P_2 time	b_i^P
T_1	5	10	7.5
T_2	2	∞	2
T_3	1	4	4
T_4	6	3	4.5
T_5	5	2	3.5
T_6	∞	4	4

Table 6.12: Original and Modified Stone Example Processing Workloads

²⁰We require each task to present the same workload to all processors it may be allocated onto.

²¹Note that Stone refers to the tasks as modules A, B, C, D, E, and F, whereas we refer to them as T_1 , T_2 , T_3 , T_4 , T_5 and T_6 , respectively

While Stone's P_1 and P_2 times for each task have arbitrary ratios, we use a linear ratio between the P_1 and P_2 times of each task (in this case 1:1) thereby conserving the total task processing workload²². Stone's two infinite processing workloads ensure that T_2 and T_6 aren't allocated to P_1 and P_2 , respectively; we achieve this via task to processor allocation infeasibility constraints (setting two T^{REF} entries to 0).

Figure 6.21: Stone Example Task-Graph

Stone's definition of the STAP gives no task-graph, or any other model for the tasks' execution. The task-graph of Figure 6.21 was constructed to make the overall communication workload of Stone's original model equal to that of our revised version. This task-graph isn't nested because of T_6 , a task that receives tokens, but that neither sends tokens nor termination notifications. The directed arcs and probabilities on Figure 6.21 constitute an execution model for the application. Using this execution model, we will allocate the tasks.

Stone's definition also didn't indicate the number of communication links (instead, just imposing a homogeneous communications cost). We assumed that one bidirectional link connects the two processors²³.

Summarising the STAP definition, we have determined b^L and T^{REF} to be the following:

²²This is required by c_j^P , as described in Section 2.1 and ignores any architectural differences making processing speeds dependent on the task being processed.

²³Stone's model does not consider the effects of more than one communication link

$$b^L = \begin{bmatrix} 0 & 0 & 4 & 0 & 0 & 12 \\ 6 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 11 & 0 \\ 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$T^{PREF} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

In addition, T^{COAL} contains neither zeroes nor ones and so does not restrict the allocation search.

Using these definitions, and the tsb lower-bound cost function, we find an optimal allocation as $a = 211112$, while applying Stone's method to this STAP gives the same allocation²⁴. Modifying b_3^P from 4 to 20 in Table 6.12 and solving again using Stone's method gives $a = 211112$, the same allocation as before, at a cost using tsb of 30. Solving this modified STAP using tsb gives $a = 211212$ at a cost of only 27, more that a 10% improvement over the result that Stone obtains.

The approach of this thesis gives broadly similar results to Stones's method, but can make use of more detailed information to get improved results. In particular, we make use of an execution model (i.e., a task-graph and a processor-graph). For STAPs such as Stone's, we constructed such a model. The solutions were the same in our version of Stone's STAP, but for a modified version, we have seen that different solutions result, Stone's having a higher cost.

The execution model construction process that was used to generate our version of Stone's STAP is unrealistic and makes it clear that one must model application execution (e.g., using task-graphs and processor-graphs) in order to realistically evaluate the performance impacts of allocation decisions. While we constructed an execution model from Stone's workload, normally the user already has an execution model for his application and is able to use it directly in our method. Stone's information could then be extracted from this execution model.

²⁴Stone's original STAP (having arbitrary speed ratios) had a solution of $a = 111112$: in which T_1 was allocated to P_1 because P_1 had half the execution cost of P_2 for T_1 .

6.5 Experience with Ma's Example

As with Stone's work described above, we have adapted²⁵ Ma's example published in [48]. The correspondence between Ma's task numbers and our task numbers is given in Table 6.13, and the GTG we constructed is given in Figure 6.22.

our's	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Ma's	22	1	2	4	8	3	5	6	9	10	16	17	7	18	13	19	14	23

Table 6.13: Correspondence between Ma's Task Numbers and Our Task Numbers

Figure 6.22: Ma Example Task-Graph

²⁵The adaptations consisted of (1) removing the independent second "interceptor" class of tasks, (2) assigning processing and communication workloads to the tasks, and (3) consecutively numbering them (several of Ma's tasks were unimplemented or in the interceptor class). Ma's tasks were incompletely specified, (e.g., machine language instructions were used to specify some task workloads) and so we generated our illustrative b^P and b^L values.

The following parameters describe the workloads of Ma's task-graph:

$$b^P = \begin{bmatrix} 0 & 1 & 1 & 0.5 & 0.2 & 0.8 & 1.1 & 2 & 0.4 & 0.1 & 1.2 & 0.4 & 0.9 & 0.1 & 0.1 & 0.2 & 0.3 & 2 & 2 \end{bmatrix}$$

$$b^L = \begin{bmatrix} 0 & 10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 9 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 8 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 20 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 15 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The standard processor-graph architectures of Subsection 6.1.1 are used. No T^{COAL} or T^{PREF} constraints were used, making the problem as difficult as possible for the allocator. Optimising the throughput bounds for one token approximates Ma's response-time-like minimisation.

Ma's example is ideal to further illustrate the results of applying the present solution techniques. Tables 6.14, 6.15 and 6.16 compare the performance of three clustering techniques on six hardware configurations (a bus having an L^s of either 0.1 or one, and either two, three or four processors). Recall that for tsb , cost is $\Lambda(a)^{-1}$.

In these tables, the entries in the column labelled modifier have three meanings, defined as follows:

L^s	clustering heuristic	modifier	time	nodes	cost	a
0.1	csp(.7,2.1)	none	2.8	45	6.6	221221222122221211
0.1	csp(.7,2.1)	imp	3.5	43	6.6	221221222122221211
0.1	csp(.7,2.1)	red	0.7	25	6.6	221221222122221211
0.1	cbc(.6,2.2)	none	6.2	119	5.0	221221222222221211
0.1	cbc(.6,2.2)	imp	8.6	130	5.0	221221222222221211
0.1	cbc(.6,2.2)	red	2.2	51	5.0	221221222222221211
0.1	cbc(.4,12)	none	3.1	58	5.0	221221222222221211
0.1	cbc(.4,12)	imp	1.6	22	5.0	221221222222221211
0.1	cbc(.4,12)	red	0.2	9	5.0	112112111111112122
0.1	none	none	7.8	147	5.0	221221222222221211
1.0	csp(.7,2.1)	none	6.4	100	3.84	221221122122221212
1.0	csp(.7,2.1)	imp	8.3	91	3.84	221221122122221212
1.0	csp(.7,2.1)	red	1.7	53	3.84	221221122122221212
1.0	cbc(.6,2.2)	none	47.0	575	3.84	121211222111211112
1.0	cbc(.6,2.2)	imp	33.7	381	3.84	121211222111211112
1.0	cbc(.6,2.2)	red	17.2	309	3.84	221121111222121212
1.0	cbc(.4,12)	none	3.1	58	5.0	221221222222221211
1.0	cbc(.4,12)	imp	1.7	22	5.0	221221222222221211
1.0	cbc(.4,12)	red	0.2	9	5.0	112112111111112122
1.0	none	none	619.0	5113	3.84	111122221221211212

Table 6.14: Modified Ma Results for $|P| = 2$ and $tsb\ Ll$

L^3	clustering heuristic	modifier	time	nodes	cost	a
0.1	csp(.7,2.1)	none	3.6	51	6.6	221221222122221211
0.1	csp(.7,2.1)	imp	4.8	49	6.6	221221222122221211
0.1	csp(.7,2.1)	red	1.1	32	6.6	221221222122221211
0.1	cbe(.6,2.2)	none	19.1	317	5.0	221221222222221211
0.1	cbe(.6,2.2)	imp	27.4	341	5.0	221221222222221211
0.1	cbe(.6,2.2)	red	6.3	115	5.0	221221222222221211
0.1	cbe(.4,12)	none	6.1	94	5.0	221221222222221211
0.1	cbe(.4,12)	imp	2.7	30	5.0	221221222222221211
0.1	cbe(.4,12)	red	0.3	12	5.0	112112111111112122
0.1	none	none	25.5	431	5.0	221221222222221211
1.0	csp(.7,2.1)	none	7.2	106	2.81	221223222122221213
1.0	csp(.7,2.1)	imp	10.2	95	2.81	221223222122221213
1.0	csp(.7,2.1)	red	2.2	56	2.81	221223222122221213
1.0	cbe(.6,2.2)	none	78.8	1102	2.71	222232223331231313
1.0	cbe(.6,2.2)	imp	48.2	581	2.71	222212223333231313
1.0	cbe(.6,2.2)	red	19.2	324	2.71	222212223311211113
1.0	cbe(.4,12)	none	5.9	90	5.0	221231223333231311
1.0	cbe(.4,12)	imp	2.4	26	5.0	221231223333231311
1.0	cbe(.4,12)	red	0.4	12	5.0	11233233333332322
1.0	none	none	192.5	2270	2.61	222223222211211113

Table 6.15: Modified Ma Results for $|P| = 3$ and *tsb Ll*

L^s	clustering heuristic	modifier	time	nodes	cost	a
0.1	csp(.7,2.1)	none	4.0	51	6.6	221221222122221211
0.1	csp(.7,2.1)	imp	5.2	49	6.6	221221222122221211
0.1	csp(.7,2.1)	red	1.3	32	6.6	221221222122221211
0.1	cbc(.6,2.2)	none	27.8	418	5.0	221221222222221211
0.1	cbc(.6,2.2)	imp	39.4	445	5.0	221221222222221211
0.1	cbc(.6,2.2)	red	8.4	133	5.0	221221222222221211
0.1	cbc(.4,12)	none	7.3	99	5.0	221221222222221211
0.1	cbc(.4,12)	imp	2.9	30	5.0	221221222222221211
0.1	cbc(.4,12)	red	0.4	12	5.0	112112111111112122
0.1	none	none	10.1	592	5.0	112112111111112122
1.0	csp(.7,2.1)	none	10.8	138	2.81	221223222122221213
1.0	csp(.7,2.1)	imp	13.2	109	2.81	221223222122221211
1.0	csp(.7,2.1)	red	3.2	70	2.81	221223222122221214
1.0	cbc(.6,2.2)	none	147.5	1737	2.36	223223222233231311
1.0	cbc(.6,2.2)	imp	89.1	908	2.36	223223222241241411
1.0	cbc(.6,2.2)	red	62.8	793	2.36	222422442244441413
1.0	cbc(.4,12)	none	7.2	96	5.0	221221222233231311
1.0	cbc(.4,12)	imp	2.8	27	5.0	221231223344241411
1.0	cbc(.4,12)	red	0.5	14	5.0	112432443344442422
1.0	none	none	290.6	3082	2.36	223223222233231311

Table 6.16: d Ma Results for $|P| = 4$ and $tsb Ll$

imp: after clustering, T^{ORDER} is rearranged, or improved, by sequencing in T^{ORDER} the tasks that have been clustered. All tasks in a cluster immediately follow the task in the cluster that appears first in the original T^{ORDER} .

red: after clustering, the STAP is reduced – as described in Section 4.5 – and the solution to this reduced STAP is mapped back to the original STAP (also as described in Section 4.5).

none: neither T^{ORDER} improvement nor STAP reduction are used. This is the base case used in much of Section 6.1. It allows the effectiveness of the imp and red modifiers to be gauged.

As would be expected from the stochastic loop STAPs discussed in Section 6.1, increasing L' made the STAP more difficult to solve. The Lf search-mode was again found to be slower than Ll and so is not shown. The three clustering heuristics shown were quite effective: in some cases reducing the time to solve the STAP by two orders of magnitude, with no degradation in the solution quality. The imp modifier (having a rearranged, or improved, T^{ORDER}) often gave better results than none, and the red modifier (reducing the STAP and mapping the reduced solution back to the original STAP) always outperformed the imp modifier. That the imp modifier often required more node creations and time than the none modifier can be explained in that it shifts those tasks to be co-allocated sequentially into T^{ORDER} . Naturally, this avoids exploring sections of the search tree that cannot lead to complete allocation. Counter-balancing this reduction is the fact that the cost bound itself often prevents the exploration of those options anyway. The extra tasks shifted up in T^{ORDER} when the imp modifier is used are tasks that would normally have low impact on the cost bound, but which now must be evaluated. On the balance, these results confirm the usefulness of the imp modifier since it doesn't dramatically increase solution time or nodes in any case and often reduces it²⁶.

²⁶for the *tsb* and most other cost techniques, both the imp and red modifiers are available; however, for NTG problems and path-length bounds, not used in these examples, only the imp modifier is allowed, since the NTG structure is lost during the STAP reduction process.

