View metadata, citation and similar papers at core.ac.uk

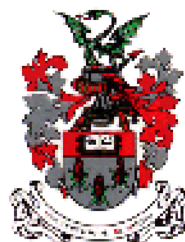
brought to you by CORE

provided by Universidade do Minho: RepositoriUM

Luís Paulo Peixoto dos Santos

Application Level RunTime Load Management: A Bayesian Approach

Tese submetida à Escola de Engenharia da Universidade do Minho
para a obtenção do grau de Doutor em Informática
(Área de Especialização em Engenharia de Computadores)



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Braga — February, 2001

Luís Paulo Peixoto dos Santos

Application Level RunTime Load Management: A Bayesian Approach

Tese submetida à Escola de Engenharia da Universidade do Minho
para a obtenção do grau de Doutor em Informática,
Área de Especialização em Engenharia de Computadores

Dissertação realizada sob a orientação do
Prof. Doutor Alberto José Gonçalves de Carvalho Proença,
Professor Catedrático do Departamento de Informática da
Escola de Engenharia da Universidade do Minho

Universidade do Minho

Escola de Engenharia

Departamento de Informática

Braga — February, 2001

É autorizada a reprodução integral desta tese, apenas para efeitos de investigação, mediante declaração escrita do interessado, que a tal se compromete.

Título: Application Level RunTime Load Management: A Bayesian Approach

Autor: Luís Paulo Peixoto dos Santos

Tese de Doutoramento em Informática, Especialidade em Engenharia de Computadores, Departamento de Informática, Escola de Engenharia, Universidade do Minho

Candidatura a Doutoramento aceite pelo Conselho Científico da Escola de Engenharia da Universidade do Minho em 08 de Fevereiro de 1995

Orientador: Alberto José Gonçalves de Carvalho Proença

Conclusão: Fevereiro de 2001

© 2001

"Senhor,

posto que o capitão-mor desta vossa frota e assim os outros capitães escrevam a Vossa Alteza a nova do achamento desta vossa terra nova, que se ora nesta navegação achou, não deixarei também de dar disso minha conta a Vossa Alteza, assim como em melhor puder, ainda que para o bem contar e falar o saiba pior que todos fazer. Mas tome Vossa Alteza a minha ignorância por boa vontade, a qual, bem certo, creia que por afremosentar nem afear haja aqui de pôr mais do que aquilo que vi e me pareceu. Da marinhagem e singraduras do caminho não darei aqui conta a Vossa Alteza, [...]"

Carta do Achamento do Brasil
Pêro Vaz de Caminha
1 de Maio de 1500

Acknowledgements[1]

A work such as a PhD thesis can not be done without the contribution of many persons besides myself. Although I try, in the next few lines, to express my gratitude to all of those that contributed to this thesis, I am certain that some will be forgotten; to these I sincerely apologise.

First of all, I would like to thank Prof. Alberto Proença, who supervised my research. His suggestions definitely contributed to the final quality of this work, and his guidance was invaluable in writing the thesis.

Many people at the Departamento de Informática, Universidade do Minho, contributed to this thesis, either in direct or indirect ways. I want to thank specially to my colleagues at the Grupo de Engenharia de Computadores, namely, Eng. João Luís Sobral, Prof. João Miguel Fernandes, Eng. António Esteves and Prof. António Pina, for their company and countless fruitful discussions.

I must also thank my family, particularly, my mother, my father and my sister, who believed in my ability to successfully complete this thesis and helped me with their love and encouragement.

And last, but not the least, I want to thank Cristina, for being always there, even in the most difficult moments. I have to apologise for the many hours and weekends I could not pay her the attention she deserves.

Braga, February, 2001

[1]This work has been partially supported by The Fundação para a Ciência e Tecnologia, grant PRAXIS XXI 2/2.1/TIT/1557/95.

Abstract

Affordable parallel computing on distributed shared systems requires novel approaches to manage the runtime load distribution, since current algorithms fall below expectations. The efficient execution of irregular parallel applications, on dynamically shared computing clusters, has an unpredictable dynamic behaviour, due both to the application requirements and to the available system's resources. This thesis addresses the explicit inclusion of the uncertainty an application level scheduling agent has about the environment, on its internal model of the world and on its decision making mechanism. Bayesian decision networks are introduced and a generic framework is proposed for application level scheduling, where a probabilistic inference algorithm helps the scheduler to efficiently make decisions with improved predictions, based on available incomplete and aged measured data. An application level performance model and associated metrics (performance, environment and overheads) are proposed to obtain application and system behaviour estimates, to include in the scheduling agent's model and to help the evaluation. To verify that this novel approach improves the overall application execution time and the scheduling efficiency, a parallel ray tracer was developed as a message passing irregular data parallel application, and an execution model prototype was built to run on a seven time–shared nodes computing cluster, with dynamically variable synthetic workloads. To assess the effectiveness of the load management, the stochastic scheduler was evaluated rendering several complex scenes, and compared with three reference scheduling strategies: a uniform work distribution, a demand driven work allocation and a sensor based deterministic scheduling strategy. The evaluation results show considerable performance improvements over blind strategies, and stress the decision network based scheduler improvements over the sensor based deterministic approach of identical complexity.

Resumo

A computação paralela em sistemas distribuídos partilhados exige novas abordagens ao problema da gestão da carga computacional, uma vez que os algoritmos existentes ficam aquém das expectativas. A execução eficiente de aplicações paralelas irregulares em *clusters* de computadores partilhados dinamicamente exibe um comportamento imprevisível, devido à variabilidade dos requisitos da aplicação e da disponibilidade dos recursos do sistema. Esta tese investiga as vantagens de incluir explicitamente no modelo de execução de um escalonador ao nível da aplicação a incerteza que este tem sobre o estado do ambiente em cada instante. Propõe–se um mecanismo de decisão baseado em redes de decisão de Bayes, complementado por uma estrutura genérica para estas redes, vocacionada para o escalonamento ao nível da aplicação; a utilização de um algoritmo de inferência probabilística permite ao escalonador tomar decisões mais eficazes, baseadas em previsões estocásticas das consequências destas decisões, geradas a partir de informação incompleta e desactualizada sobre o estado do ambiente. É proposto um modelo de desempenho da aplicação e respectivas métricas, que permite prever o comportamento da aplicação e do sistema distribuído; estas métricas são utilizadas quer no mecanismo de decisão do escalonador, quer para avaliar o desempenho do mesmo. Para verificar se esta abordagem contribui para melhorar o tempo de execução das aplicações e a eficiência do escalonador, foi desenvolvido um *ray tracer* paralelo, representativo de uma classe de aplicações baseada em passagem de mensagens com paralelismo no domínio dos dados e comportamento irregular. Este protótipo foi executado num *cluster* com sete nodos partilhados no tempo e submetidos a vários padrões sintéticos de cargas de trabalho dinâmicas. Para avaliar a eficácia da gestão de carga proposta, o desempenho do escalonador estocástico foi comparado com três escalonadores de referência: uma distribuição estática e uniforme da carga, uma estratégia orientada ao pedido e uma política de escalonamento determinística baseada em sensores. Os resultados obtidos demonstram que estratégias dinâmicas baseadas em sensores obtêm grandes melhorias de desempenho sobre estratégias que não usam informação sobre o estado do ambiente, e realçam as vantagens do escalonador estocástico relativamente a um escalonador determinístico com um nível de complexidade equivalente.

Preface

A PhD thesis must be a rigorous and original work, which contributes to extend or clarify the scientific knowledge on a given area. Hence, it is a scientific work and must adhere to a set of recommendations proposed by the scientific method.

Knowledge can be defined as intellectual and internal models that man creates to represent reality. These models enable the classification and organisation of the phenomenons observed in Nature. Scientific knowledge distinguishes itself from other kinds of knowledge by trying to classify these phenomenons based on explicative principles and by trying to achieve two ideals: rationality and objectivity.

The rationality ideal requires that all scientific theories are coherent among themselves, and do not contradict each others. This is referred to as the syntactic truth. The objectivity ideal requires that scientific theories, as theoretic models representing reality, do so with fidelity and accuracy. This is referred to as the semantic truth. Objectivity requires the possibility of experimentally testing all theories and hypothesis, as well as the possibility of a critical and intersubjective assessment by the scientific community, as a defense mechanism against the researcher subjective beliefs and expectations (pragmatic truth).

In order to produce scientific knowledge, the researcher must adhere to a set of recommendations proposed by the scientific method. However, the scientific method is not a set of magic and normative rules that guarantee that by applying them accurate and absolute scientific knowledge is produced. In fact, if these rules were magic they could not be referred to as scientific, and scientific knowledge is never absolute, rather it constantly maintains its hypothetical nature, and must be revised whenever new data becomes available. The scientific method is a critical way of producing knowledge, consisting on the formulation of well founded hypothesis, on the possibility of experimentally testing them and on submitting the results to the assessment of the scientific community. The correct scientific attitude is that which is critical of all results and established theories [4, 90]. The scientific method consists on three phases:

Problem's identification and delimitation — To begin its work, the researcher must clearly identify and delimitate the problem being tackled. It must be expressed as an interrogative statement, which questions the possible relations that may exist among

at least two known variables. Hence, the problem's delimitation defines the limits of the doubt, and contains in itself clues about the relations that may exist in its answer. If the problem is not adequately delimitated, it may be impossible to tackle.

Hypothesis formulation — The hypothesis is formulated by the researcher as an unequivocal statement, free of ambiguities. The hypothesis is a proposal for the problem's solution, or explanation of the question being tackled, and depends on the conjectures made by the researcher, given all the knowledge available. It serves as a guide for all the experimental work.

Experimentation and results' analysis — After proposing possible and plausible answers to the problem, the researcher must plan and carry out experimentations that allow him to confront the hypothesis with real data. The experimentation submits the hypothesis to a systematic and severe critique, in order to assess its correspondence with facts — semantic truth. However, an hypothesis is an universal statement, while an experimentation's result is a singular fact, or statement. It can not, therefore, conclusively prove that the hypothesis is true. It can only either conclusively prove that it is false, or corroborate it. Hence, an hypothesis can either be rejected or corroborated, but it can never be conclusively confirmed in the positive sense. The scientific knowledge always maintains its hypothetical nature.

These phases determined the structure of this thesis. Part I identifies and delimitates the problem being tackled, puts forward an hypothesis to solve this problem and presents the knowledge and related work required to both understand the problem and formulate the hypothesis. Part II presents the experimentations performed in order to either reject or corroborate this hypothesis.

Unfortunately, I did not have such an understanding of the scientific method when this work first started. Therefore, my working method did not always followed the most correct path in the right order. It slowly converged to this systematic way of proceeding, as I was learning how to carry out a scientific work. On the other hand, maybe this was not so unfortunate. In fact, more than learning about scheduling, I can say that I learned about the process of producing scientific knowledge.

Since there are no normative rules in the scientific method, the problem's identification and delimitation, the hypothesis's formulation and the experimentation planning require a lot of imagination and motivation from the researcher. These requirements are better satisfied if the working environment promotes the discussion and exchange of ideas among researchers. In the environment where this research took place, the discussion of ideas and presentation of the researchers' work could be more actively pursued. There are no organised discussion forums to debate ongoing research, which often leads to solitary work that could otherwise be richer and more motivating.

Contents

1 Introduction **1**

I Problem's Identification and Hypothesis Formulation **5**

2 Load Management: Research Goals **9**

2.1 The Load Management Problem 10

2.1.1 Definition 10

2.1.2 Objectives and Constraints 12

2.2 Application Level Scheduling 13

2.3 Static vs. Dynamic Policies 15

2.4 Incompleteness of Information and Uncertainty 18

2.4.1 Handling Uncertain Knowledge 20

2.4.2 Related Work 21

2.5 Summary 22

3 Load Management: Basic Concepts **25**

3.1 Degree of Balancing 26

3.2 Level of Complexity 26

3.3 Static versus Dynamic Policies 28

3.4 Deterministic versus Stochastic Strategies 31

3.5 Centralised versus Distributed Approaches 32

3.6 Scheduling Policy Components 34
3.6.1 Information Policy 35
3.6.2 Transfer Policy 38
3.6.3 Selection Policy 39
3.6.4 Location Policy 41
3.7 Load Management Evaluation 41
3.7.1 Performance and Efficiency 42
3.7.2 Cost Analysis 43
3.7.3 Scalability 45
3.7.4 Stability 46
3.8 Summary 48

4 Load Management: Algorithms **51**
4.1 Classification of Scheduling Algorithms 52
4.1.1 Decision Base × Migration Space 52
4.1.2 Casavant's Taxonomy 53
4.1.3 Families of Strategies 54
4.1.4 Load, Action and Solution Model 56
4.1.5 The ESR Classification Scheme 57
4.1.6 Selection of a Classification Scheme 59
4.2 Scheduling Policies 61
4.2.1 Centralised Algorithms 61
4.2.2 Nearest–Neighbour Algorithms 62
4.2.3 Random Location Algorithm 64
4.2.4 Probing and Bidding 65
4.2.5 Flexible Load Sharing Algorithm 68
4.2.6 Distributed Clustering Algorithms 68

4.2.7 Stochastic Learning Automata 69
4.2.8 Physics Based Models 70
4.2.9 Economy Based Models 71
4.3 Distributed Job Management Systems 72
4.4 Summary 74

5 Handling Uncertainty **77**
5.1 Notation 78
5.2 Beliefs Expressed as Probabilities 79
5.3 Probabilistic Models 80
5.3.1 The Joint Distribution 80
5.3.2 Local Structure and Conditional Independence 82
5.3.3 Causality 85
5.4 Bayesian Networks 85
5.4.1 Conditional Independence and d–separation 87
5.4.2 Probabilistic Inference 88
5.4.3 Bayes' Rule 89
5.4.4 Pearl's Probabilistic Inference Algorithm 90
5.4.5 Sensor Model 91
5.5 Making Decisions 92
5.5.1 Preferences and Utilities 92
5.5.2 Decision Networks 94
5.6 Knowledge Engineering 95
5.6.1 Determine the Scope of the Problem 96
5.6.2 Identify Direct Dependencies 96
5.6.3 Assign Probabilities 96
5.6.4 Assign Utilities 98

5.6.5 Model Refinement and Sensitivity Analysis 98
5.7 Applying Decision Networks to a Dynamic Scheduler 99
5.7.1 Generic Structure 99
5.7.2 The Decision Making Process 104
5.8 Summary 104

II Hypothesis' Verification 107

6 Methodology 111

6.1 The Distributed System 111
6.2 Selection of a Case Study 113
6.3 The Problem's ESR Classification 115
6.4 Performance Modelling 116
6.4.1 Performance Metrics 116
6.4.2 Environment Metrics 116
6.4.3 Scheduling Overhead Metrics 117
6.5 Reference Scheduling Strategies 120
6.5.1 Uniform Work Distribution 121
6.5.2 Demand–Driven Work Allocation 122
6.5.3 Sensor Based Deterministic Strategy 122
6.5.4 Decision Network Based Strategy 124
6.6 Synthetic Background Workload 125
6.7 Summary 129

7 Ray Tracing: a Case Study 131

7.1 Ray Tracing Algorithm 132
7.2 Illumination Model 135
7.2.1 Local Illumination Model 135

7.2.2 Global Illumination Model 139
7.2.3 Ray Tracing Rendering Equation 140
7.3 Some Ray Tracing Deficiencies 141
7.4 Acceleration Techniques 142
7.5 Parallel Ray Tracing 144
7.6 PaRT – Parallel Ray Tracer 146
7.6.1 PaRT's Architecture 147
7.7 Summary 149

8 Experimental Results **151**
8.1 Experimental Data Sets 151
8.2 Estimating the Tasks' Requirements 152
8.3 Performance Modelling 159
8.3.1 Performance Metrics 159
8.3.2 Environment metrics 159
8.3.3 Scheduling Overhead Metrics 161
8.4 Reference Scheduling Strategies 163
8.4.1 Sensor Based Deterministic Strategy 164
8.5 Decision Network Based Strategy 168
8.5.1 Laying out the Network's Topology 169
8.5.2 Assign Probabilities 176
8.5.3 Assign Utilities 185
8.5.4 Sensitivity Analysis 185
8.6 Results' Analysis 189
8.6.1 Dedicated Mode 191
8.6.2 Background Workload 196
8.6.3 Using Previous Knowledge about the Background Workload 199

8.6.4 Discussion 203
8.7 Summary 205

9 Conclusions **207**
9.1 Discussion 207
9.2 Future Work 210

Appendices **213**

A Glossary **215**
A.1 Decision Theory 215
A.2 Load Management 218

B Propagation Rules for Bayesian Networks **225**
B.1 Notation 225
B.2 Propagation Rules for Chains 227
B.3 Propagation Rules for Trees 228
B.4 Propagation Rules for Polytrees 231
B.5 An Example: The Burglary Alarm 233
B.5.1 The Probabilistic Model 233
B.5.2 Inference Algorithm without Evidence 235
B.5.3 Belief Distribution with Evidence 236

C PaRT 2.1 : User's Manual **239**
C.1 Introduction 239
C.2 Illumination Model 240
C.2.1 Rendering Equation 241
C.3 Supported Primitives 242
C.4 Neutral File Format 242

C.5 PaRT 2.1 Extensions to the Neutral File Format 246
C.6 PaRT 2.1 Usage 247
C.6.1 Installation and Requirements 247
C.6.2 Usage 248

D Results **251**

Bibliography **261**

Index **277**

List of Figures

2.1 The scheduling problem 10

2.2 An agent that perceives its environment and acts upon it 11

4.1 Decision base × Migration space 53

4.2 Casavant's hierarchical partial taxonomy 54

5.1 A simple Bayesian network 87

5.2 A simple Bayesian network 87

5.3 The burglary alarm's Bayesian network 89

5.4 Thermometer sensor model 91

5.5 Extended sensor model 92

5.6 Divorcing $\{A_1, A_2\}$ from $\{A_3, A_4\}$ 97

5.7 A generic structure for a scheduling decision network 100

5.8 The system sharing level versus the application regularity space 103

5.9 The decision making process 105

6.1 Three different distributed system's architectures 112

6.2 Background workload: the system sharing level versus the application regularity space 126

6.3 Computing throughput variation with different synthetic background workloads 128

7.1 Primary and shadow rays 133

7.2 Shadow rays and occlusion 134

7.3 Primary (V), Shadow (L), Reflected (R) and Transmitted (T) rays 134
7.4 Diffuse reflection 136
7.5 Specular reflection 138
7.6 Snell's Law 140
7.7 PaRT's architecture 147

8.1 SPD scenes – balls3, balls3c, balls4pv, teapot9 153
8.2 Execution time distribution – balls3, balls3c, balls4pv, teapot9 155
8.3 Balls3c: Estimated versus actual execution time 158
8.4 Teapot9: improvements with extended information policy 169
8.5 Ray tracing: the system sharing level versus the application regularity space 170
8.6 Bayesian network – the intersection rate sensor model 171
8.7 Decision network – the resources' capacity block 173
8.8 Decision network – the tasks' requirements block 174
8.9 The complete decision network 175
8.10 InfoIr's belief distribution given Ir = Medium 178
8.11 InfoFW's belief distribution given FW = Forn1 181
8.12 NewBalance's belief distribution given FWRatio=aMHigherb 186
8.13 NewBalance's belief distribution given FWRatio=aHigherb 187
8.14 Sensitivity analysis 191
8.15 Execution time with different scheduling strategies 193
8.16 Performance improvements with different scheduling strategies 194
8.17 Performance improvement of DN relative to det 195
8.18 TTidle% and TTdata% with *det* and DN scheduling strategies 196
8.19 Execution time with different background workloads (7 nodes) 197
8.20 Performance improvement with different background workloads (7 nodes) . 198
8.21 Performance improvement of DN relative to det (7 nodes). 199

8.22 Teapot9: performance improvement with heavy background workload . . . 199
8.23 TTidle% and TTdata% with *det* and DN scheduling strategies and different background workloads 200
8.24 Performance improvements with adaptive stochastic approach 202
8.25 Performance improvements relative to demand driven with 7 nodes 203

9.1 The decision network approach effectiveness 209

B.1 A Bayesian network with a chain topology 228
B.2 Structure of an individual node on a chain network 228
B.3 A Bayesian network with a tree topology 229
B.4 Structure of an individual node on a tree network 231
B.5 A Bayesian network with a polytree topology 231
B.6 The burglary alarm's Bayesian network 234

List of Tables

3.1 Direct costs induced by the scheduler 44

3.2 Direct costs distribution across the scheduling policy components 45

4.1 The ESR Scheduling Problem Classification Attributes 58

4.2 The ESR Scheduling Strategies Classification Attributes 60

4.3 Criteria for DJMS's evaluation 75

5.1 The joint probability distribution: an example 81

5.2 Burglary Alarm Example: $\mathbf{CPT}(A|B, E)$ 83

5.3 Burglary Alarm Example: $\mathbf{P}(B)$ and $\mathbf{P}(E)$ 84

5.4 Burglary Alarm Example: $\mathbf{P}(A, B, E)$ 84

6.1 Application space divided according to application's characteristics 114

6.2 The Scheduling Strategies' ESR classification 125

6.3 Synthetic background workloads 127

7.1 Material's properties 141

8.1 Scenes' main characteristics 152

8.2 Comparison of arithmetic and exponentially weighed averages with different α161

8.3 Intersection rate sensor model — $\mathbf{CPT}(InfoIr|AgeIr, Ir)$ 179

8.4 $\mathbf{CPT}(IrRatio|Ira, Irb)$ 180

8.5 Foreground workload sensor model — $\mathbf{CPT}(InfoFW|AgeFW, FW)$ . . . 181

8.6 $\mathbf{CPT}(FWRatio|FWa, FWb)$ 182

8.7 $\mathbf{CPT}(NB|FWRatio = aEqualb, IrRatio, Transfer)$ 184

8.8 $\mathbf{CPT}(FWRatio|FWa, FWb)$ 188

8.9 $\mathbf{CPT}(InfoIr|AgeIr, Ir, IrS)$ — sensitivity analysis 190

8.10 $\mathbf{CPT}(InfoFW|AgeFW, FW, FWS)$ — sensitivity analysis 192

B.1 Burglary Alarm Example: $\mathbf{P}(B)$ and $\mathbf{P}(E)$ 234

B.2 Burglary Alarm Example: $\mathbf{CPT}(A|B, E)$ 234

B.3 Burglary Alarm Example: $\mathbf{CPT}(N1|A)$ and $\mathbf{CPT}(N2|A)$ 235

C.1 Set of parameters describing the material properties 241

D.1 Columns's Labels 251

D.2 Balls3: dedicated mode results 252

D.3 Balls3c: dedicated mode results 253

D.4 Teapot9: dedicated mode results 254

D.5 Balls4pv: dedicated mode results 255

D.6 Balls3: results with different background workloads (7 nodes) 256

D.7 Balls3c: results with different background workloads (7 nodes) 257

D.8 Teapot9: results with different background workloads (7 nodes) 258

D.9 Balls4pv: results with different background workloads (7 nodes) 259

Chapter 1

Introduction

Parallel computing on distributed systems is becoming increasingly popular with affordable cluster computers and the exploitation of unused cycle times on interconnected workstations. There are many goals targeted by this computing paradigm: reduce applications' turnaround time, maximise throughput, increase system's utilisation or reliability, etc. These systems offer a theoretical processing power that is the sum of the capacity of the individual resources. However, in order to obtain a performance close to this theoretical limit, the workload must be effectively distributed over the available resources, to exploit and profit from their multiplicity [22]. Ensuring a good correspondence between the workload's structure — both code and data — and the characteristics of the distributed system is known as the load management, or scheduling, problem.

The complexity of each particular scheduling problem depends on the complexity of the environment, both the workload and the computing system, upon which the scheduler is required to act. The applications may exhibit either predictable or unpredictable computational requirements and data access patterns. Predictable behaviours simplify the scheduler's job, since resources can be allocated to parallel tasks based on accurate data. However, many applications do not exhibit a predictable behaviour, and the scheduler's decisions must be made based on inaccurate predictions of the workload's future requirements. These predictions are usually a function of the workload recent past behaviour.

The distributed computing system may also exhibit an unpredictable behaviour. This is specially true when systems are shared among several users and applications. The scheduler can not predict the exact overall workload submitted to the distributed system, and this workload can change frequently as a consequence of the users' activities. The performance of the distributed system resources, as perceived by the application, varies in time, as a result of several applications competing for the same set of resources. To satisfy its performance requirements, the scheduler must correctly orchestrate the distribution of the dynamically varying workload on the distributed system's resources.

This thesis addresses the problem of effectively and efficiently scheduling applications on shared distributed computing systems, particularly when the applications exhibit unpredictable computing requirements and data access patterns.

The pertinence of the scheduling problem grows as the complexity of the distributed systems increases. With the advent of the Internet and other global computing infrastructures and services — such as the Grid [50] — the distributed systems' size and heterogeneity are increasing dramatically, both at the resources' level and the range of services offered. Furthermore, the resources' availability in these computing environments changes with time in unpredictable ways, rendering the scheduler's task even more difficult. The scheduler's role is crucial in such systems, where, in addition to raw performance, cost and ownership must be taken into account, among other issues [32]. Although the present work does not address such complex distributed systems, some of the results presented may be used as preliminary work and helpful hints for such cases.

Optimally scheduling a workload on a distributed system is known to be a NP–complete problem in the general case [19, 37]. One of the reasons why the scheduling problem can not be optimally solved is the environment's limited measurability: both the application's requirements and the environment's workload can not be accurately predicted. The scheduler has to make decisions based on inaccurate information. Since the environment's complexity is high, the scheduler must use a simplified internal execution model of the world for its decision making mechanism. This simplified model summarises the objects and relationships holding on the real world, neglecting some of them. Furthermore, some of the environment's relevant aspects are inaccessible, in the sense that it is too expensive, or even impossible, to get exact and updated information about them. Therefore, it is not possible neither to have an exact, accurate and updated knowledge about the global state of the environment at any given instant, nor to make exact predictions about near future behaviour. The scheduler is required to decide and act under conditions of uncertainty about past, present and future system's states and workload profiles.

The problem tackled by this thesis is: *should the scheduling agent explicitly include the uncertainty it has about the environment on its internal model of the world and on its decision making mechanism, in order to more tightly meet its performance requirements?*

The problem of handling uncertainty on computational models of real world problems is usually solved by representing certainty, or belief, on a given proposition, or event, using probabilistic values and by combining beliefs on a set of propositions using probabilistic theory. Preferences among the outcomes of the alternative decisions available to the decision making agent are usually represented using utilities. Decision theory combines probabilities and utilities to evaluate alternative actions. One of the tools proposed by decision theory for rational decision making are decision networks, also called influence diagrams [69, 127, 143]. Decision networks require a probabilistic model of the world, where

direct causal relationships and probabilistic conditional independences among the model's variables are encoded in the network's topology. With the use of additional decision and utility variables, decision networks provide coherent prescriptions for rational decision making under uncertainty. Decision networks allow the inclusion, on the reasoning process, of the uncertainty about the environment's current state and about the consequences of the scheduling agent's selected actions, i.e., the environment's next state after each scheduling event. Furthermore, they provide an automated process of computing the expected utility of each action, therefore enabling rational decision making by selecting the action which maximises expected utility.

The hypothesis put forward by this thesis is: *decision networks, if applied to the scheduling agent's execution model and decision making mechanism, may improve its effectiveness and help to overcome the problems caused by uncertainty.*

The use of decision networks to dynamically schedule a parallel application among the nodes of a distributed shared system is an original approach and the main contribution of this thesis; the author has no knowledge of any related work where decision networks have been applied to solve this type of problem.

To limit the scope of this research, the problem was restricted to application level scheduling of divisible workloads in distributed shared systems, aiming to minimise the overall application execution time. The evaluation results were obtained on a multi–user cluster of seven workstations, interconnected by a high performance Myrinet network. Application level scheduling is performed by the application itself, in order to meet its own performance goals, while in competition with other applications that may be sharing the same set of resources. These are referred to as background workload. Divisible loads are those which can be divided into any number of segments of any fractional size, for example, by using data domain decomposition. The actual experiments were conducted using a parallel ray tracer as a case study. Image space decomposition is used, therefore the workload is arbitrarily divisible down to the finest grain of rendering a single pixel.

To verify the hypothesis suitability to solve the problem being tackled, the decision network based scheduler was submitted to a set of experimental tests and confronted with other alternative decision making mechanisms, referred to as reference scheduling strategies. The most relevant scheduling strategies focus their efforts in dynamic, runtime tasks migrations, rather than trying to generate an improved initial partitioning of workload among the processing nodes. This is mainly due to the high variability in capacity exhibited by typical shared systems. Their effectiveness, understood as the level of performance achieved, is studied and compared. Since this application level scheduler's performance goal is to minimise execution time, the performance metric used to compare the various schedulers' effectiveness on a parallel ray tracer is the time required to render some predefined sample scenes. The schedulers' efficiency, which reflects the scheduling overheads imposed upon

the distributed system, is also measured and compared. These overheads are classified as either direct or indirect overheads, depending on whether they result directly from the scheduler's activity or are indirect consequences of the scheduler's decisions.

The main goal of this thesis is to verify if decision networks can be successfully applied to the scheduler's decision making mechanism, improving its effectiveness by overcoming the problems associated with uncertainty about the environment's global state at each instant.

This work makes the following contributions to the scientific knowledge about the scheduling problem:

- proposes a generic Bayesian decision network structure, to dynamically schedule applications on distributed shared systems (chapter 5);
- suggests a performance model, oriented to application level scheduling on distributed shared systems, to assess both the scheduler's effectiveness and efficiency (chapter 6);
- proposes an execution model for a particular case study — parallel ray tracing — and experimentally validates it (chapter 8).

This thesis is structured in two parts. Part I presents the main research goals pursued by this work, describes and clarifies the main concepts related to scheduling on distributed systems, introduces the reader to the scheduling algorithms most commonly found on the literature and describes decision networks. Part II presents the methodology adopted to submit the hypothesis to experimental tests, describes the ray tracer used as a case study, presents the experiments actually performed and analyses their results. Finally, chapter 9 extracts some conclusions from the results and suggests directions for future work. The appendices include a glossary of the most relevant terminology, a detailed description of the probabilistic inference mechanism used by the stochastic schedulers, a users' manual of the ray tracer developed throughout this work and an extensive list of all experimental results.

Although this thesis was carefully read several times to eliminate all orthographic and grammar errors, this goal may not have been fully accomplished.

Part I

Problem's Identification and Hypothesis Formulation

Part I

This part presents the reasoning that led to the identification of the problem tackled by this thesis and to the formulation of the corresponding hypothesis. The global scientific knowledge, required to better understand the scheduling problem, is systematically presented, as well as the theoretical foundations of decision networks, which allowed the formulation of the hypothesis.

Chapter 2 discusses the issues that motivated the work developed throughout this thesis. Uncertainty and incompleteness of information are identified as the main reasons why an application level scheduler operating on a shared distributed system may fail to fulfill its performance goals.

Chapter 3 introduces some fundamental concepts related to load management, and presents an analysis of the overheads associated with scheduling. A thorough understanding of the structure of scheduling overheads is essential in order to design an effective and efficient scheduler.

Chapter 4 reviews some scheduling policies found on the specialised literature, discusses some fundamental attributes that a systematic classification scheme must exhibit and selects one such scheme to use throughout this thesis.

Chapter 5 presents probabilities as a mean to express partial beliefs on propositions, introduces probabilistic models and decision networks, describes the steps required to assemble a coherent and computable decision basis and suggests a generic structure for a decision network applied to the scheduling problem.

Chapter 2

Load Management: Research Goals

Contents

2.1 The Load Management Problem **10**
2.2 Application Level Scheduling **13**
2.3 Static vs. Dynamic Policies **15**
2.4 Incompleteness of Information and Uncertainty **18**
2.5 Summary **22**

Parallel or distributed systems are composed of a multiplicity of heterogeneous resources, which cooperate to solve one or more computational jobs. In such systems, a parallel application component (hereby designated as task) may be waiting for service at one resource, while other resources are idle [9, 102]. This idle–while–waiting condition suggests that the capabilities of such systems may be more efficiently exploited by adequately mapping the tasks onto the available resources. This mapping, which can be either static or dynamic, is the load manager's role.

Hwang and Xu [73] identify four sources of overhead for parallel applications, namely, parallelism management, communication, synchronisation and load imbalance overheads. The load imbalance overhead is incurred when some resources are idle, while others are busy. The load manager is responsible for minimising this overhead, while keeping other overheads (e.g. communication) within acceptable levels [37]. The major goal of load management, however, is not to equalise the loads on the nodes of a parallel or distributed computing system, but to optimise some set of predefined performance goals [11, 178].

This chapter discusses the issues that motivated the work developed throughout this thesis. The load management problem is characterised as a scheduling problem, and some fundamental requirements and constraints imposed upon an application level scheduler operating on a shared distributed system are identified.

2.1 The Load Management Problem

2.1.1 Definition

The problem of mapping a set of tasks onto a set of resources is a scheduling problem. Scheduling resources on a parallel or distributed system is a two–dimensional scheduling problem [9, 25, 172]. Local, or intranode, scheduling is concerned with scheduling within a single node. It occurs at various sub–levels, such as through the memory hierarchy, at the device and functional level, and among processors when the node is a multiprocessor system. Global, or internode, scheduling is a level above this, and is concerned with scheduling among nodes [124]. This thesis addresses global scheduling, assuming that local scheduling is performed by each node's operating system.

The scheduling problem has five components: the tasks (or relevant events related to them), the distributed system, the performance requirements, the schedule and the scheduler. The tasks' events, system's characteristics and performance requirements are the input to the scheduler, or load manager, and the mapping, or schedule, is its output.

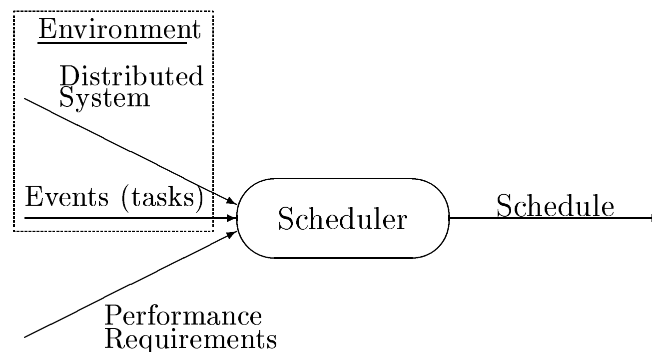


Figure 2.1: The scheduling problem

The distributed system related input refers to all system's characteristics that impact on the schedule, such as the available resources' absolute capabilities and current state. The events are messages related to the workload. These inform the scheduler of the arrival of new tasks, removal of old ones or just report a task's current state.

The distributed system and the workload, together, constitute the environment upon which the scheduler must act to achieve its performance requirements. In order to improve their effectiveness, most schedulers maintain some internal representation of the environment's current state, which is updated whenever new information about the distributed system or workload reaches the scheduler. This information is referred to as the *environment's metrics*. These metrics, which are acquired through the scheduler's sensors, determine the set of the environment's particular aspects that can be perceived, or measured, by the

scheduler. The range of available metrics and their accuracy are strongly correlated with the scheduler's effectiveness.

The requirements specify which goals the scheduler must pursue. These can include throughput maximisation, execution time minimisation, etc. (section 2.1.2).

The schedule is the actual mapping of tasks onto resources, as generated by the scheduler. It may either be an ordered list of pairs (tasks, resources) generated before execution time, if all tasks and systems' characteristics are previously known, or be dynamically generated at runtime by a set of rules. These rules specify correcting actions that redistribute tasks over the system such that some requirement is more tightly met (section 2.3). The set of actions available to the scheduler constitute the scheduler's effectual capabilities.

The scheduler is responsible for generating the schedule, based on what it knows about the environment's current state and on its performance requirements. To be able to achieve these requirements, the scheduler must have an internal execution model of the world, that adequately represents the distributed system and the workload's most relevant aspects and respective interrelationships. This execution model is used to generate estimates of future behaviours that are used as inputs to the scheduler's decision making mechanism. Being a computational representation of a real world problem, the execution model must be a simplification of the objects and relations that hold on the universe being considered. This simplification may result in inaccuracies on the generated estimates, that must be properly handled by the scheduler, so that its effectiveness does not get compromised.

The scheduler receives as input information about the system's characteristics and current state, and information about the workload. Based on this information and on its internal execution model, which represent everything the scheduler knows about the environment, it must select, at each instant, the best action to take in order to meet its performance requirements. According to this statement, the scheduler is an autonomous agent, which perceives the surrounding environment through its sensors and acts upon it through its effectors [176].

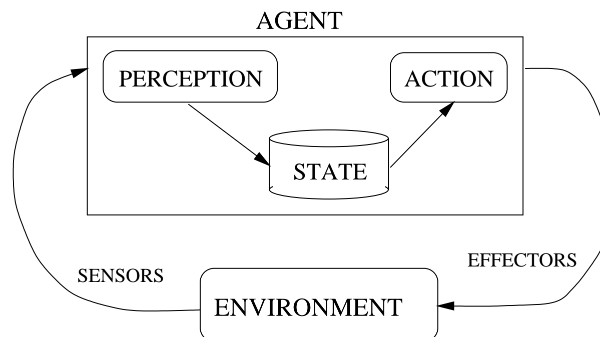


Figure 2.2: An agent that perceives its environment and acts upon it

In this context, autonomous means that the agent determines its behaviour and can act

without the intervention of other systems. This concept can be further extended if the scheduling agent's behaviour depends, at least partially, on its previous experience, i.e., if it exhibits some learning capacity.

If the agent tries to achieve its performance objectives by selecting the most adequate action for each environmental state, then it can be classified as an intelligent agent. This does not mean that the agent is omniscient and never fails. Rather, it means that the agent operates with flexibility and rationality, by choosing the action that will, with higher probability, optimise its performance, given the agent's effectual capabilities and the information it has about the environment.

This thesis addresses global scheduling of parallel applications, where tasks' requirements and arrival rates, the distributed system's current state and the scheduler correcting actions' consequences, or outcomes, may not be clearly known.

2.1.2 Objectives and Constraints

Distributed systems' schedulers exhibit a range of different performance goals. Those most usually found on the literature [11, 139, 178] are:

- maximisation of system's throughput;
- minimisation of a single parallel application's execution time (application's turnaround time or makespan);
- predictability of tasks' response times (e.g. real–time systems);
- system's reliability (e.g. fault–tolerance);
- minimisation of resources' idle time.

Much work has been developed on the context of scheduling real–time systems [134, 166, 167, 168]. One beneficial effect of load management is that it makes tasks' response times more predictable by approximating them to the mean response time, i.e., the standard deviation is reduced. In real–time systems this increase in predictability is more important than reducing the mean response time [182].

This thesis addresses minimisation of a single parallel application's execution time, running on a distributed shared system. Many distributed systems are simultaneously shared by many users and respective applications. These compete for the same set of resources, each seeking to achieve its own performance goals. Schedulers running on these shared systems schedule the execution of concurrent tasks on resources whose performance varies

dynamically due to the presence of various applications that share the same resources [14, 42, 43, 155] (section 2.2).

Many schedulers consider only problems where the tasks' requirements and data access patterns are known in advance. However, in the general case, these parameters are unknown by the scheduler. Most of the work presented on the literature also considers tasks to be independent of each others. However, this is not the case for many real applications. The scheduler must minimise both load imbalance and communication overheads [37]. A general purpose load manager should consider the following constraints [96]:

- no *a priori* knowledge about incoming tasks' requirements and data access patterns;
- no assumption about tasks' independency;
- no assumptions about the underlying network (topology, homogeneity, size, etc.).

The present work deals with problems with unknown tasks' requirements and data access patterns, and further extends these constraints by assuming that the environment's state might also not be clearly known. Gathering information about the exact current environment's state and generating accurate estimates about the workload's behaviour may be prohibitively expensive, or even impossible. The scheduler has to make decisions based on incomplete, or out–of–date, information.

2.2 Application Level Scheduling

Most distributed systems are simultaneously shared by many users and respective applications. These compete for the same set of resources, each seeking to achieve its own performance goals. This sharing introduces challenging resource management problems that are beyond state of the art in a variety of areas [50]. Applications on these shared systems schedule the execution of concurrent tasks on resources whose performance varies dynamically due to the presence of other applications competing for the same resources [14, 155]. This inherent variability on the resources' capacities, caused by the variable load imposed by several users, may impact on the application's performance in dynamic and unpredictable ways. These applications must adapt to changes in resources availability in order to meet their performance requirements [50].

A good understanding of the interplay between the application and the variable multi–user computing system is essential to achieve an effective schedule. By placing the scheduling agent at the application level, rather than at system level,the application programmer can incorporate on the scheduler some of his/her knowledge about the application behaviour and requirements [14, 124].

The present work is concerned with application level scheduling, defined as scheduling performed by the application itself in order to meet its own performance objectives, while in competition with any other applications that may be sharing the same distributed system [14, 19]. Application level scheduling is the opposite of system level scheduling, where a single scheduler controls all resources and applications, and makes all allocation decisions. A system level scheduler, however, can also be present to enforce administrative policies imposed upon the distributed system. This higher level scheduler allocates resources to applications based on a set of attributes, like the number of requested nodes, exclusive or shared CPU access, amount of memory, availability of peripheral devices, etc., and on the organisation's policies. Applications' execution may be postponed and processed in batch, whenever the requested resources become available. System level schedulers, often referred to as Distributed Job Management Systems (DJMS) (section 4.3), are usually concerned with system's throughput and reliability, rather than with application's execution time. The application level scheduler's role is to efficiently manage the resources made available by the DJMS.

Effective application level scheduling involves the integration of application–specific and system–specific data, and it depends on the dynamic interactions between the application and the computing system. The scheduler, embedded within each application, must effectively combine the data it receives about the distributed system and the tasks' states in order to meet its performance requirements.

Berman et al. [14] identify four fundamental application level scheduling requirements:

- **Effective scheduling requires both application and system specific data** – the quality of the generated schedules is highly correlated with the knowledge it has about the system's characteristics and the structure and requirements of the application.

- **Dynamic information is required to determine the environment's state** – using updated environment's state data, the scheduler can detect load imbalances and act to more efficiently exploit the available resources. Static scheduling policies may be inadequate on shared distributed systems, namely when the system's state and global workload vary widely during execution time due to the other users' activities.

- **Effective scheduling requires both application and system performance prediction** – prediction provides the basis for effective scheduling. The scheduler makes its decisions based on predictions about the tasks's requirements and the resources' capacities on the near future. Accurate prediction can be difficult, since the system's state and the application requirements vary in forms that are beyond the scheduler's control. To make predictions the scheduler must have some model of the application and system's behaviour. This model needs to accurately represent

the dynamic performance variation of the application on the underlying resources in a way that allows the scheduler to adapt application execution to the current system's state. Simple execution models can make the prediction task easier, but may fail to provide accurate predictions. A tradeoff must be found between the model's complexity and the quality of the predictions it provides.

- **All resources can be evaluated strictly in terms of the performance they deliver to the application** – from the application's perspective each resource is evaluated in terms of how much it benefits the application's execution. If the computing system is shared among several applications, some resources, e.g. processors, are probably time–shared by these applications. Since only a fraction of each resource's performance can be allocated to each application, the application level scheduler must evaluate each resource in terms of the resource's capacity it uses.

2.3 Static vs. Dynamic Policies

The degree of flexibility the scheduler must exhibit depends on how demanding are its performance requirements and on the complexity of the environment it is supposed to manage. Russel, Norvig and Weiss [143, 176] suggest that environments can be classified according to the following properties:

Accessible vs. Inaccessible – an environment is accessible if the agent's sensors give it complete, accurate and updated information about the environment's state; if the sensors do not provide enough information to completely determine the environment's state, then it is classified as inaccessible, or partially observable; this is related to the environment's *measurability*, which can be defined as a function of the total set of information available to the agent's sensors and the cost of acquiring this information;

Deterministic vs. Non–Deterministic – an environment is deterministic if its next state is completely determined by its current state and the agent's selected action; on a deterministic environment each action has a single guaranteed effect, which is known by the agent; on a non–deterministic environment the same action, performed twice on apparently identical circumstances, may appear to have entirely different effects, and in particular, may fail to have the desired effect;

Static vs. Dynamic – a static environment remains unchanged except by effect of actions triggered by the agent; if the environment changes in ways beyond the agent's control, due to other processes operating on it, then the environment is dynamic.

The environments where a scheduling agent is required to act are, usually, inaccessible, non–deterministic and dynamic.

Inaccessible: the environment may be so complex that the scheduler can not consider all the relevant aspects on its internal execution model of the world. Some of the entities and relations holding on the environment must be neglected or summarised, so that a manageable execution model is obtained. This simplification in representation hinders the scheduler from having a complete image of the environment's state. Furthermore, the information available to the agent is inaccurate because it is expensive, or even impossible, to acquire with great precision. Also this information is often outdated, because the environment is continuously changing — information aging.

Non–deterministic: the environment changes while the agent is deliberating and its selected actions are being performed. The scheduler can not be completely sure of its actions' consequences. If an environment is inaccessible, it may appear to be non–deterministic, since the agent can not keep track of its inaccessible aspects.