Ma's name	cost	a
A1	47.1	112123233131122323
A2	32.9	332322322331311112
A3	24.8	331321332331311112
A4	16.9	333123121221121211

Table 6.17: Modified Ma Results for Ma's allocations, using $|P| = 3$, $L^s = 1$ and tsb

In [48], Ma looked at the four allocations onto three processors listed in Table 6.17. As the costs for $L^s = 1$ in Table 6.15 show, none of his allocations are optimal for us (we have achieved a cost of 2.61 for no clustering heuristics and 2.81 via a range of heuristics). However, the STAP that Ma worked on is different than the one we generated (in which task workloads are different, only one class of tasks is modelled, the execution model for the tasks is different, and the tsb cost is being optimised, rather than minimising inter-processor communication subject to response time requirements). Nonetheless, we have a better system, since the response time is much faster.

As was shown by the tables, the selection of which clustering heuristics to use and the parameters for them affect the solution times for the reduced problems. The *cbc* heuristic gave the best results for slow link problems and the *csp* heuristic worked well for quick link problems, finding optimal or nearly optimal allocations²⁷. For an arbitrary STAP, the difficulty of predicting the proper heuristic to use means that several diverse heuristics should be used, combined with a non-zero bracket. The best result can be refined further by adapting the parameters to generate somewhat smaller clusters. In this way, good results can be obtained for large problems. Optimality is necessarily sacrificed by our clustering techniques. The non-zero bracket would thus not deteriorate the solution quality noticeably.

In conclusion, these results suggest that our approach to task allocation can produce the same results that Ma's approach can. For more general problems, our

²⁷Note that more than one clustering technique can be applied on the same problem to further reduce the effective number of tasks.

allocator should produce better results (higher application performance) than Ma's method because we model the application's execution.

6.6 Summary

This chapter has looked at several STAPs, both for GTGs and for NTGs. Various bounding and heuristic cost functions were used and compared on these problems. In particular, the consistent lower-bound costs of Section 5.3 were compared with the inconsistent lower-bound costs of Section 5.4, and it was seen that the consistent methods can handle larger problems than the inconsistent methods can.

The recommendations of Subsection 6.1.9 were successfully applied to a range of NTGs. As the problem size increases, clustering becomes important because of the often exponential growth in the difficulty of finding a solution. Although clustering gives only heuristic answers, a range of parameters and even manual clustering reduce the complexity of the allocation problem enough that good allocations can be found. Clustering the tasks having the biggest communication was seen to be helpful in simplifying the problem and yet yielded results close to optimal (where the optimal results could be found). Search-space reduction, based on hardware symmetry was found to simplify the problem dramatically where homogeneous processors are used. The 'load-balancing' lower-bound cost and 'bottleneck' lower-bound cost gave the best results, as did the least lower-bound last in first out search-mode. Setting the bracket to 0.01 significantly reduces the effort to find solutions, yet introduces an at most one percent increase in the cost of the allocation.

These results show the effectiveness of the bounding methods of Chapter 5 and the search techniques of Chapter 4. STAPs consisting of task-graphs ranging in size from six to more than forty tasks have been allocated onto their processor-graphs. The next chapter concludes this thesis and describes the rationale behind several possible extensions.

Chapter 7

Conclusions

The previous chapters have presented several methods for static task allocation and the results obtained using them. This chapter summarises the present work and suggests several areas for further research.

7.1 Summary

This thesis has examined the problem of static task allocation onto distributed computer systems. A difficult and realistic version of this static task allocation problem (STAP) has been introduced and analysed, in which:

- heterogeneous processors are allowed,
- heterogeneous communications links can be modelled, including unidirectional links,
- precedence constraints between the tasks are represented through the use of nested task-graphs, and
- constraints can be expressed for and against co-allocation of tasks and for and against allocation of tasks to processors.

A branch-and-bound search technique is used to select optimal or nearly optimal complete allocations based on the costs of the partial allocations from which they are derived.

Contrasting with performance functions, which are normally applied to complete allocations only, bounding functions are used by the branch-and-bound search to compare partial and complete allocations. The nested task-graph represents the execution of software and allows a path-length bounding function to be used in allocation. The structure of the graph has novel features, and a new technique has been described for the analysis of its path-length bounds. These path-length bounds, derived from the precedence constraints expressed by the nested task-graph, simplify and help to make practical the search for optimal or nearly optimal task allocations. Although the structure of the search-space is not altered by the use of a more accurate bounding function (such as path-length bounds), the amount of the search-space actually examined during the search for an optimal allocation is nevertheless reduced.

A range of performance-based bounding functions were developed and evaluated to determine which functions were the most effective. The 'load-balancing' and 'bottleneck' lower-bound functions gave the best results, with the 'bottleneck' lower-bound being recommended for use with nested task-graphs that contained split-join parallelism. Both of these bounding functions are based on the software execution model represented by the task-graph. 'Inconsistent' bounding functions, those in which one bounding function is used to quickly bound the cost of partial allocations and another is used to evaluate complete allocations, performed poorly as the size of the STAP was increased and are therefore not recommended. The 'load-balancing' and 'bottleneck' bounding functions recommended above are 'consistent,' using one evaluation technique for all allocations.

For several example STAPs, requiring that an optimal allocation be found required from two to four times the effort of finding an allocation that had a cost at most 1% higher than the optimal cost. This attribute is expected to be very useful for many applications of the allocator.

Careful study of several search strategies led to the selection of the least-lower-bound last-in-first-out strategy, used for most of the examples in this thesis. The order in which tasks are allocated was found to have a pronounced effect on the difficulty of the search. Problems that are communications-intensive are best han-

dled by allocating in order of decreasing communications workload while problems that are processing-intensive should be allocated in order of decreasing processing workload. Other task allocation orders, not studied in the present work, such as allocating tasks on a "critical path" of the task-graph first, may facilitate the search process.

In addition to the optimality and task allocation order, symmetry-based search-space reduction was evaluated as a way of controlling the complexity of the search. Search-space reduction, based on hardware symmetry was found to simplify the allocation problem dramatically where at least some of the processors were homogeneous, and their communications structure was equivalent. On a set of experiments with four homogeneous processors and a single bus, up to a 99% reduction in the difficulty of the search resulted, with no degradation in the quality of the solution. Two kinds of symmetry-based search-space reduction were examined: a static modification to the task-to-processor allocation preference matrix (T^{PREF}) and a more complicated dynamic modification. The effort associated with the dynamic modification was found to be effective, and the allocation itself may reduce the strength of this setting based on an initial analysis of the STAP. Hence, use of the dynamic setting is recommended.

The complexity of the allocation problem was evaluated for 248 task-graphs of between six and twenty tasks that were combined into five families of related allocation problems. In each of these families, the *NP*-complete nature of the problem was confirmed, even with the use of search-space reduction and task allocation ordering techniques. Interesting variations in complexity were found with increasing task-graph size (sometimes the difficulty of the allocation was even reduced as the size of the task-graph was increased).

As the size of the task-graph increases, task clustering techniques become essential to reduce the complexity of the allocation problem. Task-graphs in the range of up to roughly twenty tasks can be adequately allocated onto up to about five processors without the use of clustering techniques. Three clustering heuristics (and two complementary techniques in which tasks are scattered) were developed and compared on several large task allocation problems. Some combinations of these

3

of/de

3

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

heuristics were also evaluated, by comparing the costs of the allocations obtained and the difficulty required to obtain it. Good results were obtained for each of example on which task clustering was used; as long as the clusters were not allowed to grow too large, only a small (less than 10%, for good choices of techniques and parameters for the techniques) increase in the cost of the solution allocation resulted. Surprisingly, for one example, the heuristics were found to be superior to results obtained from clusters that were manually selected.

Comparison with several substantial task allocation examples in the literature has demonstrated that better allocations result from the more complete models of the present work.

The implementation of the methods described in chapters 4 and 5 is documented in a technical report [40]. This technical report also details the examples and results of Chapter 6 and Appendix B, including the configuration and use of the support routines and files.

This thesis has demonstrated the need not just to evaluate the performance of complete allocations, but also to bound it with functions for partial allocations that are well-adapted to the search algorithm. When combined with the well-tuned branch-and-bound optimisation search, the linear-programming bounding functions developed by this work have been shown to practically solve problems of statically allocating up to about fifty tasks onto distributed computers having less than roughly ten processors. Without the techniques developed for task ordering, search-space reduction, clustering and bounding the cost of partial allocations, the present STAP can be viewed as a large and impractically complex integer programming problem. These techniques reduce the amount of the search tree actually evaluated while maintaining good allocation quality.

7.2 Areas for Further Research

There are a number of areas where further research can be directly related to work in this thesis. Each of these areas is briefly introduced below.

Multi-Class Problems

The work reported here has dealt with a single class of tokens executing one application. Extension to the case of multiple classes of tokens executing multiple applications was straightforward, as described in sections 2.4 and 5.5. Although implemented, multi-class AMVA allocation and multi-class linear program based allocation are not discussed in this thesis. Such multi-class problems evaluate the throughput (or its capacity) for each class to find an allocation with the best overall performance. Additional work is required to evaluate the effectiveness on such multi-class problems of the approaches resulting from single class problems.

More Accurate Communications Workload Heuristics

All the bounding techniques of Chapter 5 ignore the unallocated communications workload. By ignoring some work that must be done, a lower-bound on each node's cost was found. In problems with a communications bottleneck, ignoring unallocated communications gives very loose cost bounds in the top part of the allocation tree, resulting in an "unfocussed" (and hence large) branch-and-bound search. More realistic lower-bounds could be placed on the unallocated communications work by taking into account the task allocation restrictions and the partial allocation.

Consider any partial (perhaps empty) allocation. Task allocation restrictions (i.e., T_{ij}^{COAL} and T_{ij}^{PREF}) combine with the presently allocated tasks to limit the allocation of the remaining tasks. Using these restrictions, the unallocated communications load could be given a lower-bound. For example, consider the partial allocation of three tasks to two processors (connected by one link) in which one task is already allocated to each of the processors. A lower-bound on the additional link communications needed for a complete allocation is then the minimum of the last task's communication requirements with the other two.

Consider the following two cases of ignored communication:

- communication between one allocated task and one unallocated task, and
- communication between two unallocated tasks.

Allocation restrictions could reduce the ignored communications for the first case.

The second case would require additional heuristics. While considering the first case substantially reduces the number of nodes created and has been implemented by the LP2 bound of Subsection 5.2.1, the second case would reduce the number of nodes created by considering inevitable communications etc. throughout the tree (not just near the leaves, depending on the T^{ORDER} chosen). Further research would be required to characterise when the extra time spent evaluating nodes would be repaid by evaluating fewer of them.

Searching Processor-Graphs

A straightforward extension of the allocation search already introduced would be to examine several alternative processor-graphs by building a branch-and-bound search on top of the allocator. Performance-based cost functions that identify bottlenecks would assist this search for a suitable processor-graph. Such bottlenecks would be corrected in the next processor-graph to be examined, helping to verify that the final configuration would be well utilised.

This extra level of search would use the branch-and-bound technique (with a different set of costing and branching rules), possibly requiring many task allocation searches. To reduce the problem of complexity and computational time requirements, intelligent processor-graph construction techniques would be required. For example, processor-graphs could be restricted to a single bus in computation bound problems, or to a maximum number of processors for communication bound problems.

More General Splits and Joins

While the task-graph developed can represent nested split-joins, many applications require more general inter-task synchronisation, representable using non-nested split-joins, in which the branches of a split-join may contain a join-task waiting for the completion of tasks not within the branch. This non-nestedness would require extensions to the algorithms of Appendix B.

Open Networks

This thesis has considered closed queueing networks, where no tokens enter

or leave the task-graph; several methods have been developed to maximize either the system's throughput or a bound on it. Open queueing networks, having token arrivals and departures, could be approximated by adding infinite servers to our processor-graphs, as Chapter 2 considered. Further work could improve upon this approach by extending our task-graphs and using cost functions that evaluate an allocation's response time, not its throughput.

Extending Nested Task-Graphs

NTGs, as we have defined them, are quite restrictive. More than one $SOP(G)$ and $EOP(G)$ point should be allowed for some new types of NTGs. The execution time of the NTG would then not only depend on the NTG entered, but also depend on where the token entered the NTG, complicating the mean path-length analysis but allowing additional applications to be modelled.

References

- [1] R. Agrawal and H. Jagadish, "Partitioning techniques for large-grained parallelism," *IEEE Transactions on Computers*, pp. 1627-1634, Dec. 1988.
- [2] M. Alford, "SREM at the age of eight; the distributed computing design system," *IEEE Transactions on Computers*, pp. 36-46, Apr. 1985.
- [3] F. Baccelli and Z. Liu, "On the execution of parallel programs on multiprocessor systems - a queueing theory approach," Tech. Rep. 833, INRIA-Sophia Antipolis, Apr. 1988.
- [4] Y. Bard, "Some extensions to multiclass queueing network analysis," in *4th Int. Symp. on Modeling and Perf. Eval. of Comp. Syst.*, (Vienna), pp. 51-62, Feb. 1979.
- [5] F. Baskett, K. Chandy, R. Muntz, and F. Palacios, "Open, closed and mixed networks of queues with different classes of customers," *Journal of the ACM*, pp. 248-260, Apr. 1975.
- [6] S. Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, pp. 207-214, Mar. 1981.
- [7] S. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Transactions on Software Engineering*, pp. 583-589, Nov. 1981.
- [8] S. Bokhari, "Partitioning problems in parallel, pipelined, and distributed computing," *IEEE Transactions on Computers*, pp. 48-57, Jan. 1988.

- [9] J. Buzen, "Analysis of system bottlenecks using a queueing network model," in *ACM-SIGOPS Workshop on System Performance Evaluation*, (New York), pp. 92-103, ACM, Apr. 1971.
- [10] J. Buzen, "Computational algorithms for closed queueing systems with exponential servers," *Communications of the ACM*, pp. 527-531, Sept. 1973.
- [11] D. Cheriton, "The V kernel: A software base for distributed systems," *IEEE Software*, pp. 19-42, Apr. 1984.
- [12] T. Chou and J. Abraham, "Load balancing in distributed systems," *IEEE Transactions on Software Engineering*, pp. 401-412, July 1982.
- [13] W. Chu, L. Holloway, M. Lan, and K. Efe, "Task allocation in distributed data processing," *IEEE Computer*, pp. 57-69, Nov. 1980.
- [14] W. Chu and L. Lan, "Task allocation and precedence relations for distributed real-time systems," *IEEE Transactions on Computers*, pp. 667-679, June 1987.
- [15] W. Chu and K. Leung, "Module replication and assignment for real-time distributed processing systems," *Proceedings of the IEEE*, pp. 547-562, May 1987.
- [16] W. Chu, C.-M. Sit, and K. Leung, "Task response time for real-time distributed systems with resource contentions," *IEEE Transactions on Software Engineering*, pp. 1076-1092, Oct. 1991.
- [17] Y. Dallery and K. Stecké, "On the optimal allocation of servers and workloads in closed queueing networks," *Operations Research*, pp. 694-703, July-Aug. 1990.
- [18] E. de Souza e Silva and M. Gerla, "Load balancing in distributed systems with multiple classes and site constraints," in *Proceedings of Performance 84*, pp. 17-33, Dec. 1984.
- [19] K. Doty, P. McEntire, and J. O'Reilly, "Task allocation in a distributed computer system," in *Proceedings of INFOCOM'82*, (Las Vegas), pp. 33-38, Mar. 1982.

- [20] L. Dowdy and D. Foster, "Comparative models of the file assignment problem," *ACM Computing Surveys*, pp. 287-313, June 1982.
- [21] D. Eager, E. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing," in *Proceedings of the 1988 ACM SIGMETRICS Conference*, pp. 63-72, May 1988.
- [22] D. Eager and K. Sevcik, "Performance bound hierarchies for queueing networks," *ACM Transactions on Computer Systems*, pp. 99-115, May 1983.
- [23] K. Eswaran, "Placement of records in a file and file allocation in a computer network," in *IFIP Conference Proceedings*, (Stockholm, Sweden), pp. 304-307, Aug. 1974.
- [24] E. Fernandez and T. Ngo, "Scheduling of task graphs into transputer networks," *Congressus Numerantium*, vol. pp. 79-88, 1987.
- [25] L. Ford and D. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, pp. 399-404, 1956.
- [26] J. Gaudiot, J. Pi, and M. Campbell, "Program graph allocation in distributed multicomputers," *Parallel Computing*, vol. 7, pp. 227-247, 1988.
- [27] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
- [28] K. Gordon and L. Dowdy, "The impact of certain parameter estimation errors in queueing network models," in *Proceedings of Performance 80*, pp. 3-9, 1980.
- [29] A. Goscinski, *Distributed Operating Systems: The Logical Design*. Addison-Wesley, 1991.
- [30] V. Gylys and J. Edwards, "Optimal partitioning of workload for distributed systems," in *Comcon Conference Proceedings*, pp. 353-357, Sept. 1976.
- [31] K. Haessig and C. Jenny, "Partitioning and allocating computational objects in distributed computer systems," in *INFORMATION PROCESSING 80* (S. Lavington, ed.), pp. 593-598, North Holland, 1980.

- [32] P. Heidelberger and K. Trivedi, "Queueing network models for parallel processing with asynchronous tasks," *IEEE Transactions on Computers*, pp. 1099-1109, Nov. 1982.
- [33] P. Jacobson and E. Lazowska, "A reduction technique for evaluating queueing networks with serialization delays," in *Proceedings of Performance '83*, pp. 45-59, 1983.
- [34] C. Jenny, "Process partitioning in distributed systems," in *Digest of Papers National Telecommunications Conference 77*, (Los Angeles), pp. 31:11 - 31:1-10, 1977.
- [35] C. Jenny and K. Kümmerle, "Distributed processing within an integrated circuit/packet-switching node," *IEEE Transactions on Computers*, pp. 1089-1100, Oct. 1976.
- [36] K. Kant, "Analysis and synthesis of generalized task graphs," in *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, (Dallas), pp. 570-577, Dec. 1990.
- [37] R. Karp and R. Miller, "Properties of a model for parallel computations: Determinancy, termination, queueing," *SIAM Journal on Applied Mathematics*, pp. 1390-1411, Nov. 1966.
- [38] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Transactions on Computers*, pp. 1023-1029, Nov. 1984.
- [39] F. Kaudel, "Comments on allocating programs containing branches and loops within a multiple processor system," *IEEE Transactions on Software Engineering*, pp. 471-472, Apr. 1990.
- [40] F. Kaudel, "Documentation of a task allocator," Tech. Rep. SCE-92-01, Carleton University, Jan. 1992.

- [41] W. Kohler and K. Steiglitz, "Enumerative and iterative computational approaches," in *Computer and Job-Shop Scheduling Theory* (E. Coffman, ed.), pp. 229-287, J. Wiley & Sons, 1976.
- [42] P. Krueger, T.-H. Lai, and V. Radiya, "Processor allocation vs. job scheduling on hypercube computers," in *Proceedings of the 1991 IEEE Conf. on Dist. Computing Systems*, (Arlington), pp. 394-401, May 1991.
- [43] M. Lavoie, "Task assignment in a DSP multiprocessor environment," Master's thesis, Carleton University, Sept. 1990.
- [44] H. Lee, M. Srinivasan, and C. Yano, "Characteristics of optimal workload allocation for closed queueing networks," *Performance Evaluation*, pp. 255-268, July 1991.
- [45] V. Lo, "Task assignment in distributed systems," in *Proceedings of the 4th International Conference on Distributed Computing Systems*, (San Francisco), pp. 30-39, May 1984.
- [46] T. Lowe, "Analysis of an information system model with transfer penalties," *IEEE Transactions on Computers*, pp. 469-480, May 1973.
- [47] P. Ma, "A model to solve timing-critical application problems in distributed computer systems," *IEEE Computer*, pp. 62-68, Jan. 1984.
- [48] P. Ma, E. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Transactions on Computers*, pp. 41-47, Jan. 1982.
- [49] G. Montforton, "A fast task allocator for multiprocessing systems," Master's thesis, Carleton University, July 1989.
- [50] T. Murata, R. Bhatia, and S. Shatz, "Markov chain reduction and analysis of GSPN models for task allocation in distributed systems," *Microelectron. Reliab.*, vol. 31, no. 4, pp. 727-746, 1991.
- [51] T. Ngo, "Optimal scheduling of processes for a network of transputers," Master's thesis, Florida Atlantic University, Dec. 1986.

- [52] M. Ojstersek and V. Zumer, "Automatic grain size determination for a macro dataflow real-time system," in *Proceedings of Euromicro '90 Workshop on Real Time*, (Horsholm, Denmark), pp. 122-128, June 1990.
- [53] I. Page, R. Jacob, and S. Chern, "Optimal algorithms for distributed resource allocation," Tech. Rep. UTDCS-37-90, University of Texas at Dallas, Dec. 1990.
- [54] D. Peng and K. Shin, "Modeling of concurrent task execution in a distributed system for real-time control," *IEEE Transactions on Computers*, pp. 500-516, Apr. 1987.
- [55] I. Philp, "Dynamic load balancing in distributed systems," in *Proceedings - 1990 Southeastcon*, (New Orleans), pp. 304-307, Apr. 1990.
- [56] C. Price, "The assignment of computational tasks among processors in a distributed system," in *Proceedings of the National Computing Conference*, pp. 291-296, 1981.
- [57] C. Price and S. Krishnaprasad, "Software allocation models for distributed computing systems," in *Proceedings of the 4th International Conference on Distributed Computing Systems*, (San Francisco), pp. 40-48, May 1984.
- [58] G. Rao, H. Stone, and T. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Transactions on Computers*, pp. 291-299, Apr. 1979.
- [59] M. Reiser, "Mean value analysis of queueing networks, a new look at an old problem," in *4th Int. Symp. on Modeling and Perf. Eval. of Comp. Syst.*, (Vienna), pp. 63-77, Feb. 1979.
- [60] M. Reiser, "A queueing network analysis of computer communications in networks with flow control," *IEEE Trans. on Communications*, pp. 1199-1209, Aug. 1979.
- [61] M. Reiser and S. Lavenberg, "Mean value analysis of closed multi-chain queueing networks," *Journal of the ACM*, pp. 313-332, Apr. 1980.

- [62] N. Resnick, "Network allocation using genetic algorithms." B.Eng. Project Report, Carleton University, 1989.
- [63] C. Shen and W. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Transactions on Computers*, pp. 197-203, Mar. 1985.
- [64] J. Sinclair, "Efficient computation of optimal assignments for distributed tasks," *Journal of Parallel and Distributed Computing*, vol. 4, pp. 342-362, 1987.
- [65] J. Sinclair, "Optimal assignments in broadcast networks," *IEEE Transactions on Computers*, pp. 521-531, May 1988.
- [66] H. Sips, "Task distribution on clustered parallel- or multiprocessor systems," in *Proceedings of the 4th International Conference on Distributed Computing Systems*, (San Francisco), pp. 126-130, May 1984.
- [67] C. Smith, *Performance Engineering of Software Systems*. Addison-Wesley, 1990.
- [68] J. Stankovic and I. Sidhu, "An adaptive bidding algorithm for processes, clusters and distributed groups," in *Proceedings of the 4th International Conference on Distributed Computing Systems*, (San Francisco), pp. 49-59, May 1984.
- [69] H. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Transactions on Software Engineering*, pp. 85-93, Jan. 1977.
- [70] H. Stone and S. Bokhari, "Control of distributed processes," *IEEE Computer*, pp. 97-106, July 1978.
- [71] A. Tanenbaum, R. van Ronsse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen, and G. van Rossum, "Experiences with the Amoeba distributed operating system," *Communications of the ACM*, pp. 46-63, Dec. 1990.
- [72] A. Tantawi and D. Towsley, "Optimal static load balancing in distributed computer systems," *Journal of the ACM*, pp. 445-465, Apr. 1985.

- [73] D. Towsley, "Allocating programs containing branches and loops within a multiple processor system," *IEEE Transactions on Software Engineering*, pp. 1018-1024, Oct. 1986.
- [74] V. Verma, "Multi-tasking multi-processor system performance modelling." Course Report, Carleton University, Mar. 1986.
- [75] C. Woodside and F. Kaudel, "A new method for static task allocation in distributed systems," Tech. Rep. SCE-86-8, Carleton University, Apr. 1986.
- [76] S. Wu and M. Liu, "A partition algorithm for parallel and distributed processing," in *Proceedings of the 1980 International Conference on Parallel Processing*, pp. 254-255, 1980.
- [77] H. Xin, Z. Hong, and C. Xiyao, "Heuristic software partitioning algorithms for distributed real-time applications," in *Proceedings of the 10th International Conference on Software Engineering*, (Singapore), pp. 116-121, Apr. 1988.

Appendix A

Detailed Task Allocation Experiment Results

This appendix provides the details for the results discussed in sections 6.1 and 6.3. Section A.1 contains tables that compare the performance of the allocation techniques for several stochastic loop examples. Section A.2 then presents several tables showing the tradeoff between guaranteed solution optimality and time needed to find a solution to two stochastic loop examples. Finally, Section A.3 contains results for a stochastic loop example as the number of tasks is scaled from six to twenty.

Note that none of these results use the task clustering or task-graph reduction techniques described in Chapter 4. The results are thus worst-case for the examples¹. By using these additional techniques, larger examples can be solved, as illustrated in Section 6.5.

Subsection 6.1.2 gives the definitions for each of the column headings (e.g., time) and mode (search-mode) abbreviations in the tables. Where an allocation was determined more than once (by allocating the same example several times), the average time is reported. Differences between these time values are normally less than 5%.