Dynamic: on a shared distributed system many applications are launched by several users, and exhibit varying and irregular workload and communication patterns. Furthermore, these applications are allowed to start and to terminate at any instant. This is well beyond the scheduler's control, as it depends on the work performed by the resources it is managing and on the users' activity. It is due to this property that the distributed system's resources deliver a variable performance to each application, requiring the application level scheduling agent to regularly measure the environment's current state.

A distributed shared system exhibits an unpredictable dynamic behaviour, caused by a large number of activities launched by several users, that the scheduler can neither anticipate nor control and that it is unable to characterise completely. Applications may have variable runtime requirements that the scheduler can not predict accurately, no matter how complex is its internal execution model. This suggests that, on such conditions, where both the application and the computing system exhibit dynamically varying behaviours, the scheduler must regularly update the image it has about the environment's current state.

Scheduling policies can be broadly classified as static or dynamic (section 3.3). Static policies generate the schedule before execution time, based on the system's properties and on the tasks' requirements. Dynamic policies, on the other hand, generate the schedule at runtime, using a set of rules to specify correcting actions that redistribute the workload over the system. They have the potential to outperform static policies by exploiting fluctuations in the system's state [1, 72, 105, 106]. Among dynamic policies three different approaches can be distinguished:

- those that do not consider the environment's state at each instant, deciding as if they

were blind;

- those that use environment's state information as input to their set of rules, hoping to make better decisions;
- those that go a step beyond, by using environment's state inputs to modify either its rules' parameters or the rules themselves, i.e., the scheduling agent's execution model is modified in runtime in order to better represent the external world; these are usually classified as adaptive policies.

Static and blind dynamic scheduling strategies seem inadequate for shared systems, since variations on the environment's behaviour are ignored. Strategies that regularly measure the environment's state seem more adequate, since they enable the scheduling agent to react to fluctuations on the environment's behaviour. These are referred to as *sensor based dynamic scheduling strategies*, since the agent gathers data about the environment's state through its sensors. It has been shown that adaptive policies provide good performance when the system state changes widely during execution time [24, 160, 182] and that different load management policies are best suited to handle different workload patterns [12, 29, 181]. This suggests that in order to achieve its performance requirements and deal effectively with the environment's properties, the scheduling agent must use an highly dynamic, or even an adaptive, decision making policy.

Becker and Waldmann [12] identify three opportunities for adaptive scheduling:

Correction of profile and load predictions – sophisticated approaches enable the exploitation of applications workload profile predictions. The challenge for the scheduler is to generate or improve predictions about near future workload profiles. By comparing the expected to the real system behaviour, the exactness and importance of these predictions can be evaluated and corrected.

Determination of diffuse factors in the execution model – dynamic decision making and information management incur overheads at runtime, therefore they must be based on simple execution models. Important side effects are often neglected or the correlations between the considered items and the desired results are not strong enough. Using feedback the scheduler can increase its knowledge about the system's behaviour and correct its initial world's execution model.

Control of the relationship between overhead and profit – dynamic scheduling incurs parallelism management, communication and computation overheads [73]. Adaptive approaches must minimise the efforts for scheduling and automatically optimise the relationship between cost and benefit.

2.4 Incompleteness of Information and Uncertainty

To meet its performance requirements, a sensor based adaptive or dynamic scheduler must collect a reasonable amount of information about the system's state and workload profile, and must have an internal execution model, that adequately represents the most relevant entities and respective interrelationships holding on the environment being considered. Using the available information and the execution model, the scheduling agent generates estimates about future system's behaviour, that are then used for decision making. The effectiveness of the decision making mechanism, depends on the amount and accuracy of the information, or knowledge, accumulated, and on the correctness of the execution model.

However, the scheduling agent is often uncertain about several aspects of the environment. This may happen due to the environment's complexity, which prevents the design of an accurate execution model, or due to the environment's limited measurability. In general, uncertainty and incompleteness of information has three main theoretical reasons [143]:

Theoretical ignorance – there is no complete theory for the problem being solved which describes all the relevant aspects that should be accounted for. Therefore, the agent's designer does not know all the entities and relationships that should be included on the execution model, so that it could predict exactly future system behaviours.

Laziness – the environment may be so complex, that there are too much different factors influencing the system's behaviour; it is neither feasible nor efficient to account for all of them. The agent has to make its decisions based on a simpler internal model of the world. Any computational representation of a real world problem must be a simplification of the objects and relations that hold on the universe being considered. Many of these objects and relations must be neglected or summarised so that a manageable execution model is obtained. This inescapable incompleteness in representation leads to unavoidable uncertainties about the state of the world and the consequences of actions [69].

Practical ignorance – since the environment is inaccessible and only partially measurable, the knowledge available to the decision maker is inaccurate. It is too expensive, or even impossible, to collect exact information about all factors considered in the execution model. Also, since the environment is dynamic, the information gets obsolete, because the system's state is continuously changing. This is usually referred to as information aging.

In the particular case of scheduling a parallel application on a distributed shared system, uncertainty arises from four main sources:

- the environment's complexity requires that some simplifications be included in the execution model, either by neglecting or summarising some of the environment's characteristics; therefore, the model does not provide exact and accurate predictions of the environment's near future behaviour; the environment can be classified as non–deterministic, since the scheduler can not predict the consequences of its actions; this is a consequence of laziness and theoretical ignorance;

- the workload profile and system's behaviour are unpredictable, both due to the application's characteristics and to the background workload imposed upon the distributed shared system by other users;

- it is too expensive, or even impossible, to get exact and accurate information about the system's state, i.e., the environment is not totally measurable and can be classified as inaccessible or partially–observable; this is a consequence of practical ignorance;

- the image the scheduler has about the environment's state gets obsolete with time due to the environment's dynamics [19, p. 535]; this is known as information aging and is also a consequence of practical ignorance.

Hence, at each instant the scheduling agent is uncertain about:

- the environment's exact current state;

- the accuracy of its predictions about the environment's near future behaviour;

- the outcome of its actions and, consequently, which is the most adequate action to take in order to meet its performance goals.

Uncertainty about both the current environment's state and the exact consequences of actions prevents the scheduler from exhibiting an omniscient behaviour. Furthermore, the scheduler must be prepared for the possibility of failure, due to actions that do not achieve the intended result. In order to meet its performance requirements, the scheduling agent must be able to deal effectively with this uncertainty about past, present and future system's states and workload profiles.

The problem studied throughout this thesis is: *should the scheduling agent explicitly include the uncertainty and incompleteness of information it has about the environment on its internal model of the world and on its decision making mechanism, in order to more tightly meet its performance requirements?*

2.4.1 Handling Uncertain Knowledge

The problem of explicitly handling uncertainty on computational models of real world problems has been tackled by artificial intelligence researchers following quite different approaches. The most common approach, currently, is to represent certainty (or belief) on a given statement (or event) using probabilistic values, and combine beliefs on a set of statements using probability theory. However, a few years ago, probabilistic models were considered inappropriate to represent real world problems, due to the exponential number of probabilities required to quantify the joint distribution, i.e., all possible combinations of the events being considered. As a result, a variety of alternatives were proposed. Examples of such alternatives are *default reasoning*, *certainty factors* on rule–based expert systems, *Dempster-Shafer theory* to represent ignorance and *fuzzy logic* to represent vagueness [127, 143]. The fully probabilistic approach has regained acceptance since the late 1980s, with the realization that modularity has to be introduced in the probabilistic model, allowing a large and complex problem to be split up into small, manageable sub–problems. This is achieved by imposing meaningful simplifying conditional independence assumptions among the model's variables [31]. These developments led to the discovery of powerful and efficient tools and algorithms to represent, reason and learn about probabilistic models (chapter 5).

One of such tools are decision networks. These allow the scheduling agent to enter in the model the evidence it has about some of the environment's aspects (evidence obtained through the agent's sensors), and infer the probability distribution over the environment's current state. This is a stochastic process, since the environment's state is not deterministically determined, rather the probability of the environment being in each of all the possible states is inferred. Then, for each possible action, the probability distribution of the environment's next state is inferred and the expected utility for that action is computed. The agent must then select the action which maximises the expected utility. The major advantages of decision networks is that they allow the inclusion on the reasoning process of the uncertainty about the sensor's readings, about the way the environment's entities interact, about the environment's current state and about the way this changes by effect of the agent's actions. And, most importantly, they provide an automatic process of computing the expected utility of each action, by weighing the probability that each next state occurs with the utility that state has for the agent, therefore enabling rational decision making by selecting the action which maximises expected utility.

The hypothesis put forward by this thesis is that: *decision networks, if applied to the scheduling agent's execution model and decision making mechanism, may improve its effectiveness and help overcome the problems caused by uncertainty.*

2.4.2 Related Work

On the context of load management, uncertainty has been explicitly taken into consideration by several researchers. The most common approach is to probabilistically model some of the environment's aspects.

Many authors propose approaches based on a stochastic learning automata [96, 150, 166, 183] (section 4.2.7). The suitability of each possible action, given the environment's current state, is encoded as a bidimensional stochastic matrix. This matrix's numeric values are learned using a reward–penalty reinforcement learning scheme [170]. However, only the scheduling agent's beliefs on the suitability of each action, for each environmental state, are probabilistically modelled. The environment's states are represented by deterministic quantities. Also, all the experiments presented on the specialised literature, view the jobs being scheduled as atomic entities, without any interdependencies among them.

Stankovic [165] proposes a scheduling strategy based on Bayesian decision theory. This strategy is suitable for non–deterministic, inaccessible and dynamic environments, and includes some of the mathematical tools used throughout this thesis. The environment's true state is modelled by a random variable θ, and the scheduler's sensors readings are modelled by a random variable Z. Bayes' rule is then used to compute the probability distribution over the environment's current state given the observations, $\mathbf{P}(\theta|Z)$. The probabilities needed to perform this computation, $\mathbf{P}(\theta)$ and $\mathbf{P}(Z|\theta)$, are dynamically updated at runtime. The most adequate action can then be selected, using the principle of Maximum Expected Utility and the stochastic state transition model, which computes the belief distribution over the environment's next state for each possible action, given the belief distribution over the current state, $\mathbf{P}(\theta|Z)$. The author shows that Bayesian decision theory is an effective tool for decision making under conditions of imperfect knowledge and that it can be successfully applied to the scheduling problem. However, the probabilistic model used is very simple (only two random variables), essentially because the mathematical tools required to efficiently handle complex probabilistic models were not known at that time (1985).

Schopf, Berman et al. [152, 153, 154, 155, 156] propose an application level scheduler based on a stochastic structural execution model. The application performance is decomposed according to the application's structure, i.e., into interacting component models that correspond to sub–tasks. Each of the model's variables can be either deterministic or stochastic. Deterministic quantities are represented by a single point value, whereas stochastic quantities can take a range of possible values, represented as intervals, histograms or probability distributions. The authors argue that stochastic modelling is more adequate in shared distributed systems, since both the resources capacities and the applications requirements vary with time. The proposed scheduler uses normal probability distributions to model the

environment's most relevant aspects. Using stochastic variables to model the environment, results on stochastic behaviour estimates. This particular scheduler uses this model to increase Quality of Service guarantees, by assigning tasks to those resources that exhibit lower variance, i.e., whose performance exhibits less dispersion around the average value. Although the environment's state can be stochastically modelled, no attempt is made to extend this approach to the modelling of the scheduler's actions outcomes.

2.5 Summary

Load imbalance is one source of overheads for parallel applications, since the capabilities of the available system's resources are not fully exploited. The load manager's role is to reduce this overhead, by adequately mapping tasks onto the available resources. This is a scheduling problem.

Uncertainty and incompleteness of information were identified as the main reasons why an application level scheduling agent, operating on a shared distributed system, may fail to achieve its performance goals.

Uncertainty arises from four main sources:

1. the environment's complexity is high, therefore the scheduler's internal execution model must be a simplification of the objects and relationships that hold on the real world;

2. some of the environment's relevant aspects are inaccessible, in the sense that it is too expensive, or even impossible, for the scheduler to get exact and updated information about them;

3. since the environment is non–deterministic, the scheduling agent can not be completely sure of its actions outcomes;

4. the environment is dynamic, i.e., it changes while the scheduling agent is deliberating on which action to take, and it changes by the effect of processes that are not under the scheduler's control; the information the scheduler has about the environment's current state gets obsolete as a result of the environment's dynamics.

In order to meet its performance requirements the scheduling agent must be able to deal effectively with this uncertainty about past, present and future system's states and workload profiles. The most common approach to reason with uncertain knowledge is to probabilistically model the uncertain quantities.

This thesis studies the suitability of some probabilistic tools to increase the effectiveness of application level scheduling on distributed shared systems, where the scheduling agent may be uncertain about the environment's current state and about the outcome of the actions it selects.

Chapter 3

Load Management: Basic Concepts

Contents

3.1 Degree of Balancing **26**

3.2 Level of Complexity **26**

3.3 Static versus Dynamic Policies **28**

3.4 Deterministic versus Stochastic Strategies **31**

3.5 Centralised versus Distributed Approaches **32**

3.6 Scheduling Policy Components **34**

3.7 Load Management Evaluation **41**

3.8 Summary **48**

The design of an effective scheduler requires the programmer to make a set of decisions, related to the scheduler's architecture and to the organisation of its various components. These decisions depend on the scheduler's performance goals and on the characteristics of the particular environment being managed.

In order to increase the scheduler's performance, the programmer must have a thorough understanding of the structure of the overheads induced by the scheduling agent, the reasons why these overheads exist and the potential benefits associated with them.

This chapter discusses some basic concepts involved in the design of an effective scheduler. Section 3.7 distinguishes between the scheduler's performance and efficiency, defining this last one as a measure of the scheduling overheads. An analysis of these overheads structure is presented. Scalability and stability are defined and identified as two desirable scheduling properties.

3.1 Degree of Balancing

The load balancing approach to load management consists in equalising the workload assigned to all system's resources [24]. The main concern of algorithms following this approach, is to reduce the standard deviation of some metric related to the amount of work allocated to each resource. On an heterogeneous system, the level of workload distribution can be weighed by the resources' relative processing capacities.

Load sharing, on the other hand, attempts to keep all resources busy, as long as there is work to be performed, independently of the amount of work actually assigned to each resource, minimising the idle–while–waiting condition.

These two approaches represent different degrees of balancing, from the weakest degree of load sharing to the strongest degree of global load balancing [42, 179]. It has been shown that load balancing can potentially reduce the mean and standard deviation of tasks' response times, but since it requires higher transfer rates, the higher overhead incurred may outweigh this potential performance improvement [39, 160]. Tight load balancing can be a source of instability, since useless work transfers can be initiated by the scheduler, in order to keep the amount of allocated work equalised among resources, even though no additional benefits are gained with these transfers. The particular case of cyclic useless transfers of tasks among processing nodes, is referred to as *processor trashing* (section 3.7.4).

In the context of application level scheduling on distributed shared systems, the fraction of each resource's capacity that is allocated to each application can vary significantly and frequently during execution time. Keeping the workload distribution constantly balanced can constitute a non–profitable source of overheads. A load sharing approach, where the scheduling agent decides whether or not to transfer work among resources by weighing the expected benefits with the overheads incurred with these transfers, seems more adequate. This kind of assessment is usually referred to as *profitability determination*.

3.2 Level of Complexity

One important issue is to determine what is the appropriate level of complexity for the load management policy. It has been claimed that relatively simple policies can provide substantial performance gains, while more complex ones are not likely to offer much further improvements [25, 39, 182]. Complex policies rely on detailed information about system's state and workload's behaviour. Not only is this information expensive to gather, but also some quantities cannot be precisely known, regardless of the effort expended. Furthermore, complex policies are bound to have complex negotiation policies among nodes, which may

increase communication costs. The final concern is the potential for instability. Complex policies may react to small imbalances among resources, causing a form of processor thrashing, in which all nodes spend all their time transferring tasks. Less complex policies tend to react more slowly to changes in the system's state and are, therefore, less susceptible to such instability. Eager [39] claims that:

- extremely simple load distribution yields dramatic performance improvement relatively to the no load–sharing case;

- complex policies, which try to make the best choice, do not offer much further improvements.

The experiments performed by the above cited author do not consider tasks interdependencies and were performed on small, homogeneous and dedicated distributed systems. Parallel tasks often exhibit data dependencies that, if not taken into account, may hinder the scheduler from achieving its performance goals. The distributed system's size and heterogeneity may require complex scheduling policies. However, the potential for unstable behaviour increases with the system's size, heterogeneity and with the scheduling policy complexity. Special care must be taken, therefore, to ensure that potential overheads associated with instability, information collection and sophisticated decision making do not overwhelm the benefits obtained with a more complex load management.

Becker [10, 12] proposes an hierarchical approach to load management which operates with a rather sophisticated policy. This policy considers not only processing demands and processor load, but also data affinity and data communication costs. He argues that sophisticated load management is more effective than simply assigning tasks to the least loaded node. However, he concludes that complex load management is worthwhile as long as it can keep in time with information collection and decision making. Slight congestions can completely destroy the profits of costly strategies and must be avoided. A strategy with adaptable complexity is proposed, which reduces its complexity if the scheduler becomes heavily loaded and/or unable to make the required decisions in time.

The complexity of an application level scheduler's policy is determined by the richness of the execution model it uses. A shared distributed system is, usually, a complex environment (section 2.3), with the total workload allocated to each resource varying significantly with time, due to the activity of several users. If the application being scheduled exhibits an irregular and unpredictable pattern of resources' requirements, then a relatively complex execution model is probably required to allow the scheduler to meet its performance requirements.

3.3 Static versus Dynamic Policies

Scheduling policies can be broadly classified as static or dynamic [33]. Static policies generate the schedule before execution time, based on the system's properties and on the tasks' requirements. This is also called the *mapping problem*, because a mapping function must be defined, which assigns tasks to resources before the execution begins [1, 19, 24, 85, 109]. This approach can be effective when the workload can be sufficiently well characterised before the actual execution, but it fails to adjust to fluctuations on the environment's behaviour. Dynamic policies, on the other hand, generate the schedule at runtime, using a set of rules to specify correcting actions that redistribute workload over the system.

To solve the mapping problem a mapping function must be defined which assigns each task to a processor before runtime. The distributed system is represented by an undirected graph Gp(P,L), where P and L are the set of processors and interconnection links, respectively. The parallel program is represented by a Task Interaction Graph (TIG), Gn(V,E), where V is the set of tasks and E is the set of edges representing interactions among the tasks. The TIG vertices are labelled with weighs that represent the tasks' estimated computation costs and the edges are labelled with weighs that represent the amounts of communication involved among tasks. The mapping problem is solved by a function $f : V \rightarrow P$, which associates each task i with an unique processor $q = f(i)$ [157]. The *optimal* solution for the mapping problem is known to be NP–complete [19, 37], meaning that no computationally tractable (polynomial time) algorithm exists for evaluating an optimal mapping for the general case. Instead of searching the entire solution space, one has to be satisfied with a *sub–optimal* solution. The factors that determine whether this approach is worthy of pursuit include:

- the availability of an objective function to evaluate a solution;
- the time required to evaluate this function.

Senar et al. [157] propose two cost functions which can be used as objective functions:

minimax – the cost incurred by each processor q under a certain mapping is due to computation and communication of all tasks $i : f(i) = q$ mapped onto it,

$$\text{cost}(f,\text{q}) = \sum_{i:f(i)=q} w_i + \sum_{i:f(i)=q, j:f(j)\neq q} c_{i,j}$$

w_i being task i's computational weigh, $c_{i,j}$ denoting the communication weigh for the edge that joins tasks i and j. Communication among tasks mapped onto the same

processor is modelled as having no cost. The objective function is to minimise the maximum cost across all processors,

$$\text{Obj}(f) = \min(\max_{\forall q}(\text{cost}(f,\text{q})))$$

summed – this functions tries to minimise the load imbalance cost while keeping the amount of communication to a minimum:

$$\text{cost}(f) = \sum_{\forall q}(\sum_{i:f(i)=q} |w_i - \overline{W}| + \sum_{i:f(i)=q,j:f(j)\neq q} c_{i,j})$$

$\overline{W} = \frac{\sum_{\forall i} w_i}{N}$ being the average computational weight for all N processors.

$$\text{Obj}(f) = \min(\text{cost}(f))$$

The authors use the minimax model because it assumes that communications can occur independently and in parallel on the system's several communication links. With the summed model all communications are assumed to occur sequentially, which could be preferable for systems where the processors are connected by one single communication link (ethernet–like systems). Both models, however, consider that a processor is either computing or communicating, but not doing both simultaneously.

Senar et al. do not discuss how to estimate the parameters w_i and $c_{i,j}$. Schopf and Berman [155] propose a static approach where these parameters are stochastically modelled, following normal distributions. Their goal is to increase Quality of Service guarantees in shared environments, by assigning tasks to resources that exhibit lower variance (section 2.4.2).

A common approach to sub–optimal mapping is the use of heuristics. Heuristic schedulers make use of special parameters which affect the system's performance in indirect ways. These parameters are usually very simple to calculate, although it may not be possible to prove that a first–order relationship exists between the mechanism employed and the desired result. One possible approach is presented by Casavant [24] and Monien [117]. The goal is to minimise congestion (number of logical channels mapped onto each physical link) and dilation (number of physical links one logical channel has to traverse, i.e., physical distance between two logical neighbours) and equalise the workload among all processors. Two strategies are possible:

- place processes that are able to execute concurrently on different processors, in order to maximise the degree of parallelism;
- place tasks that communicate frequently on the same processor, to increase locality.

These two strategies conflict with each other and the solution involves tradeoffs. However, some domain decomposition techniques have been developed, like the recursive bisection and its variants [16, 49], which try to identify groups of tasks with identical computational and communication requirements and map them onto the nodes. Other techniques have been studied to map some often used graphs onto the most common network topologies [37, 116, 117, 140].

Dynamic policies generate the schedule at run time, using a set of rules to specify correcting actions that redistribute workload over the system. Among dynamic policies three different approaches can be distinguished:

- those that do not consider the environment's state at each instant, deciding as if they were blind;
- those that use environment's state information as input to their set of rules, hoping to make better decisions;
- those that go a step beyond, by using environment's state inputs to modify either their rules' parameters or the rules themselves; the scheduling agent's execution model is modified in runtime in order to better represent the external world; these are usually classified as adaptive policies.

Most dynamic policies use environment's state information to make load distribution decisions, so they have the potential to outperform static policies by improving the quality of their decisions. These are referred to as sensor based dynamic scheduling strategies, since the agent gathers data about the environment's state through its sensors. Dynamic load distribution policies improve performance by exploiting short–term fluctuations in the environment's state [1, 106]. Since they must collect state information, they incur more overhead than their static counterparts, but this overhead is often well spent.

Adaptive policies adapt their activities by dynamically changing their parameters, or even their algorithms, to suit the changing environment's state [51, 52, 183]. Whereas a dynamic policy takes system–state inputs into account when making its decisions, an adaptive policy takes these inputs into account to modify either its parameters or the scheduling policy itself. The performance of scheduling policies is very sensible to their parameters, which suggests that adaptive load distribution may be able to provide good performance when the environment's state changes significantly [24, 160, 182]. It has been shown that different load management algorithms are best suited to different classes of applications [12, 181], which further suggests that in order to be effective over a wide range of workload characteristics, the scheduler must dynamically adapt itself to the current workload profile.

An application level scheduler, operating on a shared distributed system, must use a dynamic, or even adaptive, scheduling policy, in order to deal efficiently with the dynamic

characteristics of such environments (sections 2.2 and 2.3).

3.4 Deterministic versus Stochastic Strategies

Scheduling strategies can be classified as deterministic or stochastic according to the rules governing its decisions [179]. Deterministic methods act according to a set of rules that use single values, or deterministic values, to parameterise the scheduler's execution model. Stochastic methods decide using some probabilistic mechanism, in an attempt to maximise, with high probability, their performance goals. These methods often use stochastic execution models which must be parameterised with stochastic values, or probability distributions, rather than deterministic values. Stochastic values are specially adequate to quantify environment's characteristics that may vary with time and/or whose exact value can not be precisely known. Instead of quantifying these variables with exact, single–point values, a probability is assigned to each of their possible values [14, 153, 155, 156].

Ryou and Juang [144] argue that deterministic algorithms consistently outperform their stochastic counterparts, but they only consider probabilistic approaches that do not use environment's state information. Loh et al. [105] claim that stochastic models are more realistic, since they can capture an application's time varying characteristics.

Some well known stochastic strategies are random allocation and the Stochastic Learning Automata (sections 4.2.3 and 4.2.7). Stankovic [165] proposes a scheduler based on Bayesian decision theory, which learns its stochastic parameters during execution time and successfully manages the load of a small distributed system (section 2.4.2). Schopf, Berman et al. [152, 153, 154, 155, 156] propose an application level scheduler based on a stochastic structural execution model. They argue that stochastic modelling is more adequate than deterministic modelling, when the environment's state changes with time in ways that are beyond the scheduler's control. This is the case with application level scheduling in shared distributed systems.

Simulated annealing is a stochastic physical optimisation method which can be used to solve the load management problem. It simulates the random movements of a collection of vibrating atoms in a cooling process. The goal is to arrive to an optimally low energy configuration of the atoms at some low temperature. When applied to load management a configuration corresponds to a state of the system and the final configuration to the final result of applying the scheduler, hopefully a balanced state with respect to workload distribution [179].

As stated in chapter 2, a stochastic model of the environment may be more adequate than a deterministic one, when the distributed system's state may change with time and the application's near future behaviour can not be accurately predicted. Moreover, if the

outcomes of the actions available to the scheduler are not exactly known, then these can also be stochastically modelled. This is the approach taken throughout this thesis.

3.5 Centralised versus Distributed Approaches

Scheduling policies may be centralised, distributed or some hybrid form of both. Centralised approaches employ a single agent for state information collection and decision making. Becker and Zedelmayr [13] claim that centralised approaches are optimal, from a logical point of view, for several reasons:

- state information is coherent at the central agent and does not need to be replicated among the system's nodes, causing additional message traffic; allowing each node to maintain a local image of the environment's state, usually results in differences among the nodes' images, due to different information ages;
- global knowledge about the entire parallel system's state as well as about the progress and interrelationships of the application's tasks can be exploited, avoiding counter-productive decisions that would arise from different partial information and thus achieving coherence between the various decisions [75].

Distributed approaches divide the information management and decision making among the system's nodes, usually employing one agent per node. Each agent keeps information about its state and the states of some set of its neighbours. This approach is scalable because increasing the number of nodes does not increase the amount of information each node has to handle. The main disadvantages of distributed approaches are that each scheduling agent has only a partial view of the entire environment, and different agents might have different views of the environment's state due to information aging. This can lead to contradictory decisions which increase the overheads.

Although some authors claim that centralised policies achieve better results, are simpler and more effective than distributed ones [182], and scale with system's size [171], the great majority agrees that a load management policy must have some degree of distribution in order to avoid bottlenecks and thus be scalable [41, 92, 107, 125, 160, 178]. Central load management has no logical drawbacks, but it is not scalable as it will cause processing and communication overheads to grow with system's activity and size. Above some limit, centralised approaches can no longer improve the overall throughput.

Scheduling implies at least two activities which require communication among the distributed system's nodes: environment's state information gathering and task transfers. If each node is required to interact with all other nodes, then it will have to use collective

communication mechanisms — such as broadcast and global gathering — which are not scalable and create intolerable overheads and congestion in large systems.

To reduce the potential for both this communication bottleneck and processing time delays at the central agent, many approaches partition the system into sets of nodes called domains. A node only exchanges information and tasks with members of its domain. In some cases, each node's domain is restricted to its physical or logical neighbours and is static during all the execution time. These are called nearest–neighbour algorithms. These have a less stringent requirement on the spread of local workload around the system than global algorithms, therefore they are scalable with system's size. Nearest–neighbour algorithms are iterative in nature, in the sense that successively imposing local load distribution inside each domain, makes progress towards a global uniform load distribution, since domains overlap [179, 180]. One disadvantage of these schemes, is that they are usually slow in distributing load across domains when a sub–region of the system becomes suddenly overloaded (formation of hot regions with an excessive load causes the congestion problem [158]) [108, 117, 151].

Ryou and Suen [144, 169] propose an approach which does not consider physical neighbourhood, but reduces the number of nodes to which a given node must send load information updates. This set of nodes is referred to as the processor's sending set. Using balanced sending sets, the number of load information messages is minimised and load distribution can occur between any pair of nodes, either through direct negotiation or through an arbitrator node which has knowledge about the loads of those nodes.

Shin and Chang [158] propose a method based on static domains (referred to as buddy sets) and preferred lists. Each processor's preferred list indicates with which nodes it will exchange load information and tasks. Each node must select underloaded nodes to exchange tasks with, respecting its preferred list's order. The nodes appear in the various preferred lists in different order (although trying to maintain physical distance order), reducing, therefore, the probability of two, or more, overloaded nodes selecting the same underloaded node and overflowing it with task transfers (coordination problem).

An alternative approach is to apply a centralised load management policy inside each domain. Tight load management efforts are provided within the domains, while less interaction and task transfers are performed among them. This is usually called a distributed clustering approach (section 4.2.6) [10, 11, 12, 13, 125]. This clustering may be continued hierarchically with clusters of clusters, or scheduling among clusters may be performed between direct neighbours only. Hierarchical clustering tends to impose more overhead on higher level managers, while interaction with neighbours requires two different load management policies and shows counter–productive decisions similar to completely distributed schemes, although at a smaller scale. It has been shown that clustering approaches are a good compromise, since they are able to exploit some advantages of centralised strategies

and nevertheless are scalable.

The domains' sizes must be carefully selected. If these are too large, they will suffer from the same problems as centralised schemes. If they are too small, then the problems associated with distributed schemes will arise. One obvious solution is to dynamically adapt the domains' sizes. If a given cluster's scheduling agent becomes heavily loaded, unable to make decisions in reasonable time, then this cluster might be splitted into two or more clusters, each with a smaller number of elements. On the other hand, if a scheduler becomes underloaded, it can search among its neighbours for a suitable partner with whom it can merge, forming a larger cluster. Splitting the clusters when the system load is high carries some computational and communication overheads, which consume these resources exactly when they are most needed. Therefore, some alternatives should also be considered. One such alternative is for the scheduler to switch to a simpler and faster strategy, if the increase in the system's activity is supposedly of short duration. It switches back to the full strategy as soon as the congestion is handled, or the cluster is splitted if it turns out to be a long lasting situation. On the other hand, merging clusters is done when these are lightly loaded, therefore the resources consumed by this operation should not be critical.

The selection of a centralised, distributed or hybrid architecture for the scheduler, should depend on the scheduler's goals and on the system's size and overall activity. If the system's reliability is a main concern, then centralised architectures must be avoided, since the central scheduling agent's failure can compromise the entire system's performance. If the scheduler is required to scale with system's size and/or activity, then a distributed architecture should be preferred, since centralised approaches can compromise the scheduler's efficiency. Nevertheless, centralised approaches are very effective for small distributed systems, due to the central agent global and unique view of the environment's state.

3.6 Scheduling Policy Components

Scheduling policies are often described in terms of four sub–policies, or components, and the transfer mechanism used to move tasks around the distributed system [160]:

- **information policy** – set of rules to decide when, from where (which resource or application's task) and what information must be collected about the environment's state;
- **transfer policy** – set of rules to determine whether a given resource is in a suitable state to participate on a workload transfer;
- **location policy** – set of rules to identify the various partners enrolled on a given action, e.g., workload transfer;

- **selection policy** – set of rules to decide which work must be transferred among the partners identified by the location policy.

3.6.1 Information Policy

The usefulness of a scheduling policy is highly dependent on the quality of load measurement and prediction [175]. The information policy determines when, where and what, information must be collected, through the scheduling agent's sensors, about the system's resources current workload and tasks' workload profiles. Information policies can be broadly classified into three types, although hybrid versions of these may exist:

- Demand–driven policies, which imply that an agent collects information about the environment's state only when it needs that information for decision making. Information collecting is, therefore, triggered by another sub–policy. The mechanisms for demand–driven information collection are probing or bidding. Under the probing method (section 4.2.4) a node willing to participate in a workload transfer selects another node and checks whether or not it can participate in this workload redistribution. If not, the process is repeated until a suitable partner is found or the number of probes exceeds the probe limit. Though Eager et al. [39] claimed that the performance of this policy is insensitive to the probe limit's value, other authors claim that it is an essential parameter [158, 182]. It has been shown that small probe limits are appropriate [39]. Under the bidding method (section 4.2.4) a request for bids is sent to a group of nodes (eventually broadcasted) and bids are received from those willing to participate in the transfer. Bids are then evaluated to choose a suitable partner[1][25, 158, 166, 167]. These methods do not use any history information regarding past scheduling decisions or other nodes' states, hence they do not incur any overhead maintaining this information. However, information has to be collected whenever a task is to be transferred, which introduces additional delays in completing these tasks.

- Periodic policies, which collect information periodically. A fixed overhead is imposed upon the system because environment's state information is collected and maintained, regardless of whether this information is used or not. However, there is no probing delay whenever a decision has to be made. The image maintained of the environment's state may not correspond to the current environment's state due to delays in the communication network and to the periodic nature of information collection. This phenomenon is usually called information aging, and is closely related to the period with which information is collected. However, collecting information more frequently increases the overheads imposed upon the system by the scheduler and

[1] This is the location policy's responsibility

there is always the delay associated with the communication network. Also, with distributed policies, the image that a scheduling agent has about the system's state may be different from agent to agent. This may lead to complex location policies.

- State–change driven policies, which collect information about the resources' state whenever it changes. Most state–controlled algorithms classify a resource's state as one of n possible states, depending on its actual load (L) and a number of thresholds. A 3–state policy can be specified as [93, 106, 121, 178]:

$$\begin{array}{cl} L < T_1 & \text{Underloaded node} \\ T_1 < L < T_2 & \text{Mediumloaded node} \\ L > T_2 & \text{Overloaded node} \end{array}$$

Various combinations of these different information policies are possible. An information policy might be periodic, but a node willing to participate in a task transfer might poll the best candidate to confirm that its current state still corresponds to the local image.

A dynamic scheduling agent must have a set of sensors, referred to as the agent's sensorial apparatus, by means of which it acquires information about the environment's state. In computing systems, these sensors are usually based on static instrumentation, i.e., they are composed of a set of software instructions inserted into the application's code [21]. The design of a sensor must carefully balance the overhead and changes induced on the environment's behaviour by the sensor itself (level of intrusion), with the accuracy and relevance of the quantity being measured. Excessive instrumentation may perturb the system, and even change the events' order among the different tasks of a parallel application, while reduced instrumentation can compromise the measurements' accuracy. Sensors can be classified as either event–driven or time–driven [132]. Event–driven sensors are activated by the occurrence of particular events, while time–driven sensors are based on sampling, i.e., they are activated at fixed time intervals. Demand–driven and state–change driven information policies use event–driven sensors, whereas periodic policies use time–driven sensors.

A key issue is to identify suitable metrics which adequately describe a resource's current load. Such metrics are generically referred to as *load metrics* (section 6.4). A good load metric should [46, 121, 160]:

- correlate well with task response times, since it is used to predict the performance of a task if it is executed at some particular resource;
- be usable to predict the load in the near future, since the response time of a task will be affected more by the future load than by the present load;
- be relatively stable; short fluctuations in the load should be discounted;

- be relatively cheap to compute.

A number of load metrics have been presented in the literature: CPU queue length, CPU utilisation, normalised response time, I/O queue length, memory utilisation, context–switch rate, system call rate, etc.[46, 96]. It has been observed that a task at a node is likely to demand services from a number of resources (e.g. CPU, memory, disks), therefore it might be important to define load not only as a single resource in a node, but as a collection of resources. Ferrari [46] proposed a linear combination of resource queue lengths as a load metric. If an incoming task requires s_j seconds from resource r_j and the queue length of that resource is q_j then the load metric, as perceived by that job, is

$$l = \sum_{j=1}^{N} (s_j * q_j)$$

N being the number of different resources. This load index is task dependent as it takes into consideration each task's demands. However, the assumption that task's demands are known in advance is too strong in most cases. Ferrari studied also another load metric, based on a linear combination of the various resources' queue lengths. His results showed that the performance differences among the cases where metrics based on the CPU queue length alone were used, and those where I/O and memory contention were also considered, are not significant, suggesting that the CPU is the predominant resource in their hosts. Kunz [96] got similar results doing experiences with both one–dimensional load metrics (CPU queue length, available memory, context switch rate, etc.) and linear combinations of these. These results and those of [7, 121, 183] suggest that CPU queue length is one of the most adequate load metrics and that its value is not required to be very accurate. Ferrari [46] used exponentially smoothed average CPU queue length over a time window, instead of instantaneous queue length, to eliminate high–frequencies components of the rapidly changing load, which may be regarded as noise. Care should be taken when choosing the time window, because if it is too long past loads will be emphasised and performance may become worse. The optimum averaging interval is clearly dependent upon the dynamics of the workload.

Since an application level scheduler is designed by the application programmer itself, two different approaches may be taken when selecting the load metrics: these can be either application–independent or application–dependent. Application–independent load metrics have just been discussed, and they refer to system's characteristics, such as each resource's waiting queue length. Application–dependent load metrics convey more information about each resource's performance on application specific tasks. On image processing applications, for instance, each node's load can be characterised by the number of pixels processed by unit of time. If a node's background workload, imposed by other applications sharing the same set of resources, increases, then the rate at which it processes image pixels must decrease. To avoid dependencies among the load metric's actual values and the particular

image sub–region being processed by each node, image templates, equal on all nodes, can be used to compute the load metric, instead of using the actual image. This approach will, however, impose an additional overhead, since computing the load metric does not directly contribute to finish the task in hand. The benefits of using an application–dependent load metric must be carefully weighed with the overheads of computing it.

3.6.2 Transfer Policy

The transfer policy determines whether a resource is in a suitable state to participate in a workload transfer, either as a sender or a receiver. Most transfer policies are either threshold–based or relative. Threshold policies classify a resource as a sender if its load metric exceeds a threshold T_s or as a receiver if it falls below a threshold T_r [106, 160]. The choice of these thresholds is fundamental for the algorithm's performance. Clearly, the best threshold values depend on the environment's load and the task transfer cost. At low loads and/or low transfer costs, thresholds should favour task transfers; at high loads and/or high transfer costs remote execution should be avoided. Although Eager [39] states that the optimal threshold is not very sensitive to the environment's load, several techniques were studied which efficiently and in runtime adapt the threshold to the system's load [12, 34, 133].

Relative transfer policies take as input the difference among a resource's load and that of its neighbours. Resources are considered able to participate in a transfer, if their loads differ by more than some threshold δ. They may then transfer some fixed number of tasks or a fraction of the load difference [25, 94, 108, 117, 151, 160, 180].

Any transfer policy should strive to minimise remote execution activities in the system (task transfers). When the system is heavily loaded, much higher transfer delays than the average may be expected, which can severely degrade performance. Only a small percentage of the tasks needs to be remotely executed in order to achieve effective load management [92].

The transfer policy may be either periodic or event–triggered. The algorithm may periodically check if the resource's state qualifies it as a candidate for a workload transfer or not. However, the great majority of the policies proposed in the literature are event–triggered. If a resource's state changes, then a task transfer may be possible. The transfer policy might also be triggered because the node is polled by another one to check its suitability to act either as a receiver or a sender on a task transfer.

Task transfers may be sender–initiated, receiver–initiated or symmetrically–initiated. With sender–initiated policies (also known as source–initiated) resources with extra work must search for a suitable receiver; with receiver–initiated policies (also designated as server–

initiated) underloaded resources take the initiative looking for suitable senders; with symmetrically–initiated policies both senders and receivers may initiate task transfers. Sender–initiated algorithms may be ineffective at high system loads, since most of the resources are senders, hence it is unlikely that the majority of them will ever find a suitable receiver. Even worst, they might overflow some of the potential receivers with too much tasks. Even if the potential receivers are allowed to reject additional work sent to them, more control messages are being introduced and useless work is being performed in a system already highly loaded. Under sender–initiated policies the burden of initiating the activity is taken by an already overloaded resource. Under receiver–initiated policies this overhead is placed on the underloaded nodes, which seems to be more adequate. However, if the system is lightly loaded these policies will fail to find a suitable sender. How many times, or for so long, should a receiver try to find this sender? It can suspend its activity after some threshold (or timeout), but then it will fail to detect future overloaded resources, unless its activity is periodically reinitiated: a disadvantage of receiver–initiated algorithms is that the receiver is not aware that other resources became potential senders, neither these senders can notify it. An alternative approach is to allow receivers to place reservations on other resources, which will later transfer work if they are still receivers. A further disadvantage is that a request for work may arrive to an overloaded resource when it is executing all its tasks, forcing a preemptive task migration, which is expensive (selection policy). However, receiver initiated policies have the advantage of automatically turning themselves off when the system load is high (there are no receivers), decreasing the number of control messages [39, 92, 121, 160, 171, 182].

Symmetrically–initiated algorithms have the advantages and disadvantages of both sender– and receiver–initiated algorithms. A symmetrically initiated adaptive algorithm is proposed by Shivaratri [160], which switches its behaviour between sender and receiver initiated transfers according to the environment's load (section 4.2.4).

3.6.3 Selection Policy

Once a decision is made to enroll a given resource on a workload transfer as a sender, a selection policy must select the task (or tasks) to be transferred.

A workload transfer may be either preemptive or non–preemptive. Preemptive transfers involve transferring partially executed tasks. This is expensive since it requires collecting a process' execution state, changing all its communication channels, etc. Although many authors do not even consider preemptive transfers [171, 182], others claim to do it with success. Scheurer [151] presents a tool which does preemptive process migration in a Transputer network with very good results; Bozyigit et al. [17] developed the LBDCS system, which performs preemptive process migration and load balancing on a Linux net-

work. Balter [7] uses a preemptive migration strategy whose selection policy is based on UNIX process lifetime distributions on an academic environment. However, the processes considered are completely independent of each others. Non–preemptive task transfers involve only tasks that have not began execution and hence do not require transferring the task's execution state. A resource may be overloaded and yet have no tasks available for non–preemptive transfer if it is polled by a receiver. When performing preemptive task transfers, the scheduler can use dynamically generated information about a task's behaviour to reallocate it to a different resource. Using only non–preemptive transfers this is not possible, although dynamically gathered information may be used to allocate future similar tasks. Non–preemptive task transfers are sometimes designated as task placements or *one–time assignments*. However, one–time assignments usually refer to tasks that once assigned to one resource can no longer be migrated once again, independently of whether or not its execution has began.

When a parallel application entails applying the same algorithm to large numbers of data points, this data set can be partitioned and assigned to several processors, in order to be processed in the shortest possible time — data domain decomposition. This kind of computational loads can be classified based on its divisibility property. If the data set can be divided into any number of segments of any desired fractional size, then it is referred to as an *arbitrarily divisible load*. If there is a limit to the load's degree of divisibility, then it may be classified as modularly divisible [15]. A scheduler dealing with divisible loads, may decide the division of a task whose processing has already began. It just splits the task onto two or more sub–tasks, by partitioning the task's data set.

A selection policy should take several factors into consideration [7, 160, 175]:

- task transfer overheads must be minimised; non–preemptive transfers and small tasks (small amount of information) carry less overhead;

- the transferred task's execution time should be enough to justify the cost of the transfer; even if tasks' execution time is unknown, it should be possible to classify them as short or long lived tasks, and consider only the latter ones for migration; Zhou [182] has shown that some classification errors might be tolerated, since scheduling algorithms are quite robust with regard to this parameter;

- the number of location–dependent resources needed by the selected task should be minimal; these resources include specific data, i/o devices (display, keyboard, disks), etc.

3.6.4 Location Policy

The location policy selects a suitable workload transfer partner, usually using information about the resources' states. It must avoid flooding an underloaded resource with tasks (coordination problem), which can happen when one resource is simultaneously selected as a target by several scheduling agents. Some policies try to find the best partner among the domain, while others just look for an adequate partner. The random location policy (section 4.2.3) selects a random partner without using any information about the target's state. Surprisingly good results can be achieved with this policy [39]. Some location policies use probabilistic schemes instead of deterministic ones. Probabilistic schemes distribute tasks according to a set of probability distributions (section 4.2.7).

Mehra [111, 113] proposes an approach which uses one comparator neural network per resource. These learn to predict the speedup of an incoming task, using only the resource's utilisation patterns observed prior to the task's arrival. The lack of job specific information is overcome by learning to compare the relative speedup of different resources with respect to the same task, rather than attempting to predict absolute speedup. The numerous parameters of this dynamic approach are tuned using a genetic algorithm. Neural networks are trained using an off–line learning system.

The location policy may deal with some restrictions when searching for a destination resource. These restrictions may include resource requirements, tasks' precedences [11], task interrelationships and *data locality* [163]. Data locality addresses the fact that tasks require certain data items, which may have to be fetched from remote locations [12, 27, 139].

3.7 Load Management Evaluation

The load manager, or scheduler, dynamically generates a mapping, or schedule, in order to satisfy its performance goals. To generate and apply this schedule, the load manager itself consumes the resources it is managing. To evaluate a scheduling agent's design it is not enough to evaluate the quality of its decisions, but it is also necessary to take into account the resources consumed by the agent itself.