¹The hardware-symmetry-based search-space reduction techniques of Section 4.4 are used, wherever possible.

A.1 Comparison of Results

This section gives seven tables of results showing the effectiveness of the search and bounding methods of chapters 4 and 5 for several examples of the stochastic loop STAP defined in Subsection 6.1.1. Subsection 6.1.3 discusses these results.

Table A.1 reports our results for the L1P2T6 example with several L^s (link speed) values and modes (search-modes). The results in Table A.1 are plotted in figures 6.6 and 6.7.

Table A.2 reports our results for four other six task examples, showing the relative performance of six modes for these examples.

Table A.3 contains results for the L1P3T10 example with several L^s values and modes. The results in Table A.3 are illustrated by figures 6.8 and 6.9.

Table A.4 contains results for the L2P3T10 example with several L^s values and modes. The results in Table A.4 are plotted in figures 6.10 and 6.11.

Table A.5 reports our results for several other ten task examples. The results for L1P5T10 and *tslo* show what happens when the default allocator node expansion limit of 3000 is reached: searching stops and the current status is reported. The time and nodes fields are set to be greater than the current time and nodes created. $\lambda(a)$ is then the cost of the complete allocation having the greatest λ found so far and a is this allocation.

Table A.6 compares results for the L2P3T10 example for three L^s values and the *c*, *tsprob* and *tadd* modes.

A.2 Search Optimality Bracket

This section presents four tables comparing the solution search for different

L^s	mode	time	nodes	$\Lambda(a)$	a
10.0	tadd	1.3	25	0.086	112222
10.0	tall	1.4	27	0.086	112222
10.0	ts	1.0	27	0.086	112222
10.0	tsb	0.5	15	0.086	121222
1.0	tadd	1.3	25	0.08594	112222
1.0	tall	1.4	27	0.08594	112222
1.0	ts	1.0	27	0.08594	112222
1.0	tsb	0.5	15	0.08591	121222
0.3	tadd	1.3	25	0.08575	112222
0.3	tall	1.3	27	0.08575	112222
0.3	ts	1.0	27	0.08575	112222
0.3	tsb	0.5	15	0.08553	121222
0.1	tadd	1.3	25	0.08346	112222
0.1	tall	1.3	27	0.08346	112222
0.1	ts	1.0	27	0.08346	112222
0.1	tsb	0.5	15	0.0778	121221
0.03	tadd	1.4	27	0.06954	122111
0.03	tall	1.4	27	0.06954	122111
0.03	ts	0.9	27	0.06954	122111
0.03	tsb	0.7	25	0.06954	122111
0.01	tadd	1.3	25	0.04762	111111
0.01	tall	1.2	25	0.04762	111111
0.01	ts	0.8	25	0.04762	111111
0.01	tsb	0.7	25	0.04762	111111
0.001	tadd	0.6	11	0.04762	111111
0.001	tall	0.6	11	0.04762	111111
0.001	ts	0.4	11	0.04762	111111
0.001	tsb	0.3	11	0.04762	111111

Table A.1: Results for L1P2T6 using Ll

problem	mode	time	nodes	$\lambda(a)$	a
L2P2T6	tadd	3.0	50	0.08593	112222
	tall	3.2	54	0.08593	112222
	ts	2.1	54	0.08593	112222
	tsb	0.9	30	0.08586	212112
	tslo	4.9	126	0.08593	112222
	tslob	0.2	12	0.04762	222222
	tspo	1.9	54	0.08593	112222
	tspob	0.8	30	0.08586	212112
L1P3T6	tadd	1.0	15	0.11868	123321
	tall	1.4	21	0.11868	123321
	ts	1.4	27	0.11868	123321
	tsb	0.6	15	0.11868	123321
	tslo	9.0	185	0.11868	123321
	tslob	0.2	11	0.04762	111111
	tspo	1.3	27	0.11868	123321
	tspob	0.6	15	0.11868	123321
L2P3T6	tadd	3.1	43	0.1185	123321
	tall	4.3	61	0.1185	123321
	ts	4.2	79	0.1185	231132
	tsb	0.7	17	0.1185	231132
	tslo	30.6	549	0.1185	123321
	tslob	0.3	17	0.04762	222222
	tspo	4.1	79	0.1185	231132
	tspob	0.6	17	0.1185	231132
L1P4T6	tadd	1.4	18	0.1434	123443
	tall	1.4	18	0.1434	123443
	ts	1.3	22	0.1434	123443
	tsb	0.9	18	0.14045	123444
	tslo	15.6	261	0.1434	123443
	tslob	0.3	11	0.04762	111111
	tspo	1.2	22	0.1434	123443
	tspob	0.8	18	0.14045	123444

Table A.2: Results for L2P2T6, L1P3T6, L2P3T6 and L1P4T6 using $L1$

L^s	mode	time	nodes	$\lambda(a)$	a
10.0	tadd	29.0	365	0.04538	1213332221
10.0	tall	92.2	1082	0.04538	1213332221
10.0	ts	219.4	2765	0.04538	1213332221
10.0	tsb	11.3	218	0.04537	1213233233
1.0	tadd	29.2	362	0.0453	1213332221
1.0	tall	93.6	1082	0.0453	1213332221
1.0	ts	219.5	2765	0.0453	1213332221
1.0	tsb	11.3	218	0.0452	1213233233
0.3	tadd	28.6	356	0.04498	1213332221
0.3	tall	83.8	986	0.04498	1213332221
0.3	ts	219.5	2765	0.04498	1213332221
0.3	tsb	11.4	218	0.04396	1213233233
0.1	tadd	15.9	209	0.04427	1223331111
0.1	tall	34.6	452	0.04427	1223331111
0.1	ts	95.6	1424	0.04427	1223331111
0.1	tsb	18.3	350	0.04069	1123332221
0.03	tadd	25.5	331	0.03348	1233311111
0.03	tall	24.8	334	0.03348	1233311111
0.03	ts	20.0	385	0.03348	1233311111
0.03	tsb	11.9	254	0.03348	1233311111
0.01	tadd	11.8	158	0.02309	1221111111
0.01	tall	11.3	158	0.02309	1221111111
0.01	ts	7.6	161	0.02309	1221111111
0.01	tsb	6.7	155	0.02309	1221111111
0.001	tadd	1.4	19	0.01818	1111111111
0.001	tall	1.4	19	0.01818	1111111111
0.001	ts	0.9	19	0.01818	1111111111
0.001	tsb	0.8	19	0.01818	1111111111

Table A.3: Results for L1P3T10 using $L1$

L^s	mode	time	nodes	$\lambda(a)$	a
10.0	tadd	92.9	1020	0.04537	2321113332
10.0	tall	320.8	3243	0.04537	2321113332
10.0	ts	712.4	8292	0.04537	2321113332
10.0	tsb	33.4	648	0.04537	2323111133
1.0	tadd	86.2	954	0.04526	2321113332
1.0	tall	308.2	3120	0.04526	2321113332
1.0	ts	711.9	8292	0.04526	2321113332
1.0	tsb	33.4	648	0.04522	2323111133
0.3	tadd	77.2	864	0.04486	2321113332
0.3	tall	229.2	2403	0.04486	2321113332
0.3	ts	709.7	8268	0.04486	2321113332
0.3	tsb	33.4	648	0.04458	2323111133
0.1	tadd	31.6	375	0.04417	2113332222
0.1	tall	79.8	927	0.04417	2113332222
0.1	ts	209.8	2964	0.04417	2113332222
0.1	tsb	20.2	426	0.04076	2321113333
0.03	tadd	37.9	446	0.03551	2133322222
0.03	tall	43.1	527	0.03551	2133322222
0.03	ts	46.5	863	0.03551	2133322222
0.03	tsb	13.6	312	0.03551	2133322222
0.01	tadd	26.3	312	0.02309	2332222222
0.01	tall	25.3	312	0.02309	2332222222
0.01	ts	15.7	330	0.02309	2332222222
0.01	tsb	12.7	297	0.02309	2332222222
0.001	tadd	3.8	47	0.01818	2222222222
0.001	tall	3.6	47	0.01818	2222222222
0.001	ts	2.0	47	0.01818	2222222222
0.001	tsb	1.9	47	0.01818	2222222222

Table A.4: Results for L2P3T10 using Ll

problem	mode	time	nodes	$\lambda(a)$	a
L1P2T10	tadd	11.3	173	0.03301	1112222221
	tall	21.4	313	0.03301	1112222221
	ts	21.4	373	0.03301	1212221111
	tsb	6.6	153	0.03298	1222111212
	tslo	56.8	1023	0.03301	1112222221
	tslob	0.4	19	0.01818	1111111111
	tspo	20.7	373	0.03301	1212221111
	tspob	6.1	153	0.03299	1222111212
L2P2T10	tadd	24.3	326	0.03301	1112222221
	tall	49.2	626	0.03301	1112222221
	ts	45.7	746	0.03301	1112222221
	tsb	13.1	306	0.03300	2221111112
	tslo	125.0	2046	0.03301	1112222221
	tslob	0.5	20	0.01818	2222222222
	tspo	44.2	746	0.03301	1112222221
	tspob	12.1	306	0.03300	2221111112
L1P5T10	tadd	7.6	70	0.06422	1234554321
	tall	48.5	449	0.06422	1234554321
	ts	712.5	6632	0.06422	1234554321
	tsb	2.7	40	0.06422	1234554321
	tslo	>1253	>12965	>0.06242	1234435555
	tslob	0.7	19	0.01818	1111111111
	tspo	706.2	6632	0.06422	1234554321
	tspob	2.6	40	0.06422	1234554321

Table A.5: Results for L1P2T10, L2P2T10 and L1P5T10 using Ll

L^*	mode	time	nodes	$\lambda(a)$
1.0	c	0.5	29	0.01818
1.0	tadd	86.1	954	0.04526
1.0	tspob	31.1	648	0.04522
0.1	c	0.6	29	0.01818
0.1	tadd	31.5	375	0.04417
0.1	tspob	31.1	648	0.03636
0.01	c	0.5	29	0.01818
0.01	tadd	26.2	312	0.02309
0.01	tspob	31.1	648	0.00414

Table A.6: Results for L2P3T10 using *c*, *tspob* and *tadd*

search brackets (as described in Section 4.2, the search bracket ranges in value from 0 to 1 and affects the optimality guarantee of the solution). Tables A.7 and A.8 compare the effects of using different search brackets for the L1P2T10 allocation example while Tables A.9 and A.10 do likewise for the L2P3T10 allocation example. Discussion of these results is given in Subsection 6.1.5.

bracket	nodes	time	$\lambda(a)$	a
0.0	173	11.3	0.03301	1112222221
0.0001	165	10.8	0.03301	1112222221
0.001	97	5.8	0.03301	1112222221
0.01	37	2.1	0.03301	1112222221
0.05	37	2.1	0.03301	1112222221
0.25	37	2.2	0.03301	1112222221
0.9	37	2.1	0.03301	1112222221
0.99	37	2.1	0.03301	1112222221

Table A.7: Search Bracket Comparison for L1P2T10 using *tadd* Lf

bracket	nodes	time	$\lambda(a)$	a
0.0	173	11.4	0.03301	1112222221
0.0001	165	10.7	0.03301	1112222221
0.001	97	5.8	0.03301	1112222221
0.01	37	2.2	0.03301	1112222221
0.05	37	2.2	0.03301	1112222221
0.25	37	2.1	0.03301	1112222221
0.9	37	2.1	0.03301	1112222221
0.99	37	2.1	0.03301	1112222221

Table A.8: Search Bracket Comparison for L1P2T10 using *tadd Ll*

bracket	nodes	time	$\lambda(a)$	a
0.0	954	86.1	0.04526	2321113332
0.0001	930	83.8	0.04526	2321113332
0.001	726	63.7	0.04526	2321113332
0.01	315	26.6	0.04523	1213332221
0.05	315	26.6	0.04523	1213332221
0.25	315	26.6	0.04523	1213332221
0.9	315	26.6	0.04523	1213332221
0.99	315	26.6	0.04523	1213332221

Table A.9: Search Bracket Comparison for L2P3T10 using *tadd Lf*

bracket	nodes	time	$\lambda(a)$	a
0.0	954	86.0	0.04526	2321113332
0.0001	930	83.7	0.04526	2321113332
0.001	726	63.8	0.04526	2321113332
0.01	312	26.4	0.04523	1213332221
0.05	312	26.3	0.04523	1213332221
0.25	312	26.3	0.04523	1213332221
0.9	312	26.3	0.04523	1213332221
0.99	312	26.3	0.04523	1213332221