The design of an efficient scheduler requires a careful analysis of the costs involved. Only with a thorough understanding of the scheduler's cost structure, the reasons why these costs exist and the potential benefits associated with them, can the programmer intervene on the scheduler's design in order to make it more efficient. A set of metrics is required to measure the scheduling costs and the scheduler's performance; this set of metrics is referred to as the *performance model*.

3.7.1 Performance and Efficiency

Since the scheduling agent also consumes the resources whose utilisation it is trying to optimise, the scheduler itself imposes additional overheads upon the environment. These are referred to as *scheduling overheads*. These overheads may hinder the scheduler from achieving its performance goals. Scheduling overheads must, therefore, be kept under acceptable levels, so that they do not overcome the benefits achieved with the scheduler's activity. The scheduling agent measures these overheads by using a set of *scheduling overhead metrics*.

The scheduler's performance goals degree of achievement is referred to as the *scheduler's performance*, or effectiveness. This is closely related to the quality of its decisions, and can be described as the application's satisfaction with how well the scheduler manages the resources in question. The effectiveness of the scheduler's decisions can only be evaluated by using suitable *performance metrics*.

Efficiency is a measure of the scheduling overheads. It is related to the application's satisfaction in terms of how costly it is to be serviced by the scheduler and to the level of intrusion that it imposes on the system.

The desirable scheduler's behaviour is that which has the highest level of performance, while incurring the least overhead [37]. A scheduler with high performance and poor efficiency, i.e., high overheads, is preferable to one which is more efficient but achieves less performance. The scheduler's main objective is to fulfil its performance requirements. Evaluation of performance and efficiency is very difficult due to their inherent entanglement. Imposing low overheads might prevent the scheduler from meeting its performance objectives due to improper decision making, whereas imposing too much overheads will lead to the same undesired final result, since too much resources are dedicated to the managing function. In order to correctly balance these two factors the scheduler must perform, either explicitly or implicitly, a profitability determination analysis [175, 178], which estimates the potential gain obtainable with a given action and weighs it against the overheads associated with that action.

The scheduler's evaluation must be done in terms of the level of performance achievement, rather than using some second–order metrics. An example of such metric is the standard deviation of CPU time across all nodes, used to evaluate some load balancing schedulers [29, 30, 120]. The performance objective is, usually, to reduce execution time, not to balance the system's load. A well balanced system, presenting low standard deviation, may have a longer execution time than an unbalanced one, due to the overheads imposed by the scheduler. Although second–order metrics may be important indicators of the scheduler's behaviour, the designer can not forget which are its real performance objectives.

3.7.2 Cost Analysis

The design of an efficient scheduler requires a careful analysis of the costs involved, the reasons why these costs exist and what are the potential benefits associated with them. The costs induced by the scheduler depend on the environment where it operates and on its performance objectives. On a distributed computing system, scheduling overheads usually include CPU time, memory space and communication bandwidth.

Some scheduling overheads represent the resources the scheduler consumes, since they result directly from the scheduler's activity. These are referred to as *direct costs* (table 3.1). They depend on the scheduler's strategy complexity and on the frequency with which it triggers its various mechanisms.

By understanding how direct costs are distributed across the scheduling policy components, the programmer can optimise the scheduler's algorithm, in order to increase its efficiency. Table 3.2 presents the distribution of direct costs over these components.

Besides direct costs, there are also *indirect costs*. These are consequences of the scheduler's selected actions. By changing the resources allocated to the application's tasks, the scheduling agent modifies both the application's course of action and the pattern of resources' utilisation. These changes on the environment's behaviour may cause additional overheads that reduce the scheduler's efficiency.

Indirect costs depend on the application and distributed system being managed. Three different kinds of indirect costs occur for many applications, in distributed memory parallel systems:

work replication – when a task is transferred among nodes there is the possibility that some work has to be done by all nodes involved in the task migration; this is usually true in data parallel applications, where splitting a data region into several sub–regions can require that values associated with the regions' boundaries be computed at all nodes;

resources' idle times – even though the scheduler tries to minimise the resources' idle times, there are several reasons why it may fail to keep all resources busy during the whole execution time; it may, for instance, fail to find a suitable partner for a task transfer, or it may decide not to transfer additional work to an idle node, because this migration's overheads would be larger than the respective benefits; above all, it may make decisions that increase the resources' idle times due to errors in its execution model or due to poor accuracy on the environment's metrics;

remote data access overheads – the tasks' migrations decided by the scheduler might increase the amount of remote data required by the tasks, by reducing data locality.

Scheduling activity	Comments
Estimation of the environment's metrics	These metrics are the scheduler's environment inputs. The means used to compute them constitute the scheduling agent's sensory apparatus. **Benefits** – the more accurate and diverse these metrics are, the more knowledge is available about the environment's state, hence the more potential for good decision making. **Costs** – this consumes both CPU time, required to compute the metrics, and communication bandwidth, since some metrics may be related to the network subsystem.
Information messages	In a distributed system the environment's metrics are computed across the whole system and must be communicated to the scheduler. The scheduler itself may be distributed and exchange state information among its components. **Benefits** – related to the estimation of the environment's metrics. **Costs** – communication bandwidth and CPU time. Many distributed systems – hardware and software – require messages to be packed/unpacked and eventually routed by the computing nodes, which may consume CPU time.
Control messages	Depending on the protocol used by the scheduler, it may need to exchange control messages among its various components and the processes it is managing. **Costs** – communication bandwidth and CPU time.
Selection of the most adequate action	Based on what it knows about the environment's state and on its performance requirements, the scheduler must select the best action to take at each instant. **Benefits** – the better the quality of the decision making process, the higher the degree of performance achievement. **Costs** – CPU time.
Execution of the selected actions	After selecting which action to take, the scheduler must execute it. The set of available actions constitute the scheduler's effectual capabilities. This set depends on the scheduler's design, but it may include: do nothing, workload transfer, change the scheduling policy, collect more information about the environment's state. **Costs** – most actions require the engagement of one or more nodes' CPUs. They may also consume communication bandwidth, since the scheduler must communicate with the appropriate nodes and work might be transferred among them.

Table 3.1: Direct costs induced by the scheduler

Sub–policy	Role	Direct Costs
Information policy	Metrics estimation Information messages	CPU time Communication bandwidth
Transfer policy	Selection of the most adequate action	CPU time
Location policy	Selection of the most adequate action	CPU time
Selection policy	Selection of the most adequate action	CPU time
Transfer mechanism	Execution of the selected actions	CPU time Communication bandwidth

Table 3.2: Direct costs distribution across the scheduling policy components

Indirect costs may prevent the scheduler from achieving its performance objectives, and should, therefore, be taken into account on the decision making process. Indirect costs are related to the quality of the decisions made by the scheduling agent. An high level of indirect costs, that exceeds the benefits attained with the selected actions, suggests that the scheduler is not making the correct decisions.

3.7.3 Scalability

Scalability can be defined as a system's ability to achieve a performance which is proportional to its hardware and software resources' capabilities. According to Hwang [73], a computer system is scalable if it can scale up (improve its resources) to accommodate ever increasing performance and functionality requirements and/or scale down (decrease its resources) to reduce costs.

There are various dimensions of scalability, including:

size scalability – refers to adding more hardware resources, such as processors, memory, disks, etc.

software scalability – improvement of some software component, such as operating system, compiler, etc.

technology scalability – achieved by using next–generation components.

A scheduler is scalable if it is able to achieve its performance objectives as the system's resources scale. The scheduler's efficiency must be kept constant as the system size increases, i.e., it should be independent of system size [92, 125, 171].

Unfortunately, the overheads of managing a parallel system increase with its size. An application's speedup does not increase linearly with the system's resources capabilities, instead it tends to saturate [49]. It is common, however, to scale the problem's size with the system's size to keep a constant efficiency [64]. The isoefficiency function [60] studies the rate at which the problem's size must increase to keep a constant efficiency. This function is a very powerful tool to perform scalability analysis and has been applied to the load management problem [95]. The rate at which the problem's size is required to grow with respect to the number of processors to keep a constant efficiency depends on the problem being solved, the algorithm used and the machine where the application is implemented. The isoefficiency function captures the characteristics of an algorithm/architecture combination in a single expression, allowing the evaluation of a particular system for a range of problem and system sizes.

3.7.4 Stability

Stability is the scheduler's ability to detect when the effects of further actions (which consume the resources being scheduled) will not improve the environment's state. A stable algorithm will return the environment to a state of equilibrium after a perturbation from this equilibrium and, in the absence of further input, ceases to take actions which cause changes in environment's state in finite time [25]. In the context of load management, a perturbation is caused by a sudden change in the environment's behaviour, due either to the arrival or removal of tasks or background workload, or to modifications on the tasks' activity, which may cause imbalances among the nodes' loads. Instability relates to the amount of schedulable resources being consumed by the scheduler, while the environment's state is changing, but not moving towards a more stable state.

According to the queueing theoretic perspective, when the tasks' long–term arrival rate is greater than the rate at which the system can perform work, the CPU queues grow unbounded; such a system is unstable. This is the case when the sum of the load due to external work arrival and the load due to the overhead imposed by the scheduler becomes higher than the system's service capacity [25, 92, 93, 125, 133, 160, 166]. However, the scheduler should be robust enough to deal with such a condition for a finite period of time.

The scheduler's effectiveness is measured, in most cases, by task average response time. So a stable system can be defined as one whose task response time is bounded for any reasonable excitation (input) [166]. The problem is, bounded by what? Simply saying that task response time should not go to infinity, only covers a limited situation where the system enters a 100% thrashing state. Establishing a practical bound is rather difficult, since a degradation of response time is expected as the system's load increases. In reality, the task response time is, usually, a non–linear function of the system's load. Ad-

ditionally, there are requirements which are specific to the policy and to the environment being managed. It is quite easy to debate stability using vague terms, such as *"too much task movement"*, *"bounded response time"*, *"don't over or under react"*. But quantifying these terms is dependent on the scheduler designer's subjective view of what constitutes reasonable behaviour. Stankovic [166] argues that a stability definition should include:

- a requirement for boundness;
- a list of requirements for the generic issues of scheduling algorithms;
- a list of requirements for stability issues specific of the particular scheduling policy being used;
- a list of requirements for stability issues induced by the environment.

The author claims that scheduling algorithms need specific mechanisms to handle all the stability requirements and the many special cases that can occur under the diverse environment's conditions, as opposed to only tuning the algorithm's parameters. A common mistake when designing such algorithms is to focus on fine tuning the equilibrium decisions. However, the actual dynamic environment in which the algorithm is required to operate may differ from this equilibrium state.

Casavant [25] identifies four sources of instability and their effects on the environment's behaviour:

intolerance instability – if the algorithm reacts to small imbalances in load distribution, it may enter a state where tasks are continuously transferred among a given set of nodes to correct small load differences — processor thrashing. To avoid this type of instability, the definition of optimal load distribution may be relaxed, allowing the system to tolerate small imbalances.

overresponse instability – this is caused by an attempt to respond too fast to local imbalance conditions. When an imbalance is detected, the system tries to transfer a large proportion of work to achieve an optimal load distribution as soon as possible. This may increase dramatically the number of transfers, causing instability, and therefore decreasing the scheduler's performance. In fact, the author argues that an increase in load movement is always accompanied by an increase in load distribution variance and, hence, in poor performance.

high load instability – the scheduler's activity is related to the total amount of load present in the system at any given instant. As the load increases, the opportunity for load imbalances and to overreactions becomes greater. This may lead to instability. In a system with low level of load, the opportunity to create large imbalances is not

present, and this kind of instability is avoided. As the amount of load increases, a corresponding decrease in system performance might be expected. However, the decrease in performance is not proportional to the increase of load. This statistic supports the belief that under conditions of heavy system load, the scheduler may do more harm than good with respect to system performance.

invalid assumption instability – it results when the load distribution algorithm violates certain assumptions made by the designer. It may happen, for example, if the system was designed to run on a certain environment and it is actually running on one with different characteristics.

It is generally accepted that activities related to remote execution should be restricted to a small percentage of the system's activity, otherwise the system might be unstable [92, 182]. Kremien and Kramer [92] propose two measures to evaluate a scheduler's stability:

% of remote execution – related to the number of task transfers decided by the scheduler. This figure must be minimised;

hit–ratio – the ratio of successful decisions. A successful decision is one which contributes to achieve the scheduler's performance goal. An unsuccessful decision results in useless information exchange and/or task movement. The hit–ratio is measured as the ratio of successful decisions to the total number of decisions. This ratio must be maximised.

Computing the hit–ratio requires the identification of all the right and wrong decisions made by the scheduler. This may be somewhat difficult and leads to what is known as the *credit–assignment problem* [115, 170]: how to distribute credit for success among the many decisions that have been involved in producing it?

3.8 Summary

In order to design an effective application level scheduler, the application programmer must take into account a number of basic concepts, which are discussed throughout this chapter.

An application level scheduler managing a distributed shared system should employ a dynamic scheduling policy, in order to exploit fluctuations on the environment's behaviour. Furthermore, its execution model must be relatively complex, in order to deal with these environment's inherent complexity.

Scheduling policies can be described in terms of four sub–policies — information, transfer, selection and location policies — each establishing a set of rules for the various scheduling

activities. The selection of the appropriate load metrics, handled by the information policy, is of crucial importance in determining the scheduler's effectiveness. These must accurately predict the near future workload in each resource. Application level schedulers can use application–dependent load metrics, which convey more information about each resource's performance on application specific tasks than application independent metrics.

Since the scheduler itself consumes the resources it is managing, it must be evaluated by the relation between the performance obtained with its decisions and the costs with which that performance is achieved. The desirable system's behaviour is that which has the highest level of performance, while incurring the least overhead. Scheduling overheads can be divided into direct and indirect costs. Direct costs are related to the resources consumed by the scheduler's activity, while indirect costs are consequences of the scheduler's selected actions. Scheduling overheads, both direct and indirect, are identified. A thorough understanding of these costs is essential to enable the design of an effective and efficient scheduler.

Chapter 4

Load Management: Algorithms

Contents

4.1 Classification of Scheduling Algorithms 52
4.2 Scheduling Policies 61
4.3 Distributed Job Management Systems 72
4.4 Summary 74

A wide range of different scheduling policies can be found on the specialised literature. These exhibit very different performance requirements and are targeted towards different environments. Even though most of these policies do not address the problem of uncertainty, it is useful to have a global view of some of the proposals that have been made to solve the scheduling problem, in order to better understand it. This chapter presents an overview of some of the most well known scheduling policies, enhancing some of their features, whenever these are specially relevant to the current work.

A systematic classification scheme is required to enable the comparison of different scheduling policies. This classification scheme must facilitate the organisation of the policies' main properties, and must clearly distinguish the algorithm's main features from its low–level implementation details. Section 4.1 presents several classification schemes and justifies the selection of one of them.

In the last few years some packages, usually referred to as Distributed Job Management Systems (DJMS), have appeared, whose role is to schedule distributed systems' resources at the system level, ensuring high throughput, reliability, transparency and enforcing the distributed system's owner usage rules. Although the present work is focused on application level scheduling, these higher level schedulers have to deal with some similar concerns, like uncertainty about the system's state. This suggests that some of the ideas discussed throughout this thesis can also be applied to the DJMS scheduler. Section 4.3 briefly discusses some issues related to Distributed Job Management Systems.

4.1 Classification of Scheduling Algorithms

Many different scheduling algorithms are described in the specialised literature. Most of these descriptions, however, are not systematic, consisting in a mixture of text, drawings, pseudo–code and inconsistent terminology. It is quite hard to distinguish the algorithm's main features and properties from the low–level implementation details. One's ability to evaluate and compare the various algorithms is severely impaired by the absence of a common reference framework. Many authors claim that a systematic classification scheme is needed.

The goal of developing a classification scheme is to increase and organise overall knowledge about a class of problems. There are at least four properties that a classification scheme should exhibit [9]:

1. to identify the problem's most significant characteristics;
2. to clearly show the relationships among problems; one problem's solution may indicate the solution to a related problem;
3. expandability and contractibility, which allow important properties of the problem to be focused on and unimportant details to be ignored;
4. to clearly separate the problem's specification from a particular solution.

The concepts used to classify the algorithms are also useful for analysis purposes and to design new scheduling algorithms in a methodical way.

4.1.1 Decision Base × Migration Space

Luling et al. [108] claim that a very complete description of a load management policy may become too complex for a general discussion. They propose a simple approach which classifies some general characteristics of the algorithm.

Any dynamic scheduler can be separated into a decision component and a migration component. The decision component may use local load information (the resource itself and its immediate neighbours) or information from the whole system. The former is referred to as "local decision base", whereas the latter is referred to as "global decision base". The migration component may allow tasks to migrate to direct neighbours only or to any of the system's nodes. The former is referred to as "local migration space", whereas the latter is referred to as "global migration space".

According to this distinction between local and global spaces, a new ordering is introduced with respect to the decision base and the migration space. A further distinction is made

with regard to the initiator of the load distribution activity: sender (s), receiver (r) or symmetric (sr).

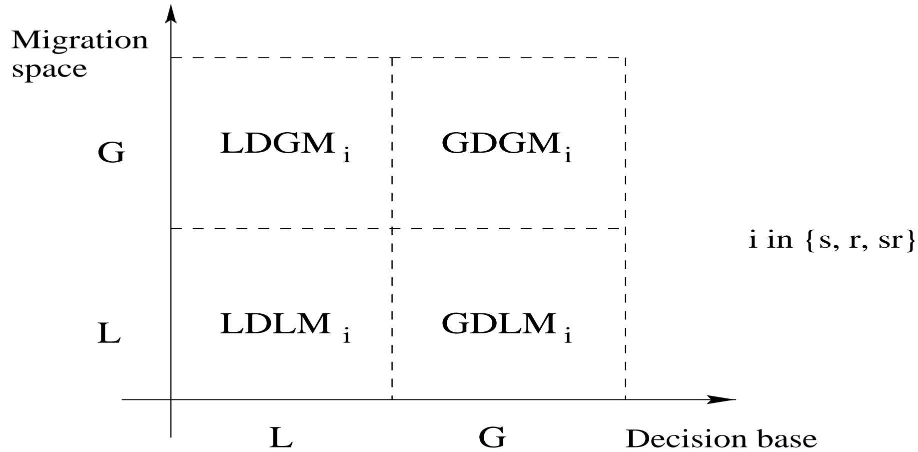


Figure 4.1: Decision base × Migration space

This classification is neither expandable nor contractible, ignores some important problem features and does not separate the problem's specification from its solution. In fact, only the scheduling policy is classified; no reference is made to the particular characteristics of the scheduling problem.

4.1.2 Casavant's Taxonomy

The taxonomy presented by Casavant and Kuhl [24] is hierarchical as long as possible in order to reduce the total number of classes, and flat when the system's description may be done in an arbitrary order. The levels in the hierarchy have been selected in order to keep the description of the taxonomy itself small, and do not reflect any ordering of importance among characteristics. This point is specially important with respect to the positioning of the flat portion of the taxonomy near the bottom of the hierarchy. The structure of the hierarchical portion of the taxonomy is shown in figure 4.2.

For the flat portion of this classification the authors propose a number of characteristics already discussed in chapter 3:

- Adaptive *versus* Non–Adaptive
- Load Balancing *versus* Load Sharing
- Bidding
- Probabilistic

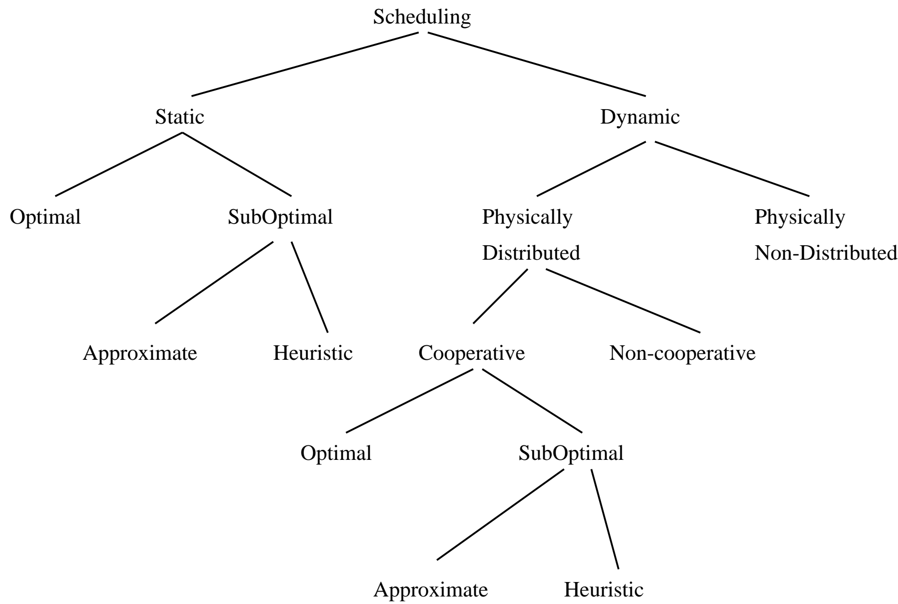


Figure 4.2: Casavant's hierarchical partial taxonomy

- One–time Assignment *versus* Dynamic Reassignment – presented as preemptive and non–preemptive task transfers

This classification identifies a set of the scheduling policy's most significant features and allows expandability and contractibility. However, it does not include any independent classification of the particular scheduling problem being solved.

4.1.3 Families of Strategies

The classification scheme proposed by Jacqmot and Milgrom [75] supports descriptions of various levels of abstraction, starting from general aspects and gradually increasing the detail down to some specific concerns. The approach reveals the existence of typical behaviours (families) and of standard building blocks for constructing their descriptions. The authors argue that two orthogonal facets should be considered separately: movement of processes and handling of information.

With regard to the various process movement policies there are four basic operations (functional units) which appear repeatedly:

- identify_source_node or identify_target_node: this corresponds to find a suitable transfer partner;
- identify_candidate_process: selection of the task/process to transfer;
- move_candidate_process: mechanism of process transfer.

According to these basic operations 4 process movement families (PM) are identified:

PM Family 1 no explicit identification of the candidate process; this identification is not the scheduler's responsibility;

PM Family 2 identification of the candidate process precedes identification of the target node;

PM Family 3 identification of the target node precedes identification of the candidate process;

PM Family 4 identification of the source node precedes identification of the candidate process.

Information management policies (IM) may also be decomposed onto a limited number of common building blocks:

- compute_set_of_correspondents
- interrogate_correspondents
- send_back_information (reply to information request)
- disseminate_information (without previous request)
- update_information_table
- compute_information

According to these blocks 3 IM families are identified:

IM Family 1 no immediate available information; information is gathered at the moment of decision and discarded afterwards;

IM Family 2 total information available; information about the nodes' states is maintained and used to search for a complementary node;

IM Family 3 partial information available; information about the nodes'states is maintained but may be out–of–date; it is used as a clue to nodes worthwhile polling for obtaining actual data.

At this stage all activities which play a major role in the PM and IM strategies have been described. Localisation aspects must now be considered. These include the distribution of a

single component onto several nodes and the degree of replication which will determine, for example, which components are centralised, totally distributed or organised into domains.

Finally, a number of specific concerns, which may or may not be addressed by a particular scheduler, are distinguished. These concerns have a major impact on the scheduler's overall quality and include: achieving global coherence among multiple activations of the scheduler's mechanisms (all actions executed must converge towards the same global objective), avoid flooding of nodes (coordination problem), avoid processor thrashing, minimise overheads due to the scheduler's activity, minimise task movements and encourage most useful task transfers.

This classification scheme, besides being very inflexible, is neither expandable nor contractible, and does not include a separate characterisation of the particular scheduling problem being solved.

4.1.4 Load, Action and Solution Model

Riedl et al. [139] propose a three models structure to classify the different ways of solving the scheduling problem:

Load model describes the workload the scheduler has to deal with, comprises a formal description of runtime environment states and describes the information available about these states; most usually the load model, which represents the knowledge about the workload, cannot be determined from the algorithm, unless explicitly described by its designers;

Action model defines all the possible actions for each of the scheduler's components; it deals with distribution actions and information collection;

Solution model sets the algorithmic context for task transfers and information collection decisions; it reflects the objectives the algorithm is supposed to pursue and the strategy it follows; only actions defined in the action model are considered.

In order to distinguish among the solution model and the solutions obtained with the implementation, the actual problem solving is addressed by the **solution making procedure**.

Finally, the authors argue that scheduling algorithms should be classified according to five criteria: objectives, type and amount of used information, initiating instance (sender, receiver, symmetric), time of activation and source of distribution (central, distributed, domains).

This classification separates the problem specification from its solution. Whereas the problem specification can be done, at least partially on the load model, the action and solution models are used to characterise the solutions. Although the authors do not include any classification of the distributed system being managed, this could possibly be done on the load model. The methodology and terminology used to characterise each of these models is left undefined by the authors. This can be made expandable and contractible by selecting an appropriate representation.

4.1.5 The ESR Classification Scheme

Baumgartner and Wah [9] present a classification scheme based on the scheduling problem definition given in section 2.1.1. This scheme completely separates the problem's specification from its solution's specification.

The problem classification establishes three groups of attributes corresponding to the scheduler's input components: the events (which refer to the workload characteristics), the surroundings (which refer to the computing system) and the scheduler's performance requirements. ESR stands for Events, Surroundings and Requirements. At this stage no attempt is made to classify the solutions to the scheduling problem. Only the problem's properties are classified.

The actual attributes used in each of these groups are not specified. Attributes can be added in order to better describe each particular problem. The classification scheme's expandability and contractability arises from this characteristic, since the set of attributes to use and their possible values can be enlarged or restricted to describe the scheduling problem at the most adequate level of abstraction.

The set of attributes used throughout this thesis are described in table 4.1. These were selected in order to enhance the problem's more relevant characteristics. The events, which succinctly describe the application's workload, are classified according to the kind of dependency among tasks, the pattern of tasks' arrivals and their resources' requirements and the decomposition used to implement the parallel application. The distributed system is classified according to the heterogeneity of the resources, their number, physical characteristics and availability — are the resources constantly available to the application or do they present a stochastic availability, maybe because they are shared with other applications? The communication model and overheads are also included. Finally, the performance requirements are specified, including the scheduler's goal and the level of performance to be attained.

The notation proposed by the authors suggests that the problem's specification must be written between brackets, as in the following example:

<table>
<tr><th>Category</th><th>Attribute</th><th>Values</th></tr>
<tr><td rowspan="4">Events
(tasks)</td><td>Dependence among tasks</td><td>independent
precedence
communication</td></tr>
<tr><td>Arrivals</td><td>static
periodic
stochastic</td></tr>
<tr><td>Resource requirements</td><td>deterministic
stochastic</td></tr>
<tr><td>Decomposition</td><td>functional
domain – indivisible tasks
domain – divisible tasks</td></tr>
<tr><td rowspan="6">Surroundings
(system)</td><td>Classes of Resources</td><td>homogeneous
heterogeneous</td></tr>
<tr><td>Number</td><td>1, n</td></tr>
<tr><td>Physical characteristics</td><td>speed
memory size</td></tr>
<tr><td>Availability</td><td>deterministic
stochastic</td></tr>
<tr><td>Communication overhead</td><td>none
deterministic
stochastic</td></tr>
<tr><td>Communication model</td><td>DSM
message passing
other</td></tr>
<tr><td rowspan="2">Requirements</td><td>Goal</td><td>execution time minimisation
throughput maximisation
real–time
fault–tolerance</td></tr>
<tr><td>Quality</td><td>any solution
sub–optimal
optimal</td></tr>
</table>

Table 4.1: The ESR Scheduling Problem Classification Attributes

$$E : \left\{ \begin{array}{l} \text{precedence dependencies} \\ \text{stochastic arrivals} \\ \text{stochastic requirements} \\ \text{functional decomposition} \end{array} \right\} - S : \left\{ \begin{array}{l} \text{7 processors} \\ \text{homogeneous resources} \\ \text{300 MHz; 128 MBytes} \\ \text{stochastic availability} \\ \text{stochastic communication overheads} \\ \text{message passing} \end{array} \right\}$$

$$R : \left\{ \begin{array}{l} \text{real–time} \\ \text{suboptimal performance} \end{array} \right\}$$

Finally, a classification of scheduling strategies is included, which allows a complete separation between the problem specification and particular solutions. This classification includes a set of attributes, which are also selected by the programmer, maintaining, therefore, the expandability and contractibility properties. Table 4.2 presents the set of attributes that are used throughout this thesis to classify the scheduling strategies, when appropriate. The Information Space attribute indicates whether the scheduling agent has an image of the entire environment's state (global) or an image of a given sub–domain (local), whereas the Migration Space attribute indicates whether a task is allowed to migrate to any suitable resource (global) or just to a particular subset (local). The separation between the problem and the solution specifications, and the utilisation of the same set of attributes to classify different problems and solutions enables the identification of relationships and differences among these. Identical solutions for different problems can be identified, as well as which are the most common characteristics appearing on solutions for problems with particular attributes. A thorough analysis of the scheduling problem as a whole, is thus made possible by this feature of the ESR classification scheme.

4.1.6 Selection of a Classification Scheme

Clearly, only the Load, Action, Solution Model and the ESR classification schemes have the attributes required for an adequate classification scheme: expandability, contractability and separation of the problem and solution specifications. Since the former leaves many aspects undefined, such as, for example, how to characterise each model, and the latter includes an explicit description of the distributed system being managed, ESR is the classification scheme selected to use throughout this thesis.

The ESR classification does not impose a set of attributes to classify each of its components. Although this characteristic can be beneficial, since the programmers have the freedom to select the most suitable attributes for their problems, it can make the comparisons of different problems and respective solutions quite difficult, if the sets of attributes used

Attribute	**Values**
Information Space	local global
Migration Space	local global
Adaptation	static dynamic adaptive
Location of Control	distributed hierarchical centralised
Kind of Transfers	one–time assignment non–preemptive preemptive divisible tasks
Decision Mechanism	deterministic stochastic
Level of Environment's State Information	none simple detailed
Application's Execution Model	none deterministic stochastic

Table 4.2: The ESR Scheduling Strategies Classification Attributes

by different people are substantially different. The set of attributes used throughout this thesis is the one presented in tables 4.1 and 4.2, therefore this problem does not exist.

4.2 Scheduling Policies

A very large number of different scheduling policies, with various performance objectives, can be found in the specialised literature. Their main focus ranges from dedicated parallel machines to shared distributed systems, from real–time constraints to execution time minimisation, from attempts to solve the scheduling problem for the general case to concerns with balancing a specific application.

This section presents some representative approaches to the scheduling problem that illustrate the concepts described throughout chapter 3.

4.2.1 Centralised Algorithms

Zhou proposes a centralised algorithm [182], designated as CENTRAL. A node is considered a sender when its workload, measured as the remaining expected execution time of all jobs allocated to this node, exceeds a given threshold T. Only jobs with an expected execution time above a threshold T_{CPU} are eligible for remote execution, which requires that the job's execution time is known in advance. To obviate this restriction it is possible to classify the jobs either as 'big' or 'small'. Periodically all nodes send their workload information to one of them, which acts both as the load information centre and the central scheduler. When this central scheduler receives a request for transfer, it selects the node with the shortest queue length and sends its identification to the requester. The author concludes that for a limited number of nodes and a relatively efficient communication medium the centralised approach to load information distribution and job placement may be simple and efficient.

Theimer and Lantz [171] propose a similar strategy, but the number of nodes that send update messages is restricted by sending a workload cutoff value to all nodes. Nodes with a workload above this value do not send any load information updates until their load falls below the cutoff value. Only the workload of nodes able to receive additional work needs to be known to the central scheduler. The authors argue that this centralised approach is highly scalable.

The Processor Farm paradigm allows underloaded nodes to request work from a central scheduler which manages a work pool. New tasks are sent directly from the central scheduler to the requesting node. The contention on the scheduler can be reduced by using a hierarchical management structure. The central scheduler can select some sub–managers, which will be responsible for a sub–pool of work and a sub–domain of the distributed sys-

tem. This is referred to as the Concurrent Access Protocol (in opposition to the Sequential Access Protocol) [3]. If nodes can dynamically generate new tasks, then the work pool becomes decentralised, but this can still be managed by a central scheduler [27, 146].

Most authors argue that centralised approaches are not scalable, because the central component is potentially a bottleneck. The functional capacity of any centralised server is bounded and cannot grow when the system where it is embedded is enlarged [92, 108, 125, 160].

4.2.2 Nearest–Neighbour Algorithms

Nearest–neighbour algorithms use only local information and make only local task transfers. Global load management is achieved due to the overlapping of the various domains. These algorithms are naturally iterative, in the sense that tasks are transferred successively between neighbour nodes, each step according to a local decision of the sending node. They rely on successive approximations to a global optimal workload distribution, and at each step need only to concern with the direction and amount of workload migration. Some algorithms select a single direction (a single nearest–neighbour), while others consider all directions (all nearest–neighbours). There are three classes of nearest–neighbour algorithms: dimension exchange, diffusion and the gradient model [29, 71, 179]. Within the dimension exchange model a node balances its workload with its neighbours one at a time. Within the diffusion model a node balances its workload with all its neighbours. It may diffuse fractions of its workload to one or more neighbours, while simultaneously requesting some workload from its other neighbours. Within gradient models each node must transfer a fraction of its workload along the direction of the nearest lightly loaded node [179].

Most iterative nearest–neighbour schedulers proposed in the literature aim to balance the workload over the distributed system's nodes. As previously discussed, load balancing can result in useless task transfers and system's instability.

Dimension Exchange

This model appeared initially with hypercube multicomputers, but it was extended to arbitrary topologies using edge–colouring. The usual approach is to equalise the workloads of the two nodes involved in a balancing operation over an edge of the network. Xu and Lau [179] showed this to be non–optimal for general network topologies and generalised the Dimension Exchange algorithm using an exchange parameter λ to control the splitting of workload among a pair of directly connected nodes. If nodes i and j are direct neighbours, then the change of load on processor i at instant t is modelled as

$$w_i^{t+1} = (1 - \lambda)w_i^t + \lambda w_j^t$$

A single value of λ is used to the entire network and its optimal value is closely related to

the network topology and size.

Diffusion

The main difference between dimension exchange and the diffusion approach is that the latter implies that a node simultaneously balances its workload with all its nearest–neighbours. It has been shown that diffusion inspired algorithms do converge to a balanced workload distribution given any initial workload distribution. However, dimension exchange algorithms seem to outperform diffusion algorithms in small and medium scale systems [179].

Scheurer et al. [94, 151] present the natural diffusion algorithm, where the fraction of load each node has to change with its neighbours is described by a double stochastic diffusion matrix P_G. This matrix is obtained by applying a Poisson operator to the adjacency matrix of the undirected connected graph G that describes the network's topology. It has been shown that, for a suitable P_G, natural diffusion converges towards an uniform distribution. However, convergence may be slow depending on P_G and on G's connectivity. An alternative approach is to use partial diffusion. The graph G is split into partitions, which may overlap. The load is balanced by carrying out a balancing step (e.g. natural diffusion) inside each partition. For many partitions of the load balancing graph the convergence rate of partial diffusion can be shown to be superior to natural diffusion.

The Gradient Model

Within this model each node is classified as low(L), normal(N) or high(H), according to its workload. Each node knows its distance to the nearest L–load node and knows in which direction it should transfer a task so that it is oriented (according to the underlying topology) towards it. Tasks are transferred to one hop away neighbours only (nearest–neighbours) [100, 108, 117].

Let $G \equiv (V, E)$ be a processor network, V being the set of processors and E the physical links. $w : V \rightarrow \{0, 1, \ldots, D(G) + 1\}$ is defined as follows: $w(i)$ is the length of the shortest path from node i to a processor which is in L–state and $D(G)$ is the diameter of G. $w(i)$ equals $D(G)+1$ if there is no L–load processor. $w(i)$ is maintained by node i. The collection of all w values is the gradient surface or pressure surface. The pressure function, at instant t, is defined as

$$p^t(i) = \begin{cases} 0 & \text{if } t = 0 \text{ or node } i \text{ is L–loaded} \\ 1 + min(D(G), min(\{p^{t-1}(j) : j \in \text{Neighbours}(i)\})) & \text{otherwise} \end{cases}$$

When a processor is highly loaded, it transfers some of its load to its nearest–neighbour in the direction of the nearest L–loaded node, which is the one with minimal pressure value.

It has been experimentally shown that this policy deals very slowly with large areas of

L–load nodes. Some extensions have been proposed to enhance the basic gradient model [108]. $\overline{w}$ is defined as $\overline{w} : V \rightarrow \{0, 1, \ldots, D(G) + 1\}$. $\overline{w}(i)$ is the length of the shortest path from node i to a node in state H. $\overline{w}(i)$ is $D(G) + 1$ if no H–load node exists. The pair $(\overline{w}(i), w(i))$ is maintained by node i. The collection of all such pairs is the extended gradient surface. The collection of all $\overline{w}(i)$ values is the suction surface. The suction function, at instant t, is defined as

$$s^t(i) = \begin{cases} 0 & \text{if node } i \text{ is H–loaded} \\ D(G) + 1 & \text{if } t = 0 \\ 1 + min(D(G), min(\{s^{t-1}(j) : j \in \text{Neighbours}(i)\})) & \text{otherwise} \end{cases}$$

Nodes in N–state can send some of its load to the neighbour with maximal suction values, which is the farthest from H-load nodes. The extended gradient model is robust with respect to load characteristics, performs well in large networks and is easy to implement.

4.2.3 Random Location Algorithm

Eager et al. [39] present a thorough study of the random location algorithm. A node is considered a sender if its CPU queue length exceeds a given threshold. The destination node is randomly selected among all the system's nodes. No exchange of state information is required, hence this is a non–cooperative algorithm. If the algorithm allows an overloaded destination node to transfer the task once again, the algorithm is referred to as "Uncontrolled Random" and the authors proved that it is unstable for a non–zero task transfer cost. No matter what the environment's average load, there is a positive probability that all nodes will be in a transferring phase simultaneously, and the system will enter a state in which all nodes are devoting all their time to transfer tasks and none to process them. Instability can be overcome by limiting the number of task migrations. The value of this transfer limit is dependent on the ratio between the task processing rate and the task transfer cost.

The random policy yields substantial performance improvement over the no–load sharing case. The degree of performance improvement is surprisingly high for such a simple policy. Zhou [182] also claims that this is a very scalable policy, which is not surprising since it doesn't collect any environment's state information. However, both referred papers present policies with better results.

Alternative approaches randomly select the destination nodes from a system's sub–domain, instead of considering the whole system [108, 158]. Kumar and Grama [95] present an isoefficiency study of a random location policy in several different machine architectures.

4.2.4 Probing and Bidding

The algorithms presented throughout this section employ a demand–driven information policy. Every time a node wishes to transfer some workload, it must inquire the possible candidate nodes about their load state.

Probing

A node willing to participate in a task transfer probes another node to find out if it is a suitable transfer partner. Nodes can be probed either serially or in parallel. A node can be selected for probing on a random basis, on the basis of previously collected information or on a nearest–neighbour basis [160].

Eager et al. [39] propose a probing algorithm which requires that a node willing to transfer a task randomly selects another node and probes it to determine whether this transfer would place it above threshold. If not, the task is transferred, otherwise another node is probed. This process is repeated until a suitable transfer partner is found or a probe limit is reached.

This algorithm usually performs substantially better than the random policy, which suggests that the overhead of collecting some amount of state information is overcome by the added benefits. Small values are appropriate for the probe limit. If p is the probability of a particular node being below threshold, then the probability that such a node is encountered at the ith probe is $p(1-p)^{i-1}$. If p is large (system lightly loaded) this value decreases rapidly: an underloaded node will be found in the first few probes. For small p (system heavily loaded) this value decreases more slowly. However, there are lots of overloaded nodes trying to find underloaded ones. Each particular node can stop probing after a few tries because another overloaded node will, with high probability, find an underloaded one. Shin and Chang argue that probing is inefficient [158], since it introduces a delay in the transferred tasks and an overloaded node might fail to find an underloaded one while there are some. However, as stated before, this happens with small probability.

Under another approach to probing [39, 182], a set of randomly selected nodes is probed in parallel and the best one (shortest CPU queue length) is selected. This policy's performance is not significantly better than the previous one, suggesting that the benefits of selecting the best target within a set do not overcome the overheads of much more message traffic and interrupting more processors. Other approaches to probing include maintaining some state information about the other nodes' states. This information can be obtained from previous probes or from a low–frequency state dissemination process or both. This state information, which can be outdated, is used as a clue to select nodes for probing [121].

Symmetrically Initiated Adaptive Algorithm

Under this probing algorithm [160], nodes are classified as senders, receivers or normal, based on information got from previous probes. Each node maintains three lists where other nodes' observed states are kept: a list of receivers, a list of senders and a list of normal nodes. Initially, each node assumes that all other nodes are receivers. Each node is allowed to initiate a task transfer either as a sender or as a receiver. The sender component of the location policy is triggered when the node's load exceeds a given threshold. It will probe the node at its receivers list head. The probed node will place the probing node in the head of its senders list, and reply with a message indicating its current state. On receipt of this reply, the sender will transfer a task if the answer indicates that the probed node is still a receiver. Otherwise, it will be removed from the receivers list and put at the appropriate list's head. Probing ends when a suitable receiver is found, the number of probes reaches a limit or the receivers list becomes empty. The receiver initiated component selects nodes for probing first from the senders list, from head to tail, using most updated information first. If this list becomes empty it will then look in the list of normal nodes and then in the receivers list, but from tail to head, using most outdated information first.

At high system loads the sender initiated components will fail to find transfer partners, so the node's receivers list will become empty. Only receivers will try to transfer tasks, which is more effective at such loads. At low system loads receiver initiated probing will be frequent and will generally fail. These failures do not adversely affect the system's performance, since extra processing and communication capacity is available. In addition, they have the positive effect of updating the receivers lists. Future senders will find receivers with the first few probes. By adapting its behaviour to system's load, this algorithm achieves improved performance over a wide range of load patterns and preserves system stability.

To avoid preemptive task transfers, which may be originated by the receiver initiated component, Shivaratri et al. propose a further extension to this algorithm [160]. Besides maintaining the three lists, each node maintains a state vector indicating to which list it belongs to at all other nodes, using information that circulates in probes and probes' replies. When a node becomes a receiver, it informs all other nodes in the system which are misinformed about its state. There are no preemptive task transfers, the sender initiated component will do any task transfer on tasks' arrivals.

Lu and Lau [106] further extend this algorithm with a guarantee and reservation protocol. A receiver that has accepted work, but has not received it yet, reserves some of its processing power surplus. It will not accept further work from other senders which would overload this node. Task migration among nodes is negotiated on batches, rather than on an individual basis.

Bidding

Bidding is a three–level contract protocol. In the first step, a node willing to transfer some of its load broadcasts a request for bids. In the second step, bids are received from those nodes willing to receive some load. In the last step, the node evaluates the bids and sends the task(s) to the node which sent the best bid. Some algorithms allow the receiving node to reactivate the bidding process if it is now overloaded [166, 167]. Other algorithms [121, 158] allow several rounds of message exchanges before actually transferring a task. This is necessary since the best bidder may become overloaded, and hence unwilling to accept another task. An underloaded node may answer several requests for bids, so it may be selected as a transfer partner by several overloaded nodes. It is possible for the bidding node to consider previous bids it sent (and were not answered yet) when it evaluates whether or not to answer yet another request for bids [25, 134]. The bidding algorithm may be extended to maintain information about previously received bids. This information is used to select a subset of nodes to which the request for bids will be sent, instead of broadcasting it. An adaptive bidding algorithm is proposed by Luling [108], where a request for bids is sent only to those nodes which are at a distance less than d from the requester. The distance d is increased (respectively decreased) if not enough bids (respectively too much bids) are received for the offered load unit within a time interval which also depends on d.

The bidding algorithm has been used in distributed real time systems, where critical tasks must meet their time deadlines. The algorithm requires that tasks deadlines and computation times are known beforehand. When a node can not guarantee that a task will meet its deadline, it issues a request for bids. This algorithm is extended with focused addressing. When a node can not guarantee a task, it selects a node which is supposed to have enough surplus, based on information collected during previous bids. The task is transferred to this focused node and a request for bids is issued with the indication that bids must be sent to the focused node [134, 166, 167].

Drafting Algorithm

Ni et al. [121] claim that the bidding algorithm is interesting due to its generality, but it creates a great deal of communication overhead which lessens its ability to reduce response time. They propose the drafting algorithm, which is a receiver–initiated bidding algorithm, that uses several techniques to reduce the number of messages exchanged. The authors claim that the drafting algorithm alleviates many drawbacks encountered in the bidding algorithm [108, 144].

4.2.5 Flexible Load Sharing Algorithm

The flexible load sharing algorithm (FLS) [93] partitions a system into sets of nodes, called domains. A node only exchanges state information with and transfer tasks to members of its domain. A node determines which other nodes to include in its domain based on their load state: overloaded(O), underloaded(U) or medium loaded(M). Domain membership is symmetrical: two nodes are candidate for inclusion in each other's domain if one is overloaded while the other is underloaded; if node A belongs to node B's domain, then node B also belongs to A's domain. When one node changes state, it informs the other nodes in its domain and that node is discarded. The selection of domain's members is based on messages which contain state–information and is done periodically. To account for the fact that membership selection can be costly, the domain size is limited. Due to information aging, remote nodes' state–information may be outdated. Thus, domain membership is treated only as a clue and a request–reply protocol is used, which allows nodes to refuse new tasks. If a supposedly underloaded node refuses a task, then it is removed from the sender's domain and the task is executed locally.

4.2.6 Distributed Clustering Algorithms

Ozden et al. [125] propose an algorithm that privileges non–intrusiveness. A scheduling algorithm is non–intrusive if the overhead that it induces on each resource is less than the bound on overhead defined by the resource owner. Nodes are grouped into clusters, and a centralised approach is applied for load management within each cluster. The set of cluster managers may change over time to handle changes on intrusiveness' bounds. When a manager receives a task request which can not be satisfied by any node within the cluster, it polls other cluster managers. While coherent global state information would allow minimisation of polling, maintaining global state information would introduce the scalability problems inherent to centralised algorithms. On the other hand, each manager must keep a record of which clusters are currently available for polling to guarantee that the intrusiveness bounds are observed. When a manager detects that it is in danger of violating its cluster's intrusiveness constraints, it removes itself from the load sharing pools, notifying the other managers. The author argues that this policy scales with system size, satisfies bounds on intrusiveness and is able to regulate and control overheads.

Ahmad and Ghafoor [1] propose a distributed clustering approach where the multiprocessor system is partitioned into independent symmetric regions (spheres) centred at some control points (schedulers). The schedulers optimally schedule tasks within their spheres. To identify the spheres and respective centres a combinatorial structure, known as the Hadamard Matrix, is used. It is shown that their approach yields better response time and resource utilisation than a fully distributed nearest–neighbour algorithm.

The Hicon concept [10, 11, 12, 13] provides dynamic scheduling support on heterogeneous workstation networks. It employs distributed clustering, each cluster being managed by a central scheduler and inter–cluster scheduling being done using a decentralised policy. The centralised approach provides sophisticated load control, while the decentralised coupling of clusters ensures scalability. The strategy used takes into account system and application state information, data communication costs between cooperating tasks, access to remote data, inter task dependencies and exploits load profile estimations. It includes several automatic adaptation techniques, like adjustment of migration thresholds, correction of task size estimations, correction of tasks remote data wait time estimations, correction of CPU utilisation estimations and adaptation of the scheduling algorithm complexity.

4.2.7 Stochastic Learning Automata

The basic stochastic learning automata is a probabilistic algorithm. The suitability of each possible action is encoded as a probabilistic value in a data structure referred to as the probability vector. Initially all these probabilities are equal, since nothing is known about the desirability of each action. After an action is executed, an answer will be received indicating whether the action was 'good' or 'bad'. Based on this response the probability vector is updated. This process is known as reinforcement learning [170, 183]. The automaton's learning behaviour occurs in the following manner:

- if the response to a certain action i is favourable, then increase the probability of that action:

$$\begin{cases} p_j = p_j - a * p_j & j \neq i \\ p_i = p_i + \sum_{j \neq i} (a * p_j) & j = i \end{cases}$$

  where a is a reward constant.

- if the response to a certain action i is not favourable, then decrease the probability of that action, using a penalty constant b.

This algorithm has the advantage [166] of modelling the environment's response history in the probability vector, and does not require highly time consuming calculations to make decisions.

This basic scheme has been extended with the introduction of environmental states [96, 166]. Each node knows the states of all other nodes (over or underloaded). Each environmental state is an arrangement of the nodes' states. Each node has a different probability vector for each environmental state and, at each instant, uses the vector corresponding to the current state. When the indication of reward or penalty is received after an action's execution, the probability vector in effect at the time the decision was made (and

not necessarily in effect now) is updated. This extended scheme requires that each node periodically checks its status and broadcasts it.

Schaerf, Shoham and Tennenholtz [150] have studied the utilisation of a distributed multi-agent reinforcement learning system in the context of load management. Jobs are submitted to the agents, which select the appropriate resources. Each agent is a stochastic learning automata, which uses only local information to update its probability vector. The agents never communicate with each other, so they have no knowledge about the environment's state. The learning process is controlled by a parameter w, which determines the weigh of new information on the probability vector. The decision making process is controlled by another parameter n, which biases the decision towards resources that performed better in the past. The larger the value of n the stronger this bias is. Large values of n do not allow the agent to exploit improvements on the capacity or workload of the various resources. The two parameters n and w interplay in the sense that highly exploratory activity (low n) must be matched with giving greater weigh to more recent experience (high w). This is known as the *exploration* versus *exploitation* problem on intelligent agents. Should the agent exploit the knowledge it has about the environment's behaviour on the past or explore new actions in an attempt to improve its effectiveness? The authors show that good results can be achieved using only local information and that the naive use of communication might deteriorate the scheduler's effectiveness.

In the stochastic learning automata presented on the literature, only the desirability of each action, for each environmental state, is probabilistically modelled. The environment's states themselves are modelled using deterministic quantities. In most real cases, the scheduling agent is uncertain of the outcomes of actions and of the environment's current state. Therefore, both these aspects should be probabilistically modelled.

4.2.8 Physics Based Models

Physical optimisation algorithms are based on analogies with physical systems. The load management problem is mapped onto some physical system, which is then solved using techniques from experimental and theoretical physics. G. Fox and P. Coddington [53] apply physics based algorithms to manage the load of scientific simulation applications. The data elements are treated as particles free to move around in the "space" of the parallel machine. Minimising the total execution time requires the minimisation of

$$\max_{\text{nodes } i} C_i$$

where C_i is the total time for computation and communication on node i. This minimax problem is replaced by a least squares minimisation of

$$E = \sum_i C_i^2$$

Labelling the data elements with m and communication between any two data elements m, m' with (m, m'), they show that

$$C_i^2 = \sum_{(m,m'),m\in i} \text{Comm}(m, m') + \sum_{(m,m')\in i} \text{Calc}(m)\text{Calc}(m')$$

In physics, this equation describes an Hamiltonian, or energy function, which must be minimised to achieve the most efficient decomposition of data onto the distributed system's resources. The last term in the Hamiltonian is zero, unless data elements m and m' are on the same node i. This acts as a repulsive force, which spreads the data elements throughout the system. The first term acts as an attractive force among those data elements which need to communicate with one another. This term is proportional to the amount of communication, so that heavy communicating elements stay near one another. For dynamic problems, where data elements change with time, the Hamiltonian will also change and data elements will have to be redistributed periodically. In this case it is the time averaged Hamiltonian $\overline{H}$ that must be minimised:

$$\overline{H}(t, t_{av}) = \int_t^{t+t_{av}} H(u)du$$

The relative importance of each of the terms of the Hamiltonian is governed by the ratio $t_{\text{comm}}/t_{\text{calc}}$, which is a characteristic of the distributed system. The Hamiltonian can be minimised using simulated annealing.

Hui and Chanson [71] propose an hydrodynamic approach for load balancing an heterogeneous parallel system. Each node i is modelled as a cylindrical liquid container, whose cross–sectional area corresponds to the capacity c_i of the node, the communication links are modelled as liquid channels among the cylinders, the workload is represented as liquid and the load balancing algorithm describes the liquid flow. Global fairness is achieved when the heights of the liquid columns in the cylinders are equal. When global fairness is achieved, there is no more liquid movement among the cylinders, therefore the system is stable. In physical terms, the load is balanced across the system when the global potential energy (GPE) is minimised. The GPE is defined as the sum of potential energies (PE) of all the nodes, where $\text{PE}_i = \frac{c_i h_i^2}{2}$, h_i being the height of liquid in cylinder i. This is a nearest–neighbour algorithm as load (liquid) is exchanged only among neighbouring nodes (cylinders).

4.2.9 Economy Based Models

Scheduling resources in a large distributed system is a complex task, similar to allocating resources in human economies. This similarity motivated researchers to apply economy theory to load management [44, 45]. The hope of this research is that concepts that evolved over thousands of years in human societies will also prove effective on distributed systems.

Economic models consist of two types of agents, suppliers and consumers, which selfishly attempt to achieve their own goals. In a computer system, consumers are applications and suppliers are resources needed by the applications, such as CPU time, memory, etc. A consumer attempts to optimise its individual performance criteria by obtaining the resources it requires, while a supplier's goal is to optimise its profit derived from its choice of resource allocations to consumers.

Pricing is the technique used to coordinate the selfish behaviour of agents. Consumers are endowed with money they use to buy resources and suppliers charge consumers for the use of its resources. The price charged by a resource is determined by its supply and the agents' demand.

Resource scheduling is based on the following mechanism. Each agent computes its demand from its utility function and its budget constraints. The aggregate demand from all the agents is sent to suppliers, which then compute new prices. This process continues iteratively until an equilibrium price is achieved. Another form of resource allocation is bidding. With this mechanism the highest bidder gets the resource and the resource's current price is determined by the bids.

A fundamental requirement of economic models is that consumers know its resource requirements (e.g. CPU time) before execution time. This is not realistic for all applications. Furthermore, it is assumed that consumers compete among each others to acquire system resources and that the system's performance criteria is determined by some combination of the individual consumers' performance criteria. This might not be the case when one is interested on a single application's execution time.

4.3 Distributed Job Management Systems

The ever growing interest on using clusters of available and inexpensive computers has led to the appearance of load management systems, which manage the distribution of independent jobs across the cluster. A cluster is defined as an heterogeneous collection of computers on a network, that can function as a single computing resource through the use of additional system management software [87]. This additional software is often referred to as the Distributed Job Management System (DJMS). The DJMS performs some scheduling tasks usually associated with system level scheduling, such as resource allocation at the job level and processes' checkpointing [19]. It assumes many of the traditional operating system responsibilities at the network level, managing distributed computing resources, while preserving an unified system image of what amounts to a virtual supercomputer. All computing resources are made transparently available to all users on the network according to the organisation's policies [130]. Distributed job management has evolved from, and

exhibits similar requirements to, job scheduling on mainframes. The challenge is to build a system that provides mainframe–like levels of robustness and reliability in today's fragile and immature distributed computing environments [74].

There are several commercial and public–domain DJMSs currently available, the most popular being [73, 87, 126]: COmputing in DIstributed Networked Environments (CODINE), Condor [40, 103, 104], Network Queueing System (NQS) [67, 89], Distributed Job Manager (DJM), Distributed Queueing System (DQS), LoadBalancer, LoadLeveler [110], Tivoli Maestro [74], Load Sharing Facility (LSF) [73, 129, 130], Hector [142], Portable Batch System (PBS) and Task Broker. A qualitative comparison of some of these systems can be found in [81, 83, 87].

Key requirements of a DJMS include reliability, security, fault tolerance, efficient use of available resources, heterogeneous platform support and ergonomic administration tools [47, 74, 147].

Most of the available DJMS's allow the user to specify the resources to be used (number of nodes, exclusive CPU access, peripheral devices, etc.) and produce reports about resource usage which can be used both by the users and administrators. The users' requests might be postponed and processed in batch whenever the requested resources become available.

The need for DJMS's has increased with the growing community of users who are now concerned with the throughput of their applications, rather than response time. Scientific and engineering users measure the power of the system by the amount of work performed on a fixed amount of time. Response time makes no sense as the amount of data to process is virtually unbounded. The concern here is not about the instantaneous system's performance, but about the amount of computing harnessed over a long period of time. These users' computing needs are satisfied by High Throughput Computing (HTC) environments, that can take advantage of workstations' idle times. DJMS's can support these environments by opportunistically using idle resources.

Reliability is a very strong requirement for these long–lived applications. Work already done should not be lost in the event of an hardware failure. DJMSs use checkpointing to ensure reliability. A checkpoint of an executing program is a snapshot of its state, which can be used to restart the program from that point at a later time. Besides providing reliability, checkpointing also enables preemptive migration. Although checkpointing is an expensive mechanism, both in space and time, any attempt to deliver HTC has to rely on it due to the opportunistic nature of resource usage [103, 104].

To improve system's throughput or minimise application's turnaround time some DJMSs, like CODINE, LSF, Dynamic PVM, PBS and MARS, provide dynamic load balancing facilities [2, 47, 57, 103, 104, 124]. In order to make scheduling decisions, the DJMS monitors the current system and network load. New jobs are placed on lightly loaded

nodes and existing jobs are migrated from heavily loaded machines to less loaded ones. One of the major requirements for providing a migration facility is transparency: a process's execution should continue as if the migration never took place. For parallel applications this transparency should also hold for the migrated process's communication partners.

Since the performance goals that need to be satisfied for each site differ considerably, DJMSs offer the means to install multiple scheduling algorithms and to activate them on the fly. A set of different scheduling schemes is usually provided to the system's administrator, including First-Come-First-Served, maximisation of system's throughput, etc.. In addition, the scheduler's library is often designed as an application programming interface, allowing the development of site specific scheduling strategies, addressing individual needs [47].

Recently, DJMS's have evolved to allow for global load sharing by scheduling jobs not only within clusters, but also among them [40, 129]. This enables multi–organisational resource sharing, but leverages several problems like security and resource ownership enforcing, limiting the intrusion level and long–distance communications.

Table 4.3 presents a set of criteria (based on the ones proposed by Kaplan and Nelson [87] and James Jones [82]) which can be used to compare DJMSs.

DJMSs are targeted towards system level scheduling rather than application level scheduling, although some of them include some dynamic scheduling possibilities. Their main goals are, usually, maximising system's throughput and reliability. Most of these systems see the parallel application as a single job, rather than considering its various tasks. They are, therefore, unable to dynamically reschedule the application's tasks in order to meet the application's particular performance requirements. This is often performed by the application level scheduler, embedded within the application's code, which manages the resources allocated by the DJMS.

Uncertainty about the environment's behaviour, however, may be present both at the application and system level. This suggests that some of the ideas discussed throughout this thesis can also be applied to the DJMS scheduler.

4.4 Summary

The effective comparison of different scheduling strategies requires a classification scheme with some fundamental properties, like expandability and contractability, which allow the specification to be done at the desirable level of detail, and a clear separation between the specification of the problem being solved and the particular solutions found to that problem. This separations enables a systematic analysis of different classes of problems and respective solutions, allowing the identification of similarities and differences among

Heterogeneous computers support
Batch support
Interactive applications support
Parallel applications support
Message passing support
Checkpointing
Automatic load balancing
Preemptive process migration
User specified execution time/date
Exclusive use of CPU for job
User specified resources
Job runtime limits
Administrator specified restrictions on resource usage
Relinking applications not required
Graphical user interface
No single point of failure
Cluster's dynamic configuration
Information about resources usage
Information about jobs status
Input/output redirection
Supported operating systems and machines
Resource scheduling policies
Scheduling priorities support
Fault tolerance support
Security
Level of intrusion
Inter–cluster scheduling support

Table 4.3: Criteria for DJMS's evaluation

them. The ESR classification scheme exhibits these properties, hence it was selected as the one to be used throughout this thesis.

Section 4.2 presents an overview of several scheduling policies, with different solutions to the scheduling problem. This global view of various solutions to the challenge of effectively and efficiently manage the workload of a distributed system, helps in better understanding the scheduling problem.

Finally, Distributed Job Management Systems are briefly discussed. Although these are mainly focused on system level scheduling, many difficulties faced by these systems are shared with application level schedulers. Uncertainty about the distributed system's current state is one of these shared difficulties.

Chapter 5

Handling Uncertainty

"Thus, the more precisely the position is determined, the less precisely the momentum is known, and conversely."

Uncertainty Principle
Werner Karl Heisenberg, 1927

Contents

5.1 Notation **78**
5.2 Beliefs Expressed as Probabilities **79**
5.3 Probabilistic Models **80**
5.4 Bayesian Networks **85**
5.5 Making Decisions **92**
5.6 Knowledge Engineering **95**
5.7 Applying Decision Networks to a Dynamic Scheduler **99**
5.8 Summary **104**

The environment where an application level scheduling agent is required to operate is often complex, non–deterministic, dynamic and inaccessible. This means that the agent can neither have a complete, accurate and updated image of the system's state, nor make exact predictions about near future system's behaviour. In every instant there are important questions about the system's state, tasks' requirements and their current execution progress for which no categorical answers can be found. The agent has to act under uncertainty, and it is this uncertainty that hinders the scheduler from fully achieving its performance goals.

Decision theory provides principles and tools for rational decision making under uncertainty. It combines utility with probability in the evaluation of an action. Probability provides a way of summarising the uncertainty that comes from laziness and ignorance.

Utility theory is used to represent and reason about preferences among system's states. Whereas judgements about the likelihood of states are quantified by probabilities, judgements about the desirability of an action's consequences are quantified by utilities.

Decision networks are one of the tools provided by decision theory. They provide coherent prescriptions for choosing actions and meaningful guarantees of the quality of these choices. Decision networks, also called influence diagrams, constitute a general mechanism for rational decision making under uncertainty.

The hypothesis put forward by this thesis is that decision networks, if applied to the scheduling agent's execution model, may improve its effectiveness and help overcome the problems caused by uncertainty.

This chapter begins by explaining why beliefs about uncertain quantities can be expressed and combined as probabilities, and describes some fundamental properties that allow the design of computationally tractable probabilistic models of real world problems. Bayesian belief networks are introduced in section 5.4, and extended in section 5.5 to allow automated decision making. Section 5.6 describes the fundamental steps required to assemble a coherent and computable decision basis. Finally, section 5.7 discusses how decision networks may be applied to the scheduling problem and presents a generic structure for such a network.

5.1 Notation

Random variables are denoted by capital letters (e.g X, Y, Z), whereas specific values taken by these variables are represented by lowercase letters (e.g. x, y, z). A discrete random variable X may take on values from a finite domain D_X. The number of different values a variable X can take, i.e., the cardinality of its domain D_X, is labelled by $\#D_X$.

Sets of variables are denoted by boldfaced uppercase letters (e.g. $\mathbf{U}$), and assignments of values to these variables are denoted by boldfaced lowercase letters (e.g. $\mathbf{u}$). If $\mathbf{Z}$ stands for the set of variables $\{X, Y\}$, then $\mathbf{z}$ represents the assignment $\{x, y\} : x \in D_X, y \in D_Y$.

$P(x)$ is used as a short notation for the probability $P(X = x), x \in D_X$. $P(\mathbf{z})$, for the set $\mathbf{Z} = \{X, Y\}$, means

$$P(\mathbf{Z} = \mathbf{z}) = P(X = x, Y = y), x \in D_X, y \in D_Y$$

i.e., the probability that $X = x$ **and** $Y = y$.

The probability distribution of a variable X over its domain $D_X = \{x_1, x_2, \ldots, x_n\}$ is denoted by the boldfaced operator $\mathbf{P}(X)$, representing the vector

$$\mathbf{P}(X) = \{P(X = x_1), P(X = x_2), \ldots, P(X = x_n)\}, \sum_i P(X = x_i) = 1$$

To distinguish between the fixed conditional probability table that describes the variable X probability distribution as a tabular function of all possible combinations of some other random variables, and a variable's probability distribution given what is known, the former is referred to as $\mathbf{CPT}(X|Y, Z, \ldots)$ and the latter as $\mathbf{P}(X|Y = y, Z = z, \ldots)$.

The conditional probability $\mathbf{CPT}(X|Y)$ is represented by a two–dimensional matrix with $\#D_Y$ rows and $\#D_X$ columns. If $D_X = \{x_1, x_2\}$ and $D_Y = \{y_1, y_2, y_3\}$ then,

$$\mathbf{CPT}(X|Y) = \begin{bmatrix} P(x_1|y_1) & P(x_2|y_1) \\ P(x_1|y_2) & P(x_2|y_2) \\ P(x_1|y_3) & P(x_2|y_3) \end{bmatrix}$$

where the values across each row add up to one, i.e., $\sum_{i=1}^{2} P(x_i|y_j) = 1, \forall y_j \in D_Y$.

5.2 Beliefs Expressed as Probabilities

Due to the inherent uncertainty present in any model of a real world problem, the agent's knowledge is, at best, a degree of belief on the environment's most relevant aspects. Probability theory provides a language for making statements about uncertainty, making explicit the notion of partial belief and incomplete information. Probability assigns numerical values to these degrees of belief, supplying an agent's designer with a way of summarising the uncertainty that comes from laziness and ignorance.

Viewing probability as a belief on a given statement is the approach taken by the subjectivist school. The frequentist approach is that probabilities can only come from counting experimental results. Probability is, therefore, the frequency with which a given event occurs. The objectivist view is that probabilities are real aspects of the universe, i.e., objects' tendencies to behave in certain ways [31, 118, 143].

One common criticism of the Bayesian definition of probability is that probabilities seem arbitrary. Why should degrees of belief satisfy the rules of probability? Many different sets of properties that should be satisfied by degrees of belief have been suggested. Each of these sets leads to the same set of rules: the rules of probability. This provides a particularly strong argument for using probability to measure and combine beliefs [66].

The approach taken throughout this work is that beliefs can be expressed as probabilities, and, therefore, that anyone can assign a probability to a given event, even if that person has never experienced it. This probability merely reflects that individual's own belief that the event will occur.

The belief on a given event, or statement, depends on what is known about the totality of other events, or statements, that are relevant to this one. Beliefs are context–dependent. One is willing to change its beliefs once more relevant information becomes available.

Probabilities are also context–dependent. The assignment of a probability to an event depends on what is known about the relevant aspects of the universe. Hence the existence of conditional probabilities, $P(A = a|B = b) = x$, which means that the probability of A being a is x, given that all that is known is $B = b$. If a new fact $C = c$ becomes known and is relevant to A, then the new probability of $A = a$ must be expressed as $P(A = a|B = b, C = c) = y$. Even prior probabilities, $P(A = a) = x$, which do not seem to be conditioned on any previous knowledge, are, in fact, conditioned by some background knowledge $K = k$, and should be written as $P(A = a|K = k) = x$. In practice, K is omitted because it is assumed to be static, i.e., it does not change in any way that could be relevant to our belief in A.

Probability theory, being unique in the way it handles context–dependent information and uncertainty, is the language used to express beliefs and, furthermore, is the tool used to process and combine these beliefs, providing a coherent account of how the belief on a set of statements must change in the light of additional information [127]. It can be used as an inference mechanism to compute the likelihood of a given statement, given the beliefs on a set of related statements.

5.3 Probabilistic Models

5.3.1 The Joint Distribution

In order to apply probability theory to a given problem, a probabilistic model of the world has to be built. A probabilistic model consists of a set of stochastic variables that can take particular values with certain probabilities. Each variable represents some important aspect of the world being modelled. The values each variable can take on are the variable's domain. This can be a binary domain, a multi-valued domain or a continuous domain. The variable's probability distribution over its domain specifies the designer's degree of belief that the variable will take on a particular value. An atomic event is an assignment of particular values to all the variables in the model. In other words, a complete specification of the world's state.

The joint distribution assigns probabilities to all possible atomic events. If a probabilistic model consists of 3 discrete variables, namely, A, B, C, with $D_A = \{T, F\}$, $D_B = \{S_1, S_2, S_3\}$ and $D_C = \{T, F\}$, then the joint distribution could be as shown in Table 5.1.

The joint distribution allows direct access to the probability of any atomic event and allows the computation of the probability of any sentence formed with the model's variables. For instance, to compute $P(A = T \wedge C = F)$ its is enough to marginalise B out of table 5.1,

A	B	C	$P(A, B, C)$
T	S_1	T	0.05
T	S_1	F	0.015
T	S_2	T	0.025
T	S_2	F	0.1
T	S_3	T	0.01
T	S_3	F	0.05
F	S_1	T	0.04
F	S_1	F	0.1
F	S_2	T	0.2
F	S_2	F	0.15
F	S_3	T	0.2
F	S_3	F	0.05

Table 5.1: The joint probability distribution: an example

by adding up all lines where $A = T$ and $C = F$:

$$P(A = T \wedge C = F) = \sum_{S_i} P(A = T, B = S_i, C = F) = 0.015 + 0.1 + 0.05 = 0.12$$

More formally, the marginal probability of a given sentence can be computed from a joint distribution by using equation 5.1.

$$P(X_i = x_i) = \sum_{Xi=xi} P(X_1, \ldots, X_n) \tag{5.1}$$

Similarly, the conditional probability $P(X_i = x_i | X_j = x_j)$ can be computed using equation 5.2 and the two marginal probabilities $P(X_i = x_i, X_j = x_j)$ and $P(X_j = x_j)$.

$$P(X_i = x_i | X_j = x_j) = \frac{P(X_i = x_i, X_j = x_j)}{P(X_j = x_j)} \tag{5.2}$$

The joint distribution, although being a complete specification of a probabilistic model, presents a number of problems that render it inadequate for probabilistic inference. The size of the joint distribution grows exponentially with the number of variables. The joint of a model with n Boolean variables requires the specification of $2^n - 1$ independent values. The last value is not independent because the joint is required to add up to 1, i.e., the model's state must be one of all the possible atomic events. Real world problems usually require a large number of variables, which result on an unthinkably large joint distribution. Even though the memory space required to store it may not be a problem on modern computing systems, computing marginal probabilities requires summing across a very large number of variable combinations. Furthermore, the exponentially large number of probabilities

required must either be assessed by experts on the problem's domain, or be computed from historical data. However, the majority of these probabilities are very difficult to assess since they represent the likelihood of conjunctions of propositions which lack psychological meaningfulness. The expert issuing these probabilities may either never have seen a case where the atomic event he is assessing occurred, or may never have related some of the model's variables with each others. The probabilities the expert is required to issue would not be reliable, since the joint does not reflect his natural way of thinking about the problem.

5.3.2 Local Structure and Conditional Independence

The huge number of numerical values required to build the joint distribution, together with the fact that marginalisation of the joint is non–intuitive and lacks explanatory power, have been the main arguments against the use of probabilistic models to handle uncertainty. However, since the late 1980s, the fully probabilistic approach has steadily gained acceptance, with the realisation that "brute force" manipulations of high–dimensional problems could never become neither technically feasible nor acceptable [127]. The path ahead was to find some way of introducing modularity, enabling a large and complex model to be split up into small manageable pieces [31]. Since most real systems exhibit local structure, in the sense that each of the system's subcomponents interacts directly with only a restricted set of other components, regardless of their total number, modularity may be introduced on the model by representing these local interactions. On a probabilistic model of such a system each stochastic variable X_i is directly influenced by at most k other variables. The remaining variables carry no informational relevance to X_i once the relevant ones are known. Probability theory handles the notion of informational relevance with the concept of conditional independence.

Two variables A and B are conditionally independent given a third variable Z if

$$\mathbf{P}(A|Z, B) = \mathbf{P}(A|Z)$$

which means that once Z is known, the belief in A does not change with the discovery of B.

To build a probabilistic model of a given problem it is not enough to identify the relevant stochastic variables; the direct dependencies that hold among these variables must also be identified. The probabilistic model must then be encoded on such a way that when the belief on a variable X_i is being computed, the ignorable is recognisable, or better yet, the unignorable is quickly identified. The inference process is speeded up, since only a subset of the entire model must be consulted.

Since the probabilistic model is structured in terms of direct influences among the variables,

all that is needed to quantify this model is to assess the strengths of these direct influences. This is achieved by quantifying the conditional probability table $\mathbf{P}(X_i|Parents(X_i))$. The conditional probability table (CPT) specifies the likelihood that the event $X_i = x_i$ – i.e., that X_i takes on a particular value x_i belonging to its domain – occurs, given any combination of the variables that directly influence it, and that are denoted by $Parents(X_i)$. These conditional probability tables are referred to as $\mathbf{CPT}(X_i|Parents(X_i))$, throughout this thesis.

Suppose one wants to build a probabilistic model of a burglary alarm which can go off due to a burglary or due to an earthquake (example due to Judea Pearl [127]). Three Boolean variables are identified, namely, Burglary (B), Earthquake (E) and Alarm (A). B and E directly influence A, so $Parents(A) = \{B, E\}$. B and E do not bear any directly influence on each other, at least on the context of this example. The direct influence of A's parents on it is quantified by $\mathbf{CPT}(A|B, E)$, requiring four independent values, which are given by table 5.2.

$\mathbf{CPT}(A\|B, E)$		A	
B	E	T	F
T	T	0.95	0.05
T	F	0.9	0.1
F	T	0.3	0.7
F	F	0.01	0.99

Table 5.2: Burglary Alarm Example: $\mathbf{CPT}(A|B, E)$

Probability is used to quantify the strength of the relationship among the parents and their children. It is also used to summarise all those factors that were not considered on the probabilistic model, but that can, nevertheless, influence the system's behaviour. For example, the probability that the alarm goes off even if neither a burglary nor an earthquake occur is 0.01. This value represents all those other reasons that the model's designer believes can make the alarm go off, but that were not explicitly considered in the model.

Those stochastic variables for which no parents were identified are quantified by their prior probabilities $\mathbf{P}(X_i)$. As previously discussed (section 5.2), prior probabilities are conditioned on some background knowledge K, which is assumed to be static and is not explicitly represented.

The conjunction of these local estimates of direct influences, or CPTs, specifies a complete and consistent probabilistic model, on the basis of which any probabilistic query can be answered. In fact, the joint distribution over all the model's variables $(X_1, X_2, \ldots, X_n)$ is

given by

$$\mathbf{P}(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} \mathbf{P}(X_i | Parents(X_i)) \tag{5.3}$$

By explicitly integrating the notion of conditional independence on the probabilistic model, the number of independent numeric values needed to quantify it is drastically reduced.

Returning to the Alarm example, the prior probabilities of B and E must still be specified. These require one independent number each (see table 5.3).

$\mathbf{P}(B)$	T	F
	0.40	0.60

$\mathbf{P}(E)$	T	F
	0.05	0.95

Table 5.3: Burglary Alarm Example: $\mathbf{P}(B)$ and $\mathbf{P}(E)$

This model requires the assessment of 6 independent values, whereas the direct assessment of the joint distribution requires 7 independent values. The joint can be calculated using equation 5.3.

$$\mathbf{P}(A, B, E) = \mathbf{P}(A|B, E)\mathbf{P}(B)\mathbf{P}(E)$$

A	B	E	$\mathbf{P}(A, B, E)$
T	T	T	0.019
T	T	F	0.342
T	F	T	0.009
T	F	F	0.0057
F	T	T	0.001
F	T	F	0.038
F	F	T	0.021
F	F	F	0.5643

Table 5.4: Burglary Alarm Example: $\mathbf{P}(A, B, E)$

Although, due to its low complexity, this example does not illustrate the reduction in the number of probabilities that must be assessed, it does show that the required probabilities are far more meaningful than the joint and that the direct dependencies existing on the model are usually easy to identify.

Experience has shown that it is usually easy for an expert to decide which direct dependencies hold in a problem [127, 143]. People tend to judge the notion of direct relevance with

clarity and conviction, even though they might not be able to provide precise numerical estimates of probabilities. This suggests that relevance and dependence are far more basic to human reasoning than the numerical values attached to probability judgements. Therefore, a probabilistic model built using these concepts is more likely to correctly reflect the real problem than one using only numerical representations of probabilistic information, as the joint distribution, due to its lack of psychological meaningfulness.

5.3.3 Causality

Probabilistic models which integrate the notion of direct dependencies can constitute a sound and complete inference mechanism if the model is constructed causally, i.e., if each variable's parents are its direct causes, as identified by the model's designer. Choosing a causal variable ordering facilitates the expert's task of assessing each variable's conditional probability table, since the resulting model represents the expert's natural way of reasoning about the problem. Choosing a non–causal variable ordering will result on direct dependencies that require difficult and unnatural probability judgements. Moreover, a causal model minimises the number of relationships that must be considered while the model is being built. The designer ends up having to specify fewer probabilities, and these have psychological meaning [127, 128].

The use of direct causal knowledge provides the crucial robustness, clarity, soundness and completeness needed to make probabilistic systems effective in the real world.

Although the identification of conditional independencies among the model's variables renders it more adequate for probabilistic inference, inappropriate assumptions of conditional independence can lead to noticeable loss of effectiveness. The model's designer must carefully analyse its decisions to ignore tenuous direct causal relationships, since these can lead to models that do not represent reality with accuracy enough. It has been shown that problems with assuming conditional independence can grow as the number of variables in the model increases. The potential benefits of considering a large number of variables can be overwhelmed by the proportional increases in the missing dependencies [69].

5.4 Bayesian Networks

Once identified, direct causal relationships must be encoded on such a way that they are quickly recognisable and that they are maintained as a stable part of the model, independently of the numerical assignment of probabilities.

Directed graphs, or networks, can be viewed as inference engines which represent direct causal relationships. The graphs' nodes are the model's stochastic variables. The links

between the nodes represent local causal dependencies. The strengths of these dependencies are quantified by the conditional probability tables, which are external to the network.

The advantage of network representations is that local causal relationships are encoded directly as neighbouring nodes. The network topology, once built, displays a consistent set of direct and indirect dependencies, and preserves them as a stable qualitative characteristic of the model, independently of any particular assignment of quantitative information, namely conditional probabilities. The network topology can be thought of as an abstract knowledge base, that holds in a variety of settings, representing the domain's general structure of causal relationships [69, 127, 143].

These networks are known as Bayesian belief networks [35, 69, 76, 77, 78, 80, 84, 122, 127, 128, 143]. A Bayesian network is a directed acyclic graph where each node represents a random variable, or uncertain quantity. The directed arcs represent direct causal influences between the linked nodes. The strength of these influences is quantified by assigning to each variable X_i a conditional probability table $\mathbf{CPT}(X_i|Parents(X_i))$, which represents the belief on the event $X_i = x_i$, given any combination of the parents, or direct causes, of X_i.

Formally, a Bayesian network is an annotated directed acyclic graph, constituted by the pair $B = (G, \Theta)$, that encodes a joint probability distribution of a set of random variables $\mathbf{U}$. G is a directed acyclic graph whose vertices correspond to the random variables $U_1, \ldots, U_n$ and whose edges represent direct dependencies among the variables. Θ represents the set of numerical parameters that quantify the network. It contains parameters $\theta_{u_i|Parents(U_i)} = P_B(u_i|Parents(U_i)), \forall_{u_i \in D_{U_i}, U_i \in U}$, where $Parents(U_i)$ denotes the set of parents of U_i in G. A Bayesian network B defines a unique joint probability distribution over $\mathbf{U}$ given by $\mathbf{P_B}(U_1, \ldots, U_n) = \prod_{i=1}^{n} \mathbf{P_B}(U_i|Parents(U_i))$ [55].

A belief network can be built incrementally as follows:

1. choose the set of relevant variables X_i that describe the domain;
2. choose an ordering for the variables such that causes always precede its effects;
3. while there are variables left:
   (a) add a node to the network for the next variable X_i;
   (b) set $Parents(X_i)$ to some minimal set of nodes already in the network, which are perceived as direct causes of X_i, and draw arrows from each of these nodes to X_i;
   (c) define the conditional probability table $\mathbf{CPT}(X_i|Parents(X_i))$;

5.4.1 Conditional Independence and d–separation

The absence of direct links among nodes are qualitative expressions of probabilistic independence of various kinds. Conditional independence among two variables depends on what is known about the entire domain.

Two variables with no parents are independent of each other if nothing is known about their successors. They become conditionally dependent of each other when information is acquired about any of its common successors. This phenomenon reflects a prevailing pattern of human reasoning: if a cause for a symptom is discovered, then the belief on other possible causes is reduced, although many causes may well be present at the same time. This is known as one cause "explaining away" the perceived effect.

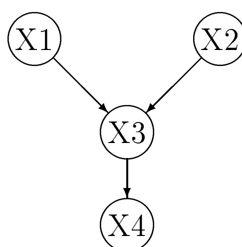


Figure 5.1: A simple Bayesian network

The Bayesian network of figure 5.1, illustrates the fact that $\mathbf{P}(X1) = \mathbf{P}(X1|x_2)$ if nothing is known about the remaining variables. If evidence is entered about one of their successors, e.g., if the exact value of X_4 is known, then the probabilities which must be considered are $\mathbf{P}(X_1|x_4)$ and $\mathbf{P}(X_1|x_4, x_2)$. These are no longer equal, i.e., X_1 is conditionally dependent of X_2 given X_4, since the discovery of X_2 "explains away" X_4, diminishing the belief on X_1.

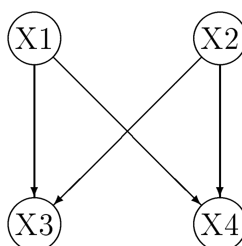


Figure 5.2: A simple Bayesian network

Variables with common parents are dependent of each others if nothing is known about their common ancestors. A person's belief on the possibility of a disease's symptom increases if other symptoms are present. From figure 5.2 it can be seen that $\mathbf{P}(X_3) \neq \mathbf{P}(X_3|x_4)$.

However, X_3 and X_4 become conditionally independent of each other if one of the causes, e.g., X_1, is discovered. The belief on the symptoms depends solely on the way the disease works, not on the presence of other symptoms, therefore $\mathbf{P}(X_3|x_1) = \mathbf{P}(X_3|x_4, x_1)$.

A variable is conditionally independent of its indirect predecessors if its parents are known. From figure 5.1, $\mathbf{P}(X_4|x_3) = \mathbf{P}(X_4|x_3, x_2) = \mathbf{P}(X_4|x_3, x_1) = \mathbf{P}(X_4|x_3, x_1, x_2)$. The belief on the symptoms of a given disease does not change with information about how the disease was caught, once it is known whether or not the disease is present.

Finally, a variable is conditionally independent of the remainder of the network given its Markov blanket. A variable's Markov blanket is defined as its immediate causes, its immediate successors and its immediate successors' parents.

An obvious question is if it is possible, for any belief network, to read whether a node X is conditionally independent of a node Y, given a set of evidence **E**. The answer is yes, and the method is given by the concept of direction–dependent separation or d–separation [77, 84, 127, 143, 148]. Two variables X and Y in a belief network are said to be d–separated, if for all paths between X and Y, there is an intermediate variable V, such that:

1. the connection is serial, $\rightarrow V \rightarrow$, or diverging, $\leftarrow V \rightarrow$, and the state of V is known;
2. the connection is converging, $\rightarrow V \leftarrow$, and neither V nor any of its descendants are known.

If, given all the evidence available **E**, two variables X and Y are d–separated, then changes on the belief of X have no impact on the belief of Y, and conversely. Therefore, if X and Y are d–separated by **E**, then they are conditionally independent of each other given **E**, i.e.,

$$\mathbf{P}(X|\mathbf{E}, Y) = \mathbf{P}(X|\mathbf{E})$$

5.4.2 Probabilistic Inference

A fully specified Bayesian network constitutes a model of the environment, rather than, as with rule based expert systems and neural networks, a model of the reasoning process [70, 128]. Furthermore, it constitutes a complete probabilistic model of the relevant variables, i.e., it specifies a joint distribution over them (equation 5.3). Hence, the network contains all information necessary to answer all probabilistic queries about the model.

The basic task of a probabilistic inference engine is to compute the posterior probability distribution for a set of query variables given the probability distribution for some evidence variables. Bayesian networks are flexible enough so that any node can serve as either a query or an evidence variable. Belief networks can make four kinds of inference:

diagnostic inference – given the effect what is the belief on causes;

causal inference – given the causes what is the belief on effects;

intercausal inference – how does the belief on some causes of a common effect changes, given other causes and the common effect – *explaining away*;

mixed inferences – combinations of any of the above inferences.

The burglary alarm example (section 5.3.2) can be extended to include the possibility that the neighbours of the alarm's owner, N_1 and N_2, call him if they hear the alarm. The model includes the possibility that they think they heard the alarm when in fact they did not, and conversely. However, the reasons why this can happen are not explicitly modelled. They are summarised on the conditional probability tables associated with each variable. Following a causal ordering, this problem can be represented by the Bayesian network presented on figure 5.3.

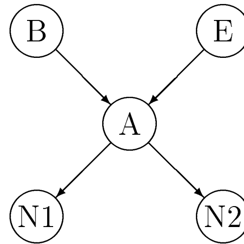


Figure 5.3: The burglary alarm's Bayesian network

Computing the probability that a burglary actually occurred, given that neighbour 1 called – $\mathbf{P}(B|n_1)$ – is a diagnostic query. The probability that N_2 calls, given that an earthquake occurred is a causal inference – $\mathbf{P}(N_2|e)$ –, whereas the probability that the alarm rang given that an earthquake occurred and both N_1 and N_2 called – $\mathbf{P}(A|e, n_1, n_2)$ – is a mixed inference.

5.4.3 Bayes' Rule

The heart of Bayesian inference techniques lies in the inversion formula, universally known as Bayes' Rule,

$$\mathbf{P}(H|\mathbf{E}) = \frac{\mathbf{P}(\mathbf{E}|H)\mathbf{P}(H)}{\mathbf{P}(\mathbf{E})} \tag{5.4}$$

which states that the belief on an hypothesis, or cause, H, given the available evidence, or effects of this cause, $\mathbf{E}$, can be computed by multiplying the previous belief on the hypothesis, $\mathbf{P}(H)$, by the likelihood that $\mathbf{E}$ will materialise given the hypothesis, $\mathbf{P}(\mathbf{E}|H)$. This

rule enables diagnostic inferences, from effects to causes, on Bayesian networks where the available information is causal, i.e., from causes to effects. The denominator of equation 5.4 hardly enters into consideration, since it is a normalising constant α that can be computed by requiring that $\mathbf{P}(H|\mathbf{E}) + \mathbf{P}(\neg H|\mathbf{E}) = 1$. Equation 5.4 can be rewritten as

$$\mathbf{P}(H|\mathbf{E}) = \alpha \mathbf{P}(\mathbf{E}|H)\mathbf{P}(H) \tag{5.5}$$

A further advantage of Bayesian networks is that new evidence can be added as soon as it arrives, piece by piece, without recomputing the previously held beliefs. When evidence E_1 arrives, the belief on H is computed according to equation 5.5

$$\mathbf{P}(H|E_1) = \alpha \mathbf{P}(E_1|H)\mathbf{P}(H)$$

As new evidence E_2 arrives, the belief on H is recomputed using the previously held belief $\mathbf{P}(H|E_1)$

$$\mathbf{P}(H|E_1, E_2) = \alpha \mathbf{P}(E_2|H, E_1)\mathbf{P}(H|E_1) \tag{5.6}$$

Since E_2 and E_1 are conditionally independent given their causes H, equation 5.6 becomes

$$\mathbf{P}(H|E_1, E_2) = \alpha \mathbf{P}(E_2|H)\mathbf{P}(H|E_1) \tag{5.7}$$

This process is known as Bayesian updating.

5.4.4 Pearl's Probabilistic Inference Algorithm

Pearl [127] proposes an algorithm for probabilistic inference on Bayesian networks that views the impact of each new piece of evidence as a perturbation that propagates through the network via message passing between the nodes, without any external supervision. This algorithm guarantees that equilibrium is reached in time proportional to the longest path in the network, and that at equilibrium each variable is given a belief measure equal to its posterior probability distribution, given all the available evidence. This algorithm is suitable only for singly connected networks. In such networks there is one and only one undirected path between any two nodes. If the network contains loops, i.e., undirected cycles, messages may circulate indefinitely around these loops, and the algorithm will not converge to equilibrium. To handle such cases, different methods must be used, like clustering, conditioning or stochastic simulation [77, 127, 143]. Furthermore, all stochastic variables must be discrete. Appendix B contains a comprehensive description of Pearl's inference algorithm.

Throughout the remainder of this work the dynamic values of the nodes' inferred probabilities, $\mathbf{P}(X|\mathbf{E})$ will be referred to as $\mathbf{BEL}(X)$, which reflects the belief distribution accorded to X by all evidence, $\mathbf{E}$, received so far. So

$$\mathbf{BEL}(X) = \mathbf{P}(X|\mathbf{E})$$

5.4.5 Sensor Model

In the general case, an agent acquires information about the environment through its sensors, sets the respective evidence variables and infers its belief distribution on the state of the world. To be effective the agent must include the possibility that its sensors return noisy and/or incorrect readings. This is the role of the sensor model, which is implemented through the conditional probability table $\mathbf{CPT}(E|X)$ associated with the percept node [143].

The direction of the causal relationship is the crucial element: the state of world X causes the sensor to take on a particular value E. The inference process will go the other way: given evidence arriving from the percept node, it is propagated to the state variable.

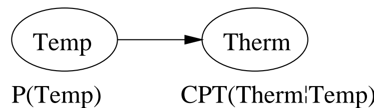


Figure 5.4: Thermometer sensor model

Figure 5.4 presents a belief network where a thermometer measures the environment's temperature. The thermometer's readings depend on the environment's temperature. The sensor model is implemented through $\mathbf{CPT}(Therm|Temp)$. The great advantage of causal ordering is that $\mathbf{CPT}(Therm|Temp)$ only depends on the way the thermometer works, on its characteristics and range of supported temperatures. It does not depend on the particular characteristics of the environment being measured. These are encoded on the prior probabilities $\mathbf{P}(Temp)$, which describe the probability distribution over a range of temperatures for that particular environment. The direct specification of $\mathbf{CPT}(Temp|Therm)$ would require that both the thermometer and the environment's characteristics are encoded on the CPT, loosing clarity and generality. Furthermore, it would also require the specification of $\mathbf{P}(Therm)$, which lacks meaningfulness.