Table A.10: Search Bracket Comparison for L2P3T10 using *tadd Ll*

A.3 Problem Size

This section presents two tables comparing the results of six modes for eight sizes of task-graphs allocated onto the L1P3 processor-graph. Table A.11 shows 6, 8, 10 and 12 task results, and Table A.12 shows 14, 16, 18 and 20 task results. Where a question mark appears in the $\lambda(a)$ and a fields, this means that no complete allocation was yet evaluated, implying that many more nodes need to be created before the solution is found. These results are discussed in Subsection 6.3.1 and four of the modes are plotted in figures 6.15 and 6.16. Results for these eight STAPs are also shown for another two modes, *tsec* and *tsecb* in Table A.13.

problem	mode	time	nodes	$\lambda(a)$	a
L1P3T6	tadd	1.1	15	0.11868	123321
	tall	1.5	21	0.11868	123321
	ts	1.4	27	0.11868	123321
	tsb	0.6	15	0.11868	123321
	tspo	1.4	27	0.11868	123321
	tspob	0.6	15	0.11868	123321
L1P3T8	tadd	3.5	48	0.0692	12321333
	tall	11.0	144	0.0692	12321333
	ts	15.8	261	0.0692	12321333
	tsb	1.2	27	0.06914	12331223
	tspo	15.3	261	0.0692	12321333
	tspob	1.1	27	0.06914	12331223
L1P3T10	tadd	29.2	362	0.0453	1213332221
	tall	96.9	1082	0.0453	1213332221
	ts	219.1	2765	0.0453	1213332221
	tsb	11.3	218	0.0452	1213233233
	tspo	215.7	2765	0.0453	1213332221
	tspob	10.7	218	0.0452	1213233233
L1P3T12	tadd	55.5	623	0.032	112233332211
	tall	836.0	7388	0.032	112233332211
	ts	>916	>8999	>0.032	112233332211
	tsb	1.9	33	0.03194	123323211112
	tspo	>904	>8999	>0.032	112233332211
	tspob	1.8	33	0.03194	123323211112

Table A.11: Results for L1P3 Stochastic Loops with 6, 8, 10 and 12 Tasks

problem	mode	time	nodes	$\lambda(a)$	a
LIP3T14	tadd	135.1	1433	0.02379	12232333111111
	tall	>1365	>8998	?	?
	ts	>1133	>8999	>0.02379	12232333111111
	tsb	2.7	39	0.02374	12333221121131
	tspo	>1111	>8999	>0.02379	12232333111111
	tspob	2.6	39	0.02374	12333221121131
LIP3T16	tadd	821.1	7342	0.01836	1213333222211111
	tall	>1359	>8998	?	?
	ts	>1399	>9000	>0.01835	1231313322222111
	tsb	>1015	>9000	>0.01830	1221333321211323
	tspo	>1373	>9000	>0.01835	1231313322222111
	tspob	>986	>9000	>0.01830	1221333321211323
LIP3T18	tadd	>1412	>8998	>0.0146	12221113333332221
	tall	>1345	>8998	?	?
	ts	>1648	>9000	>0.0146	123332312221111111
	tsb	4.7	51	0.0146	123332322111211111
	tspo	>1627	>9000	>0.0146	123332312221111111
	tspob	4.6	51	0.0146	123332322111211111
LIP3T20	tadd	>1409	>8998	>0.01189	12222311113333333333
	tall	>1439	>8998	?	?
	ts	>1978	>9000	>0.01189	12333232221111131111
	tsb	6.2	57	0.01188	12333322221111113112
	tspo	>1957	>9000	>0.01189	12333232221111131111
	tspob	6.0	57	0.01188	12333322221111113112

Table A.12: Results for LIP3 Stochastic Loops with 14, 16, 18 and 20 Tasks

problem	mode	time	nodes	$\lambda(a)$	a
L1P3T6	tsec	2.2	27	0.11868	123321
	tsecb	1.3	15	0.11868	123321
L1P3T8	tsec	23.7	261	0.06920	12321333
	tsecb	3.3	27	0.06914	12331223
L1P3T10	tsec	299.3	2765	0.04530	1213332221
	tsecb	33.6	216	0.04521	1212333323
L1P3T12	tsec	>1290	>8999	>0.03200	112233332211
	tsecb	8.2	33	0.03194	123323211112
L1P3T14	tsec	>2223	>8999	>0.02379	12232333111111
	tsecb	12.8	39	0.02374	12333221121131
L1P3T16	tsec	>2948	>9000	>0.01835	1231313322222111
	tsecb	>2558	>9000	>0.01830	1223113332231213
L1P3T18	tsec	>3125	>9000	>0.01460	123332312221111111
	tsecb	26.8	51	0.01460	123332322111211111
L1P3T20	tsec	>3695	>9000	>0.01189	12333232221111131111
	tsecb	37.1	57	0.01188	12333322221111113112

Table A.13: Results for L1P3 Stochastic Loops with *tsec* and *tsecb*

Appendix B

Nested Task-Graph Algorithms

This appendix discusses several algorithms for nested task-graphs (NTG), which were defined in Section 2.3. Section B.1 explains five algorithms used together to derive $\pi_{ii'}$, the task successor probabilities that are specified for general task-graphs (GTGs), but not for NTGs. Section B.2 finds $b_{ii'}^L$ for NTGs from $\bar{b}_{ii'}^L$, as was specified in Section 2.3, and Section B.3 explains the algorithms used to find the path-lengths of SJ NTGs and to find a lower-bound on the path-lengths of SJ NTGs; these path-length bounds are used in Chapter 5 to find throughput capacity bounds on NTG allocations.

B.1 Finding $\pi_{ii'}$ for Nested Task-Graphs

This section describes the algorithms used to find $\pi_{ii'}$ for the GTGs created by the NTG conversion process of Subsection 2.3.2. $\pi_{ii'}$ is the probability that a token leaving T_i executes $T_{i'}$ next (i.e., that $T_{i'}$ is an *immediate successor* to T_i). $\pi_{ii'}$ is the sum of the probabilities of all the paths in the NTGs leading directly from T_i to $T_{i'}$ (i.e., with no intermediary tasks). Since NIL task-graphs allow tokens to traverse OR task-graphs without executing any tasks, there can be many such paths. Because of LOOP task-graphs, there may even be an infinite number of such paths. Using the hierarchical nature of our NTGs, this section derives several algorithms that are used together to find $\pi_{ii'}$. Before continuing the development of

these algorithms, we introduce the following NTG notation:

G is a NTG. Where several NTGs are referred to, subscripts are used (e.g., G_1 and G_2).

$\text{type}(G)$ is the type of NTG G (i.e., one of LOOP, NIL, OR, SEQ, SJ, or TASK). For example, if G is a SJ task-graph, then $\text{type}(G) = \text{SJ}$,

$|G|$ is the cardinality of NTG G . Based on $\text{type}(G)$, we define $|G|$ as follows:

LOOP: $|G| = 1$,

NIL: $|G| = 0$,

OR: $|G|$ is the number of parallel branches in G ,

SEQ: $|G|$ is the number of sequential NTGs in G (at least two),

SJ: $|G|$ is the number of parallel branches in G plus two (for the split-task and the join-task),

TASK: $|G| = 1$.

$|G|$ is thus the number of NTGs within G plus one per task at the outermost level. Cardinality is used to access the subgraphs and tasks in G .

G^h is the h^{th} primary subgraph within G . G^h is also a NTG, and is defined as follows, based on $\text{type}(G)$:

LOOP: G^1 is the NTG that may be executed multiple times,

NIL: G^h is undefined,

OR: G^1 to $G^{|G|}$ refer to the different parallel branches in G . G^h is the h^{th} choice, with G^1 being the upper one,

SEQ: G^{h+1} is always executed after $G^h \forall h \in \{1, \dots, |G| - 1\}$,

SJ: G^1 is the split-task and $G^{|G|}$ is the join-task. G^2 to $G^{|G|-1}$ refer to the different parallel branches in G ; G^2 – the upper branch – sends a token to $G^{|G|}$ when it is finished, while all the other branches only send termination notifications.

TASK: G^h is undefined.

$\text{outer}(G)$ is the NTG containing G as a primary subgraph. Since neither a NIL task-graph nor a TASK task-graph contain any NTGs, $\text{type}(\text{outer}(G)) \in \{\text{LOOP}, \text{OR}, \text{SEQ}, \text{SJ}\}$,

$G_{MIN}(T_i, T_{i'})$ is the lowest level NTG containing both T_i and $T_{i'}$ (i.e., if $G = G_{MIN}(T_i, T_{i'})$ then $\forall h \in \{1, \dots, |G|\}$, $\text{not}(T_i \in G^h \wedge T_{i'} \in G^h)$). Note that $\text{type}(G_{MIN}(T_i, T_{i'})) \notin \{\text{NIL}, \text{LOOP}\}$ and that $G_{MIN}(T_i, T_{i'})$ is a TASK task-graph only if $T_i = T_{i'}$,

$\text{EOP}(G)$ is the point labeled EOP at the end of G ,

$\text{SOP}(G)$ is the point labeled SOP at the start of G ,

$\text{LOOP}(G)$ is the point labeled LOOP just after $\text{SOP}(G)$. We assume that $\text{type}(G) = \text{LOOP}$,

$\overline{\text{LOOP}}(G)$ is the point labeled $\overline{\text{LOOP}}$ just before $\text{EOP}(G)$. We assume that $\text{type}(G) = \text{LOOP}$,

$\pi^{LOOP}(G)$ is the probability of taking the $\overline{\text{LOOP}}(G)$ to $\text{LOOP}(G)$ arc (e.g., $\pi^{LOOP}(G) = 1$ for the LOOP task-graph shown on Figure 2.9). We assume that $\text{type}(G) = \text{LOOP}$, and

$\pi^{OR}(G, G^h)$ is the probability of a token at $\text{SOP}(G)$ executing branch G^h before reaching $\text{EOP}(G)$ (e.g., where G is the OR task-graph shown in Figure 2.9 and $T_2 \in G^1$, $\pi^{OR}(G, G^1) = .2$). We assume that $\text{type}(G) = \text{OR}$.

The above notation is used in the algorithms to find the following four probabilities, which are then used in the algorithm to find $\pi_{it'}$ for NTGs, the main result of this section.

$\pi^{NIL}(G)$ is the probability of a token at $SOP(G)$ going to $EOP(G)$ via a task-free path¹ remaining completely within G ,

$\pi^{RET}(G)$ is the probability of a token at $EOP(G)$ going to $SOP(G)$ via any task-free path remaining completely outside G ,

$\pi^{SOP}(G, T_i)$ is the probability of a token at $SOP(G)$ going to T_i via any task-free path remaining completely within G . We assume that $T_i \in G$, and thus $\text{type}(G) \neq \text{NIL}$, and

$\pi^{EOP}(T_i, G)$ is the probability of a token at T_i going to $EOP(G)$ via any task-free path remaining completely within G . We assume that $T_i \in G$, and thus $\text{type}(G) \neq \text{NIL}$.

We will now describe five algorithms: one each for finding $\pi^{NIL}(G)$, $\pi^{RET}(G)$, $\pi^{SOP}(G, T_i)$, $\pi^{EOP}(T_i, G)$ and π_{in} . The algorithm for finding $\pi^{NIL}(G)$ is used by all the other algorithms, and so is introduced first, while the π_{in} algorithm uses all the other algorithms, and so is discussed last, in Subsection B.1.5.

B.1.1 Algorithm to Find $\pi^{NIL}(G)$

$\pi^{NIL}(G)$ is the probability of no task being executed during the execution of NTG G (i.e., that a task-free path is taken during the execution of G). Where G is a SJ or TASK task-graph, this probability is clearly zero. For a NIL task-graph, $\pi^{NIL}(G)$ is one. For SEQ task-graph G , $\pi^{NIL}(G)$ is the product of $\pi^{NIL}(G^h)$ for all its primary subgraphs G^h . Where G is an OR task-graph, $\pi^{NIL}(G)$ is the weighted sum of the probabilities of no tasks being executed in each of its primary subgraphs. The case of G being a LOOP task-graph is the most complicated, because a token entering G may repeatedly skip through G^1 and loop back from $\overline{LOOP}(G)$ to $LOOP(G)$ until the token finally exits G . The probability of going from $SOP(G)$ directly to $EOP(G)$ without looping back is $\pi^{NIL}(G^1)(1 - \pi^{LOOP}(G))$.

¹recall that a task-free path in a NTG was defined in Subsection 2.3.2 and consists of any sequence of arrows, connected only by points or by NIL task-graphs, that leads from task to task.

The probability of a path looping back once is $\pi^{NIL}(G^1)\pi^{LOOP}(G)\pi^{NIL}(G^1)(1 - \pi^{LOOP}(G))$. Summing the probabilities of all such paths, we find that:

$$\pi^{NIL}(G) = \pi^{NIL}(G^1)(1 - \pi^{LOOP}(G)) \sum_{n=0}^{\infty} [\pi^{LOOP}(G)\pi^{NIL}(G^1)]^n$$

Recalling the identity $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x} \forall 0 \leq x < 1$, and realizing that $\pi^{LOOP}(G) < 1$ for all LOOP task-graphs (except for the implicit outer response LOOP task-graph G_r , where $\pi^{NIL}(G_r^1) = 0$ anyway) allows us to eliminate the above infinite sum, yielding:

$$\pi^{NIL}(G) = \frac{\pi^{NIL}(G^1)(1 - \pi^{LOOP}(G))}{1 - \pi^{LOOP}(G)\pi^{NIL}(G^1)}$$

Summarising, $\pi^{NIL}(G)$, for any G , is determined by the following recursive algorithm:

case type(G):

$$\text{LOOP: } \pi^{NIL}(G) = \frac{\pi^{NIL}(G^1)(1 - \pi^{LOOP}(G))}{1 - \pi^{LOOP}(G)\pi^{NIL}(G^1)}$$

$$\text{NIL: } \pi^{NIL}(G) = 1$$

$$\text{OR: } \pi^{NIL}(G) = \sum_{h=1}^{|G|} \pi^{OR}(G, G^h) \pi^{NIL}(G^h)$$

$$\text{SEQ: } \pi^{NIL}(G) = \prod_{h=1}^{|G|} \pi^{NIL}(G^h)$$

$$\text{SJ: } \pi^{NIL}(G) = 0$$

$$\text{TASK: } \pi^{NIL}(G) = 0$$

$\pi^{NIL}(G)$, as defined above, forms the basis for the following four algorithms.

B.1.2 Algorithm to Find $\pi^{RET}(G)$

$\pi^{RET}(G)$ is the probability that a token leaving $EOP(G)$ will have executed no task by the time it returns to $SOP(G)$ (i.e., that it follows a task-free path from $EOP(G)$ to $SOP(G)$). In order for the token to return, G must be contained in a LOOP task-graph (e.g., in Figure 2.10, a token leaving T_2 may return immediately via a task-free path by following the $\overline{\text{LOOP}}$ to LOOP arc having probability v). As was discussed in Section 2.3, all NTGs are implicitly

contained in an outermost LOOP task-graph whose primary subgraph is a sequence of the reference-task and the NTG. Let G_o be the NTG within which G is a primary subgraph. Where G_o is a SJ task-graph, $\pi^{RET}(G)$ is clearly zero, since all tokens going through a SJ task-graph must execute both the split-task and the join-task. Where G_o is a SEQ task-graph, $\pi^{RET}(G)$ is the probability of a token from EOP(G) reaching EOP(G_o), coming back to SOP(G_o), and then reaching SOP(G), all without executing a task. $\pi^{RET}(G)$ is found recursively by climbing the hierarchy of NTGs within which G is contained. For G being a primary subgraph of an OR task-graph (say $G = G_o^h$), we must consider that a token may leave G , repeatedly return to SOP(G_o), and take a branch other than G before it finally reaches SOP(G) or executes a task. The probability of a token going from EOP(G) to SOP(G) with no loops through other OR branches is $\pi^{RET}(G_o)\pi^{OR}(G_o, G)$. The probability of a path with one loop through another OR branch is $\pi^{RET}(G_o) \left(\sum_{h' \neq h} \pi^{OR}(G_o, G_o^{h'}) \pi^{NIL}(G_o^{h'}) \right) \pi^{RET}(G_o) \pi^{OR}(G_o, G)$. Summing the probabilities of all such paths gives:

$$\pi^{RET}(G) = \pi^{RET}(G_o) \pi^{OR}(G_o, G) \sum_{n=0}^{\infty} \left[\left(\sum_{h' \neq h} \pi^{OR}(G_o, G_o^{h'}) \pi^{NIL}(G_o^{h'}) \right) \pi^{RET}(G_o) \right]^n$$

Replacing the infinite sum simplifies this to:

$$\pi^{RET}(G) = \frac{\pi^{RET}(G_o) \pi^{OR}(G_o, G)}{1 - \pi^{RET}(G_o) \sum_{h' \neq h} \pi^{OR}(G_o, G_o^{h'}) \pi^{NIL}(G_o^{h'})}$$

Where G_o is a LOOP task-graph, there are two cases: $\pi^{LOOP}(G_o) = 1$ or $\pi^{LOOP}(G_o) < 1$. The first case is only allowed in the outer LOOP task-graph and has $\pi^{RET}(G) = 1$. For the second case, clearly:

$$\pi^{RET}(G) = \pi^{LOOP}(G_o) + \pi^{RET}(G_o)(1 - \pi^{LOOP}(G_o)).$$

Summarising, $\pi^{RET}(G)$ is found by the following recursive algorithm:

let $G_o = \text{outer}(G)$ and let $G = G_o^h$

case type(G_o): (since $G \in G_o$, type(G_o) $\notin$ {NIL, TASK})

$$\text{LOOP: } \pi^{RET}(G) = \begin{cases} 1 & \text{if } \pi^{LOOP}(G_o) = 1 \\ \pi^{LOOP}(G_o) + \pi^{RET}(G_o) \times (1 - \pi^{LOOP}(G_o)) & \text{otherwise} \end{cases}$$

$$\text{OR: } \pi^{RET}(G) = \frac{\pi^{RET}(G_o) \pi^{OR}(G_o, G)}{1 - \pi^{RET}(G_o) \sum_{h' \neq h} \pi^{OR}(G_o, G_o^{h'}) \pi^{NIL}(G_o^{h'})}$$

$$\text{SEQ: } \pi^{RET}(G) = \left[\prod_{h'=h+1}^{|G_o|} \pi^{NIL}(G_o^{h'}) \right] \pi^{RET}(G_o) \prod_{h'=1}^{h-1} \pi^{NIL}(G_o^{h'})$$

$$\text{SJ: } \pi^{RET}(G) = 0$$

Note that this algorithm recurses outwardly in the overall NTG structure until either an SJ task-graph is found or until the outermost LOOP task-graph is found; this approach contrasts markedly with that of the π^{NIL} algorithm above or the π^{SOP} and π^{EOP} algorithms below that recurse inwardly. The π^{RET} algorithm is used in Subsection B.1.5 to find π_{in} for any NTG.

B.1.3 Algorithm to Find $\pi^{SOP}(G, T_i)$

$\pi^{SOP}(G, T_i)$ is the probability of a token from $SOP(G)$ reaching T_i via any task-free path remaining completely within G ($T_i \in G$). Where G is a TASK task-graph, $\pi^{SOP}(G, T_i)$ is clearly one, while for a SJ task-graph, $\pi^{SOP}(G, T_i)$ is zero unless T_i is the split-task. Where G is a SEQ task-graph and T_i is in G^h , $\pi^{SOP}(G, T_i)$ is the probability of skipping all the primary subgraphs before G^h times $\pi^{SOP}(G^h, T_i)$. For an OR task-graph with T_i in G^h , $\pi^{SOP}(G, T_i)$ is $\pi^{SOP}(G^h, T_i)$ times the probability of executing that branch. When G is a LOOP task-graph, we must (as with $\pi^{NIL}(G)$) consider that a token may enter G and repeatedly skip through G^1 and loop back from $\overline{LOOP}(G)$ to $LOOP(G)$ until it finally executes T_i (or another task). The probability of a token going from $SOP(G)$ to T_i without looping back to $LOOP(G)$ is $\pi^{SOP}(G^1, T_i)$. The probability of a path that loops back once is $\pi^{NIL}(G^1) \pi^{LOOP}(G) \pi^{SOP}(G^1, T_i)$. Summing the probabilities of all such paths gives:

$$\pi^{SOP}(G, T_i) = \pi^{SOP}(G^1, T_i) \sum_{n=0}^{\infty} \left[\pi^{NIL}(G^1) \pi^{LOOP}(G) \right]^n$$

Replacing the infinite sum simplifies this to:

$$\pi^{SOP}(G, T_i) = \frac{\pi^{SOP}(G^1, T_i)}{1 - \pi^{NIL}(G^1) \pi^{LOOP}(G)}$$

Summarising, $\pi^{SOP}(G, T_i)$ for any G and any T_i in G can be found by the following recursive algorithm:

case type(G): (since $T_i \in G$, type(G) $\neq$ NIL)

$$\text{LOOP: } \pi^{SOP}(G, T_i) = \frac{\pi^{SOP}(G^1, T_i)}{1 - \pi^{NIL}(G^1)\pi^{LOOP}(G)}$$

$$\text{OR: } T_i \in G^h, \pi^{SOP}(G, T_i) = \pi^{OR}(G, G^h)\pi^{SOP}(G^h, T_i)$$

$$\text{SEQ: } T_i \in G^h, \pi^{SOP}(G, T_i) = \left[\prod_{h'=1}^{h-1} \pi^{NIL}(G^{h'}) \right] \pi^{SOP}(G^h, T_i)$$

$$\text{SJ: } \pi^{SOP}(G, T_i) = \begin{cases} 1 & \text{if } G^1 = T_i \\ 0 & \text{otherwise} \end{cases}$$

$$\text{TASK: } \pi^{SOP}(G, T_i) = 1$$

This algorithm is used in Subsection B.1.5 to find $\pi_{ii'}$ for any NTG.

B.1.4 Algorithm to Find $\pi^{EOP}(T_i, G)$

$\pi^{EOP}(T_i, G)$ is the probability of a token from T_i reaching EOP(G) via any task-free path remaining completely within G ($T_i \in G$). Where G is a TASK task-graph, $\pi^{EOP}(T_i, G)$ is clearly one, while for a SJ task-graph, $\pi^{EOP}(T_i, G)$ is zero unless T_i is the join-task. Where G is a SEQ task-graph and T_i is in G^h , $\pi^{EOP}(T_i, G)$ is the probability of skipping all the primary subgraphs after G^h times $\pi^{EOP}(T_i, G^h)$. For an OR task-graph, with T_i in G^h , $\pi^{EOP}(T_i, G)$ is simply $\pi^{EOP}(T_i, G^h)$. When G is a LOOP task-graph, we must again consider that a token may leave T_i and repeatedly loop back from $\overline{\text{LOOP}}(G)$ to $\text{LOOP}(G)$ and skip through G^1 until it finally exits G (or executes a task in G^1). The probability of a token going from T_i to EOP(G) with no loop back to $\text{LOOP}(G)$ is $\pi^{EOP}(T_i, G^1)(1 - \pi^{LOOP}(G))$. For one loop back, the probability is $\pi^{EOP}(T_i, G^1)\pi^{LOOP}(G)\pi^{NIL}(G^1)(1 - \pi^{LOOP}(G))$. Summing the probabilities of all such paths gives:

$$\pi^{EOP}(T_i, G) = \pi^{EOP}(T_i, G^1)(1 - \pi^{LOOP}(G)) \sum_{n=0}^{\infty} \left[\pi^{LOOP}(G)\pi^{NIL}(G^1) \right]^n$$

Replacing the infinite sum simplifies this to:

$$\pi^{EOP}(T_i, G) = \frac{\pi^{EOP}(T_i, G^1)(1 - \pi^{LOOP}(G))}{1 - \pi^{LOOP}(G)\pi^{NIL}(G^1)}$$

Summarising, the following recursive algorithm finds $\pi^{EOP}(T_i, G)$ for any G and any T_i in G :

case type(G): (since $T_i \in G$, type(G) $\neq$ NIL)

$$\text{LOOP: } \pi^{EOP}(T_i, G) = \frac{\pi^{EOP}(T_i, G^1)(1 - \pi^{LOOP}(G))}{1 - \pi^{LOOP}(G)\pi^{NIL}(G^1)}$$

$$\text{OR: } T_i \in G^h, \pi^{EOP}(T_i, G) = \pi^{EOP}(T_i, G^h)$$

$$\text{SEQ: } T_i \in G^h, \pi^{EOP}(T_i, G) = \pi^{EOP}(T_i, G^h) \prod_{h'=h+1}^{|G|} \pi^{NIL}(G^{h'})$$

$$\text{SJ: } \pi^{EOP}(T_i, G) = \begin{cases} 1 & \text{if } G^{|G|} = T_i \\ 0 & \text{otherwise} \end{cases}$$

$$\text{TASK: } \pi^{EOP}(T_i, G) = 1$$

This algorithm is used below, along with the three other algorithms of Section B.1, to find $\pi_{ii'}$ for any NTG.