If the sensor gives a perfect report of the current state, then the sensor's CPT is purely deterministic. On a more realistic model, noise and sensor's errors are reflected in the probabilities of incorrect readings.

If the agent has several sensors to measure the same state variable, the resulting inference process is called data fusion. Integrating the readings from multiple sensors provides greater accuracy, since these are conditionally independent of each others, given the measured variable actual value. Although they are not unconditionally independent — they should all return approximately the same value — they are correlated only in the sense that they depend on the quantity being measured.

The sensor model can be extended to include additional variables, which represent the sensor's condition. To measure a computer's workload, for example, additional variables

can be included to account for the various factors that can influence either the sensor's readings or the actual workload's value. Inaccuracies associated with information's age can be included in the model by means of an extra variable, **Information Age**, that changes the belief distribution over the actual workload, according to the time elapsed since the last time the sensor was actually activated. Also, if that computer's workload is known to follow some regular pattern across the day and/or the week — maybe because it is regularly used on some specific task — this information can be included on the model, by means of additional variables that directly determine the actual workload (figure 5.5).

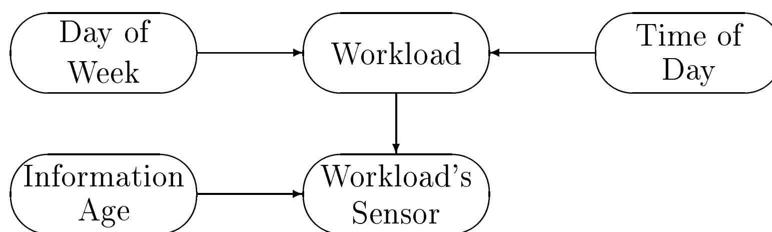


Figure 5.5: Extended sensor model

Extended models enable accurate estimates of the variable being measured and can perform diagnosis in case of failure [143].

5.5 Making Decisions

5.5.1 Preferences and Utilities

A decision is an allocation of resources under the control of the decision–making agent, which changes the state of the environment. A decision implies an action, that, on a non–deterministic environment, may have several different outcomes. Each particular outcome is a completely specified state of the environment. To be able to select among different alternative actions, an agent must have preferences for the different possible outcomes.

Utility theory is used to represent and reason about preferences. It says that each state has an utility for an agent, and that the agent will prefer states with higher utility. Whereas judgements about the likelihood of states are quantified by probabilities, judgements about the desirability of an action's consequences are quantified by utilities.

Utility is a function that maps system's states to real numbers [77, 127, 143]. The agent's preferences among different states are captured by this function, which assigns a single number to express the state's desirability. Utility imposes a preferential ordering on the system's states. Every utility function can be normalised, such that the most preferable

state has an utility $U(S) = 1$, and the least preferable one has an utility $U(S) = 0$.

Most real problems require the system's state to be characterised by many different variables, or attributes. In such cases, it is necessary to resort to multiattribute utility theory, in order to specify the utility function. If the system's state is described by variables $X_1, \ldots, X_n$, the multiattribute utility function is usually an additive value function

$$U(S) = \sum_{i=1}^{n} \alpha_i X_i$$

Additive functions are a natural way of expressing an agent's preferences, and are valid in many real world problems. These functions can be safely used when the attributes exhibit mutual preference independence, i.e., when each attribute does not affect the way in which the agent trades off the other attributes against each other. Two attributes X_1 and X_2 are preferentially independent of a third attribute X_3, if the preference among the outcomes (x_1, x_2, x_3) and (x_1', x_2', x_3) does not depend on the particular value x_3 for attribute X_3. However, when mutual preference independence does not strictly hold, an additive function can still be a good approximation to the agent's preferences [143].

Decision theory combines utilities with probabilities, providing principles for rational inference and decision making [69]. A rational method of choosing among actions, is to weigh the utility of each of the various possible outcomes of each action with the probabilities that these outcomes will occur.

Let $Result_i(a)$ be a possible outcome of a nondeterministic action a, where the index i ranges over all the different possible outcomes of a. The expected utility of action a, given the agent's available evidence $\mathbf{E}$ about the state of the world, is given by

$$EU(a|\mathbf{E}) = \sum_i P(Result_i(a)|\mathbf{E}, Do(a))U(Result_i(a)) \tag{5.8}$$

where:

$Do(a)$ is the proposition that a is executed in the current state;

$P(Result_i(a)|\mathbf{E}, Do(a))$ is the probability that this particular $Result_i(a)$ occurs given $\mathbf{E}$ and $Do(a)$;

$U(Result_i(a))$ is the utility of the outcome $Result_i(a)$.

The fundamental idea behind decision theory is that the agent is rational if and only if it selects the action that yields the highest expected utility, averaged over all possible outcomes of that action. This is the principle of Maximum Expected Utility (MEU)

$$action \leftarrow \arg_a[\max_{a \in A}[EU(a|\mathbf{E})]] \tag{5.9}$$

where A is the set of available actions.

If an agent follows the MEU principle, trying to maximise an utility function that correctly reflects its performance objectives, then the agent exhibits rational behaviour [143].

Rational behaviour must be distinguished from omniscient behaviour, where the agent never fails. A good decision may lead to a bad result. Alternatively, a random decision can lead to a successful result. Such is the nature of deciding under uncertainty. Decision theory strives for good decisions that, on average, lead to good outcomes.

5.5.2 Decision Networks

Decision networks provide coherent prescriptions for selecting actions and meaningful guarantees of the quality of these selections. Decision networks, also called influence diagrams [6, 79, 88], constitute a general mechanism for rational decision making. These networks combine belief networks with additional node types for actions and utilities.

Decision networks require three kinds of knowledge:

- causal knowledge about how events and actions influence the world's state;
- knowledge about which actions are feasible in any given set of circumstances;
- knowledge about how desirable the consequences of an action are;

The set of actions available to the agent at any given instant can be represented by variables that are under the full control of the decision–making agent, unlike the random variables discussed so far. Selecting an action amounts to impose the value of the decision variable, rather than determine it probabilistically. This setting alters the probability distribution of another set of stochastic variables in the network, known as the consequences of the decision variable. The utility function can then be evaluated, taking into account the probability distribution over those variables that directly affect utility.

Decision networks are Bayesian networks with two additional node types:

Decision nodes – represent choices available to agent. Their values are imposed to represent actions.

Utility nodes – represent the utility function to be optimised. Its parents are those variables that directly affect utility. The table associated with this node is a tabulation of the agent's utility as a function of the attributes that determine it.

Actions are selected by evaluating the decision network for each possible setting of the decision node. Once this node is set, it behaves exactly like a chance node that has been set as an evidence variable. The algorithm for evaluating decision networks is as follows:

1. Set the evidence variables;
2. For each possible setting of the decision node:
   (a) Set the decision node to that value;
   (b) Propagate the beliefs through the network to compute the new distribution over the relevant variables;
   (c) Compute the expected utility for this action $EU(a|E)$;
3. Choose the action with the highest expected utility;

This is a straightforward extension of the belief network inference algorithm (sections 5.4.2 and 5.7.2) [127, 143].

5.6 Knowledge Engineering

Decision theory provides principles for probabilistic inference and rational decision making under uncertainty. However, it does not tell how to apply these principles to real problems in a tractable manner. This is the realm of decision analysis [69].

Decision analysis is an engineering discipline that addresses the pragmatics of applying decision theory to real problems. Decision theory does not help the model's designer in the task of identifying which environmental aspects should be modelled, which direct dependencies hold among them, what level of detail and granularity should be used to reason about the decision problem or what utility function and probability distribution should be selected. Decision analysis, in contrast, addresses these issues directly.

Decision analysis provides a set of techniques for focusing attention on the problem's relevant aspects, both on the modelling and on the decision making phases [69, 127, 143].

Knowledge engineering is the process by which expert knowledge is obtained and represented on decision making systems. Although this term is not usually associated with decision analysis, most of their fundamental activities are similar.

The core of decision analytic knowledge engineering is assembling a coherent and computable decision basis. A decision basis is the complete model of a decision problem, consisting of components that represent states, relationships, alternatives and preferences.

Decision networks allow the representation of a decision basis and provide mechanisms for decision making based on this representation.

The knowledge engineering process for building decision theoretic systems entails several steps [143], described along the next sections.

5.6.1 Determine the Scope of the Problem

Decide which of the environment's aspects will be explicitly modelled as stochastic variables, which will be each variable's domain and which actions will be available to the agent. This requires a thorough understanding of the problem's domain. The number of variables implies a tradeoff between accuracy and computational tractability. The variables' domains must be carefully chosen. Large domains require the assessment of a large number of conditional probabilities. Continuously valued variables can often be discretised without loosing effectiveness.

The set of alternative actions has an huge effect on the overall value of the model. The generation of new actions is often worth an extensive reasoning and careful analysis [69].

5.6.2 Identify Direct Dependencies

This step entails the laying out of the network's topology. Variables judged to be direct causes of another variable are connected with arrows, following a causal ordering. Those variables that are direct consequences of actions must be identified, as well as those that directly determine the environment's state utility.

Laying out the network's topology amounts to identify conditional independencies among variables. Although conditional independence renders the model simpler and tractable, care must be taken with wrong independence assumptions, as these can jeopardise the model's effectiveness (section 5.3.2).

5.6.3 Assign Probabilities

Decision networks require, for each variable X_i, the assessment of $\mathbf{P}(X_i|Parents(X_i))$. A frequent concern with Bayesian models is the availability of probabilities, as well as the number of probabilities required, which can be huge if the model is complex.

People are notoriously bad numerical estimators. Although it is usually easy for a domain expert to decide which direct conditional dependence relationships hold in a domain, it is very difficult to the same expert to decide on exact values for the probabilities associated

with these dependencies. However, sensitivity analysis often reveals that these numbers need only to be specified approximately. As long as the ratio between the probability of an event occurring or not occurring, given the same evidence, roughly reflects genuine experience, valid conclusions will still be reached.

The important point to note is that the value of decision networks resides both on its qualitative part, i.e., the network's topology, and its quantitative part, i.e., the variables' conditional probabilities and utility functions. The network topology specifies which direct and indirect dependencies must be considered, and probability theory gives rules of how numbers must be combined. These remain unchanged independently of the accuracy of the actual probability estimates. If the rules by which exact numbers combine are strongly believed, then the same combination rules can be used on rough estimates of these numbers [127].

The assessment of $\mathbf{P}(X_i|Parents(X_i))$ is facilitated by the causal ordering required on Bayesian models. The assessment of these parameters amounts to estimating the likelihood that the event $X_i = x_i$ will occur, given any combination of X_i direct causes, $Parents(X_i)$. This kind of assessments points to psychological meaningful cause–effect relationships and is easier to quantify than diagnostic relationships or conjunctions of a large number of propositions.

In fact, diagnostic knowledge, i.e., from effects to causes, is often much more tenuous than causal knowledge and is dependent of the particular environment where the agent is required to operate (section 5.4.5).

Conditional probabilities assessment becomes particularly difficult when a variable has a large number of parents. Quantifying the combined influence of these interacting causes is a very demanding task. Several techniques have been developed to ease this task.

One of such techniques is to divorce a set of parents $\mathbf{S}$ from the remaining parents, by introducing a mediating variable M, and making M a child of $\mathbf{S}$ and parent of the former child of $\mathbf{S}$ (see figure 5.6).

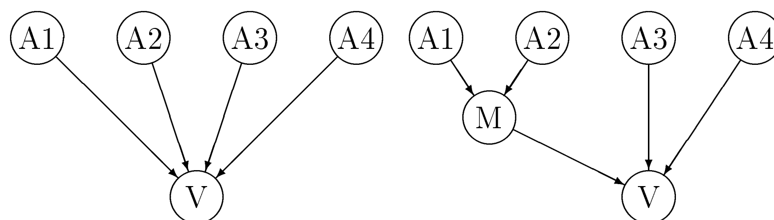


Figure 5.6: Divorcing $\{A_1, A_2\}$ from $\{A_3, A_4\}$

Divorce and mediating variables' inclusion is mainly a matter of convenience. Although it

will ease the assessment of conditional probabilities, it may increase the model's complexity to a level that may jeopardise performance [77].

Often the causal relationship among several parents and their childs falls into a category that can be modelled by canonical distributions, requiring only the identification of the distribution and a few parameters. One of the most common of such distributions is the noisy–OR. This can be applied when any member of a set of causes is likely to independently cause the effect, and this likelihood does not diminish when several of the causes are present simultaneously. This distribution requires the specification of the probability that the effect will not occur when one cause is present, for each of the causes. This probability is referred to as the *inhibitor*. All the remaining probabilities can be computed according to some well defined rules [69, 127, 143].

On the other hand, some of the required conditional probabilities might be computed based on previous experience, by counting the number of occurrences of a given effect for each combination of its causes. A Bayesian model can learn its numerical parameters by adopting this frequentist approach.

5.6.4 Assign Utilities

The model's designer must specify an utility function which translates system's states to real numbers that express the states' desirability and which maintains the agent's preferential order among these states. The utility function must correctly reflect the agent's performance goals. By selecting the action that maximises its expected utility, the agent will, in average, achieve its goals.

5.6.5 Model Refinement and Sensitivity Analysis

Sensitivity analysis is used to determine which variables, uncertainties and assumptions have the most influence on the system's behaviour. It involves exploring the space of possible models, in order to build a model that is both effective and computationally tractable. Sensitivity analysis to variables removal, continuous quantities discretisation, conditional independence assumptions and changes on the conditional probabilities and utility function should be performed in order to increase confidence on the agent's decisions [66, 69, 143].

The most usual analysis is to check whether the best decision is sensitive to small changes on the assigned probabilities and utilities, by varying these parameters and observing the agent's decisions. If small changes lead to significantly different decisions, then it may be worthwhile to spend more resources in order to build a more robust model.

The assessment of probabilities is often a very demanding task and the expert is often uncertain about the distribution he is providing. There is always a tradeoff between assigning a probability based on a current state of understanding and expending additional effort to come up with a better estimate. Sensitivity analysis helps in this task of achieving a confident tradeoff. If the model proves to be robust with respect to conditional probabilities, no further probability refinements are required.

5.7 Applying Decision Networks to a Dynamic Scheduler

This section discusses the application of decision theory, and more particularly, of decision networks, to the scheduling problem. There is a large number of different approaches to design a decision network, which selects the action that will, with higher probability, maximise the scheduling agent's performance goals. The most adequate approach to each case depends on the particular characteristics of the application, the distributed system and the performance goals.

5.7.1 Generic Structure

The scheduling problem can be briefly described as the problem of allocating resources to tasks, in such a way that a given set of performance goals is met. To properly match resources and tasks, the scheduler must know to some extent both the resources' capacities and the tasks' requirements. Even when these are not explicitly represented in the scheduler's execution model, they are implicitly known with some detail: a scheduler would not assign a disk to a task which requires calculations, since the disk has no computing capacity. Solving the scheduling problem requires that some resources are allocated to the scheduler itself; therefore, some direct overheads are incurred to generate an appropriate schedule. The resulting allocation of resources to tasks is suboptimal in most cases, resulting in additional indirect overheads. The scheduler must strive to minimise these overheads, since they can compromise its ability to achieve its performance goals. Four entities can be identified as having an important role on the scheduling process: tasks' requirements, resources' capacities, performance goals and scheduling overheads.

The decision network designer must also take into consideration that he is developing an agent which acts upon the environment, changing its state. The agent's goal at each instant is to select the action that leads the environment from its current state to the most desirable next state, that is, the one which has a maximum expected utility. To select this action, the agent needs to represent the environment's current state, the actions available to the agent, the state transition model (which estimates the next state for each action,

given the current state), the next state and the utility function.

From the last two paragraphs it is possible to identify the relevant entities that can be represented in a decision network. Figure 5.7 presents a generic structure for such a network. The network topology complies with causal relationships: the sensors' readings are an effect of the relevant quantities actual values, the next state is a function of both the current state and the selected action.

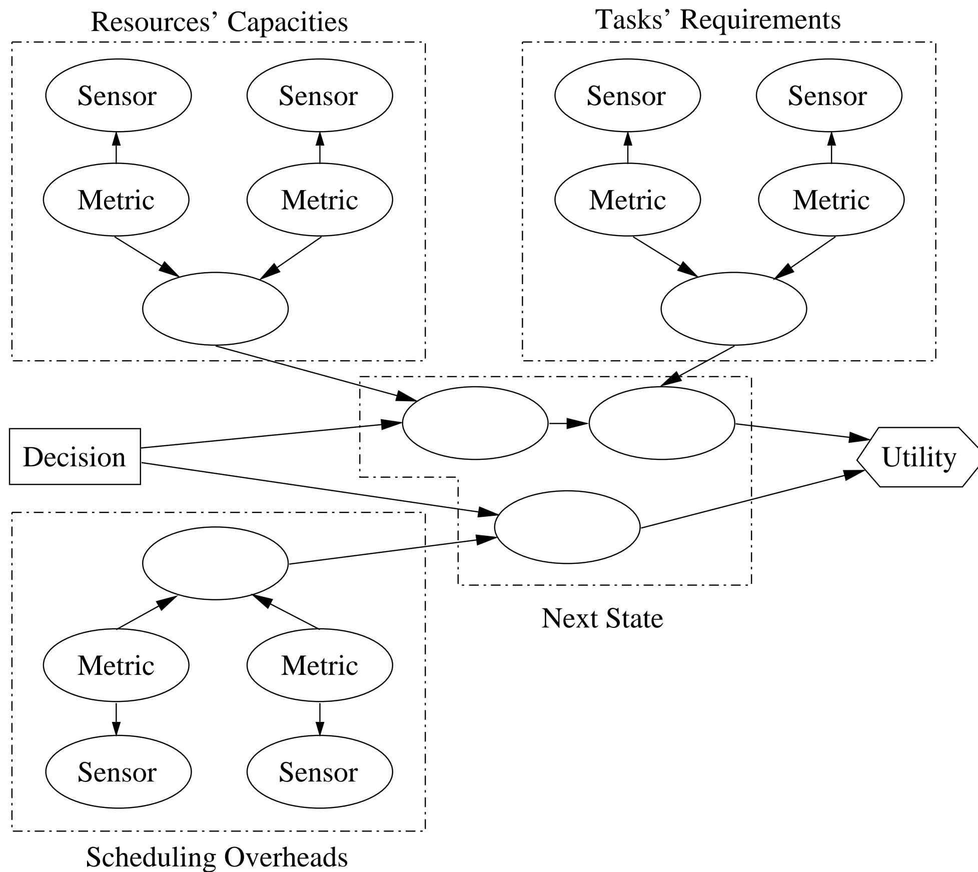


Figure 5.7: A generic structure for a scheduling decision network

Six different blocks can be identified:

- the resources' capacities, tasks' requirements and scheduling overheads blocks, built with stochastic variables that describe the agent's belief on the environment's current state, as perceived by its sensors;
- the decision variable, that lists all the actions available to the scheduling agent;
- the next state block, constituted by stochastic variables, that describe the agent's belief on the outcome of each action;
- the utility variable, that computes the expected utility of each action.

The environment's current state includes the tasks' requirements, the resources' capacities and the expected scheduling overheads. These are directly perceived through the agent's sensors and quantified by specific metrics. The sensors, however, are prone to imprecisions, and should not be blindly trusted. Using the capabilities of Bayesian networks, these errors may be explicitly modelled, by including the sensors' models in the network. The sensors' readings are entered on the network using evidence variables, and the belief distributions over the metrics, modelled as stochastic variables, are inferred, using the conditional probability tables that quantify the sensors' models. The sensors' readings themselves are deterministic values, but the agent's belief on the relevant metrics is quantified by stochastic vectors, since it is uncertain about the sensors' correctness. The reasoning behind this process is that imperfect information about the metrics is obtained through perfect information about the sensors.

The environment's current state may also be described by higher level variables, which represent more abstract concepts, such as the distributed system degree of load balancing or the communication network availability.

The variety of available sensors determines the type of information that can be included in the scheduler's execution model. These can include the node's computing throughput, the network latency and bandwidth, the amount of work completed at the sampling time, the tasks' communication volume, the tightness of the coupling among tasks, the available memory, etc. The set of available sensors may also depend on the instant the scheduler is acting. On an initial scheduling step, before starting the application, the scheduler may have no information about the tasks' requirements and decide based only on the resources' capacities. On a later step, and if task migration is possible, i.e., if the scheduler is not restricted to do one–time task assignments, it may also use information about the tasks' behaviour to redistribute the workload.

The actions available to the decision making agent are represented using a decision variable. This must list all possible actions, which depend on the particular workload and on the distributed system being managed; it may include: assign task T_1 to node n_2, migrate task T_2 from node n_3 to node n_5, transfer 35% of node n_1 workload to node n_4, etc. The agent will assess the network for all possible actions and select the one which maximises the expected utility.

The state transition model describes the state that will occur from each action, given the current state. On a decision network it is composed by all conditional probability tables (CPT) associated with the stochastic variables that describe the next state. These CPTs can be either assessed by an expert, or the agent may learn them during execution, using dynamically gathered data.

The next state is composed by a set of one or more variables that describe the resulting

state of the most relevant aspects of the environment, after each action is executed. These should be quantities that are affected by the actions being selected, since the agent uses them to quantify the desirability of each action. On a scheduling agent, these can include the tasks' estimated new completion times, the resulting degree of load balancing, the actions' expected overheads, etc.

Finally, the utility function is included by means of an utility variable, and represents the performance goals degree of achievement for each action. It maps the environment's next states into real numbers that express the desirability of each state. The most desirable states must be assigned an higher utility, while the less desirable ones are assigned low utilities. The utility function can be normalised, meaning that the most desirable state has an utility of 1, and the least desirable has an utility of 0.

The main difference between deterministic and stochastic models lies in the representation of the relevant quantities. Deterministic models generate a single estimate of the environment's state — which is assigned a probability, or belief, of 1 — and, based on this estimate, generate a single estimate for the environment's next state, for each action. Stochastic models, on the other hand, use stochastic variables, and, consequently, are not restricted to assign a certainty of 1 to the value of a given variable. A belief distribution is used instead, specifying a probability for each of the variable's possible values. This allows the explicit inclusion in the model of uncertainties arisen from:

- imprecisions in the sensors' readings;
- information aging, which occurs within dynamic environments;
- incomplete models, which do not include all the relevant aspects of the environment;
- uncertainty about the actions' outcomes — non–deterministic environments.

It is this thesis' hypothesis that, by explicitly dealing with these uncertainties, a decision network based scheduling agent will be able to make more rational decisions, that will, with higher probability, increase its performance goals degree of achievement.

Discussion

A particular decision network may not include all the blocks that describe the environment's state. The selection of the relevant blocks to include, and respective metrics, depends on three major factors:

- the uncertainty associated with each metric, and with how does it influence other entities considered on the execution model; if there is no uncertainty, it makes less sense to include it on the decision network, although deterministic conditional probability tables could be used;

- the capability to measure the quantity described by the metric; if it can not be measured, then probably it should not be included, since the sensor's variable can not be instantiated with evidence;
- the strength of the influence each entity has on the environment's behaviour; if an entity is considered irrelevant, then including it on the decision network will not contribute to a better decision making, and will render the decision network more complex, requiring more time both to design it and to infer the belief distribution.

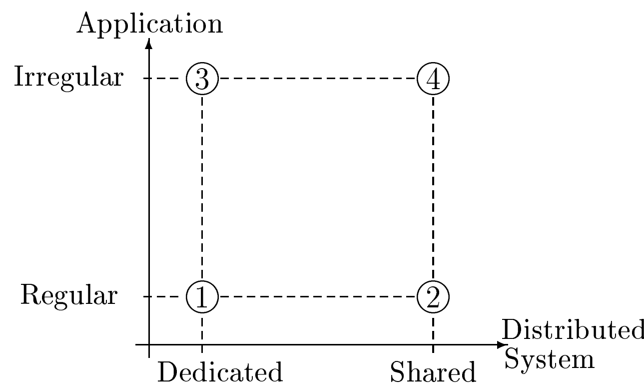


Figure 5.8: The system sharing level versus the application regularity space

Figure 5.8 presents the environments where a scheduler may be required to act, classified along two axis: the system sharing level and the application's workload regularity. Four particular points are highlighted, corresponding to the extreme values along each axis. Uncertainty about the environment's behaviour increases from the lower left corner (point 1) to the upper right corner (point 4).

If an application has a regular workload and the distributed system is dedicated to that single application (point 1), then both the tasks' requirements and the resources' capacity can be accurately estimated. In such cases, the uncertainty about the environment's behaviour is not significant. Tasks and resources can be matched using deterministic models; it is worthless to employ decision networks.

If an application has a regular workload, but the distributed system is dynamically shared among several users or applications (point 2), then the resources' capacity can no longer be estimated with accuracy. In such cases, the decision network must include the resources' capacity block. The tasks' requirements block can be either discarded or simplified. Including a very simple tasks' requirements block, built only with a single variable for each relevant metric, allows the inclusion on the execution model of the uncertainty about how will that task perform on a resource whose capacity is uncertain.

Point 3 illustrates the case where the system is dedicated, but the application's workload is irregular. The resources' capacity block can be composed by a single variable for each

metric, while the tasks' requirement block must include all the uncertainties about the applications' behaviour.

The upper right corner (point 4) illustrates the more complex case, where the application has an irregular workload and the distributed system is dynamically shared. The decision network must include both blocks, since there is uncertainty about their behaviours. This thesis further investigates this case, as described throughout Part II.

The scheduling overheads block should be included in the decision network, if these overheads are relevant to the decision making process, i.e., if their magnitude can compromise the scheduler's effectiveness and if they can be measured or estimated.

5.7.2 The Decision Making Process

Figure 5.9 illustrates the decision making process using decision networks. At each iteration, the agent starts by entering on the network all the evidence available about the environment's state (step a). This is acquired through its sensors. Since its is uncertain about the sensors' correctness, it will then infer a belief distribution over the relevant metrics (step b), using each sensor model, embedded in the network as conditional probability tables associated with the sensors' variables. The agent's belief distribution over the environment's current state may now be inferred, since all the relevant metrics are already known (step c). The inference algorithm performs these two steps in a single phase. The stochastic functions that describe how these higher level variables are influenced by the metrics are embedded in the network as conditional probability tables.

Once the belief distribution over the environment's current state is known, the agent may infer, for each action, which is the belief distribution over the environment's next state (step d). This is achieved by using the state transition model, represented as a set of conditional probability tables associated with the next state stochastic variables. The agent may then compute the expected utility for each action (step e). A rational agent will select the action which maximises its expected utility (step f).

5.8 Summary

This chapter describes decision networks, a tool proposed by decision theory for rational decision making under uncertainty. It also presents decision analytic knowledge engineering, whose main goal is to assemble a coherent and computable decision model of a real world problem. Appendix B presents with detail the inference algorithm used throughout this work.

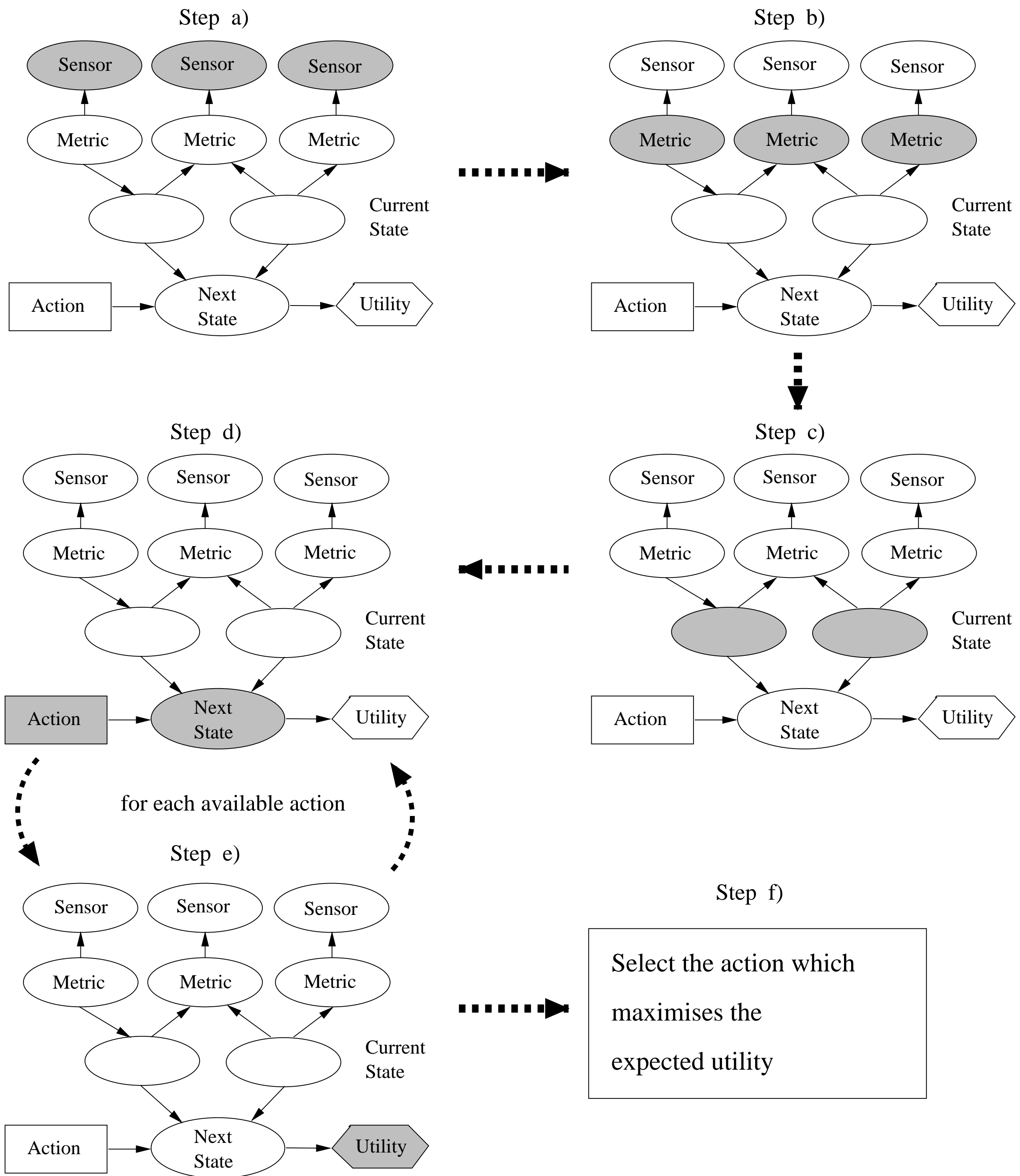


Figure 5.9: The decision making process

The hypothesis put forward by this thesis is that decision networks, if applied to the scheduling agent's execution model, may improve its effectiveness and help overcome the problems caused by uncertainty; section 5.7 discusses how this paradigm may be applied to the scheduling problem, and presents a generic structure for such a decision network. Part II verifies the hypothesis that decision networks, when applied to an application level scheduling agent's execution model, may improve its performance and efficiency.

Part II

Hypothesis' Verification

Part II

The hypothesis forwarded by this work is that: *decision networks, if applied to the scheduling agent's execution model and decision making mechanism, may improve its effectiveness and help overcome the problems caused by uncertainty.*

This hypothesis must be submitted to a systematic experimental testing, to verify if real facts do corroborate it. Since an hypothesis is an universal proposition, and experimental results are singular facts, or propositions, these can never conclusively confirm the hypothesis. They can either demonstrate that the hypothesis is false, in which case it must be rejected, or they can corroborate it, in which case it is temporarily accepted. Scientific knowledge always maintains its hypothetical nature.

It is not enough, however, to demonstrate that a single hypothesis does solve the problem being tackled. It is necessary to compare the main hypothesis's performance with alternative hypothesis. If it solves the problem better, or as well as, the alternative solutions, then it is not rejected. There is no advantage in accepting an hypothesis which performs worst than other possible explanations.

Experimentation is an attempt to demonstrate that the hypothesis is false, or that it is unable to explain real facts as well as other alternative explanations. The hypothesis can only be corroborated if it survives these rejection tests [90].

In order to guarantee that experimentation results are valid, some principles of rigorous testing must be employed [68]:

- to insure objectivity in experimental measurements;
- to employ effective controls to isolate what is being measured;
- to document all environmental factors that may affect results;
- to provide enough details to permit other researchers to replicate the results;
- to employ unambiguous notation;
- to validate the models and results with additional experiments and studies;
- to reason from the obtained results to explore fundamental underlying principles.

Part II of these thesis presents the experimentations performed to test the hypothesis, and compares the performance and efficiency of the stochastic scheduler with alternative approaches.

Chapter 6 describes the method used to perform the experimentations, the alternative scheduling strategies with which the stochastic approach is compared and the metrics used to assess the results.

Chapter 7 describes the parallel ray tracer used as a case study for all experiments, while chapter 8 presents the actual evaluation tests, the execution models used by the scheduling agents and the conclusions drawn from these experimentations.

Appendix D presents all the obtained experimental results.

Chapter 6

Methodology

Contents

6.1 The Distributed System 111
6.2 Selection of a Case Study 113
6.3 The Problem's ESR Classification 115
6.4 Performance Modelling 116
6.5 Reference Scheduling Strategies 120
6.6 Synthetic Background Workload 125
6.7 Summary 129

The effectiveness of the decision network based scheduler is assessed by performing a number of controlled experiments. The performance and efficiency of this scheduler are compared with those obtained using three other reference scheduling strategies. Since it is not possible to evaluate all different applications and workloads running on different distributed computing systems, a particular application — representative of a broader class of applications — and distributed system are used throughout all experiments.

This chapter describes the conditions under which the experiments are performed. The application and the distributed system are identified and classified. The scheduling policies and the metrics used to evaluate their efficiency and effectiveness are briefly described. The experiments are repeated with a set of different synthetic background workloads, which are also presented. However, this chapter does not present any results; these are presented throughout chapter 8 according to the particular execution model used on each experiment.

6.1 The Distributed System

Scalable distributed or parallel computing systems are converging towards three general architectures, whose differences lie in the level of resources sharing [73] (figure 6.1).

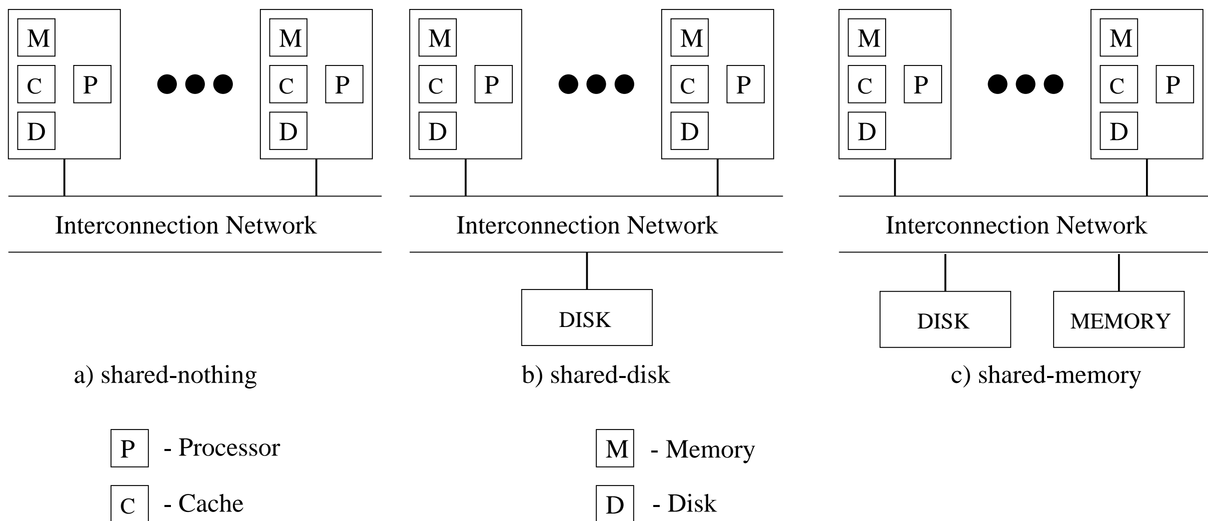


Figure 6.1: Three different distributed system's architectures

The shared–nothing architecture consists on several nodes connected by an interconnection network, but that do not share any resources. Each node can have more than one processor, in which case that node's architecture is referred to as **s***ymmetric* **m***ulti***p***rocessor* (SMP). Processors within SMP nodes have an high level of resources sharing, but usually have private caches. The shared–disk architecture differs from the previous one in that some disks can be shared by several, or even all, nodes. On shared–memory architectures even the main memory becomes shared. Hybrid versions of these architectures are common.

A parallel system's memory architecture can be either centralised or distributed. Central memories are usually shared and are also known as **u***niform* **m***emory* **a***ccess* (UMA) systems, in the sense that memory accesses take roughly the same time, independently of the node that is accessing it. Two types of UMA systems are the parallel vector processor (PVP) and SMP systems. Since each processor has its own cache, maintaining cache coherence is a major issue on UMA parallel systems.

Distributed memories may be either shared or non–shared. Examples of non–shared distributed memory systems, also referred to as **no**–**r***emote* **m***emory* **a***ccess* (NORMA), are loosely coupled clusters of workstations (COW) and tightly coupled massive parallel processing systems (MPP). These are usually programmed using message passing.

Distributed shared memory systems (DSM) have special hardware that enables all local memories to be addressed as a single global address space. Since memory access times depend on the location of the data items being accessed, these architectures are usually referred to as **n***on*–**u***niform* **m***emory* **a***ccess* (NUMA). The memory access mechanisms differ among architectures, including **c***ache*–**c***oherent NUMA* (CC–NUMA) and **c***ache*–**o***nly* **m***emory* **a***rchitecture* (COMA).

Clustering is becoming an widely spread solution to distributed processing, due to its

cost–effective approach of using available workstations and networks to assemble a scalable distributed system. Although scalable clusters may require additional investments on high speed networks, medium–size clusters can be assembled with equipment available at most organisations, by installing additional system software, which is usually free of charge.

The experiments were performed on a cluster of seven personal computers, connected by a FastEthernet network with a communication bandwidth of 100 Mbit/sec and a Myrinet network with a communication bandwidth of 1.2 Gbit/sec. All the results presented throughout this thesis were obtained using the Myrinet network. The computers are uni–processor machines based on Intel Pentium II processors, running either at 350 or 400 MHz, with 128 MBytes of central memory each. All workstations run LINUX and share a common disk by NFS. PVM 3.4 is used as the parallel execution environment.

Only seven nodes were available at the time these experiments were performed. Although the current work could be more challenging if conducted on a larger distributed system, it is felt that the results achieved are valid enough to be used as a starting point for future work with larger systems. This cluster will have additional nodes soon, and evaluation on these larger systems is already planned.

This cluster can be classified as a shared–disk, NORMA, loosely coupled distributed system. It is a small, distributed, heterogeneous shared system, in the sense that it is simultaneously used by several students and researchers. Dedicated access to a single user is not guaranteed, although it may be possible by using the system at late hours. Its heterogeneity arises not only from the processors' clock frequency, but also from the different background workloads that can be assigned to each node at the same instant.

6.2 Selection of a Case Study

Parallel applications can be classified according to several characteristics, which result from an interplay among the application's algorithm properties and the particular parallel programming paradigm and decomposition method selected for each implementation. The selection of the actual programming paradigm and decomposition method, which are tightly related, further depends on the characteristics of the distributed or parallel system being used [49, 73].

Hwang and Xu [73] suggest that the application space can be partially ordered along four dimensions, which are not totally independent: regularity in the algorithm, degree of parallelism, computational granularity and interaction overhead (table 6.1).

Since the authors do not clarify the meaning of the algorithm regularity dimension, for the purpose of this thesis it is understood as regularity in the algorithm's workload, i.e., whether

	Regular	Irregular		Regular	
High	1	2	3	4	Large Grain
Interaction	5	6	7	8	Small
Low	9	10	11	12	Grain
Interaction	13	14	15	16	Large Grain
	Low Parallelism		High Parallelism		

Table 6.1: Application space divided according to application's characteristics

or not the workload depends on the particular data being processed. Another possible interpretation is the homogeneity of the algorithm across all processes that constitute the parallel application. This characteristic is addressed with the decomposition method attribute.

Most current parallel or distributed environments favour applications falling on square 16, i.e., applications that exhibit coarse granularity, low interaction overheads, regular algorithmic workload and high degree of parallelism.

Parallel applications can be further classified according to an additional attribute: the decomposition method selected to implement the application. The programmer can, in general, opt for either *functional* or *domain decomposition*. Functional decomposition involves decomposing the application's algorithm into several different functional units and assigning these units to different processing nodes. Domain decomposition consists on applying the same algorithm to different subsets of data. These are also referred to as data parallel applications. Although the particular decomposition method to select on each case depends on both the application and parallel system characteristics, domain decomposition is usually simpler to implement and exhibits more scalability. Functional decomposition requires the development of several different processes and the application's degree of parallelism is bounded by the number of functional units. Domain decomposition, on the other hand, requires the development of few different processes — Single Program Multiple Data (SPMD) or Few Programs Multiple Data (FPMD) programming paradigms — and the degree of parallelism depends on the data set, which, in many cases, can be made arbitrarily large. Hybrid decompositions are also possible and very common.

Since it is impossible to test the thesis' hypothesis against all applications' classes, one of them has to be selected. The selected case study falls into either square 11 or 15 of table 6.1 and uses domain decomposition. It consists on a ray tracer, which renders photo–realistic images from a 3–dimensional description of the world, using a global illumination model. Since image space decomposition is used, i.e., the image is divided into subregions to generate parallel tasks, and each of the image's pixel can be autonomously rendered, the application exhibits high parallelism and low interaction overheads. The grain size

can be arbitrarily divided, therefore either small or large grains are possible. This can be classified as an arbitrarily divisible load (section 3.6.3). Since the computational workload required to render each pixel depends on the particular subregion of the world traversed by the primary and secondary rays, the algorithmic workload is irregular. This particular ray tracer can process complex 3–dimensional scenes, whose description does not fit into each node's local memory, on the target distributed memory parallel system. The ray tracer implements, by software, a read–only distributed shared memory (DSM) system, which allows data items stored on remote nodes' memories to be fetched. This operation can increase the interaction overheads among nodes. At the renderer level, the distributed system is a read–only NUMA. For further details about ray tracing see chapter 7.

6.3 The Problem's ESR Classification

This section presents the ESR problem classification according to the attributes presented in table 4.1. On a ray tracer parallelised over the image space, i.e., where each pixel can constitute one task (domain decomposition), and where image consistency is not used to speed up the renderer, tasks are independent of each other and the maximum number of possible tasks is previously known — the image's dimensions determine the set of possible tasks. The computational effort required to render each pixel can not be previously known without heavy preprocessing or pre–sampling; these alternatives are not considered in this work since they are too application dependent. Furthermore, the execution time varies over the image being rendered (figure 8.2). Therefore, the resource requirements must be stochastically modelled.

$$E : \left\{ \begin{array}{l} \text{independent tasks} \\ \text{static arrivals} \\ \text{stochastic resource requirements} \\ \text{domain decomposition — divisible tasks} \end{array} \right\} - S : \left\{ \begin{array}{l} 2 \ldots\ 7 \text{ processors} \\ \text{heterogeneous resources} \\ 350 \ldots\ 400 \text{ MHz; } 128 \text{ MBytes} \\ \text{stochastic availability} \\ \text{stochastic comm. overheads} \\ \text{message passing} \end{array} \right\}$$

$$R : \left\{ \begin{array}{l} \text{execution time minimisation} \\ \text{suboptimal performance} \end{array} \right\}$$

The distributed system has seven processors, whose only difference is the CPU clock frequency. However, these resources must be classified from the application's perspective. They are heterogeneous, since they can have different background workloads and, hence, present different computing throughputs to this particular application; the distributed system is shared among several users, therefore its resources' availability is stochastic. The communication mechanism is based on message passing, and its overheads are stochastic,

since the communication medium may be busy and message packing times vary as a function of message size and processor utilisation. The performance requirement is suboptimal execution time minimisation.

According to this classification, the applications' resource requirements, the resources' availability and the communication overheads follow stochastic distributions, instead of having deterministic values that can be precisely determined. This clearly identifies two sources of uncertainty, as presented throughout chapter 2: the scheduling agent can not predict exactly neither the application's near future behaviour nor the distributed system's state.

6.4 Performance Modelling

In order to both make decisions and assess its efficiency and effectiveness, the scheduling agent must collect a set of metrics. This set of metrics is referred to as the scheduler's performance model. These metrics are acquired through the agent's sensors, and constitute all the dynamic external information it has about the environment's state and the quality of the schedule it is generating. They can be subdivided into three groups: performance, environment and scheduling overheads metrics.

6.4.1 Performance Metrics

The performance metrics evaluate the scheduler's performance goals degree of achievement, or effectiveness. Since the goal of the schedulers used throughout this thesis is to minimise the application's execution time, this quantity is used as the performance metric and denoted by T_{exec} [101].

6.4.2 Environment Metrics

The environment metrics are used to update the image the scheduler has about the environment's current state, and may include the distributed system's state, the application's current state of execution and workload profiles. These quantities can be directly used by the agent's decision making mechanism. Environment metrics can be further subdivided into two subgroups: foreground workload and resources' capacity metrics.