B.1.5 Algorithm to Find $\pi_{ii'}$

$\pi_{ii'}$ is the probability that $T_{i'}$ is an immediate successor of T_i . We now describe an algorithm to find $\pi_{ii'}$ that uses the previous algorithms for $\pi^{NIL}(G)$, $\pi^{RET}(G)$, $\pi^{SOP}(G, T_i)$ and $\pi^{EOP}(T_i, G)$. Both T_i and $T_{i'}$ are contained in some NTG G . We let $G = G_{MIN}(T_i, T_{i'})$ and so G can only be an OR, SEQ, SJ or TASK task-graph. G can only be a TASK task-graph if $T_i = T_{i'}$, in which case $\pi_{ii'}$ is clearly $\pi^{RET}(T_i)$. Where G is a SJ task-graph, only a few possible cases allow $\pi_{ii'}$ to be non-zero. If T_i is the join-task, or if $T_{i'}$ is the split-task, then $\pi_{ii'}$ is zero, since tokens having executed the join-task leave the SJ task-graph and since tokens may only enter the split-task from $SOP(G)$. If T_i and $T_{i'}$ are both in the branches of the SJ task-graph, then $\pi_{ii'}$ must be zero since they must be in different branches and since each branch is an independent NTG. If $T_{i'}$ is the join-task, then $\pi_{ii'}$ is zero unless T_i is in G^2 (then $\pi_{ii'}$ is $\pi^{EOP}(T_i, G^2)$). G^2 is the only branch to send a token to the join-task (the other branches only send termination notifications). When T_i is the split-task and $T_{i'}$ is in branch G^h , then $\pi_{ii'} = \pi^{SOP}(G^h, T_{i'})$. If T_i is the split-task and $T_{i'}$ is the join-task, then $\pi_{ii'} = \pi^{NIL}(G^2)$ since only a token entering G^2 can reach the join-task.

When G is an OR task-graph, then we must consider a token from T_i leaving G , returning and looping through any of the branches and any

LOOP task-graphs G is contained within before executing T_i . Since T_i and $T_{i'}$ must be in different branches of G (say G^h and $G^{h'}$, respectively) the probability of a token going from T_i to $T_{i'}$ and only passing EOP(G) once is $\pi^{EOP}(T_i, G^h) \pi^{RET}(G) \pi^{OR}(G, G^{h'}) \pi^{SOP}(G^{h'}, T_{i'})$. Any positive number of passes by EOP(G) is possible. Summing them all gives $\pi_{ii'}$ as:

$$\pi^{EOP}(T_i, G^h) \pi^{RET}(G) \pi^{OR}(G, G^{h'}) \pi^{SOP}(G^{h'}, T_{i'}) \sum_{n=0}^{\infty} \left[\left(\sum_{h''=1}^{|G|} \pi^{OR}(G, G^{h''}) \pi^{NIL}(G^{h''}) \right) \pi^{RET}(G) \right]^n$$

Replacing the infinite sum simplifies this to:

$$\pi_{ii'} = \frac{\pi^{EOP}(T_i, G^h) \pi^{RET}(G) \pi^{OR}(G, G^{h'}) \pi^{SOP}(G^{h'}, T_{i'})}{1 - \pi^{RET}(G) \sum_{h''=1}^{|G|} \pi^{OR}(G, G^{h''}) \pi^{NIL}(G^{h''})}$$

The case where G is a SEQ task-graph is the most complicated since T_i may be in a primary subgraph either before or after that of $T_{i'}$. When T_i appears before $T_{i'}$, then the following sum represents $\pi_{ii'}$:

$$\pi^{EOP}(T_i, G^h) \left[\prod_{h''=h+1}^{h'-1} \pi^{NIL}(G^{h''}) \right] \pi^{SOP}(G^{h'}, T_{i'}) \sum_{n=0}^{\infty} \left[\left(\prod_{h''=h'}^{|G|} \pi^{NIL}(G^{h''}) \right) \pi^{RET}(G) \prod_{h''=1}^{h'-1} \pi^{NIL}(G^{h''}) \right]^n$$

Replacing the infinite sum simplifies this to:

$$\pi_{ii'} = \frac{\pi^{EOP}(T_i, G^h) \left[\prod_{h''=h+1}^{h'-1} \pi^{NIL}(G^{h''}) \right] \pi^{SOP}(G^{h'}, T_{i'})}{1 - \pi^{RET}(G) \prod_{h''=1}^{|G|} \pi^{NIL}(G^{h''})}$$

When T_i appears after $T_{i'}$ in G , we find the following sum for $\pi_{ii'}$:

$$\pi_{ii'} = \pi^{EOP}(T_i, G^h) \left[\prod_{h''=h+1}^{|G|} \pi^{NIL}(G^{h''}) \right] \pi^{RET}(G) \left[\prod_{h''=1}^{h'-1} \pi^{NIL}(G^{h''}) \right] \pi^{SOP}(G^{h'}, T_{i'}) \sum_{n=0}^{\infty} \left[\left(\prod_{h''=h'}^{|G|} \pi^{NIL}(G^{h''}) \right) \pi^{RET}(G) \prod_{h''=1}^{h'-1} \pi^{NIL}(G^{h''}) \right]^n$$

Replacing the infinite sum simplifies this to:

$$\pi_{ii'} = \frac{\pi^{EOP}(T_i, G^h) \left[\prod_{h''=h+1}^{|G|} \pi^{NIL}(G^{h''}) \right] \pi^{RET}(G) \left[\prod_{h''=1}^{h'-1} \pi^{NIL}(G^{h''}) \right] \pi^{SOP}(G^{h'}, T_{i'})}{1 - \pi^{RET}(G) \prod_{h''=1}^{|G|} \pi^{NIL}(G^{h''})}$$

These possible values for $\pi_{ii'}$ are summarised in the algorithm below:

if $T_i = T_{i'}$ then $\pi_{ii'} = \pi^{RET}(T_i)$ (type($G_{MIN}(T_i, T_{i'})$) = TASK)
 else let $G = G_{MIN}(T_i, T_{i'})$ (type(G) $\notin$ {LOOP, NIL, TASK})
 let $T_i \in G^h, T_{i'} \in G^{h'}$ ($h \neq h'$)
 case type(G):

$$\text{OR: } \pi_{ii'} = \frac{\pi^{EOP}(T_i, G^h) \pi^{RET}(G) \pi^{OR}(G, G^{h'}) \pi^{SOP}(G^{h'}, T_{i'})}{1 - \pi^{RET}(G) \sum_{h''=1}^{|G|} \pi^{OR}(G, G^{h''}) \pi^{NIL}(G^{h''})}$$

SEQ: if $h < h'$ then $(T_i \text{ is "before" } T_{i'} \text{ in } G)$

$$\pi_{ii'} = \frac{\pi^{EOP}(T_i, G^h) \left[\prod_{h''=h+1}^{h'-1} \pi^{NIL}(G^{h''}) \right] \pi^{SOP}(G^{h'}, T_{i'})}{1 - \pi^{NIL}(G) \pi^{RET}(G)}$$

else $(T_i \text{ is "after" } T_{i'} \text{ in } G)$

$$\pi_{ii'} = \frac{\pi^{EOP}(T_i, G^h) \left[\prod_{h''=h+1}^{|G|} \pi^{NIL}(G^{h''}) \right] \pi^{RET}(G)}{\left[\prod_{h''=1}^{h'-1} \pi^{NIL}(G^{h''}) \right] \pi^{SOP}(G^{h'}, T_{i'}) \left(1 - \pi^{NIL}(G) \pi^{RET}(G) \right)^{-1}}$$

SJ: if $(h = |G| \vee h' = 1)$ then $\pi_{ii'} = 0$ (no backwards flow)

elseif $(h > 1 \wedge h' < |G|)$ then $\pi_{ii'} = 0$ (no flow between branches)

elseif $h = 1$ then

$$\pi_{ii'} = \begin{cases} \pi^{NIL}(G^2) & \text{if } h' = |G| \\ \pi^{SOP}(G^{h'}, T_{i'}) & \text{otherwise} \end{cases}$$

else $(h \neq 1, \text{ hence } h' = |G|)$

$$\pi_{ii'} = \begin{cases} \pi^{EOP}(T_i, G^2) & \text{if } h = 2 \\ 0 & \text{otherwise} \end{cases}$$

This algorithm for finding $\pi_{ii'}$ is used in Subsection 2.3.2 to help convert NTGs to GTGs. The algorithms of this section are also used in the next section to find $b_{ii'}^L$ for NTGs, again helping to convert NTGs to GTGs.

B.2 Finding $b_{ii'}^L$ for Nested Task-Graphs

In any task-graph, the interaction from T_i to $T_{i'}$ is described by:

- the probability of a message being send from T_i to $T_{i'}$, that is $\pi_{ii'}$ for the direct passage of a token from T_i to $T_{i'}$, and $\pi_{ii'}^{NOT}$ for the passage of a notification² from T_i to $T_{i'}$.

²a (termination) notification is sent from tasks at the end of all but one branch of SJ task-graphs, when they have finished execution. Subsection 2.3.1 describes this further.

- the mean communication workload per message, $b_{ii'}^L$.

This section describes an algorithm to convert from $\bar{b}_{ii'}^L$ to $b_{ii'}^L$ for NTGs. For GTGs, in Section 2.1, we developed a simple relation to find $\bar{b}_{ii'}^L$ from $b_{ii'}^L$ and $\pi_{ii'}$; for NTGs, $b_{ii'}^L$ is not always obvious, and so we describe how it can be found using $\bar{b}_{ii'}^L$, which is always reasonable to specify as part of the NTG, and $\pi_{ii'}$, which was derived in Subsection B.1.5. First, the two ways that communication workloads can be executed are described. Then, we state and explain a simple formula converting $\bar{b}_{ii'}^L$ to $b_{ii'}^L$ using the probabilities of these two ways. The probability of the first way is simply $\pi_{ii'}$; an algorithm to find the second probability is given below in Subsection B.2.1.

$b_{ii'}^L$ is the average communication workload from T_i to $T_{i'}$ per execution of T_i , while $\bar{b}_{ii'}^L$ is the average communication workload from T_i to $T_{i'}$ each time the communication occurs. There are only two ways for T_i to have a non-zero communication workload with $T_{i'}$; first, $T_{i'}$ may be an immediate successor of T_i and secondly, T_i may send a termination notification to $T_{i'}$. These two possibilities are mutually exclusive; if $T_{i'}$ can receive a termination notification from T_i then it will never receive a token from T_i , and *vice-versa*. In both cases, T_i sends a communication workload averaging $\bar{b}_{ii'}^L$ to $T_{i'}$.

$\pi_{ii'}$ is the probability that $T_{i'}$ is an immediate successor of T_i and was determined in Section B.1. We define $\pi_{ii'}^{NOT}$ as the probability that $T_{i'}$ receives a termination notification from T_i . For each execution of T_i , the probability that T_i communicates to $T_{i'}$ is then $\pi_{ii'} + \pi_{ii'}^{NOT}$. Using this probability, we see that:

$$b_{ii'}^L = \bar{b}_{ii'}^L (\pi_{ii'} + \pi_{ii'}^{NOT})$$

Since neither the values of $\pi_{ii'}$ nor the values of $\pi_{ii'}^{NOT}$ are obvious for many NTGs, it was necessary to specify $\bar{b}_{ii'}^L$ for NTGs and to derive $b_{ii'}^L$ for the common application description, the GTG. The next subsection develops an algorithm to find $\pi_{ii'}^{NOT}$ for NTGs.

B.2.1 Finding $\pi_{ii'}^{NOT}$ for Nested Task-Graphs

This subsection develops a simple algorithm to find $\pi_{ii'}^{NOT}$, the probabil-

ity that T_i sends a termination notification to $T_{i'}$ for NTGs. The algorithm uses $\pi^{NIL}(G)$ and $\pi^{EOP}(T_i, G)$ as developed in Section B.1.

Termination notifications are sent to join-tasks from all but one parallel branch in its SJ task-graph. If $T_{i'}$ is not a join-task, then $\pi_{ii'}^{NOT}$ is zero. $\pi_{ii'}^{NOT}$ is also zero because a join-task never sends to itself.