Foreground workload metrics

These values are used to measure each resource's current workload that was directly assigned by the application level scheduler. They must correlate well with tasks' response

times, since they are used to predict a task's performance if executed at a given resource. Throughout chapter 8, the foreground workload is quantified by the pair $(T_{elapsed}, W_{\%})$. $T_{elapsed}$ is the time elapsed since the task was assigned to that resource, until the sampling instant; and $W_{\%}$ is the percentage of work already executed. These metrics are used to estimate the time required to finish the current task, T_m.

Resources' capacity metrics

These metrics are used to assess the resources' dynamic availability; they measure the background workload on each resource, in contrast to the foreground workload metrics, which measure the workload directly assigned by the application level scheduling agent. The background workload is defined as the workload assigned to a resource by other processes that do not belong to the application being scheduled. This workload may be generated by other applications, eventually owned by different users, operating system processes and daemons that share the distributed system's resources. These values must correlate with the performance degradation that results from sharing the resource with several processes and can be either application–independent or application–dependent.

The set of resources' capacity metrics that are actually used by the scheduling agent depend on the set of resources that are considered relevant to achieve the scheduler's performance goals. Depending on the particular characteristics of the distributed system and application, these may include, for example, the processors, the memory hierarchy and the communication network. Each of these resources's capacities can be measured, respectively, by the computing throughput, the memory capacity and the disk space and the communication bandwidth and latency.

Since the ray tracer is low demanding on interactions among nodes when compared to the computing requirements (see section 6.2), and the distributed system has a low latency network, the communication overheads are neglected on a first approach. The results presented throughout chapter 8 will show the correctness of this option.

The most time consuming task performed by a ray tracer consists on intersecting straight lines (rays) with the objects present in the scene being rendered; hence, a processor's computing throughput can be measured by the rate at which it is able to compute these intersections, expressed in intersections per second. This metric must be inversely proportional to the background workload. Such metric is used by the scheduler proposed throughout chapter 8, and is referred to as Ir (intersection rate).

6.4.3 Scheduling Overhead Metrics

These metrics are used to quantify the overheads, both direct and indirect, imposed by the scheduler upon the distributed system. They can be used in two different ways: either

by the scheduler's designer to analyse *a posteriori* the scheduler's efficiency and eventually change its algorithm, or by the scheduling agent itself to automatically adapt its strategy in order to minimise overheads and maximise benefits, i.e., to optimise its efficiency. Metrics for direct costs and for the three types of indirect costs identified in section 3.7.2 are presented next.

Direct Costs

Direct costs represent the resources directly consumed by the scheduler. These are incurred every time the agent initiates some scheduling activity and can be measured by counting how often each particular activity is triggered and what is the average cost of each of these activities.

The list of scheduling activities that incur direct costs is presented in table 3.1. The implementation of these metrics depends on the application and scheduling policy used, being described in section 8.3. The scheduling agents developed throughout these experiments measure direct costs associated with information messages and the execution of the scheduler's selected actions. Two of these metrics are fully reported in appendix D: #T, the number of tasks, and #TS, the number of information messages. The number of tasks required to finish the application is directly related to the scheduler's activity: a large number of tasks indicates that the scheduler had to intervene frequently, imposing large migration overheads, and may be a clue to instability [92]. A third metric, T_{sched}, is also computed in run–time to assess the time required to migrate a task between two processing nodes.

An obvious direct cost is the CPU time spent by the scheduling agent in its decision making process, referred to as T_{decide}. This cost is directly proportional to the complexity of the scheduling strategy used. T_{decide} is so small for the schedulers used throughout this thesis, that it is not taken into account in the analysis of these schedulers' efficiency.

Indirect Costs

Work replication — occurs when several processing nodes must perform the same work in order to complete their respective tasks. The work replication penalty is quantified by Pen%, which measures the number of additional computations required by a particular scheduling strategy compared to some reference scheduling strategy; it is computed as

$$Pen\% = \frac{W_{sched} - W_{ref}}{W_{ref}} * 100 \tag{6.1}$$

Details about how W is quantified are given in section 8.3.

Resources' idle times — occur when the scheduler fails to keep the resources busy. Resources' idle times are quantified by TTidle, which measures the total time spent by all

processing nodes waiting for tasks. TTidle is given by

$$TTidle = \sum_{i}^{N} Tidle_i \tag{6.2}$$

where N is the number of processors and $Tidle_i$ is the time processor i spent waiting for tasks. This includes the time waiting for the application to finish when some processors already performed their last task. TTidle measures the time waiting for tasks specific to the application being scheduled and assigned to the processors by the application level scheduling agent. An high TTidle does not mean that the processor was idle, since it may be busy processing other applications's tasks; furthermore, an high TTidle can result from the scheduler being unable to use all the available resources effectively, but can also be the result of a rational decision made by the scheduler, since the overheads of using some resources can be larger than the associated benefits, probably due to a very low processing capacity caused by a large background workload.

$$TTidle\% = \frac{TTidle}{N * T_{exec}} * 100 \tag{6.3}$$

measures the percentage of the aggregated execution time that was spent waiting for tasks.

The standard deviation of the nodes' busy times, $Tbusy_i = T_{exec} - Tidle_i$, referred to as StdDev, is also presented for all experiments, and is given by

$$StdDev = \sqrt{\frac{\sum_i^N (Tbusy_i - \overline{Tbusy})^2}{N}} \tag{6.4}$$

If StdDev is 0, then all processing nodes had work allocated for identical time periods. This does not mean, however, that all nodes were busy during the whole execution time, since they may have been idle for identical time periods, but in different instants. StdDev gives an hint about the average degree of load balancing among nodes, but not of the dynamic instantaneous state of the system with respect to load balancing.

Remote data access overheads — result from the need to fetch data from remote locations. These are quantified by TTdata, which measures the total time spent by all processing nodes waiting for remote data items, instead of performing useful work. TTdata is given by

$$TTdata = \sum_{i}^{N} Tdata_i \tag{6.5}$$

where N is the number of processors and $Tdata_i$ is the time processor i spent waiting for remote data.

$$TTdata\% = \frac{TTdata}{N * T_{exec}} * 100 \tag{6.6}$$

measures the percentage of the aggregated execution time that was spent waiting for remote data. Remote data fetching is an undesirable overhead, that the scheduler may try to minimise by exploiting data locality.

Although the overhead metrics can be used by the scheduling agent in its decision making mechanism to optimise its efficiency, the schedulers proposed throughout this work do not make such use; these are expected to be too low to significantly compromise the scheduler's effectiveness. Only T_{sched} is considered by the scheduler in run time; the remaining metrics are used only after completion to evaluate the scheduler's efficiency. The experimental results will show the correctness of this approach.

6.5 Reference Scheduling Strategies

To assess the effectiveness of the decision network approach to the scheduler's decision making mechanism, its results are compared with those of three other scheduling strategies: a static uniform data distribution, a demand–driven approach and a sensor based deterministic dynamic scheduling strategy, referred to as "det", that uses dynamically gathered information about the environment's state. All these are application level schedulers, whose performance goal is to minimise execution time. The schedulers can have either a centralised or a distributed architecture. Although centralised strategies are optimal from a logical point of view [13], they often exhibit scalability problems, since the central agent may become a bottleneck (section 3.5). On a system with seven nodes, however, the central scheduling agent does not become a contention point; all these schedulers have a centralised source of control, employing a single agent for decision making.

Parallel applications with a structure similar to ray tracing with image space decomposition have a static maximum number of parallel tasks, which is known at the beginning of execution, and determined by the resolution of the image being rendered. This is in contrast with applications that dynamically generate new tasks [36, 49, 141]. When the whole set of possible tasks is known at the beginning, the scheduler can either initially allocate the entire workload to the processing nodes and then subdivide the tasks and reallocate them in run–time, or perform the work subdivision and allocation on demand. The approaches followed in this assessment include some of these alternatives: the static data distribution does not reallocate tasks, the demand driven allocates tasks on demand and the sensor based deterministic approach performs an initial allocation of workload and then reallocates it in run–time, using task migration.

This initial allocation of workload to resources is equivalent to static scheduling. Rather complex strategies can be used to perform this step. The appropriate complexity depends on the performance goals, and on what is known about the tasks' requirements and the distributed system's capacities and dynamics. If these two latter factors are known, at least to some extent, then the scheduler may spend additional efforts on this step to produce a suitable initial allocation, by using an adequate partitioning strategy (section 3.3). Decision networks could also be used on this initial step (section 5.7). An optimised

initial allocation may reduce the number of task migrations required in execution time. This approach does not seem very adequate for the present work, since the tasks' requirements are unknown for the ray tracer and the distributed system is shared among several users. Although some preprocessing could be done to estimate the tasks' requirements [61, 137, 138], this was not applied because it is too application dependent; furthermore, the high variability in capacity exhibited by resources in a shared system, may render this initial allocation inadequate soon after execution begins. This led to a simple approach to the initial allocation of work: the image to be rendered is divided on as many sub–regions as the number of processing nodes, and each region (task) is allocated to each node. Since the scheduler is dynamic, it can redistribute this workload according to the environment's state at each instant, by performing task migration at a later scheduling episode.

This section briefly presents each of the four scheduling strategies. Further details are presented on the chapter describing the actual experiment, since these depend on the particular details of the scheduler's internal execution model, the decision making mechanism and the ray tracer. Table 6.2, at the end of this section, presents the ESR classification of the four scheduling strategies, according to the attributes presented in table 4.2.

6.5.1 Uniform Work Distribution

This is a static work allocation strategy. The scheduler does not spend any effort at run–time to optimise execution time: it does not collect environment's state information and does not make dynamic scheduling decisions. The scheduler finishes its role before execution time. The image to be rendered is divided into as many vertical strips, with identical widths, as the distributed system's nodes. Each of these strips is assigned to one node, without any further work redistribution.

This approach corresponds to the worst case and its results are used as an upper bound for execution time. If any of the dynamic schedulers presents worst execution times than uniform distribution, then its scheduling overheads clearly exceed the benefits. If the workload is uniformly distributed across the image space, then an uniform decomposition may result in an even distribution of workload among the processing nodes, thus achieving the best results, since no scheduling overheads are incurred at run–time; the results obtained in a dedicated system with two nodes with a fairly simple scene (balls3), presented in figure 8.16 and table D.2, are an example of this situation. With two nodes, uniform work distribution slightly outperforms both the demand–driven and the sensor based deterministic scheduling strategies, because the total workload gets almost evenly distributed between the two nodes, as shown in figure 8.2.

6.5.2 Demand–Driven Work Allocation

This strategy follows the processor farm paradigm. Processing nodes demand additional work from a central scheduling agent, whenever they finish their previously allocated task. The central scheduling agent initially subdivides the image into a number of subregions (tasks) larger than the number of processing nodes, and assigns one of these regions to each processor. On completion of a task, another one is requested to the central scheduler. This process is repeated until the whole image has been generated. The demand–driven approach is labelled as “dd” in all tables and graphics where results are presented.

If the number of tasks is large enough, the system can be fairly well balanced, since nodes with larger computing throughput will get more work. However, since no workload redistribution takes place after assigning tasks to processors, it is possible that a slow/overloaded processor gets a specially heavy image subregion and takes a considerable longer time to render it, while the remaining nodes have already finished the allocated work and stay idle waiting for the slower node to complete. The potential for this to happen decreases as the number of tasks increases; however, having too much tasks increases scheduling overheads, since more tasks' assignments have to be made, and work replication and remote data accesses may increase. For these experiments, the image is subdivided into 25 tasks for systems with 2 or 3 nodes, and into 64 tasks if the number of nodes is larger than 3.

The demand–driven strategy does not require extended information about the environment's state: the central scheduling agent only needs to be informed that a processor has finished the previous task. This reduces direct scheduling costs, since only one information message is required per task, no effort is spent computing the environment's metrics and the decision making mechanism is extremely simple.

6.5.3 Sensor Based Deterministic Strategy

Both the sensor based dynamic deterministic scheduler and the decision network based scheduler use the same information about the environment's state and have the same set of possible actions to intervene on the environment's behaviour. In other words, they have identical sensorial apparatus and effectual capabilities. The main difference between them, is that the former uses a deterministic execution model, whereas the latter uses a stochastic one. The main motivation behind this approach is to enable direct comparisons between deterministic and stochastic approaches of identical complexity.

The sensor based deterministic scheduling strategy is labelled as “det” in all tables and graphics. Since the actual strategy details depend on the particular execution model being used, the sensorial apparatus and the set of actions available, only a general description can be presented here. A detailed description is presented in chapter 8.

This scheduling strategy uses information about the current system's state and workload profile to make predictions about near future behaviour. Initially, the image being rendered is divided into as many vertical strips of identical widths as the number of system's nodes, and each of these subregions is assigned to one node. This is identical to uniform work allocation. However, application processes are now able to send information messages to the central scheduler, informing it of the environment's state. This is the information policy. Using this information and its transfer policy, the central scheduling agent classifies the nodes as either potential work suppliers or receivers, for dynamic task migration.

The selection policy is used to determine the amount of work to transfer between two nodes, one sender and one receiver, aiming to minimise total execution time. It works in a pair by pair basis, i.e., one supplier only transfers work to one receiver in each scheduling step. This can be classified as a myopic view, since the benefits could be larger if the surplus workload was allowed to be distributed among several receivers. However, this more complex approach could also have non–desirable effects, such as increasing the number of tasks' migrations and the execution model's complexity, thereby increasing the time necessary to make decisions and the time it takes to design and test the scheduler. For each pair (supplier,receiver) the selection policy computes the amount of workload that should be transferred, based on the nodes' respective computing throughputs. This is expressed as a percentage of the workload at the supplier, and is a function of the nodes' relative computing throughput.

The location policy is then used to select the most suitable (supplier, receiver) pairs. For each potential receiver, the location policy selects the supplier which maximises the gain in execution time, T_{gain}, according to equation 6.7

$$T_{gain} = \max(T_{ms}, T_{mr}) - \max(T'_{ms}, T'_{mr}) \tag{6.7}$$

where T_{ms} and T_{mr} are the estimated time required to finish the supplier and the receiver currently allocated tasks, T'_{ms} and T'_{mr} are the expected execution times of the tasks assigned to the supplier and receiver, respectively, if the percentage of work computed by the selection policy is actually transferred. The supplier selected with equation 6.7 is accepted only if this gain is larger than the time required to migrate a task between two processing nodes multiplied by a constant. This time is measured at run–time and is referred to as T_{sched}. This is a direct scheduling overhead metric. The constant value was arbitrarily set to 6 throughout all experiments to ensure that the migration gain is significantly larger than the migration cost.

After selecting a supplier, the scheduler initiates the task migration mechanism. The supplier may refuse to transfer some of its workload, since, due to information aging, it is possible that it has already finished its currently allocated task. If this happens then the next supplier is selected. Upon reception of an acknowledgement message from the supplier, both the supplier and the receiver ids are removed from their respective lists, and

the scheduler proceeds with the next potential receiver. This process is repeated until one of the candidates' lists, built by the transfer policy, becomes empty.

6.5.4 Decision Network Based Strategy

The deterministic scheduling strategy uses deterministic quantities to make decisions about the suitability of a given task migration; the decision network based strategy, labelled as "DN", models some parts of the environment as random variables. The interactions among these variables are stochastically modelled, by means of conditional probability tables. Modelling part of the environment with random variables allows the explicit integration of the uncertainty that may exist about their current values and of the exact way these variables interact.

The DN strategy uses an execution model equivalent to that of the deterministic strategy. The information policy is exactly the same. The location and selection policies are now implemented mainly by evaluating the decision network. It is evaluated for each pair of nodes, recommending the most adequate action for this pair, expressed in terms of the percentage of work that should be transferred between the two nodes and on which direction. For each pair of nodes, the sensor's readings are entered on the network as evidence and the belief distribution over the environment's current state is inferred. Then, for each alternative action, both the probability distribution over the environment's next state and the expected utility of this action are computed. Decision networks provide an automatic process of performing all these inferences on a small number of steps. The action which maximises the expected utility for each pair of nodes is then selected. This produces a list of recommended actions, one for each pair of nodes.

This approach requires the assessment of the decision network C_2^n times, which may represent a large overhead, since it grows with $\mathcal{O}(n^2)$:

$$C_2^n = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2} \approx \mathcal{O}(n^2)$$

To reduce the number of evaluations of the decision network, hence keeping the direct cost of decision making, T_{decide}, within acceptable values, it is evaluated only with pairs of potential suppliers and receivers, as identified by the transfer policy, which is, therefore, deterministic.

Assessing the decision network once for each pair of nodes may be considered a myopic approach, since the scheduler only considers those two nodes' states to recommend an action. This is similar to what happens with the sensor based deterministic scheduling strategy. The alternative is to build a decision network where the states of all nodes are considered at once, and the most appropriate action would possibly entail various workload movements through the system. However, building such a network would be much more

demanding, at least with respect to the probabilities' specification, and different networks would be required for systems with different number of nodes. It could also result in a larger number of tasks migrations.

The resulting list of recommended actions is then sorted by descending order of gain in execution times, T_{gain}, given by

$$T_{gain} = \max(T_{ms}, T_{mr}) - \max(T'_{ms}, T'_{mr})$$

where T'_{ms}, T'_{mr}, T_{ms} and T_{mr} are the estimated execution times on both the supplier and the receiver, with and without executing the action, respectively. One recommended action may be refused by the scheduler if its estimated gain, T_{gain}, is less than T_{sched} multiplied by a given constant. The location policy is, thus, very similar to the one used in the deterministic approach, reflecting the fact that the execution models used on both strategies are very similar. The scheduler then tries to execute these actions one by one following the ordering just imposed. Once an action has been effectively executed, i.e., not refused by the supplier, all actions where either the supplier or the receiver appear are removed from this list, assuring that no node is involved in more than one action per scheduling step.

Table 6.2 presents these four scheduling strategies classification, according to the attributes discussed in section 4.1.5.

Characteristic	**Attributes**			
	Uniform	**dd**	**det**	**DN**
Information Space	—	global	global	global
Migration Space	—	global	global	global
Adaptation	static	dynamic	dynamic	dynamic
Location of Control	centralised	centralised	centralised	centralised
Kind of Transfers	one–time assignments	one–time assignments	divisible loads	divisible loads
Decision Mechanism	deterministic	deterministic	deterministic	stochastic
Environment's State Information	none	simple	detailed	detailed
Application's Execution Model	none	none	deterministic	stochastic

Table 6.2: The Scheduling Strategies' ESR classification

6.6 Synthetic Background Workload

To evaluate the scheduling agents' effectiveness on distributed shared systems, it is necessary to experiment them under a variety of background workloads. *Background workload*

refers to all those processes that are not under the application level scheduling agent's control, such as other applications, eventually belonging to different users, and operating system processes. Conversely, the processes under the scheduling agent direct control are referred to as the *foreground workload*.

Characterising and generating a realistic and general background workload is a complex issue [20, 21, 22, 23], if not an elusive goal [38], that falls outside the scope of this thesis. To simplify this issue, a few synthetic background workloads are generated, by overloading some of the distributed system's resources with a synthetic program [112].

Since the schedulers used throughout chapter 8 only take into account the nodes' computing throughput in their execution models, these are the resources that are overloaded in these experiments. For this purpose, a synthetic program, referred to as "CPUSpoiler", has been developed, which performs floating point operations during all its life time. On a time sliced environment CPUSpoiler creates a background workload that steals CPU time from other processes. Each workstation background workload can be made heavier by launching several copies of CPUSpoiler.

Three different synthetic background workload patterns, referred to as Light, Medium and Heavy backgrounds, are used throughout this work. Together with the dedicated mode, these workload patterns simulate different degrees of system sharing among applications, as illustrated in figure 6.2. These simulate 10 virtual users per node, each of them launching new processes with a given frequency. The mean arrival time between consecutive processes of the same user, hereby referred to as the mean arrival time, and the life time of each process can be selected, allowing the simulation of different background workloads, with different CPU demands.

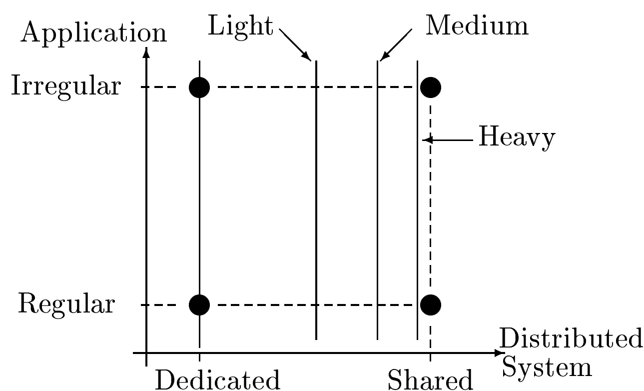


Figure 6.2: Background workload: the system sharing level versus the application regularity space

The arrival time is measured as the time elapsed since this user's last process terminated until a new process is spawned. Each virtual user can only have an active process at each

instant. The arrival time is exponentially distributed with parameter α, which means that the mean arrival time is given by $\frac{1}{\alpha}$ [114]. The synthetic workload can be made heavier by decreasing each user's mean arrival time.

Each process life time is measured as the time elapsed since the process is launched until it is terminated. The CPUSpoiler program is used to simulate the users' processes. The life time is also exponentially distributed. The synthetic workload can be made heavier by increasing each process' mean life time.

The three different background workload patterns are presented in table 6.3, sorted by increasing order of CPU demand. These workloads are spatially heterogeneous, since two out of the seven nodes are submitted to lighter background loads on average.

Mode	Comments
Dedicated	No synthetic background workload. Only system processes coexist with the application being scheduled.
Light	Mean life time = 20 seconds Nodes 0 and 2: Mean arrival time = 80 seconds Remaining Nodes: Mean arrival time = 60 seconds
Medium	Mean life time = 20 seconds Nodes 0 and 2: Mean arrival time = 60 seconds Remaining Nodes: Mean arrival time = 40 seconds
Heavy	Mean life time = 20 seconds Nodes 0 and 2: Mean arrival time = 20 seconds Remaining Nodes: Mean arrival time = 5 seconds

Table 6.3: Synthetic background workloads

To ensure the experiments' reproducibility all the values needed to generate the workloads, i.e., the processes' arrival and life times, were previously generated and saved. These are then used to drive the simulations, ensuring that all experiments are identical with respect to the synthetic background workload.

Figure 6.3 shows the behaviour of an application–dependent computing throughput metric, the intersection rate (described with detail in section 8.3.2), when the distributed system is submitted to these background workloads. The figure illustrates the relative computational weigh of the selected synthetic background workloads. Larger values represent larger available computing power. In order to keep the graphics readable only four nodes are shown. The remaining nodes exhibit a similar behaviour.

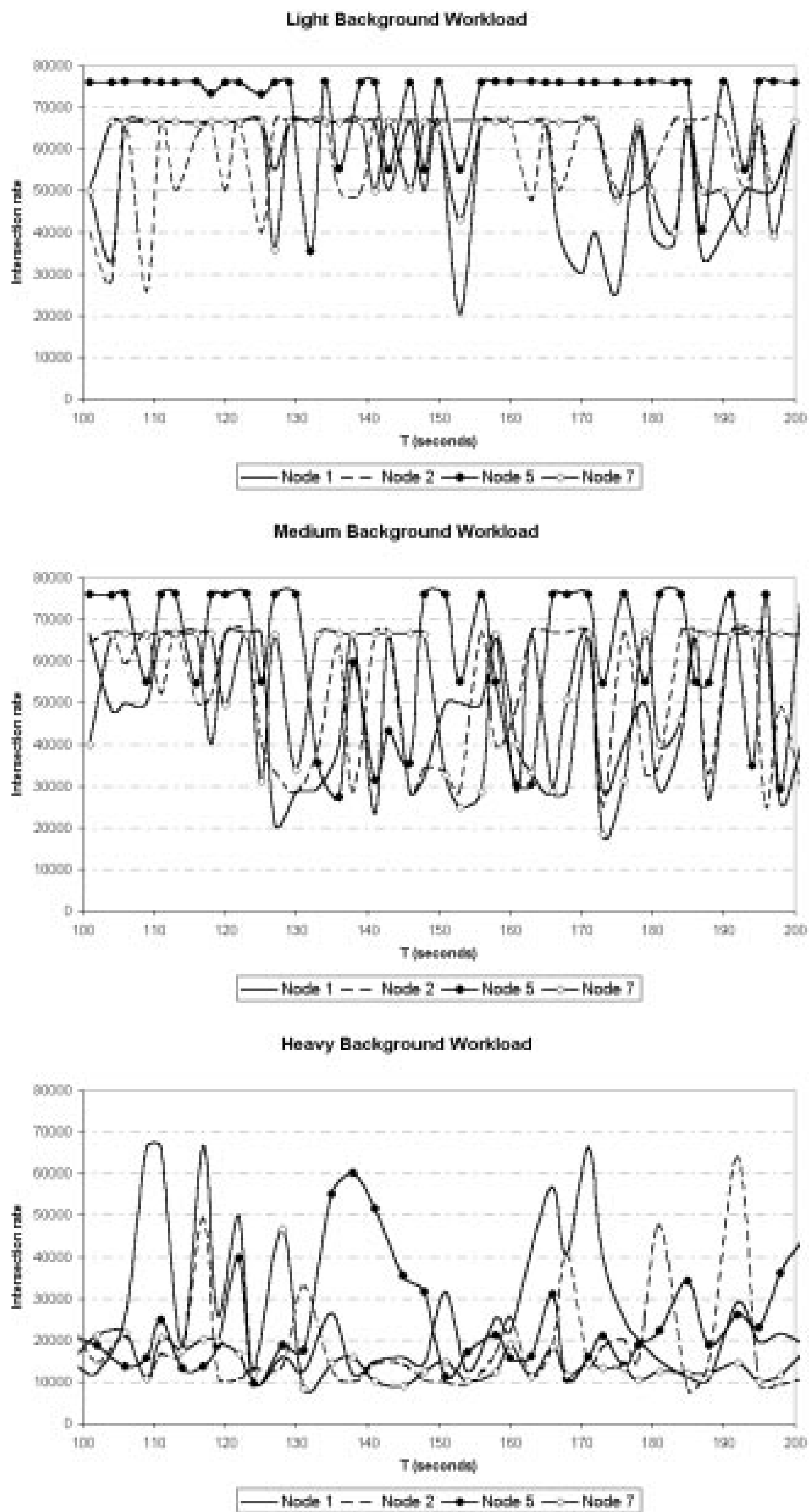


Figure 6.3: Computing throughput variation with different synthetic background workloads

6.7 Summary

This chapter describes the environmental setup used throughout the experiments performed to verify the proposed hypothesis. This can be summarised as follows:

- the distributed computing system is a small cluster, with seven nodes, that can be simultaneously shared among several users; the nodes are interconnected by a low latency, high bandwidth Myrinet network;

- a ray tracing application is used as the case study; this application exhibits an irregular algorithmic workload, low interaction among tasks and high parallelism; it has been partitioned into parallel tasks using image space decomposition, which provides arbitrarily divisible loads; the scenes being rendered may be arbitrarily complex, which may require that data is distributed among the nodes' memories; this feature can increase the interaction among nodes;

- the experimental results of three different reference scheduling strategies are compared with those of a decision network based scheduler; all these schedulers perform application level scheduling, aim to minimise execution time and have a centralised architecture;

- the workload is initially allocated to the various processing nodes following a straight forward partitioning strategy, which does not consider neither the tasks' requirements nor the resources' capacities; the sensor based dynamic schedulers focus their efforts on workload redistribution at run time by performing tasks migrations;

- the sensor based schedulers do not include scheduling overheads on their decision making process, with the exception of the time required to migrate a task between two nodes; scheduling overheads are used to evaluate the scheduler's behaviour after the application completes;

- the only environment metrics used are the nodes' computing throughput and the tasks' estimated time to finish; the computing throughput is justified by the high computing requirements exhibited by ray tracers; other resources' capacity metrics, like communication bandwidth and latency, are neglected, since an high performance network is used; no attempt is made to subdivide the tasks' estimated time to finish in its various components, such as the computation time, the remote data fetching time, etc.

- the distributed system is submitted to 3 different synthetic background workload patterns, which simulate shared environments.

The proposed decision network fits into the generic decision network proposed in section 5.7, with the following modifications:

- scheduling overheads are not explicitly considered on the selection policy; this block is not present in the actual decision network;
- the processing nodes are considered in a pair–wise basis; each of the resources' capacities and tasks' requirements metrics include two sensors and respective stochastic metrics, one for each node;
- the next state is described by a single variable related to the expected degree of load sharing between the two nodes being considered at each inference step (section 8.5.1).

Chapter 7

Ray Tracing: a Case Study

Contents

7.1 Ray Tracing Algorithm 132
7.2 Illumination Model 135
7.3 Some Ray Tracing Deficiencies 141
7.4 Acceleration Techniques 142
7.5 Parallel Ray Tracing 144
7.6 PaRT – Parallel Ray Tracer 146
7.7 Summary 149

Ray tracing is a computer graphics method used to render photo-realistic images from three–dimensional descriptions of the world. This is achieved by simulating light behaviour. Ray tracing is one rendering algorithm among others, such as radiosity and particle tracing [135, 136]. These algorithms differ in the way light paths are approximated, which means that the resulting light effects vary from algorithm to algorithm.

Rendering algorithms are computationally very demanding, with execution times that can be intolerably long, depending on the scene's complexity and on the illumination model's richness. To speed up ray tracing many approaches have been proposed, including parallel processing. Since in ray tracing one pixel's computations are completely independent of any other pixel, and the data structures needed to describe the world are not modified, this algorithm's parallelisation is straightforward. If the world's description is made available to all processing nodes, then an image space problem decomposition can be used, assigning different image regions to each node [61]. Ray tracing falls in the class of embarrassingly parallel problems. Since the workload associated with each pixel may vary widely across the image [61, 135], the only issue left to be addressed is load management. However, there are no dependencies among the various tasks and nodes, hence this problem can be easily and efficiently solved using some form of demand driven work allocation [27, 135, 136].

Unfortunately, having the whole scene description directly available to all processing nodes may be impossible. On a distributed memory parallel system this requires replicating the data across the processors, which limits the problem size to that of the memory available at each node. To handle larger scenes this database has to be distributed across the nodes. A distributed scene implies that processors either migrate tasks or fetch remote data, whenever a given datum can not be found at the processor's local memory. This incurs overheads that prevent naive ray tracing algorithms from scaling with the number of processors [27, 135, 136].

When the scene data does not fit into each processor's local memory, the problem of efficiently parallelising ray tracing becomes more difficult. Work allocation to the processors can no longer be efficiently done based solely on a processor's availability. Issues such as data access costs must also be considered, to ensure that the time spent fetching remote data or migrating tasks is minimised. The problem of efficiently matching tasks and data becomes even more complex due to the unpredictability of the data access patterns exhibited by ray tracing tasks.

This complexity motivated the choice of ray tracing with large scene descriptions as a case study for this work.

This chapter presents PaRT, a parallel ray tracer developed purposely as an experimental testbed for this work. It begins by introducing the ray tracing algorithm and the Phong illumination model and identifies some deficiencies associated with this algorithm. Parallel ray tracing is then discussed, and section 7.6 describes PaRT's architecture.

7.1 Ray Tracing Algorithm

All rendering algorithms try to solve the rendering equation which models light behaviour [48, 59, 86, 135, 136, 174]. The rendering equation is formulated as [159]:

$$L_0(x,\Theta_0) = L_e(x,\Theta_0) + \int_{all x'} v(x,x') f_r(x,\Theta_0',\Theta_0) L_0(x',\Theta_0') \cos\Theta_i \frac{\cos\Theta_0' dA'}{\|x'-x\|^2} \qquad (7.1)$$

This equation states that the outgoing radiance L_0 at surface point x in direction Θ_0 is equal to the self-emitted radiance L_e plus the incoming radiance from all points x' reflected into direction Θ_0. $v(x,x')$ is a visibility term, being 1 if x' is visible from x and 0 otherwise. Θ_0' is the direction between x and x'. The material properties of surface point x are represented by the bi–directional reflection distribution function $f_r(x,\Theta_0',\Theta_0)$, which returns the amount of radiance reflected into direction Θ_0 as a percentage of the incident radiance from direction Θ_0' (represented by $L_0(x',\Theta_0')$). The cosine terms translate surface points in the scene into projected solid angles.

The rendering equation is a recursive integral that must be solved numerically. Ray tracing

is a numerical method that approximates this equation by sampling the world from the observer's point of view.

Ray tracing is a view dependent algorithm. The basic ray tracing algorithm follows a ray from the view point, through each pixel of an imaginary image plane and into the scene (figure 7.1). These rays are referred to as *primary rays*.

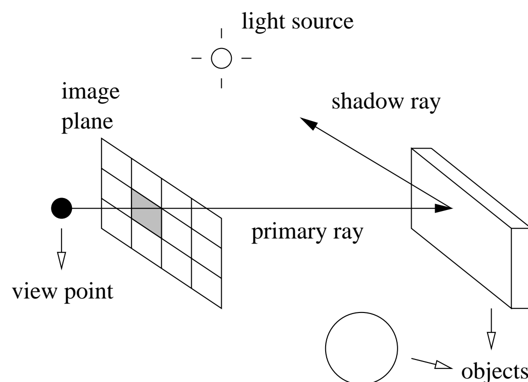


Figure 7.1: Primary and shadow rays

The closest intersection of the primary ray with an object in the scene must then be calculated in order to shade this pixel. If an intersection point is found, then the appropriate illumination model is applied, otherwise the pixel is shaded as background.

Primary rays perform hidden surface removal. In a naive ray tracer each ray must be tested against every object in the scene for intersection. If the ray intersects more than one object then the nearest intersection point is selected.

Once an intersection point is found, shadow rays are spawned towards each light source. If an object is found in the path between the intersection point and the light source, then this one is considered occluded and does not contribute to that point illumination. Otherwise, a local illumination model is brought into play and that light source contribution to the pixel is calculated (section 7.2.1). Figure 7.2 shows object O1 directly lit by light source L1, while object O3 occludes light source L2. Object O2, on the other hand, is not lit by any of the light sources.

Some ray tracers extend this model in order to account for transparent objects. The local transparency coefficients of all objects intersected by the shadow ray are multiplied by the light source intensity for each wavelength (section 7.2.1). If the product for a wavelength reaches zero, then that wavelength does not contribute to that point illumination. If the products for all wavelengths reach zero, then the intersection process may terminate, since that light source does not contribute to that point illumination. This model is known as *filtered transparency*, since objects are treated as transparent filters that selectively pass different wavelengths [48]. Filtered transparency ignores *refraction*, i.e., light rays are not bent as they go through the objects.

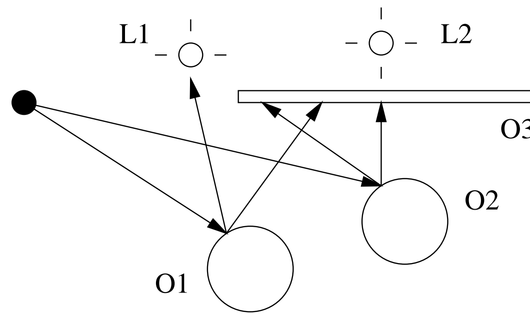


Figure 7.2: Shadow rays and occlusion

Including filtered transparency on the shading model increases the computational load associated with shadow rays. The intersection process can no longer terminate as soon as the first object intersecting the shadow ray is encountered. It only terminates when the computed products for all wavelengths reaches zero or if no intersecting object is found.

Ray tracing is a global illumination algorithm that also models light incident on a surface after interaction with other objects. Most ray tracers include perfect specular reflection and perfect specular transmission on their global illumination models (section 7.2.2). These phenomena are modelled by shooting new rays into the reflected and/or transmitted directions (figure 7.3). These rays are treated exactly the same way as primary rays. Hence, ray tracing is a recursive algorithm. This recursive process is repeated, generating a tree of rays, until a pre–established maximum depth is reached or no additional rays need to be spawned. Light arriving to the surface through each of these secondary rays is multiplied by either a global reflection coefficient or a global transmission coefficient, to model the surface's material properties (section 7.2.2).

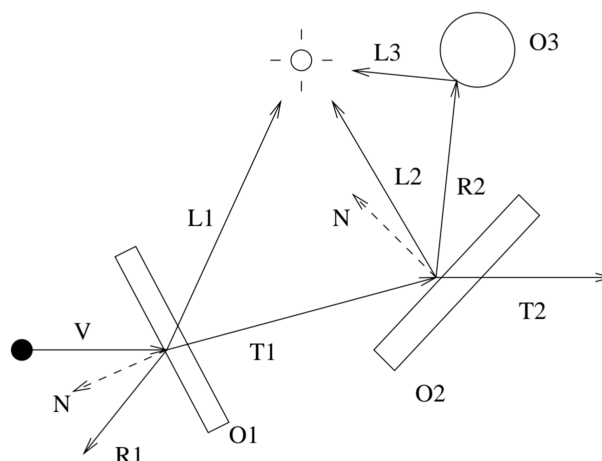


Figure 7.3: Primary (V), Shadow (L), Reflected (R) and Transmitted (T) rays

Figure 7.3 shows the tree of rays generated for one pixel. Although object O1 is not directly lit by the light source (it is self-occluding), object O2 is seen through it by specular transmission and object O3 is seen on O2's surface by specular reflection. No further rays

are spawned on O3's intersection point with ray R2, either because its global transmission and reflection coefficients are zero, or because the tree maximum depth has been reached.

7.2 Illumination Model

The *illumination model* expresses the factors that determine an object's colour at a given point and specifies the equations by which these factors are combined. An object's colour at a given point depends on the object's position and orientation relative to other objects, the view point and the light sources. It also depends on the object's material properties and on the light sources' characteristics. Although illumination models are often referred to as shading models, this term should be reserved for the broader framework in which an illumination model fits [48]. The shading model determines when and which illumination model is applied. Some shading models invoke an illumination model for some pixels, and shade the remaining pixels by interpolation.

Most ray tracers model both local and global illumination. *Local*, or first-order, *illumination models* consider only direct interactions of an object with light arriving from a light source. If light incides on a surface after interaction with another object, then that illumination is referred to as global. *Global illumination* arises from the interaction of light with reflective or transparent objects [48, 59, 173, 174].

Glassner [59] presents four light transport modes by which light interacts with objects: perfect diffuse reflection, perfect specular reflection, perfect diffuse transmission and perfect specular transmission. Ray tracers model some of these light transport mechanisms on either their local or global illumination model, or even on both of them. However, these perfect light transport modes are idealised models of reality, since there are no perfect specular or diffuse surfaces. Many techniques used in computer graphics include simplifications that have no firm ground theory, but produce acceptable results and are computationally tractable in useful time.

7.2.1 Local Illumination Model

Local illumination models consider only the first interaction of light with an object.

Many local models are presented on the literature. These include the Cook and Torrance Model, models completely based on a physical surface roughness simulation and models completely based on wave theory [48, 174]. The model used throughout this work – the Phong model – is a completely empirical approach to specular and diffuse reflection that approximates more precise theoretical models.

The Phong model is a linear combination of three terms: ambient light, diffuse reflection and specular reflection. This model will be extended by an extra term to account for the spread that a light source produces when viewed through a transparent object.

Diffuse Reflection

Surfaces exhibiting diffuse reflection, also known as Lambertian reflection, reflect light with equal intensity to all directions. For a given surface the brightness at point $\dot{P}$ depends only on:

- the angle Θ between the direction $\vec{L}$ to the light source and the surface's normal $\vec{N}$ at point $\dot{P}$ (figure 7.4);
- the surface's material properties;
- the light source's characteristics.

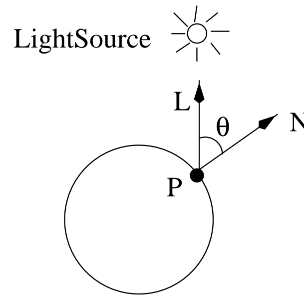


Figure 7.4: Diffuse reflection

The diffuse reflection illumination equation is

$$I_d = I_i k_d \cos\theta = I_i k_d (\vec{L} \cdot \vec{N}) \tag{7.2}$$

where I_i is the incident light's intensity (arriving directly from the light source), k_d is the material's diffuse reflection coefficient and $(\vec{L} \cdot \vec{N})$ is the dot product between $\vec{L}$ and $\vec{N}$. k_d is a constant between 0 and 1 and varies from one material to another.

Most ray tracers handle colour by considering three different wavelengths: red (R), green (G) and blue (B). k_d can be wavelength dependent. Equation 7.2 can be extended to support multiple light sources and wavelenght dependent reflections, becoming:

$$\begin{aligned} I_{d,R} &= k_{d,R} \sum_j I_{i,j,R} (\vec{L_j} \cdot \vec{N}) \\ I_{d,G} &= k_{d,G} \sum_j I_{i,j,G} (\vec{L_j} \cdot \vec{N}) \\ I_{d,B} &= k_{d,B} \sum_j I_{i,j,B} (\vec{L_j} \cdot \vec{N}) \end{aligned} \tag{7.3}$$

where $k_{d,R}$, $k_{d,G}$ and $k_{d,B}$ are the material's diffuse reflection coefficients for each wavelength, $I_{i,j,R}$, $I_{i,j,G}$ and $I_{i,j,B}$ are the incident light intensities from light source j for each wavelength, and $\vec{L_j}$ is the normalised direction vector from point $\dot{P}$ towards light source j.

The angle Θ must be in the range $[0 \ldots 90]$ degrees, otherwise the surface is self–occluding, since light cast from behind a surface does not contribute to diffuse reflection.

Ambient Light

Ambient light is included in the Phong model to account for the interaction of multiple diffuse reflections of light from the many surfaces present in the scene. The effect of this global diffuse interactions is modelled as a diffuse, non-directional light source that reaches every point of the environment. The Phong model assumes that ambient light impinges equally on all surfaces from all directions and approximates it as a constant. Ambient light is wavelength dependent and its equations are

$$\begin{aligned} I_{a,R} &= I_{amb,R} k_{a,R} \\ I_{a,G} &= I_{amb,G} k_{a,G} \\ I_{a,B} &= I_{amb,B} k_{a,B} \end{aligned} \tag{7.4}$$

where $I_{amb,R}$, $I_{amb,G}$ and $I_{amb,B}$ are the wavelength dependent intensities of ambient light present in the environment and $k_{a,R}$, $k_{a,G}$ and $k_{a,B}$ are the surface's material ambient reflection coefficients and range between 0 and 1.

More accurate solutions for modelling global diffuse interactions are discussed on section 7.3.

Specular Reflection

The highlight seen on shiny surfaces is the reflex of a light source. It is caused by specular reflection and depends on the viewpoint. Specularly reflected light is emitted mostly in the direction of $\vec{R}$, which is $\vec{L}$ mirrored about $\vec{N}$. $\vec{R}$ is given by

$$\vec{R} = 2\vec{N}(\vec{N} \cdot \vec{L}) - \vec{L} \tag{7.5}$$

Specular reflection is thus a function of Ω, the angle between the viewing direction $\vec{V}$ and the mirror direction $\vec{R}$ (figure 7.5).

The Phong model assumes that maximum specular reflection occurs when Ω is zero, and falls off rapidly as Ω increases. This falloff is approximated by $\cos^{ns}\Omega$, where ns is the material's specular reflection exponent. A value of 1 provides a broad highlight, whereas higher values simulate a sharp highlight. For a perfect mirror ns is infinite. Specular reflection is given by

$$I_{s,R} = k_s \sum_j I_{i,j,R} (\vec{R_j} \cdot \vec{V})^{ns}$$

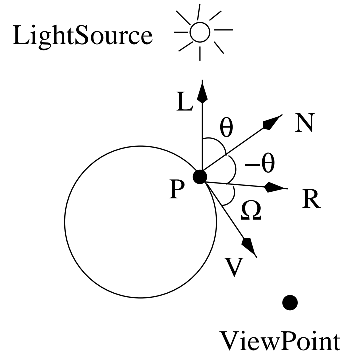


Figure 7.5: Specular reflection

$$\begin{aligned} I_{s,G} &= k_s \sum_j I_{i,j,G}(\vec{R_j} \cdot \vec{V})^{ns} \\ I_{s,B} &= k_s \sum_j I_{i,j,B}(\vec{R_j} \cdot \vec{V})^{ns} \end{aligned} \tag{7.6}$$

where k_s is the material's local specular reflection coefficient and is not dependent on the wavelength, since highlights are modelled as being of the same colour as the incident light.

Once again Θ_j must be in the range $[0 \ldots 90]$ degrees, otherwise the surface is self-occluding and won't exhibit the specular highlight.

Specular Transmission

This term is introduced as an extension to the Phong model and it accounts for light that is seen through a transparent object due to a light source standing behind it. This term is evaluated only to those light sources laying behind the object ($\Theta > 90$), whereas diffuse and specular reflection are evaluated to those standing in front of the object ($0 < \Theta < 90$).

Light is assumed to be maximally transmitted along the transmission direction $\vec{T}$, but is allowed to spread over this direction. This model is identical to the one used to allow specularly reflected highlights to spread over the maximum reflection direction. The broadness of this spread is controlled with nt, which is the material's specular transmission exponent. This term is wavelength dependent and is given by

$$\begin{aligned} I_{t,R} &= k_{t,R} \sum_j I_{i,j,R}(\vec{N} \cdot \vec{H'_j})^{nt} \\ I_{t,G} &= k_{t,G} \sum_j I_{i,j,G}(\vec{N} \cdot \vec{H'_j})^{nt} \\ I_{t,B} &= k_{t,B} \sum_j I_{i,j,B}(\vec{N} \cdot \vec{H'_j})^{nt} \end{aligned} \tag{7.7}$$

$\vec{H'}$ is the orientation that the surface should have to maximally transmit light in the viewer's direction and is a function of $\vec{L}$, $\vec{V}$ and the material's relative index of refraction

– IOR or η. IOR is discussed with more detail in section 7.2.2. $\vec{H'}$ is given by [174]:

$$\vec{H'} = \frac{-\vec{L} - \eta\vec{V}}{\eta - 1} \tag{7.8}$$

The complete equations for the local illumination model are:

$$\begin{aligned}
I_{l,R} &= I_{amb,R}k_{a,R} + \sum_j I_{i,j,R}[k_{d,R}(\vec{L_j} \cdot \vec{N}) + k_s(\vec{R_j} \cdot \vec{V})^{ns} + k_{t,R}(\vec{N} \cdot \vec{H'_j})^{nt}] \\
I_{l,R} &= I_{amb,R}k_{a,G} + \sum_j I_{i,j,G}[k_{d,G}(\vec{L_j} \cdot \vec{N}) + k_s(\vec{R_j} \cdot \vec{V})^{ns} + k_{t,G}(\vec{N} \cdot \vec{H'_j})^{nt}] \\
I_{l,B} &= I_{amb,B}k_{a,B} + \sum_j I_{i,j,B}[k_{d,B}(\vec{L_j} \cdot \vec{N}) + k_s(\vec{R_j} \cdot \vec{V})^{ns} + k_{t,B}(\vec{N} \cdot \vec{H'_j})^{nt}]
\end{aligned} \tag{7.9}$$

7.2.2 Global Illumination Model

Illumination due to light that reaches a point $\dot{P}$ after reflection from and transmission through other surfaces is computed using the rules defined by the global illumination model.

Ray tracing includes mechanisms to model perfect specular reflection and perfect specular transmission. Reflection and transmission rays are conditionally spawned into the scene from each intersection point. These are referred to as *secondary rays*. A reflection ray is spawned if the material's global reflection coefficient k_{gs} is greater than 0, and a transmission ray is spawned if the material's global transmission coefficient k_{gt} is greater than 0. Secondary rays are then traced as if they were primary rays, recursively spawning shadow, reflection and transmission rays (figure 7.3). The rays form a tree that is traced until a pre-specified maximum depth is reached or no additional rays need to be spawned. The reflected and transmitted rays' contributions are computed by applying the local illumination model at their respective intersection points and are then weighed by k_{gs} or k_{gt}.

The direction of the reflected ray, $\vec{R}$, is $\vec{V}$ mirrored about $\vec{N}$, and is given by equation 7.10. This is different from the $\vec{R}$ vector used in the local illumination model, which is $\vec{L}$ mirrored about $\vec{N}$.

$$\vec{R} = 2\vec{N}(\vec{N} \cdot \vec{V}) - \vec{V} \tag{7.10}$$

The direction of the transmitted ray $\vec{T}$ is a function of the indices of refraction (IOR) of the materials through which the light passes, η_i and η_t. A material's IOR is the ratio of the speed of light in vacuum to the speed of light in the material. In practice, this means that the direction of $\vec{T}$ is different from the direction of $\vec{V}$. The relation between the angle of incidence Θ_i and the angle of refraction Θ_t (figure 7.6) is given by Snell's Law:

$$\eta = \frac{\eta_t}{\eta_i} = \frac{\sin\Theta_i}{\sin\Theta_t} \tag{7.11}$$

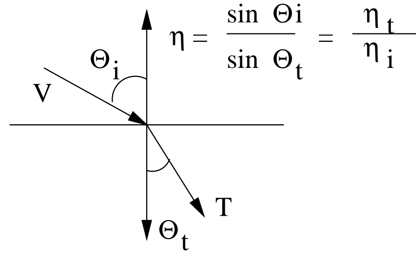


Figure 7.6: Snell's Law

The transmitted ray $\vec{T}$ is given by

$$\vec{T} = \frac{1}{\eta}\vec{V} - (\cos\Theta_t - \frac{1}{\eta}(\vec{V}\cdot\vec{N}))\vec{N} \tag{7.12}$$

$$\cos\Theta_t = \sqrt{1-\sin^2\Theta_t} = \sqrt{1-\frac{1}{\eta^2}(1-(\vec{V}\cdot\vec{N})^2)} \tag{7.13}$$

Although η varies with the light's wavelength and even with temperature, most ray tracers model it as wavelength independent to avoid shooting a different transmission ray for each frequency. The global illumination model can be written as

$$I_g = k_{gs}I_s + k_{gt}I_t$$

where k_{gs} and k_{gt} are the material's global specular reflection and transmission coefficients, I_s and I_t are the reflected and the transmitted ray contributions, respectively.

7.2.3 Ray Tracing Rendering Equation

The final equations used by a ray tracer that implements the illumination model presented above are :

$$\begin{aligned}
I_R &= I_{amb,R}k_{a,R} + \sum_j I_{i,j,R}[k_{d,R}(\vec{L_j}\cdot\vec{N}) + k_s(\vec{R_j}\cdot\vec{V})^{ns} + k_{t,R}(\vec{N}\cdot\vec{H'_j})^{nt}] + \\
&\quad k_{gs}I_{s,R} + k_{gt}I_{t,R} \\
I_G &= I_{amb,G}k_{a,G} + \sum_j I_{i,j,G}[k_{d,G}(\vec{L_j}\cdot\vec{N}) + k_s(\vec{R_j}\cdot\vec{V})^{ns} + k_{t,G}(\vec{N}\cdot\vec{H'_j})^{nt}] + \\
&\quad k_{gs}I_{s,G} + k_{gt}I_{t,G} \\
I_B &= I_{amb,B}k_{a,B} + \sum_j I_{i,j,B}[k_{d,B}(\vec{L_j}\cdot\vec{N}) + k_s(\vec{R_j}\cdot\vec{V})^{ns} + k_{t,B}(\vec{N}\cdot\vec{H'_j})^{nt}] + \\
&\quad k_{gs}I_{s,B} + k_{gt}I_{t,B}
\end{aligned}$$

The parameters presented on table 7.1 are needed to describe each material's properties and to apply the illumination model just described.

Symbol	Description
$k_{d,R}$, $k_{d,G}$, $k_{d,B}$	Diffuse reflection coefficients
$k_{a,R}$, $k_{a,G}$, $k_{a,B}$	Ambient reflection coefficients
k_s	Local specular reflection coefficient
ns	Specular reflection exponent
$k_{t,R}$, $k_{t,G}$, $k_{t,B}$	Local specular transmission coefficients
nt	Specular transmission exponent
η	Index of refraction relative to the vacuum
k_{gs}	Global specular reflection coefficient
k_{gt}	Global specular transmission coefficient

Table 7.1: Material's properties

7.3 Some Ray Tracing Deficiencies

Despite claims of realism, ray tracing uses an empirical illumination model that fails to accurately represent all light transport mechanisms, resulting in images that have a clear ray tracing signature [175]. This section discusses some of these deficiencies.

Global Diffuse Interaction

The global illumination model used by most ray tracers does not include light interaction among diffuse surfaces. This is approximated by a constant, referred to as ambient light. Modelling diffuse interaction by extending the ray tracing method would mean that at each intersection point a large number of rays would be spawned and the algorithm would quickly become intractable. However, much of the light that is seen on real scenes is due to global diffuse interactions, as are shadow's soft edges. Either radiosity or backwards ray tracing can be used to account for this phenomenon [175]. Backwards ray tracing refers to ray tracing from the light sources, instead of from the view point.

Specular to Diffuse Interaction

This problem is similar to the previous one: conventional ray tracing does not account for light incident on a diffuse surface coming from an interaction with a specular one. An intersection with a diffuse surface is tested only for light source visibility, by spawning shadow rays; light arriving from indirect routes, such as a specular reflection on another surface, is completely ignored. This light transport mechanism can also be modelled by performing a previous pass with a backward ray tracing algorithm [175].

Aliasing

Classical aliasing artifacts are caused by inadequate sampling of high frequency information

by a low frequency sampling signal. In a point–sampled image, frequencies greater than the Nyquist limit are inadequately sampled; high frequency patterns appear as low frequency patterns and interferences are formed. Since primary rays are uniformly spaced, ray traced images suffer from aliasing artifacts. One possible solution is to supersample the image. Supersampling is the process by which aliasing artifacts are reduced by increasing the frequency of the sampling grid followed by averaging down — the aliasing artifacts are not eliminated, they are smoothed. The two main drawbacks of supersampling are that:

- there is a practical limit to the frequency at which supersampling may be done; since the spatial frequency spectrum of the 3D scene can extend to infinity, aliasing artifacts are not eliminated, just reduced;

- spawning additional primary rays is an expensive process, that can increase the rendering times beyond acceptable limits.

Alternative solutions to the aliasing problem include adaptive supersampling and stochastic sampling [59, 175].

7.4 Acceleration Techniques

The naive ray tracing algorithm is a brute force approach, that tests every spawned ray against all objects in the scene. This is a very inefficient algorithm, since intersection tests are expensive and most of them fail. Several alternative efficiency schemes have been developed, which either try to reduce the number of rays or exploit coherence to reduce the number of intersection tests. This section presents some of these acceleration techniques.

Adaptive Depth Control

In a naive ray tracer, secondary rays are spawned at each intersection point to account for global reflection and transmission, generating a tree of rays for each primary ray. This tree only terminates when a ray hits nothing or a maximum trace depth is reached. Since the contribution of each secondary ray is multiplied by a constant that ranges between 0 and 1, rays at greater depth contribute a decreasing amount to the pixel value — the effort spent calculating rays deep down the tree may have an imperceptible effect on the final image.

Adaptive depth control is implemented by accumulating the product of the global transmission or reflection coefficients along the tree of rays, and stop spawning secondary rays if this value falls below a certain threshold [175].

Exploiting Coherence

Coherence is a property of the environment that expresses the degree to which parts of the environment exhibit local similarities. Coherence may be exploited by acceleration techniques. There are four types of coherence which are commonly used in the context of ray tracing. Object coherence is the most fundamental one; it expresses the fact that objects are connected, smooth and bounded and that different objects are usually disjoint in space. Image coherence is the view–dependent version of object coherence. The objects' projections in the image plane exhibit the same degree of connectedness, smoothness and boundness as the original 3D objects. Ray coherence means that rays with nearly the same origin and direction are likely to trace similar paths through the environment, intersecting the same object. Frame coherence means that successive frames of an animation are likely to be similar if the time difference is small.

Applications of object and ray coherence are given in the next sections. Image coherence is often applied to shade pixels in the middle of a surface; pixels at the surface's edges are shaded with a regular illumination model, while the remaining pixels are shaded by interpolation. Frame coherence can be applied to speed up rendering of consecutive frames; two frames are accurately rendered, while intermediate frames are interpolated [59].

Occluding Object Buffer

Ray coherence states that rays with nearly the same origin and direction tend to intersect the same objects. Shadow rays spawned from neighbouring intersection points on the same surface towards the same light source are likely to exhibit coherence among them. There is an high probability that the object that occludes a surface from a light source for a given intersection point, is the same that occludes it for nearby intersection points. If this object's identifier is saved on an occluding object buffer and if this object is tested for nearby intersection points, there is a good chance that this object is intersected and no further intersection tests are required. Ray coherence contributes to reduce the number of intersection tests.

Bounding volumes

Bounding volumes reduce the number of intersection tests by exploiting object coherence. An object with an arbitrary complexity can be enclosed in a simple bounding volume, such as a sphere; rays are tested for intersection against the simpler volume. If a ray does not intersect the bounding volume, then it does not intersect the complex object, and no more intersection tests are required. Spheres have commonly been used as bounding volumes, because intersecting a ray with a sphere is computationally simple. If the object contains a large number of polygons, this scheme can improve efficiency significantly, since it saves unnecessary tests of the ray against each polygon.

Bounding volumes can be extended hierarchically; by enclosing a number of bounding volumes within a larger bounding volume, many objects can be eliminated from further consideration with a single intersection test [48, 59, 175].

3D Space SubDivision

Object coherence implies space coherence, because objects are connected. The rational behind exploiting space coherence is simple. The 3–dimensional space occupied by the scene is subdivided into 3–dimensional regions. Regions are processed in order along the ray from its origin. Only objects contained within the current region must be tested for intersection. If no intersections occur, then the next region along the ray direction is processed. The first region where an intersection is found is the one nearer to the ray origin, therefore no further regions must be visited.

Several different approaches to 3D spatial subdivision have been proposed. All these approaches require a top–down preprocessing step, that partitions the space into non–overlapping 3 dimensional regions, also known as voxels (volume elements). Uniform space partitioning requires that all voxels are of the same size, while non–uniform space partitioning allows voxels of different sizes. The most common approach is to use an octree of non–uniform voxels, where these are hierarchically subdivided into smaller voxels, until a minimum voxel dimension is reached or the number of objects within a voxel falls below a predefined threshold [48, 59, 175].

7.5 Parallel Ray Tracing

Ray tracing is a computationally expensive algorithm, both due to its recursive nature and to the arbitrary complexity of the scene being rendered. In addition to acceleration techniques that reduce the number of operations required to render a scene, parallel processing is often used to speed up ray tracing times. To write a parallel ray tracer two major issues must be handled: problem decomposition and load distribution.

A decision has to be made whether the problem is decomposed into separate functional parts (functional decomposition), or if identical algorithms are applied to different parts of the data (domain decomposition). The latter tends to be more suitable for parallel rendering purposes [136], therefore this discussion focuses on domain decomposition.

In the context of ray tracing, domain decomposition can be done either in object space or image space [61, 101, 136]. Object space entails subdividing the 3D scene space into voxels, and allocating these voxels to the processing nodes; each processor stores a subset of the whole database, rather than all the objects in the scene. Parallel tasks are executed by the processors holding the required data; whenever a ray enters a different voxel, the

respective task is migrated to the appropriate processor. Image space decomposition entails subdividing the 2D image plane into a number of distinct regions, and allocating these regions to the processors. Since each of these regions can be rendered independently of all others, parallel ray tracing with image space decomposition is an embarrassingly parallel problem [61]. This last statement presupposes that the whole scene database may be replicated and stored at each node's local memory; with complex 3D scenes this may not be the case, and the database must be distributed among the nodes. These will need to request non–local data items, increasing the nodes' interaction and, consequently, communication overheads [27].

Object space and image space decompositions may suffer from load imbalance problems. With object space decomposition, both the number of objects per voxel and the number of rays traversing each voxel are likely to vary. Certain parts of the scene attract more rays than others; this is mainly related to the view point and the location of the light sources. Having multiple voxels per processor, representing non–adjacent regions of the scene, may be a good initial allocation of data to reduce load imbalances. Dynamic load distribution can also be applied, by shifting the voxels' boundaries in runtime. The problem with data redistribution is that data accesses are highly irregular, both in space and time; tuning such systems is very difficult and if data is redistributed too often, the data communication overhead may dominate computation times. With image space decomposition, load imbalances may occur due to differing complexities associated with different areas of the image; areas with many objects' projections tend to require more computing power than areas with less projections. Static load balancing may be performed by either allocating to each processor many non–adjacent regions of the image, or by trying to estimate, before execution, the requirements of each region. Dynamic load balancing can follow many different approaches: tasks may be assigned to the processor on demand (demand driven) or task migrations may be performed in runtime. If the scene's database has to be distributed across the processors' memories, then image space decomposition may incur high overheads due to remote data fetching.

Reinhard et al. [135, 136] propose an hybrid load management approach that tries to reduce both load imbalances and communication overheads due to remote data fetching. The scene's database is distributed across the node's memories, allowing the rendering of complex scenes. Tasks that are not computationally very intensive, but require access to a large amount of data, are rendered using object space decomposition, i.e., tasks migrate among processors to access the required data; these are referred to as data driven tasks. On the other hand, tasks that require a relatively small amount of data are processed on demand. By assigning demand driven tasks to processors that attract few data driven tasks, the load is balanced.

As discussed in section 6.2, image space decomposition has been selected for this work, be-

cause it is a very common model in parallel processing and quite application–independent. It is expected that the results obtained with this work are of value beyond the scope of ray tracing.

7.6 PaRT – Parallel Ray Tracer

PaRT is a simple ray tracer used as a testbed for this work. It is not a state–of–the–art ray tracer, but it includes some of the features discussed on the previous sections.

PaRT's illumination model is the one discussed on section 7.2. Primary rays are shot through the pixels' corners and the four corners' contributions are averaged to obtain the pixel colour. This means that 513x513 primary rays must be fired to render a 512x512 image. Adaptive supersampling is supported to reduce the aliasing effect on the final image (section 7.3). A fifth ray is shot through the pixel's center if any two corners differ by more than a given threshold. This ray's value is averaged with the corners' rays using formula 7.14.

$$I_{pixel} = \frac{4 * I_{center} + I_{corner1} + I_{corner2} + I_{corner3} + I_{corner4}}{8} \tag{7.14}$$

Adaptive supersampling reduces aliasing artifacts by rising the sampling frequency, but it is computationally expensive, since more rays are spawned. This feature can be turned on or off through a command line switch.

PaRT includes some acceleration techniques to improve execution time over exhaustive ray tracing: adaptive depth control, occluding object buffer and non-uniform 3D space partitioning.

PaRT's input are scene descriptions written in Neutral File Format (NFF) [65]. A few extensions to NFF have been added to allow the specification of all the illumination model's parameters. PaRT's output is the rendered image in Targa format and a text file containing several statistics about the rendering process. It can also display, in run-time, the image being rendered, if the system where it runs has XWindows and Tcl/Tk installed. A detailed description of PaRT, version 2.1, can be found on PaRT's manual, included as appendix C.

PaRT is a parallel ray tracer, running over PVM 3.3. It can handle scene descriptions larger than each node's local memory, by including a mechanism that allows processes to address remote data items. It also spawns processes whose only function is to distribute the workload among the nodes. PaRT is, therefore, a suitable package to use as an experimental testbed for the current work.

7.6.1 PaRT's Architecture

PaRT runs over PVM 3.3 [58]. It starts in the root node and spawns processes on each of the Parallel Virtual Machine's nodes. Figure 7.7 depicts the processes' structure.

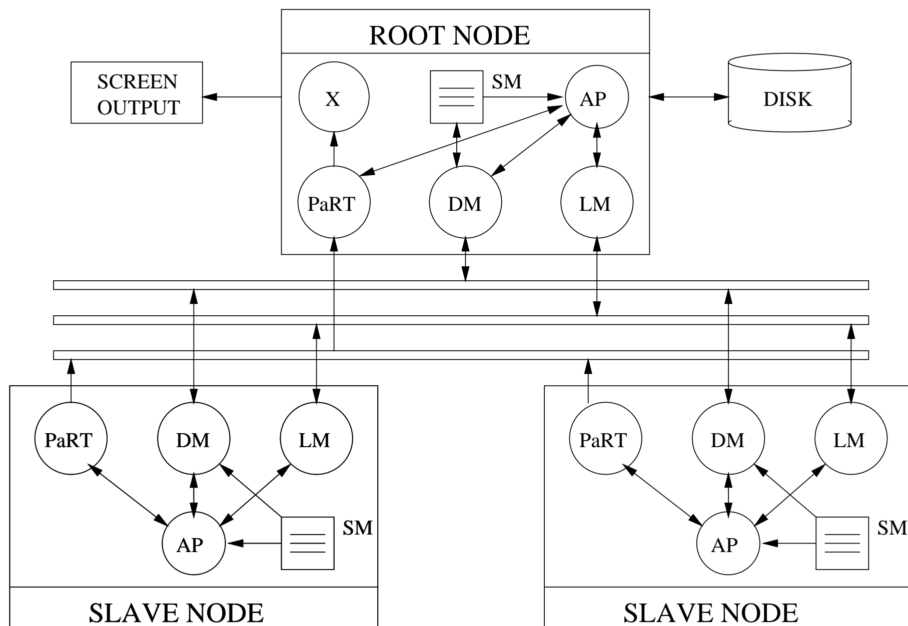


Figure 7.7: PaRT's architecture

PaRT 2.1 is composed of 5 different processes:

PaRT – this process plays different roles depending on whether it runs in the root node or in a slave node. In slave nodes it collects statistics and results from the local Application Process (AP) and sends them to the PaRT process running on the root. This one collects results and statistics from both its local AP and remote slave PaRT processes. It then displays the final statistics, saves the rendered image to disk and sends image data to the XWindows process if the appropriate command line switch has been used.

AP – Application Process – these processes perform all the calculations related to the ray tracing algorithm. All other processes perform auxiliary work, such as collecting results or supplying AP processes with tasks and data.

LM – Load Management – these processes perform work allocation and redistribution among the APs. The number of LM processes spawned and respective algorithm depend on the particular scheduling policy being used. APs execute a main loop where they wait for messages sent by the local LM. If it is a END–OF–WORK message, the AP process finishes. Otherwise, it must be a TASK message, containing the top left coordinates and dimensions (width and height) of the image's region to

be rendered. Therefore, PaRT only supports image space decomposition. When the AP finishes processing its work, it sends the results and statistics to the local PaRT process and requests more work from the local LM. This process can order task transfers by sending to the AP a SPLIT–TASK message containing the identification of the receiver and the percentage of the remaining work that must be sent to this receiver. This percentage determines the number of rows that will be sent. The AP keeps the rows with lower y coordinate, and sends those with larger y. It refuses to transfer work if it only has one row left to render. One row is, therefore, the finest grain of work that can be transferred. Tasks can thus be classified as modularly divisible.

X - XWindows display process – receives image data from the PaRT process running in the root and displays it on a X window. This process is not launched unless the appropriate command line switch is used.

DM – Data Management – this process supplies the local AP with data about the objects present in the scene being rendered. It is not launched unless the entire scene description does not fit on each node's local memory. If it does, then the whole data is replicated on each node. Otherwise, a different fraction of the scene's description is sent to each DM. It is stored in a memory segment (SM) shared by the DM and the AP. This fraction of the scene's description is referred to as the resident set. Whenever the AP needs to access a particular object's description, it first searches in the resident set for a local copy. This is efficiently done by using the object and node's identifiers. If that object's description does not belong to the resident set, then the AP searches it on a local cache also stored in the shared memory segment. The object's description could have been placed on this cache by the DM on a previous access to the same object. If a local copy can not be found, then the AP issues a data request to the DM, which will locate the object's owner using its identifier and an hashing function. A remote data request is sent to the appropriate DM, which will reply with the requested data. This will then be placed in the cache, and the AP, which was idle waiting for data, can resume its work. The partitioning of the entire data set in several resident sets is made sequentially, without using any estimates of the number of accesses each resident set will suffer. More sophisticated approaches could use statistics obtained from building the spatial subdivision to derive a cost distribution. This cost distribution can then be used to calculate a data distribution, such that voxels with higher cost are replicated over all nodes' resident sets [137, 138] (section 8.2). The DM processes implement, by software, a distributed shared memory with non–uniform memory access times (NUMA), since the time required to access a remote data item depends on that item's localisation, on the CPUs' current loads and on the communication network availability.

An obvious improvement to this architecture is to use a multithreading approach, where several AP's coexist at each node. While some processes are waiting for remote data, others can continue performing useful work. This form of multithreading allows overlapping of computational activities with data management tasks and can lower execution times. Multithreading, however, presents some drawbacks. The number of pending remote data requests increases with the number of threads, which in turn increases contention on the communication medium and on the data management processes, therefore increasing data access times. Furthermore, the multiple threads share the same data cache, but do not necessarily exhibit locality among their data requests, which can result in cache trashing. These overheads, associated with context switch costs, can result in a loss of performance above a certain number of Application Processes per node [145, 146]. Since the main goal of the current work is to study scheduling issues, not multithreading issues, this feature has not been included in PART.

Since the ray tracing algorithm does not modify the scene description, no coherency problems arise among multiple copies of the same data item, that can exist simultaneously across the various nodes. The Data Manager algorithm is, therefore, very simple.

7.7 Summary

This chapter introduced ray tracing and discussed some issues related to it. Ray tracing with scene descriptions larger than each node's local memory capacity will be used as a case study for the ideas presented throughout this thesis.

PaRT, a Parallel Ray Tracer, presented on section 7.6, was developed to serve as an experimental testbed for the current work. PaRT includes data management processes that enable processes to address remote data items. It also spawns processes whose only function is to distribute the workload among the nodes.

Chapter 8

Experimental Results

Contents

8.1 **Experimental Data Sets** **151**
8.2 **Estimating the Tasks' Requirements** **152**
8.3 **Performance Modelling** **159**
8.4 **Reference Scheduling Strategies** **163**
8.5 **Decision Network Based Strategy** **168**
8.6 **Results' Analysis** **189**
8.7 **Summary** **205**

This chapter presents the results obtained using a decision network as the scheduling agent's decision making mechanism. This scheduler's performance is compared with three reference scheduling strategies. Results are obtained using the ray tracer described in chapter 7 as a case study. Throughout these experiments only processing costs are explicitly taken into account by the scheduler's decision making mechanism; it makes no efforts to minimise communication overheads due to remote data accesses.

8.1 Experimental Data Sets

Four different scenes are used to test the ray tracer. These are taken from the Standard Procedural Database (SPD), developed by Eric Haines [65]. The SPD package is meant to act as a set of basic test scenes for ray tracing algorithms. Different ray tracers can be used with these scenes, and the respective results compared to assess both performance and geometrical correctness. The scenes used throughout this experiment are balls3, balls3c, balls4pv and teapot9 (figure 8.1). The main characteristics of each scene are described in table 8.1. They range from simple scenes (balls3 and balls3c), whose description entirely

fits into each node's local memory, to the more complex scene, balls4pv, whose size requires the distribution of data across the nodes' memories. The execution time distribution over the various images' regions depends on the observer's position and concentrates on those regions where objects' images are projected. Figure 8.2 illustrates the fact that although none of the experimental scenes exhibits an homogeneous distribution of workload across the entire image, balls4pv is the one with the most concentrated workload.

Name	Characteristics	Comments
balls3	**Dimensions:** 512x512 **Nbr. of Objects:** 821 **Memory required:** 341.3 KB **Nbr. of light sources:** 3	All objects fit into each node's local memory, whose capacity is 2 MBytes. The workload is mainly concentrated on the image's centre, corresponding to the region filled with the spheres.
balls3c	**Dimensions:** 512x512 **Nbr. of Objects:** 3280 **Memory required:** 1256 KB **Nbr. of light sources:** 3	All objects fit into each node's local memory. The workload is mainly concentrated on the image's top and to the left. It diminishes as y increases.
balls4pv	**Dimensions:** 256x256 **Nbr. of Objects:** 7382 **Memory required:** 2826 KB **Nbr. of light sources:** 3	Only 55% of the objects belong to each processor resident set. The remaining ones must be fetched from other nodes' memories. The workload is concentrated on the image's top left corner.
teapot9	**Dimensions:** 512x512 **Nbr. of Objects:** 5193 **Memory required:** 1988 KB **Nbr. of light sources:** 3	79% of the objects belong to each processor resident set. The workload is concentrated on the region occupied by the teapot and the base polygons.

Table 8.1: Scenes' main characteristics

8.2 Estimating the Tasks' Requirements

The static uniform distribution and the demand–driven schedulers do not use any internal execution model on its decision making mechanisms. On the other hand, the dynamic sensor based schedulers use an execution model both to predict the tasks' execution times and for decision making. They use the same information about the environment's state and have the same set of possible actions. This model only takes into account the tasks' estimated time to finish, the nodes' computing throughput and the time required to transfer tasks among nodes. Data access overheads are completely ignored, therefore no attempt is made to profit from, or improve, data locality. This option results from two facts:

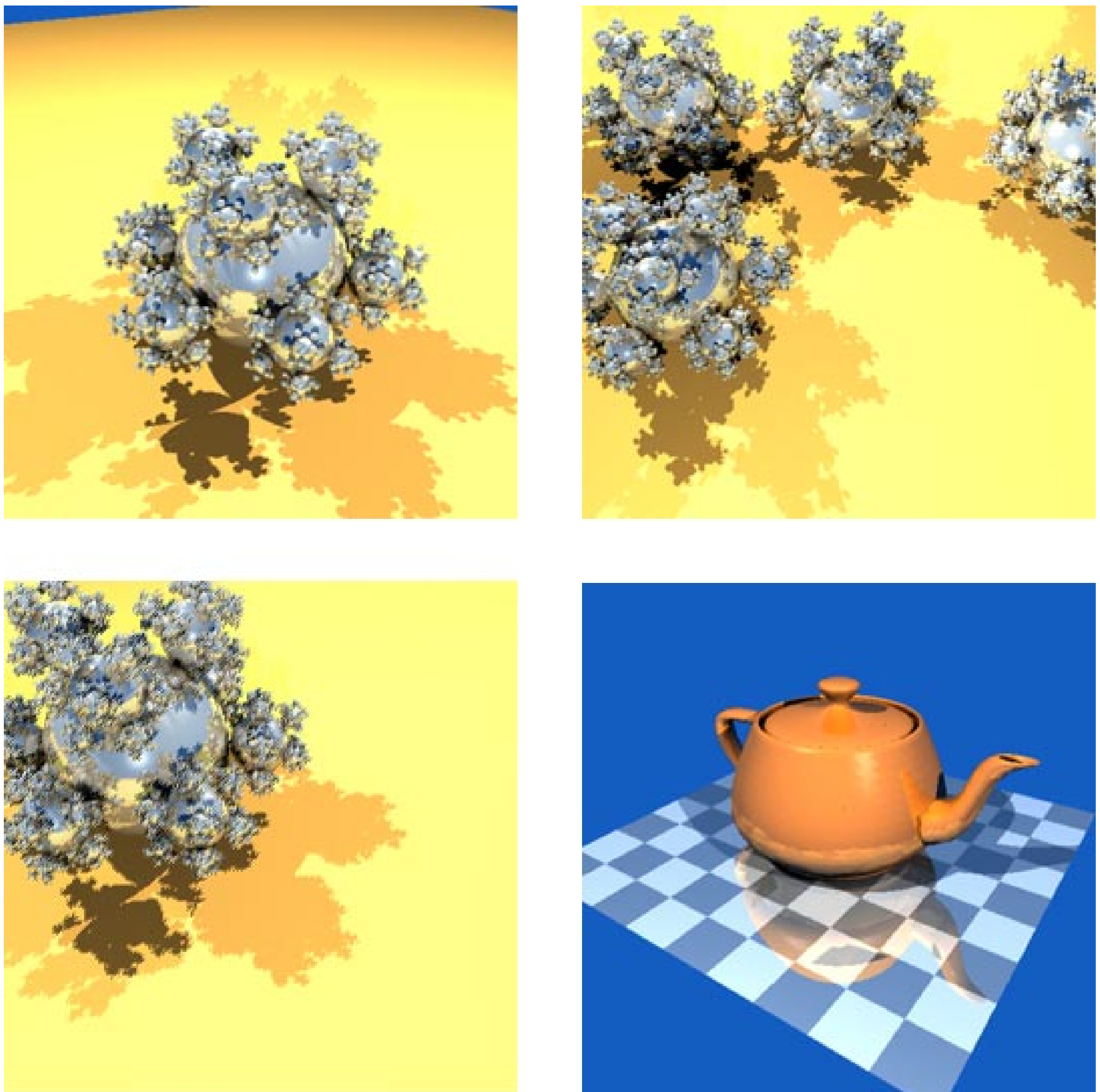


Figure 8.1: SPD scenes – balls3, balls3c, balls4pv, teapot9

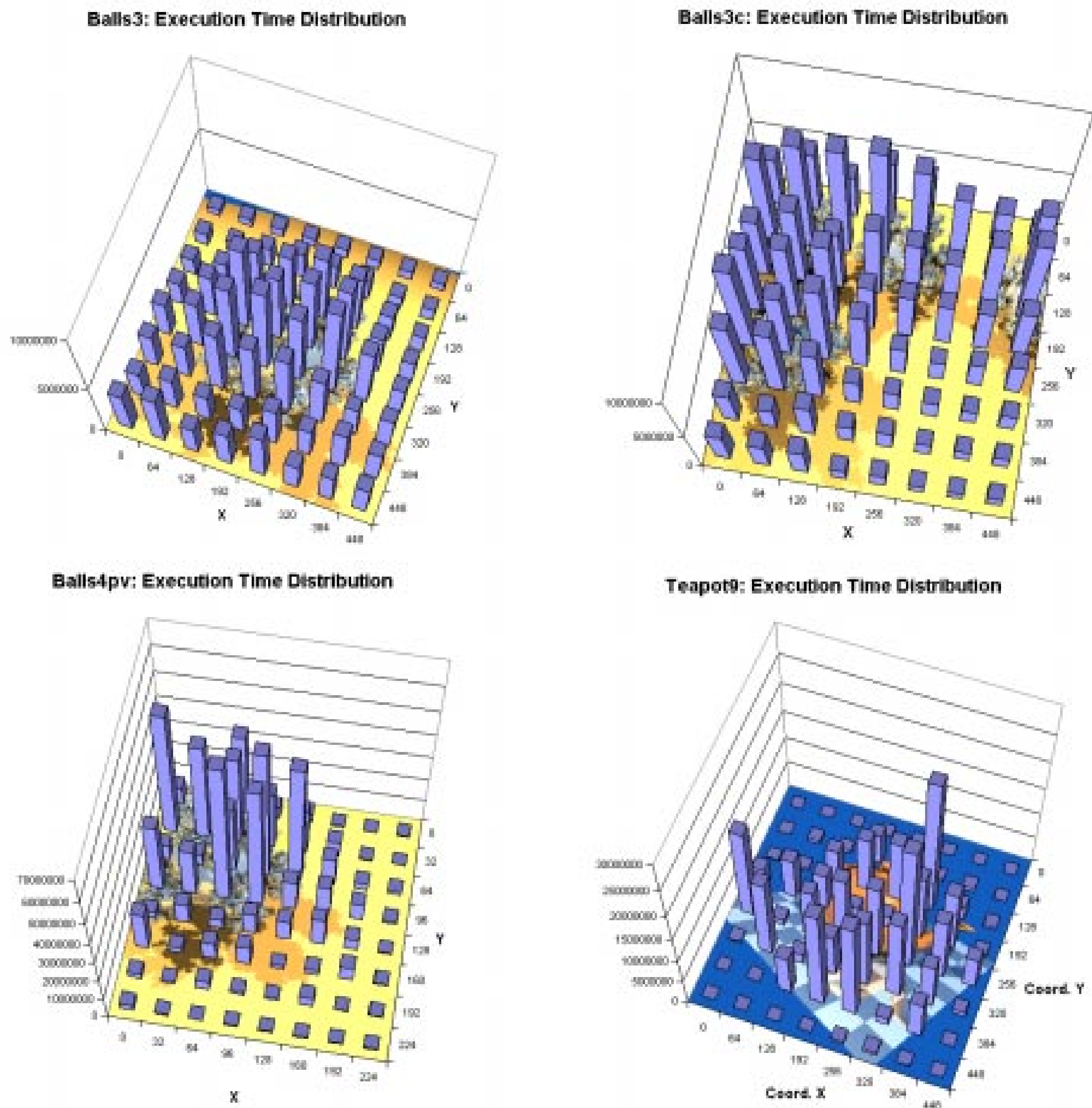


Figure 8.2: Execution time distribution – balls3, balls3c, balls4pv, teapot9

ray tracing is an application exhibiting low interactions among nodes (section 6.2) and the distributed system has a low latency/high bandwidth communication network. The results presented throughout this chapter will show the relevancy of including these overheads on the scheduler's execution model. The main difference between these two schedulers is that the former, as its name suggests, models the environment using deterministic quantities, while the latter uses a stochastic model.

The schedulers have a centralised architecture. The scheduler running in the root node receives messages from all nodes about system's state and tasks' workload profiles, and decides which actions to take based on this information and on its internal execution model. These messages include the following values:

- the task's top left corner coordinates — $x0, y0$;
- the task's dimensions — width (w) and height (h);
- the number of rows that have been processed up to this moment — h_p;
- the time elapsed since this task's execution began — $T_{elapsed}$;
- the time the node spent idle, waiting for the load manager to send the current task — $Tidle$;
- the time spent waiting for remote data necessary to render the current task — $Tdata$;
- the node's computing throughput, as perceived by the application process (AP), expressed on intersections per second; this metric is referred to as the Intersection Rate — Ir.

This information includes foreground workload metrics ($x0, y0, w, h, T_{elapsed}, h_p$), resource's capacity metrics (Ir) and indirect scheduling overhead metrics ($Tidle, Tdata$). Messages are sent by the application processes under certain conditions, which depend on the particular scheduling strategy being used.

Upon reception of an information message, the scheduler computes an estimate of the time required for that task to finish, T_m. This is used to quantify each node's current foreground workload, and is a function of the currently assigned load, $T_m = f(T_{elapsed}, W_\%)$. $W_\%$ represents the percentage of the task that has been completed up to the current moment. These are given by

$$\begin{aligned} W_\% &= \frac{h_p}{h} \\ T_m &= \frac{T_{elapsed}}{W_\%} * (1 - W_\%) \end{aligned} \tag{8.1}$$

This component of the execution model is identical for the two schedulers. It has some drawbacks:

- the time spent waiting for remote data is not explicitly taken into account; the scheduling agent completely ignores this issue, so it may fail to generate accurate predictions and does not try to reduce this overhead by exploiting data locality;
- the time required to finish each task is computed as if the tasks' requirements were constant, i.e., the time necessary to render each row is always the same; this is not true for several reasons: the time required to render each pixel strongly depends on the objects being intersected, hence varying across the image [61] (figure 8.2), and the resources' capacities may change at any moment due to fluctuations on the background workload.

Figure 8.3 shows the estimated versus actual T_m for one region of the balls3c scene. It clearly shows that equation 8.1 does not predict T_m with great accuracy. To reduce these errors there are several alternatives: the execution model may be improved to account for the dynamics of the tasks' requirements, or the scheduler might have the possibility of revising old decisions and appropriately redistribute the workload over the nodes to respond to changes in tasks' T_m predictions. This is the approach taken with these schedulers.

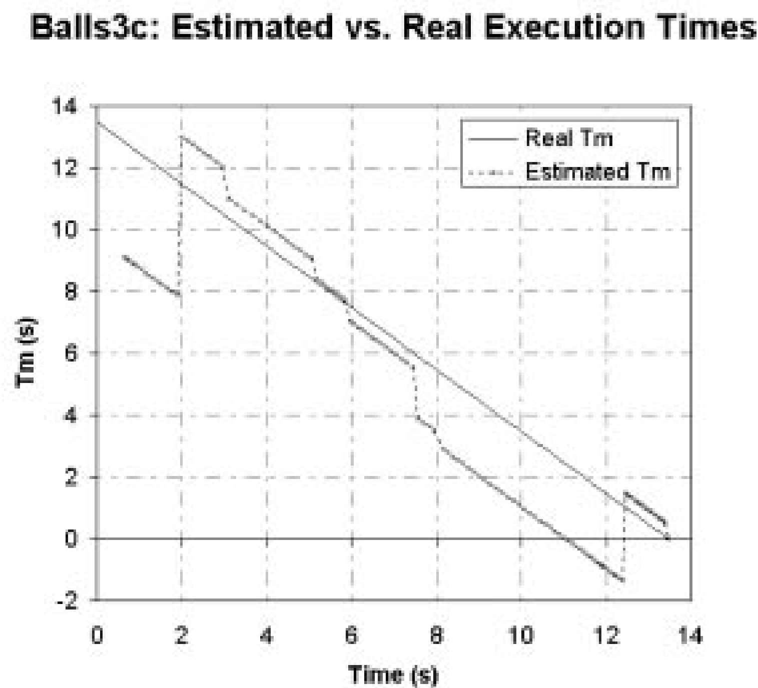


Figure 8.3: Balls3c: Estimated versus actual execution time

More complex execution models could have been used, which would generate more accurate predictions of tasks' execution times [161, 162]. An obvious approach is to render a low

resolution image of the scene, to obtain a reasonable estimate of the total cost of ray tracing a larger image. Other alternatives include estimate an average cost per ray on basis of the geometrical properties of the scene using statistics gathered during the space partitioning phase [138], and estimate the number of rays using geometrical and surface properties and the position of the light sources [137]. However, these models are very application–dependent. Since the main purpose of this work is to study scheduling issues, not ray tracing ones, the simpler execution model was preferred.

8.3 Performance Modelling

This section details the metrics used by the scheduling agents throughout this experiment. These are the same metrics presented in section 6.4, but described with the particular details imposed by the scheduler's execution model. They include performance, environment and scheduling overhead metrics.

8.3.1 Performance Metrics

This application level scheduler's performance goal is to reduce the application's execution time; therefore T_{exec} is the metric used to assess the scheduler's performance. It is measured as the time elapsed from the beginning of the rendering process until the complete image is stored in the root node's central memory. The time spent with the space partitioning process, performed before the rendering process, is not included, since it has not been optimised and the scheduler does not act upon this process. In order to reduce errors measuring T_{exec}, all application processes begin rendering at exactly the same time, once space partitioning is completed, by performing a global synchronisation operation.

8.3.2 Environment metrics

These metrics are used to update the image the scheduler has about the environment's state. They are subdivided into two subgroups: foreground workload metrics and resources' capacity metrics.

Foreground workload metrics

These metrics measure each resource current foreground workload. The pair $(T_{elapsed}, W_{\%})$ is used to estimate the time required to finish the task currently assigned to each processor, T_m, using equation 8.1.

Resources' capacity metrics

Both sensor based scheduling agents decide the amount of work to transfer between two nodes — the sender and the receiver — based on the nodes' computing throughput, as perceived by the application processes. On a ray tracer, the most adequate metric to measure the computing throughput is the intersection rate, (Ir), expressed on intersections per second, since it spends most of its time performing intersections between rays and objects.

This metric could be computed by counting the number of intersections each node has performed and dividing it by the time required to perform them. However, this approach revealed itself very sensible to the characteristics of the image region being rendered. In fact, the time necessary to perform an intersection with an object depends on the object's type. Intersecting a straight line with a sphere is much faster than with a polygon. Furthermore, if an image's region requires fewer intersections than other, the measured intersection rate will be much lower since the processor will spend proportionally more time managing auxiliary data structures. Also, this approach does not allow the intersection rate's measurement when the node has no work allocated.

Since the intersection rate is not taken into account to compute T_m, its absolute value is not important. What really matters is the two nodes' relative intersection rates. The approach taken is to perform the intersection of a straight line with an object built specifically for this purpose. For the *deterministic* and decision network based schedulers, all nodes measure its current intersection rate in intervals of 7.5 seconds. These are, therefore, time–driven sensors. Measurements have shown that, using this strategy, between 0.8% and 1.1% of the application's execution time is spent computing this metric. This is an acceptable intrusion level, since the benefits of using this metric outweigh the overheads of measuring it.

The value reported by each intersection rate sensor to the scheduling agent is the average of all the intersection rate measurements done during the application execution. For any generic quantity Q, the arithmetic average of $k+1$ measurements, denoted by $\overline{Q_{k+1}}$, is given by

$$\overline{Q_{k+1}} = \frac{1}{k+1}\sum_{i=1}^{k+1} r_i \qquad (8.2)$$

$$= \overline{Q_k} + \frac{1}{k+1}[r_{k+1} - \overline{Q_k}] \qquad (8.3)$$

where r_i is the ith measurement and $Q_1 = r_1$. Expression 8.3 enhances the fact that $\overline{Q}$ is updated each time a new reading r_{k+1} is received. This expression has the general form

$$NewEstimate \leftarrow OldEstimate + \alpha * [NewReading - OldEstimate]$$

and all measurements are given the same weigh $\frac{1}{k+1}$, which is more clearly seen in equation

8.2. This method of averaging is adequate when the quantity being measured is stationary, i.e., it tends to float around some particular value. If that quantity is not stationary, such as the nodes' intersection rates, then it is more adequate to weigh recent readings heavier than long–past ones. This can be done by using a constant α, $0 <= \alpha <= 1$. Each reading r_i is now given the weigh $\alpha(1-\alpha)^{k-i}$, which depends on how many observations ago, $k-i$, it was observed. In fact, the weigh decreases exponentially according to the exponent of $(1-\alpha)$. This average is referred to as an *exponential recency–weighed average* [170]. Table 8.2 compares the behaviour of the arithmetic average with exponentially weighed averages with different α.

k	r_k	$\overline{Q_k}$ (arithmetic)	$\overline{Q_k}(\alpha = 0.1)$	$\overline{Q_k}(\alpha = 0.5)$	$\overline{Q_k}(\alpha = 0.9)$
1	5	5.00	5.00	5.00	5.00
2	4	4.50	4.90	4.50	4.10
3	6	5.00	5.01	5.25	5.81
4	7	5.50	5.21	6.13	6.88
5	6	5.60	5.29	6.06	6.09
6	1	4.83	4.86	3.53	1.51
7	6	5.00	4.97	4.77	5.55
8	3	4.75	4.78	3.88	3.26
9	2	4.44	4.50	2.94	2.13
10	3	4.30	4.35	2.97	2.91

Table 8.2: Comparison of arithmetic and exponentially weighed averages with different α

It clearly shows that with $\alpha = 0.1$ the average reacts very slowly to changes in the quantity's tendency, as for $k = 8$, where r_k starts to float around 3. On the other hand, with $\alpha = 0.9$, recent readings are given an heavy weigh, therefore the average reacts very quickly to changes in the readings. It fails to filter out atypical values, like for $r_6 = 1$. Using $\alpha = 0.5$ seems to be a good compromise, since it reacts quickly to changes in the readings tendency, and reasonably filters out atypical values. Therefore, the intersection rate readings are exponentially recency–weighed with $\alpha = 0.5$.