For $\pi_{ii'}^{NOT}$ to be non-zero, both T_i and $T_{i'}$ must be contained in a SJ task-graph G , with $T_{i'}$ being the join-task. We let $G = G_{MIN}(T_i, T_{i'})$ and so $\pi_{ii'}^{NOT}$ is zero if G isn't a SJ task-graph or if $T_{i'} \notin G^{|G|}$. If T_i is in G^2 , then $\pi_{ii'}^{NOT}$ is also zero because tasks in G^2 can at most generate tokens reaching the join-task. If T_i is in G^h , $2 < h < |G|$, then $\pi_{ii'}^{NOT}$ is just the probability that T_i can generate a token reaching $EOP(G^h)$ (i.e., $\pi_{ii'}^{NOT} = \pi^{EOP}(T_i, G^h)$). The only other case is that T_i may be the split-task and one branch contains a task-free path from its SOP to its EOP (recall that at most one branch G^h has a non-zero $\pi^{NIL}(G^h)$). If G^2 is the only branch with the non-zero $\pi^{NIL}(G^h)$, then $\pi_{ii'}^{NOT}$ is zero since G^2 never sends a termination notification. In general, for this case of $T_i = G^1$, and $T_{i'} = G^{|G|}$:

$$\pi_{ii'}^{NOT} = \sum_{h=3}^{|G|-1} \pi^{NIL}(G^h)$$

The possible values for $\pi_{ii'}^{NOT}$ are summarised in the algorithm below:

if $T_i = T_{i'}$ then $\pi_{ii'}^{NOT} = 0$

else let $G = G_{MIN}(T_i, T_{i'})$ (type(G) $\notin$ {LOOP, NIL, TASK})

let $T_i \in G^h, T_{i'} \in G^{h'}$

case type(G):

OR: $\pi_{ii'}^{NOT} = 0$

SEQ: $\pi_{ii'}^{NOT} = 0$

SJ: if ($h' \neq |G| \vee h = 2$) then $\pi_{ii'}^{NOT} = 0$ (termination notification impossible)

elseif $h = 1$ then $\pi_{ii'}^{NOT} = \sum_{h''=3}^{|G|-1} \pi^{NIL}(G^{h''})$

else $\pi_{ii'}^{NOT} = \pi^{EOP}(T_i, G^h)$ ($2 < h < |G|$)

This $\pi_{ii'}^{NOT}$ algorithm was used earlier in this section to find $b_{ii'}^L$ for NTGs from $\bar{b}_{ii'}^L$ (recall that $b_{ii'}^L = b_{ii'}^L(\pi_{ii'} + \pi_{ii'}^{NOT})$, where $\pi_{ii'}$ was determined in Subsection B.1.5).

The next section describes the method used to find $\bar{t}$, the mean path-length bound, and δ , those tasks contributing to $\bar{t}$, for NTGs.

B.3 Finding $\bar{t}$ and δ in Nested Task-Graphs

This section presents a "critical-path"-like algorithm used to find the mean path-length bound, $\bar{t}$, for both partially and fully allocated NTGs. In addition, this algorithm determines the tasks contributing to $\bar{t}$ (i.e., those T_i with $\delta_i = 1$). Recall that t , the mean path-length, is the same as $\bar{t}$ for $\overline{SJ}$ NTGs, and hence this algorithm also finds t for $\overline{SJ}$ task-graphs.

First, we modify the critical-path problem to apply to our NTGs. Standard critical-path problems either have time associated with executing each node or with executing each arc. In contrast with these problems, for NTGs, time is spent both on the arcs, communicating, and at the nodes, processing. NIL task-graphs represent no problem by themselves, but when they are used in OR task-graphs contained in LOOP task-graphs, complications arise. OR task-graphs represent probabilistic choices (not relevant to the problems minimising the time to complete a network of dependent activities from which critical-path analysis arose). LOOP task-graphs also represent probabilistic re-execution of a NTG, not allowed by critical-path analysis. Thus, the analysis for NTG mean path-length bounds to be described below extends critical-path analysis in several ways.

Each NTG G can be converted into a corresponding GTG $\overline{G}$ via the procedure outlined in Subsection 2.3.2, using the algorithms of sections B.1 and B.2. From $\overline{G}$, we find the task visit-ratios, $y_i \forall i \in T$, via the procedure of Subsection 5.1.1, and the task and intertask workloads per response: w_i^P and w_{ii}^L . With w_i^P and w_{ii}^L so defined, we propose the following two relaxations to simplify finding $\bar{t}(G)$. First, $\overline{G}$ ignores the synchronisation delays at any join-tasks in G , and thus using $\overline{G}$ gives a lower-bound on t for SJ NTGs. The second relaxation is to transform $\overline{G}$ into $\overline{G}'$, wherein some SJ task-graph branches are ignored, again giving a lower-bound on t for SJ NTGs. In addition to using the notation of Section B.1, the $\bar{t}(G)$ algorithm uses the following notation:

$t(G)$ is the mean path-length of G , if G is a $\overline{\text{SJ}}$ NTG (i.e., $t(G)$). Otherwise, G is a SJ NTG and $\bar{t}(G)$ is a lower-bound on the mean path-length of G .

$T(G)$ is the set of all tasks in G .

$T^{IN}(G)$ is the set of all tasks that can receive a token directly from $\text{SOP}(G)$. This may include tasks in NTGs not in G , but rather outside of G that can be reached because of NIL task-graphs,

$T^{OUT}(G)$ is the set of tasks in G which can send a token directly to $\text{EOP}(G)$,

$\text{seq}(G_1, \dots, G_n)$ is the SEQ task-graph composed of the sequence of all the argument NTGs

(i.e., if $G = \text{seq}(G_1, \dots, G_n)$ then $G_1 = G^1$, $G_n = G^n$, etc.),

$t^P(T_i)$ is the per response processing time of T_i , defined as follows. If T_i has been allocated (i.e., if $(T_i \in A \wedge a_{ij} = 1)$), then $t^P(T_i) = w_i^P / c_j^P$, where c_j^P is the speed of the processor that T_i has been allocated to. If T_i has not been allocated ($T_i \notin A$), then $t^P(T_i) = w_i^P / \max(c^P)$, which is a lower-bound for $t^P(T_i)$ after T_i is allocated.

$t^L(T_i, T_{i'})$ is the per response communication time from T_i to $T_{i'}$ on its allocated link, if any (i.e., if $(T_i, T_{i'} \in A \wedge a_{ij} = a_{i'j'} = 1, j \neq j', l_{jj'} = l)$ then $t^L(T_i, T_{i'}) = w_{ii'}^L / c_l^L$), or zero otherwise,

$t^L(S_i, S_{i'}) = \sum_{h \in S_i} \sum_{h' \in S_{i'}} t^L(h, h')$ is the total communication time from tasks in set S_i to tasks in set $S_{i'}$.

Note that both $t^P(T_i)$ and $t^L(T_i, T_{i'})$ give a lower-bound on the workloads for partially allocated task-graphs³, and thus reduce the $\bar{t}(G)$ bound for partially allocated task-graphs further than for fully allocated task-graphs. In the following two subsections, we present algorithms to find $t(G)$, $\bar{t}(G)$ and $\delta(G)$ for both $\overline{\text{SJ}}$ and SJ NTGs.

³because a lower-bound is used on the t^P of unallocated tasks and communication between T_i and $T_{i'}$ is ignored unless both T_i and $T_{i'}$ are allocated.

The algorithm for SJ NTGs in Subsection B.3.2 generates $t(G)$ for $\overline{\text{SJ}}$ NTGs, but of course less efficiently than the $\overline{\text{SJ}}$ NTG $t(G)$ algorithm of Subsection B.3.1. Each of these algorithms assumes that a task's communication work is not overlapped with its processing work (i.e., that a task may not both communicate and process at the same time).

B.3.1 Algorithm to Find $t(G)$ and $\delta(G)$ for $\overline{\text{SJ}}$ Nested Task-Graphs

For $\overline{\text{SJ}}$ NTG G , $\delta_i = 1$ for all $T_i \in T(G)$, and $t(G)$ is found below by summing all the w_{ii}^L and w_i^P workloads. The following recursive procedure finds $t(G)$ and $\delta(G)$ for $\overline{\text{SJ}}$ NTGs by descending the NTG hierarchy, adding the average per response workloads of the tasks to that of all the communication paths that could exist between tasks⁴.

case type(G): (type(G) $\neq$ SJ since G is a $\overline{\text{SJ}}$ NTG)

LOOP: $t(G) = t(G^1) + t^L(T^{OUT}(G^1), T^{IN}(G^1))$

NIL: $t(G) = 0$

OR: $t(G) = \sum_{h=1}^{|G|} t(G^h)$

SEQ: $t(G) = \sum_{h=1}^{|G|} t(G^h) + \sum_{h=2}^{|G|} t^L(T^{OUT}(G^{h-1}), T^{IN}(G^h))$

TASK: $t(G) = t^P(T(G))$

Note that $\delta(G)$ was not used in the above $t(G)$ algorithm; because G is a $\overline{\text{SJ}}$ NTG, all δ_i values are 1:

$$\delta_i = 1 \quad \forall i \mid 1 \leq i \leq |T(G)|$$

⁴nested task-graphs allow only structured patterns of $\bar{b}_{ii}^L$, communication workloads to be non-zero.

B.3.2 Algorithm to Find $\bar{t}(G)$ and $\delta(G)$ for SJ Nested Task-Graphs

The case of G being a SJ NTG is more complicated than the $\overline{\text{SJ}}$ NTG case considered above because of parallelism in the execution of the branches. To ensure that $t(G)$ is a lower-bound on the path-length of G , we consider the workloads of only one branch of any SJ task-graph⁵. To find the tightest lower-bound ignoring all but one branch of any SJ task-graph, a branch with the longest path-length is considered. All other branches have δ_i set to zero for all their member tasks. Since this $\bar{t}(G)$ algorithm is recursive, a SJ task-graph that sets all but one of its branches' δ_i values to zero may have all its tasks' δ_i values set to zero because it was contained in another SJ task-graph and the branch it was in was not considered. The following recursive procedure is similar to that for $t(G)$ given above, but works for both SJ and $\overline{\text{SJ}}$ NTGs, and sets $\delta_i = 0$ for those tasks not considered in its path-length bound $t(G)$. For $\overline{\text{SJ}}$ NTGs, $\bar{t}(G) = t(G)$.

initialise δ to a vector of ones (i.e., $\delta_i = 1 \forall i \mid 1 \leq i \leq |T(G)|$)

case type(G):

LOOP: $\bar{t}(G) = \bar{t}(G^1) + t^L(T^{OUT}(G^1), T^{IN}(G^1))$

NIL: $\bar{t}(G) = 0$

OR: $\bar{t}(G) = \sum_{h=1}^{|G|} \bar{t}(G^h)$

SEQ: $t(G) = \sum_{h=1}^{|G|} \bar{t}(G^h) + \sum_{h=2}^{|G|} t^L(T^{OUT}(G^{h-1}), T^{IN}(G^h))$

SJ: $G_h \equiv \begin{cases} \text{seq}(G^1, G^{|G|}) & \text{if type}(G^h) = \text{NIL} \\ \text{seq}(G^1, G^h, G^{|G|}) & \text{otherwise} \end{cases} \forall h \mid 1 < h < |G|$

$h^* \in \{2, \dots, |G| - 1\}, \bar{t}(G_{h^*}) = \max_{1 < h < |G|} \bar{t}(G_h) \quad (G^{h^*} \text{ is the branch kept})$

⁵including the workloads of all the branches of a SJ task-graph would not consider any parallelism in the execution of the branches.

$$\forall h \in \{2, \dots, |G| - 1\} \mid h \neq h^*, \delta_i \equiv 0 \forall T_i \in T(G^h)$$

$$\bar{t}(G) = \bar{t}(G_{h^*})$$

$$\text{TASK: } \bar{t}(G) = \bar{t}^P(T(G))$$

For SJ NTG G , removing the tasks with $\delta_i = 0$ and the arcs connected to them gives a related $\overline{\text{SJ}}$ NTG⁶, called G' , for which $\bar{t}(G') = \bar{t}(G)$. $\overline{G}$, the GTG derived from G earlier in this section can similarly be converted into $\overline{G}'$, the GTG derived from G' , by simply removing the tasks with $\delta_i = 0$ and all arcs connected to them. Figure B.1 summarises this relationship between G and $\overline{G}'$. Although two paths exist from G to $\overline{G}'$, they both first apply Subsection 2.3.2 to find the workloads and then apply the algorithm of this subsection to find δ_i . Note that to preserve the control flow of the NTG⁷, when going from G to G' , $\pi_{ii'}$ should everywhere be replaced by $\pi_{ii'} + \pi_{ii'}^{\text{NOT}}$. Similarly, when going from $\overline{G}$ to $\overline{G}'$, $\pi_{ii'}$ should also be replaced by $\pi_{ii'} + \pi_{ii'}^{\text{NOT}}$, where $\pi_{ii'}^{\text{NOT}}$ is found using NTG G .

Figure B.1: Relationships between Task-Graphs G , $\overline{G}$, G' and $\overline{G}'$

This recursive $\bar{t}(G)$ procedure can be used to evaluate both partial and full allocations of NTGs for the path-length bound discussed in Subsection 5.3.4.

⁶all the SJ task-graphs in G are effectively converted into SEQ task-graphs (i.e., all but one branch of the SJ task-graph is removed) or deleted (i.e., had $\delta_i = 0$ for all its tasks) in G' , as in the $\bar{t}(G)$ algorithm given above.

⁷because the upper SJ task-graph branch, which conveys all tokens to the join-task, may have δ_i for all its tasks set to zero by the above algorithm.

END

22|1 2|92|

FIN