This procedure revealed itself as being able to detect changes on the processor's workload. The intersection rate's measured value diminishes when additional applications are launched on the same processor, i.e., when the background workload increases, as depicted by figure 6.3.

8.3.3 Scheduling Overhead Metrics

These metrics are used to quantify the direct and indirect overheads the scheduler imposes upon the system.

Direct costs

Direct costs depend directly on the scheduler's activity. Hence, they can be measured by counting how many times the scheduler's mechanisms responsible for these overheads are activated and by measuring the average cost per activation.

#T – number of tasks that have been processed. A number of tasks much larger than the number of nodes indicates that the percentage of remote execution is high, i.e., the scheduler had to intervene frequently in the environment. This may be a clue to instability.

T_{sched} – time required to transfer a task between two nodes. It is measured as the time elapsed since the scheduler initiates the migration mechanism, until an acknowledgement is received from the target node. It is used by the scheduling agent to decide if the execution time potentially gained with a given task transfer outweighs the direct cost of actually transferring that task. The product of this metric by the number of tasks can be used to quantify the total task migration direct overhead.

#TS – number of information messages sent to the central scheduling agent by the application processes.

A general classification of direct costs is presented in table 3.1. The metrics **#T** and T_{sched} are related to the cost of executing the actions selected by the scheduler, while **#TS** is related to the cost of sending information messages from the processing nodes to the central scheduling agent.

Indirect costs

Indirect costs are consequences of the scheduler's actions, and are related to the quality of its decisions. Three different indirect costs are measured:

work replication penalty (Pen%) – since each pixel's value is computed as the average of its four corners, splitting an image's region into various subregions requires that boundaries are computed by both nodes that have assigned contiguous regions. This results on additional primary rays being spawned. The work replication penalty, ($Pen\%$), is measured as the percentage of additional primary rays that must be fired, compared to an uniform screen space decomposition over the same number of nodes. With an uniform screen decomposition, each node gets a vertical strip of the image.

$$Pen\% = \frac{\#PrimRays_{sched} - \#PrimRays_{unif}}{\#PrimRays_{unif}} * 100$$

resources' idle times (TTidle, TTidle%, StdDev) – $Tidle_i$ is the time node i spent waiting for tasks, including the time spent waiting for the application to finish, when

node i already performed its last task. All nodes terminate at the same time, even though some of them could have no work allocated for some time. $TTidle$ is the sum of idle times for all nodes

$$TTidle = \sum_{i=1}^{n} Tidle_i$$

$TTidle\%$ is the percentage of idle time for all n nodes, compared with the aggregated execution time

$$TTidle\% = \frac{TTidle}{n * T_{exec}} * 100$$

$StdDev$ measures the standard deviation of the nodes' busy times. If the standard deviation is 0, then it means that all processors had work allocated for identical time periods. This does not mean, however, that all nodes had work during the whole execution time, since they may have been idle for identical time periods. If a given execution of an application presents lower $TTidle\%$ and larger $StdDev$ than another execution of the same application, this means that the processing nodes had work allocated for a larger fraction of the execution time, but that idle times were not uniformly distributed by all processing nodes.

remote data access times (TTdata, TTdata%) – $Tdata_i$ is the time node i spent waiting for remote data items, instead of performing useful work. $TTdata$ is the sum of $Tdata_i$ for all nodes

$$TTdata = \sum_{i=1}^{n} Tdata_i$$

$TTdata\%$ is the percentage of remote data access time for all n nodes, compared with the aggregated execution time

$$TTdata\% = \frac{TTdata}{n * T_{exec}} * 100$$

Both the deterministic and the stochastic scheduler use T_{sched} as a gauge to decide whether or not a given task transfer is profitable. The remaining overhead metrics are not used in run time. They are, however, used throughout this chapter to analyse the scheduler's efficiency.

8.4 Reference Scheduling Strategies

To assess the decision network approach's effectiveness, its results are compared with those of three other scheduling strategies: a static uniform data distribution, a demand–driven approach and a sensor based deterministic dynamic scheduling strategy, referred to as *det*, that uses information about the current system's state and workload profile to make predictions about future behaviour, in an attempt to improve its effectiveness.

These scheduling strategies are briefly described in section 6.5, and their classification according to the ESR classification scheme is also presented. The static uniform work distribution and the demand–driven approaches are described with enough detail, thus no further descriptions are required. The deterministic scheduling strategy, on the other hand, depends on the particular execution model being used, therefore it is described with greater detail throughout this section.

Setting the Constants

Some of these strategies require the specification of a few constants which regulate the scheduler's behaviour. These constants are set empirically, using common sense to select the most adequate values. A more rigorous alternative would be to run the scheduler with several different combinations of these constants' values and select the one which maximises the scheduler's effectiveness. Throughout this work the empiric approach was preferred, keeping in mind, however, that their values must be carefully selected. The reasoning that led to each constant value is presented, whenever it is introduced.

8.4.1 Sensor Based Deterministic Strategy

This scheduling strategy uses information about the current system's state and workload profile to make predictions about near future behaviour, in an attempt to improve effectiveness. It can therefore be classified as a dynamic non–adaptive strategy. Initially, the image is divided into as many vertical strips as the system's nodes, and each of these strips is sent to a different node. This behaviour is identical to that of uniform distribution. The scheduler spends no effort trying to generate an optimised initial workload distribution. It focus on run time task migration, since the tasks' requirements are unknown before execution time and the resources' capacities may exhibit high variability. Application processes are now allowed to send information messages to inform the central scheduler of both that node's capacity and the tasks' requirements (section 8.2). Using this information, the scheduler classifies the nodes as either potential work suppliers or receivers. It then computes the amount of work, if any, to be transferred among these nodes.

The time elapsed since a task migration is decided until it is effectively completed, is used as a gauge to decide whether or not a given node can be classified either as a potential supplier, or a potential receiver, and to determine the expected profitability of a potential task migration. This metric is referred to as T_{sched}, and is dynamically recalculated at runtime. This is a direct cost, thoroughly described in section 8.3.3.

Information Policy

Whenever an application process finishes rendering one row of its allocated task, it consults a set of rules to decide whether it should send an information message to the scheduler.

These rules are defined as follows:

- if the elapsed time since this node's intersection rate last measurement is longer than a pre–defined value (7.5 seconds, as justified in section 8.3.2), then measure it again;
- if no information message about the current task has been sent yet, then send one;
- if the current intersection rate changed more than a pre–defined value relatively to the last information sent, then send one information message; some different pre–defined values were tried and the scheduler revealed itself quite insensitive to this parameter; therefore, this value is set to 15% throughout this experiment.

An information message is also sent upon each task's completion. According to these rules the information policy can be classified as state–change driven (section 3.6), since although the intersection rate is measured periodically, the scheduling agent is informed only if it changes significantly.

Whenever the scheduling agent receives an information message, it computes the task's expected time to terminate, T_m (section 8.2), and updates all other tasks' T_ms. To be able to update other tasks' T_ms, a time stamp is stored whenever an information message is received. T_m is updated by subtracting the time elapsed since the last message was received from that node.

$$\begin{aligned} T_{message} &= TimeNow - TimeStamp \\ T_m &= OldT_m - T_{message} \end{aligned}$$

Every information message carries the number of lines that have been processed up to this moment, h_p. When T_m is updated, h_p must also be updated. This updated value is referred to as h_{pcurr}

$$h_{pcurr} = \frac{T_{message}}{T_m} * (h - h_p) + h_p$$

where T_m and h_p are the values effectively received with that node's last message, and not the updated values.

Transfer Policy

Upon reception of an information message, and after updating the information available about the system's state, the scheduler tries to identify potential work suppliers and receivers. The transfer policy used is threshold–based (section 3.6); the thresholds depend on T_{sched}, the average time required to transfer a task among two nodes.

A node is classified as a potential receiver if it is not currently enrolled on any task transfer, it has no work allocated or if it has a small estimated T_m.

A node is classified as a potential supplier if it is not currently enrolled on any task transfer, if it still has more than 2 lines to process and if it has a large estimated T_m.

The value of T_m is used by the decision agent according to the following rules:

$$\begin{aligned} T_m &< K_{recv} * T_{sched} \Rightarrow \text{potential receiver} \\ T_m &> K_{supp} * T_{sched} \Rightarrow \text{potential supplier} \end{aligned}$$

The values of K_{recv} and K_{supp} were set to as 2 and 10, respectively. The value 2 allows the selection as receivers of nodes which are probably about to finish their currently allocated task, while 10 assures that the potential supplier has enough work allocated to justify considering it for a task transfer, and that this transfer does compensate the direct overheads of task migration (T_m must be an order of magnitude larger than T_{sched}).

Potential receivers are then sorted. Inactive nodes come first, sorted by descending order of computing throughput (Ir), followed by nodes with allocated work, sorted by ascending order of T_m. Potential suppliers are sorted by descending order of T_m.

Location and Selection Policies

After identifying candidates for potential tasks' migrations, the scheduler tries, for each receiver, to find the most adequate supplier and the appropriate amount of work to transfer. It computes the number of lines to transfer from the supplier to the receiver, h_t, so that the amount of work on the supplier, NI_s, measured on number of intersections still to perform, is equivalent to the amount of work on the receiver, NI_r, given their relative intersection rates, $\frac{Ir_r}{Ir_s}$. Therefore,

$$\frac{NI_s}{NI_r} = \frac{Ir_s}{Ir_r} \Leftrightarrow NI_r = \frac{Ir_r}{Ir_s} * NI_s \tag{8.4}$$

The number of predicted intersections in both the receiver and the supplier, are given by

$$\begin{aligned} NI_r &= \frac{NI}{h_p} * (h_t + 1) \\ NI_s &= \frac{NI}{h_p} * (h_{ms} - h_t) \end{aligned}$$

where NI is the number of intersections performed by the supplier during this task, h_p is the number of lines that required NI intersections, and $h_{ms} = h - h_{pcurr}$ is the number of lines still to process for this task. To compute NI_r, one unit is added to h_t to account for work replication, required by the fact that the pixel's values are computed by averaging its four corners. Therefore, replacing in equation 8.4,

$$\frac{NI}{h_p} * (h_t + 1) = \frac{Ir_r}{Ir_s} * \frac{NI}{h_p} * (h_{ms} - h_t)$$

and, trivially,

$$h_t = \frac{\frac{Ir_r}{Ir_s} * h_{ms} - 1}{\frac{Ir_r}{Ir_s} + 1} \tag{8.5}$$

The computation of the appropriate amount of work to transfer between a given pair of nodes, h_t, constitutes this scheduling strategy's selection policy.

After obtaining h_t, the new estimated execution times on both the supplier, T'_{ms}, and the receiver, T'_{mr}, are computed, using equations 8.6 and 8.7,

$$T'_{ms} = \frac{T_{elapsed}}{h_p} * (h_{ms} - h_t) \tag{8.6}$$

$$T'_{mr} = \frac{Ir_s}{Ir_r} * \frac{T_{elapsed}}{h_p} * (h_t + 1) \tag{8.7}$$

The supplier which presents the maximum gain in execution time, T_{gain}, according to equation 6.7 (repeated below), is selected for this task transfer.

$$T_{gain} = \max(T_{ms}, T_{mr}) - \max(T'_{ms}, T'_{mr})$$

where T_{ms} and T_{mr} are the estimated execution times in the supplier and the receiver if the work migration does not take place. However, this supplier is accepted only if this gain is larger than T_{sched} multiplied by a pre–defined constant, K_{accept}, which is set to 6 throughout all experiments. Its purpose is to assure that only profitable enough migrations, compared to the cost of migrating tasks among nodes, are performed. Solving equation 6.7, by means of which the most suitable supplier for the current receiver is found, constitutes this scheduling strategy's location policy.

After selecting a supplier, the scheduler initiates the task migration mechanism. If, for any reason, the supplier refuses to transfer work, then the next supplier is selected. Upon reception of an acknowledgement message from the supplier, both the supplier and the receiver ids are removed from their respective sorted lists, and the scheduler proceeds with the next potential receiver. This process is repeated until one of the candidates' lists becomes empty.

Migration Mechanism

After selecting the pair of nodes to enroll on the task transfer, and the amount of work to migrate, h_t, the scheduler notifies the supplier, indicating the percentage of the remaining work that it must transfer, $W_{\%} = \frac{h_t}{h_{ms}} * 100$, and to which node.

Upon reception of a task migration message, the supplier computes the number of rows to transfer, given its current available work. If the number of lines still to process is less than 2, then the supplier refuses this operation. Otherwise, it sends to the receiver the coordinates of the new task, and reduces its own task to account for the transferred work.

Upon reception of a task transfer, the receiver begins processing its newly allocated work, after notifying the central scheduler that the task was received.

Extended Information Policy

Previous results obtained using the above described scheduling strategy, revealed some deficiencies on the execution model. Due to the dynamics of the tasks' requirements, the tasks' predicted execution times, T_m, may present very significative errors (figure 8.3). These errors' worst consequence is that, in some cases, the scheduling agent may estimate that a given node is just finishing its task, when actually it will take longer to finish. This happens when the scheduler has estimated an execution time, T_m, equal to 0 for that task, but the task did not finish yet and the respective application process does not send new information messages. This node is not selected as a potential supplier and an opportunity for better distributing the workload is missed.

To overcome this deficiency, the information policy has been extended, by allowing the scheduling agent to directly ask for updated information, whenever a node should have already terminated its task, but no further information was received. Hence, it can be classified as an hybrid state–change and demand driven policy. This extended approach has two main advantages:

- potential suppliers are more easily identifiable, due to more accurate information;
- the scheduling agent is no longer restricted to redistribute the workload only when it receives an information message; it can stop waiting for these messages at any instant, send an information request and enter a redistribution step.

Figure 8.4 shows the improvements obtained with the extended information policy, labelled *det*, compared with the previous information policy (labelled *det1*) for teapot9. Unless explicitly stated, all results presented through the remainder of this chapter and labelled as *det* were obtained using this extended information policy.

8.5 Decision Network Based Strategy

While the *deterministic* scheduling strategy uses deterministic quantities to make decisions about the suitability of a given task migration, the decision network based strategy (DN) models some parts of the environment as random variables; the interactions among these variables are also stochastically modelled, by means of conditional probability tables.

This strategy's information and transfer policies are identical to those of the sensor based deterministic strategy; the selection and location policies are implemented mainly by evaluating the decision network. This is evaluated for each pair of nodes, one identified by the transfer policy as a potential receiver of work and the other as a potential work supplier (section 6.5.4). The decision network recommends the most adequate action for each of

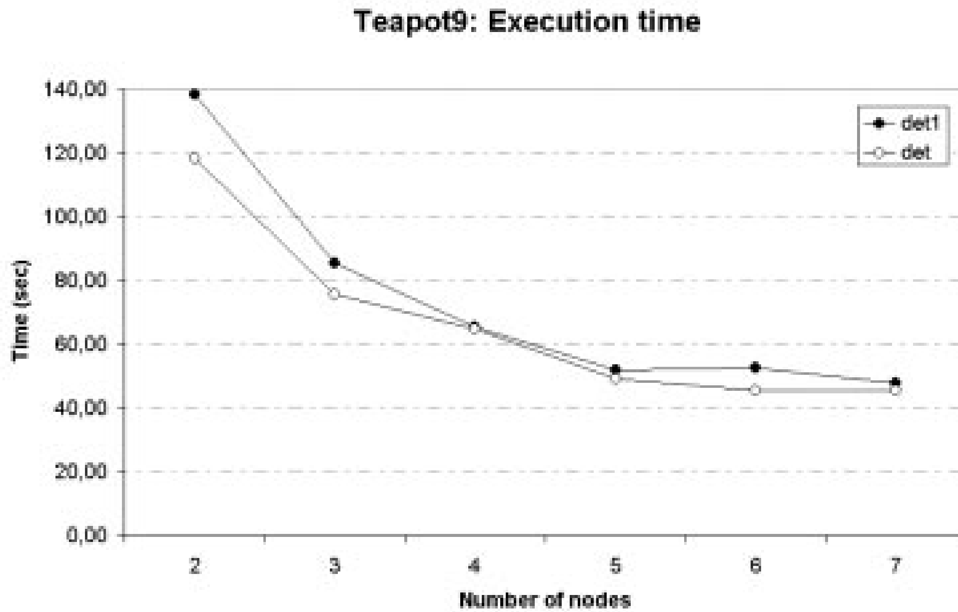


Figure 8.4: Teapot9: improvements with extended information policy

these pairs, expressed in terms of the amount of work to transfer between them and in which direction.

The list of recommended actions is sorted by descending order of gain in execution times, T_{gain}, given by equation 6.7. Some of these actions may be discarded by the scheduler if its estimated gain, T_{gain}, is less than $K_{accept} * T_{sched}$, where K_{accept} is a pre–defined constant set to 6, as previously discussed. The scheduler will then try to execute these actions, assuring that no node is involved in more than one action per scheduling step.

The next sections describe the knowledge engineering process followed to build the decision network used throughout this experiment.

8.5.1 Laying out the Network's Topology

This step entails identifying those aspects of the problem that must be modelled and what are the direct causal relationships holding among them. Once built, the network's topology constitutes a knowledge base that represents the model's qualitative characteristics. These are preserved independently of the assignments of quantitative information. No effort is made through this section to describe or quantify how the various variables are influenced by each others. These causal relationships are just identified and represented on the network by means of direct arcs between causes and respective effects. The strengths of these relationships, quantified by the conditional probability tables, are specified in the next section.

The decision network proposed throughout this section follows the generic structure proposed in section 5.7, with some modifications:

- the selected case study involves an application with irregular workload running on a shared system, as illustrated in figure 8.5; in these cases, the scheduling agent will be uncertain about both the tasks' requirements and the resources' capacity; to properly handle these cases, a decision network must include two blocks to describe the environment's state: the tasks' requirements block and the resources' capacity block (section 5.7.1);

- the scheduling overheads are considered irrelevant for the decision making process, i.e., it is assumed that they are not large enough to compromise the scheduler's effectiveness, hence, this block is not included in the network; the experimental results will show the correctness of this assumption;

- the sensors' model includes the age of the information; the older the information, the more uncertain the agent is about its accuracy;

- the environment's next state is described by a single stochastic variable, that represents the expected degree of load sharing between the two nodes being considered.

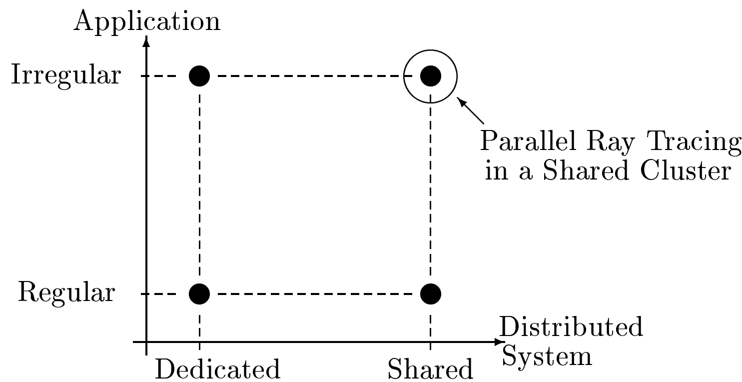


Figure 8.5: Ray tracing: the system sharing level versus the application regularity space

Throughout this model two metrics are used to describe each node's state: the intersection rate and the node's current foreground workload. This latter quantity is measured in terms of the estimated time required to finish the current task – T_m – and is given by equation 8.1. Each of these metrics can be seen as acquired through the scheduling agent's sensors. To account for the possibility that these sensors can return noisy information, a model has to be built for each sensor, that tries to capture the uncertainty associated with the various readings.

The Resources' Capacity Block

Figure 8.6 presents the Bayesian network used to model the intersection rate sensor. The actual current intersection rate is modelled by node **Ir**. However, this value can not be directly measured. It is known only through node **InfoIr**, which represents the sensor's readings. The sensor works exactly the same way as described in section 8.3.2; imperfect information about the desired quantity is only obtainable through perfect information about the sensor. There is no uncertainty on the value of **InfoIr**, its value is completely known just by reading the sensor. The uncertainty is about the value of **Ir**, and is modelled by the conditional probability table $\mathbf{CPT}(InfoIr|Ir)$. The notion of causality is crucial: it is the actual intersection rate that determines the sensor's reading. $\mathbf{CPT}(InfoIr|Ir)$ represents the probability that the sensor will return a certain value, given that the actual intersection rate takes a particular value. Inference will then be performed the other way around: given the sensor's reading, which is the actual intersection rate?

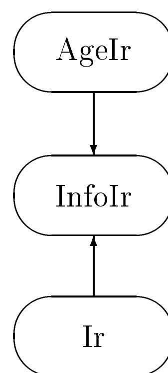


Figure 8.6: Bayesian network – the intersection rate sensor model

Since the sensor also consumes resources, it can not be constantly activated to return updated readings. The variable **AgeIr** models the uncertainty that arises from the fact that the information returned by the sensor may be outdated. The older this information, the less precise it is. This uncertainty is modelled by the conditional probability table $\mathbf{CPT}(InfoIr|AgeIr, Ir)$.

One of the most fundamental properties of a variable is the set of values it can take on, i.e., its domain. Most algorithms for probabilistic inference are designed for discrete variables, and this is also true for Pearl's algorithm, presented in section 5.4.4 and appendix B. To use this algorithm, the decision network must include only discrete variables, even if this requires that conceptually continuous quantities are discretised for the purposes of reasoning. Experience has shown that, in most cases, continuous quantities can be discretised without loosing effectiveness. The discrete domain's cardinality, however, must be carefully selected. Large domains allow a more accurate discretisation of the continuous

quantity, but require the specification of a huge number of probabilities and the inference process becomes heavier. Throughout this experiment, continuous variables are discretised by three to five points. Sensitivity analysis to this aspect could be performed to determine its relevancy.

Let D_X represent the finite domain of a random discrete variable X, and $\#D_X$ represent the domain's cardinality.

D_{AgeIr} = {Current, Recent, OutDated}

The age of a node's intersection rate information is classified as a function of the time elapsed since that node's last information message arrival, $T_{message}$, according to the following rule:

$$\text{AgeIr} = \begin{cases} \text{Current} & \Leftarrow \quad T_{message} <= 10 * T_{Sched} \\ \text{Recent} & \Leftarrow \quad 10 * T_{Sched} < T_{message} <= 30 * T_{Sched} \\ \text{OutDated} & \Leftarrow \quad T_{message} > 30 * T_{Sched} \end{cases} \tag{8.8}$$

D_{InfoIr} = {VeryLow, Low, Medium, High, VeryHigh}

This variable's value is computed as a function of the intersection rate measured for each particular node, Ir_i, and the average intersection rate over all n nodes, $\overline{Ir} = \frac{\sum_{j=1}^{n} Ir_j}{n}$, according to rule 8.9. The decision network does not take into account the nodes' absolute intersection rates, but their relative values.

$$\text{InfoIr} = \begin{cases} \text{VeryLow} & \Leftarrow \quad Ir_i <= 0.7 * \overline{Ir} \\ \text{Low} & \Leftarrow \quad 0.7 * \overline{Ir} < Ir_i <= 0.9 * \overline{Ir} \\ \text{Medium} & \Leftarrow \quad 0.9 * \overline{Ir} < Ir_i <= 1.1 * \overline{Ir} \\ \text{High} & \Leftarrow \quad 1.1 * \overline{Ir} < Ir_i <= 1.3 * \overline{Ir} \\ \text{VeryHigh} & \Leftarrow \quad Ir_i > 1.3 * \overline{Ir} \end{cases} \tag{8.9}$$

Ir has exactly the same domain as **InfoIr**. As previously stated, there is never evidence available about this variable's exact value. Therefore, its value is not directly set. The belief distribution over this variable is computed by the inference engine, based on evidence about **AgeIr** and **InfoIr**.

Since the network is used to determine the best action to take given a pair of nodes' states, it must model both nodes' intersection rate sensors. Therefore, there are two sensor models, subscripted with a and b, representing the two nodes. These two nodes relative intersection rates are represented by an additional node, referred to as **IrRatio**, as depicted in figure 8.7.

IrRatio represents the scheduling agent's belief distribution over node a's intersection rate

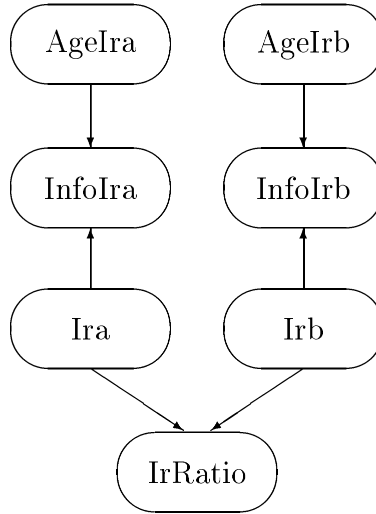


Figure 8.7: Decision network – the resources' capacity block

relative to that of node b. Its domain is

$$D_{IrRatio} = \{\text{aMHigherb, aHigherb, aEqualb, aLowerb, aMLowerb}\}$$

where aMHigherb stands for node a presents a much higher intersection rate than node b.

The Tasks' Requirement Block

The foreground workload sensor model follows the same basic idea as the intersection rate sensor model. There are two sensors, a and b, one for each of the nodes being considered. Each sensor has three random variables: **AgeFW**, **InfoFW** and **FW**. The age variables have exactly the same meaning and domain as **AgeIr**, and evidence is entered using rule 8.8. **InfoFW** represents the workload sensor readings, T_m. The foreground workload is determined using equation 8.1 and the node is then classified, according to rule 8.10, as:

NoWork – no work currently allocated;

Recp – potential receiver due to its light load;

Forn1 – lightly loaded supplier;

Forn2 – heavily loaded supplier.

$$D_{\text{InfoFW}} = \{\text{NoWork, Recp, Forn1, Forn2}\}$$

$$\text{InfoFW} = \begin{cases} \text{NoWork} & \Leftarrow \text{ no work currently allocated} \\ \text{Recp} & \Leftarrow T_m <= 2 * T_{Sched} \\ \text{Forn1} & \Leftarrow 2 * T_{Sched} < T_m <= 10 * T_{Sched} \\ \text{Forn2} & \Leftarrow T_m > 10 * T_{Sched} \end{cases} \quad (8.10)$$

FW is identical to **Ir**, in the sense that this is the quantity upon which decisions are made, but its actual value is not directly observable. The agent's belief distribution over **FW** is inferred, based on the evidence available from the sensor and on this information's age. Its domain is

$$D_{\text{FW}} = \{\text{NoWork, Recp, Forn1, Forn2}\}$$

The information available about the two nodes' workload is merged in **FWRatio**, as depicted by figure 8.8. It represents the scheduling agent's belief distribution over node a's foreground workload relative to that of node b. Its domain is

$$D_{\text{FWRatio}} = \{\text{aMHigherb, aHigherb, aEqualb, aLowerb, aMLowerb}\}$$

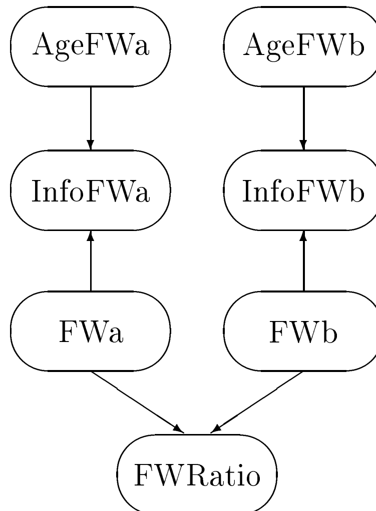


Figure 8.8: Decision network – the tasks' requirements block

The Complete Decision Network

The belief distributions over variables **IrRatio** and **FWRatio** describe the current environment's state, with respect to the two nodes being considered, given the information available as evidence, and obtained through the agent's sensors, and the age of this information.

Any action decided by the scheduling agent, that transfers workload between these nodes, must be based on this information and will change the environment's state. For each

alternative action, the scheduling agent must infer the probability distribution over the environment's next state and compute the respective expected utility. The decision network provides an automatic process of performing these inferences on a small number of steps. To allow this operation additional variables are required, to represent the alternative actions, the environment's next state and the expected utility.

The network is completed (figure 8.9) by adding three variables: **Transfer** is a decision variable and represents the alternatives available to the scheduling agent; **NewBalance** is a random variable and represents the scheduling agent's belief distribution on the system's next state based on what is known about the actual state and on the selected action; **Gain** is an utility variable that computes the expected utility of each action, based on its different possible outcomes and on the probabilities that these occur.

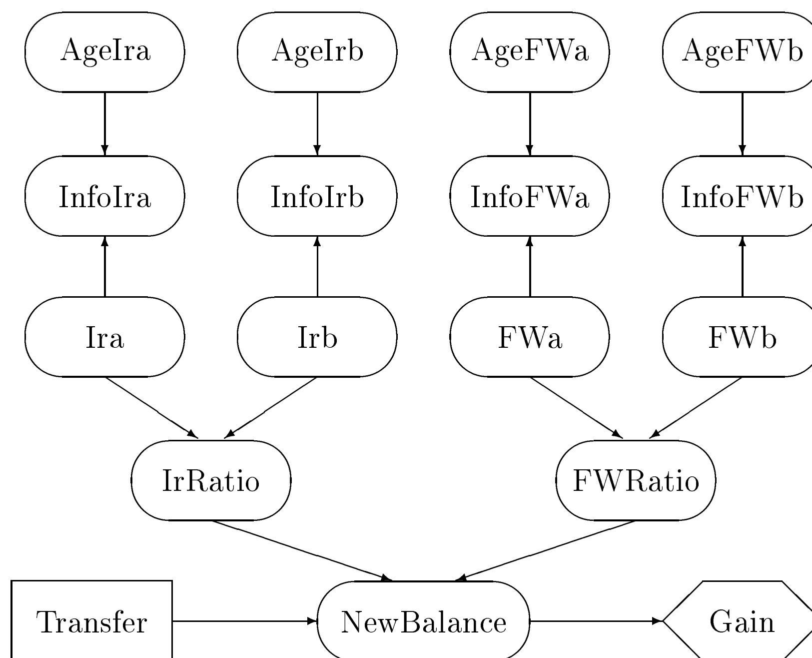


Figure 8.9: The complete decision network

The **Transfer** variable models the alternatives available to the scheduling agent. After entering all the available evidence about the current environment's state, the scheduling agent successively sets this variable to each alternative action, in order to infer its expected utility. By setting this variable, the belief distribution over the next state changes, to express the action's expected consequences. The action's expected utility can then be computed, by averaging each next state utility with the probability that it occurs. The value of this decision variable is externally set by the agent, because it represents an external intervention, which is extrinsic to the environment.

The alternatives available to the scheduling agent are represented in terms of the percentage of work to transfer among nodes a and b and in which direction, i.e., from a to b or from b to a. **Transfer**'s domain is

$$D_{Transfer} = \{\text{a2b75, a2b50, a2b25, NoTransfer, b2a25, b2a50, b2a75}\}$$

where a2b75 stands for transfer 75% of node a's remaining task to node b, and b2a50 stands for transfer 50% of node b's remaining task to node a.

NewBalance models the agent's belief distribution on which will be the environment's next state, given its current state and the selected action. The next state is a stochastic function of the current state – modelled by **IrRatio** and **FWRatio** – and the action taken at this time step. The conditional probability table associated with this variable, **CPT**$(NewBalance|IrRatio, FWRatio, Transfer)$, constitutes the environment's state transition model. **NewBalance**'s domain contains three possible values, related to the future degree of load sharing among the two nodes being considered:

$$D_{NewBalance} = \{\text{VGood, Good, Bad}\}$$

Gain is an utility node, with a corresponding utility function, which translates system's states to real numbers that express the agent's preferences among these states (section 5.5.1). Since the next state is described by a single random variable, **Gain** is a single–argument function of **NewBalance**.

Conditional Independence Analysis

The evidence available at each inference step is

$$E = \{\text{AgeIra, InfoIra, AgeIrb, InfoIrb, AgeFWa, InfoFWa, AgeFWb, InfoFWb, Transfer}\}$$

where **Transfer** is directly set by the inference engine. The sensors, as expected, are d–separated from each others given E. This means, for example, that changing **InfoFWa** does not change **BEL**(FWb), since they are d–separated by **FWRatio**. This is assured by rule 2 of d–separation: the connection between the 2 sensors is converging on **FWRatio**, and neither **FWRatio**, nor any of its descendants is known (section 5.4.1). According to the same rule, the intersection rate sensors and the foreground workload sensors are d–separated by variable **NewBalance**.

8.5.2 Assign Probabilities

Once the network's topology is built, the strengths of the direct causal relationships among the variables connected by arcs must be specified. These strengths are quantified by assigning to each variable X_i a conditional probability table, **CPT**$(X_i|Parents(X_i))$, which

represents the belief on the event $X_i = x_i$, given any possible combination of the parents, or direct causes, of X_i.

Since people are bad numerical estimators, one frequent concern with Bayesian networks is the availability of probabilities. The notion that probabilities merely reflect an individual's own belief on a given statement and that these numbers need only to be approximately specified, allows an expert on the field to informally assess these conditional probabilities. As long as the ratio between the probability of an event occurring or not occurring, given the same evidence, roughly reflects the expert's belief, valid conclusions will still be reached (sections 5.2 and 5.6.3). This section presents the probability tables obtained by direct assessment. The refinements that resulted from performing sensitivity analysis are presented in section 8.5.4.

The Resources' Capacities Block

The intersection rate sensor model requires the specification of the following probabilities: $\mathbf{P}(AgeIra)$, $\mathbf{P}(AgeIrb)$, $\mathbf{CPT}(InfoIra|AgeIra, Ira)$, $\mathbf{CPT}(InfoIrb|AgeIrb, Irb)$, $\mathbf{P}(Ira)$ and $\mathbf{P}(Irb)$.

Since both sensors, a and b, are identical, the probabilities specified for one of them also hold for the other one. Hence, only $\mathbf{P}(AgeIr)$, $\mathbf{P}(Ir)$ and $\mathbf{CPT}(InfoIr|AgeIr, Ir)$ are studied throughout this section.

The scheduling agent always knows the age of the information it is using: it is a function of the time elapsed since that node's last information message arrival. Hence, in every inference step **AgeIr** is instantiated with evidence. Since **AgeIr** does not have parents, its prior probabilities $\mathbf{P}(AgeIr)$ do not have any influence on the network's remaining variables. Therefore, $\mathbf{P}(AgeIr)$ do not have to be specified.

The prior probabilities of **Ir**, $\mathbf{P}(Ir)$, on the other hand, can strongly influence the belief distribution over **Ir** given **AgeIr** and **InfoIr**, $\mathbf{BEL}(Ir)$. This belief is biased towards the prior probabilities distribution. This bias' magnitude depends on the uncertainty associated with the sensor model, $\mathbf{CPT}(InfoIr|AgeIr, Ir)$. The more accurate the sensor, the less $\mathbf{BEL}(Ir)$ is biased towards the prior probabilities of **Ir**. In the case of a deterministic sensor, there is no bias towards the prior probabilities, since there is no uncertainty associated with the sensor's readings. Throughout this experiment no knowledge is assumed about the nodes' intersection rates. Therefore, the prior probabilities of each node's intersection rate are assigned an uniform distribution $\mathbf{P}(Ir) = \{0.2, 0.2, 0.2, 0.2, 0.2\}$. However, these probabilities can be learned by observing the node's behaviour and updating this probability vector (section 8.6.3).

$\mathbf{CPT}(InfoIr|AgeIr, Ir)$ represents the expert's belief distribution over the values returned by the sensor, given the information's age and the actual intersection rate. If the sensor

is known to exhibit some biased behaviour, then this information can be encoded on this table. For instance, if the sensor often overestimates the actual intersection rate, then a larger probability value can be given to account for this fact. The sensor used throughout this particular experiment does not present such biased behaviour. It does, however, present some noise that increases with the age of information. This noise is represented by allowing the probability distribution over **InfoIr** to disperse around **Ir**'s actual value. This dispersion increases with the information's age. Figure 8.10 presents the initial probability assessments for $\mathbf{CPT}(InfoIr|AgeIr, Ir = Medium)$, for all possible different values of **AgeIr**.

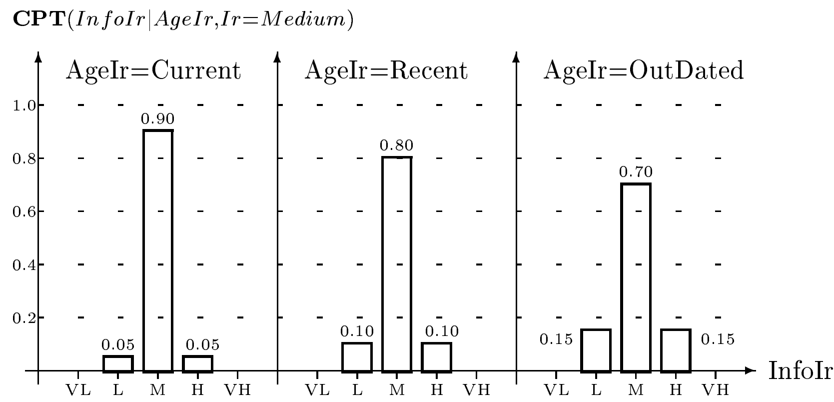


Figure 8.10: InfoIr's belief distribution given Ir = Medium

The conditional probability table $\mathbf{CPT}(InfoIr|AgeIr, Ir)$, presented on table 8.3, was obtained by applying this line of reasoning for all possible values of **Ir**. This is the initial proposal for this quantitative parameter. It will be refined by performing sensitivity analysis throughout section 8.5.4.

$\mathbf{CPT}(IrRatio|Ira, Irb)$ is a deterministic table, i.e., comprising only 0's and 1's, that combines the beliefs on Ira and Irb onto the appropriate value of $IrRatio$ (table 8.4). However, $\mathbf{BEL}(IrRatio)$ is a stochastic vector, since both $\mathbf{BEL}(Ira)$ and $\mathbf{BEL}(Irb)$ are stochastic vectors.

The Tasks Requirements Block

The reasoning behind the foreground workload sensor model is identical to the intersection rate model. The prior probability $\mathbf{P}(AgeFW)$ does not need to be specified, since **AgeFW** has no parents and it is always instantiated with evidence. The prior probability $\mathbf{P}(FW)$, if known, should be included in the model, since it biases $\mathbf{BEL}(FW)$. Once again this bias depends on the uncertainty associated with the sensor model, $\mathbf{CPT}(InfoFW|AgeFW, FW)$. Since nothing is known about **FW**, $\mathbf{P}(FW)$ is assigned an uniform probability distribution.

$\mathbf{CPT}(InfoFW|AgeFW, FW)$ must obey certain rules, that were not present in the inter-

$\mathbf{CPT}(InfoIr\|AgeIr, Ir)$		$InfoIr$				
Ir	$AgeIr$	VeryLow	Low	Medium	High	VeryHigh
VeryLow	Current	0.950	0.050	0.000	0.000	0.000
	Recent	0.900	0.100	0.000	0.000	0.000
	OutDated	0.850	0.150	0.000	0.000	0.000
Low	Current	0.050	0.900	0.050	0.000	0.000
	Recent	0.100	0.800	0.100	0.000	0.000
	OutDated	0.150	0.700	0.150	0.000	0.000
Medium	Current	0.000	0.050	0.900	0.050	0.000
	Recent	0.000	0.100	0.800	0.100	0.000
	OutDated	0.000	0.150	0.700	0.150	0.000
High	Current	0.000	0.000	0.050	0.900	0.050
	Recent	0.000	0.000	0.100	0.800	0.100
	OutDated	0.000	0.000	0.150	0.700	0.150
VeryHigh	Current	0.000	0.000	0.000	0.050	0.950
	Recent	0.000	0.000	0.000	0.100	0.900
	OutDated	0.000	0.000	0.000	0.150	0.850

Table 8.3: Intersection rate sensor model — $\mathbf{CPT}(InfoIr|AgeIr, Ir)$

section rate sensor model:

- if the sensor reports that the node is in state NoWork, then there is no uncertainty about this proposition; NoWork corresponds to the node having no work allocated, and this only happens when the node already sent the results of its last task, and the scheduler did not yet send a new task; this is something the scheduler can be completely sure of;
- these sensors also report some noise, which increases with information's age; however, the scheduling agent also assumes that the foreground workload diminishes with time, since the node is performing its work; consequently, the probability of the sensor returning values larger than the current foreground workload increases with the information's age – see figure 8.11 for an example with **FW**=Forn1.

The initial assessment of the conditional probability table $\mathbf{CPT}(InfoFW|AgeFW, FW)$, presented on table 8.5, was obtained by applying this line of reasoning for all possible values of **FW**. It will be refined by performing sensitivity analysis throughout section 8.5.4.

$\mathbf{CPT}(\mathit{FWRatio}|FWa, FWb)$ is a deterministic table, i.e., containing only 0's and 1's, that combines the beliefs on **FWa** and **FWb** onto the appropriate value of **FWRatio** (table 8.6). To avoid work transferences among two nodes that present similar foreground workloads, two special cases must be considered:

$\mathbf{CPT}(IrRatio \mid Ira, Irb)$		$IrRatio$				
Ira	Irb	aMHigherb	aHigherb	aEqualb	aLowerb	aMLowerb
VeryLow	VeryLow	0	0	1	0	0
	Low	0	0	0	1	0
	Medium	0	0	0	0	1
	High	0	0	0	0	1
	VeryHigh	0	0	0	0	1
Low	VeryLow	0	1	0	0	0
	Low	0	0	1	0	0
	Medium	0	0	0	1	0
	High	0	0	0	0	1
	VeryHigh	0	0	0	0	1
Medium	VeryLow	1	0	0	0	0
	Low	0	1	0	0	0
	Medium	0	0	1	0	0
	High	0	0	0	1	0
	VeryHigh	0	0	0	0	1
High	VeryLow	1	0	0	0	0
	Low	1	0	0	0	0
	Medium	0	1	0	0	0
	High	0	0	1	0	0
	VeryHigh	0	0	0	1	0
VeryHigh	VeryLow	1	0	0	0	0
	Low	1	0	0	0	0
	Medium	1	0	0	0	0
	High	0	1	0	0	0
	VeryHigh	0	0	1	0	0

Table 8.4: $\mathbf{CPT}(IrRatio \mid Ira, Irb)$

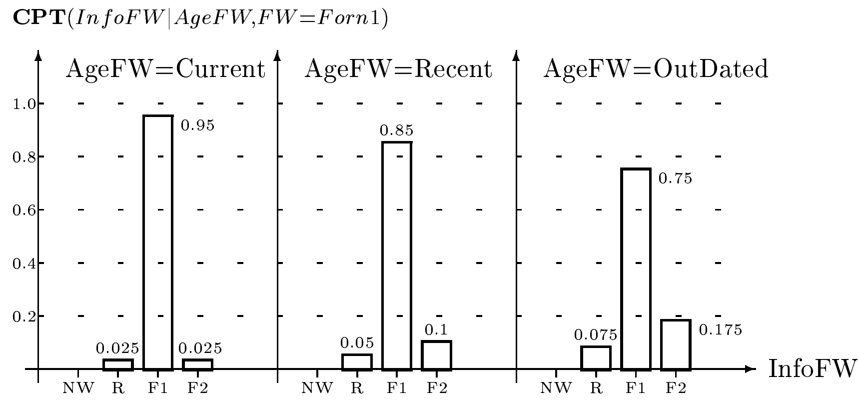


Figure 8.11: InfoFW's belief distribution given FW = Forn1

CPT$(InfoFW \mid AgeFW, FW)$		*InfoFW*			
FW	*AgeFW*	NoWork	Recp	Forn1	Forn2
NoWork	Current	1.000	0.000	0.000	0.000
	Recent	1.000	0.000	0.000	0.000
	OutDated	1.000	0.000	0.000	0.000
Recp	Current	0.000	0.975	0.025	0.000
	Recent	0.000	0.900	0.100	0.000
	OutDated	0.000	0.825	0.175	0.000
Forn1	Current	0.000	0.025	0.950	0.025
	Recent	0.000	0.050	0.850	0.100
	OutDated	0.000	0.075	0.750	0.175
Forn2	Current	0.000	0.000	0.025	0.975
	Recent	0.000	0.000	0.020	0.980
	OutDated	0.000	0.000	0.010	0.990

Table 8.5: Foreground workload sensor model — **CPT**$(InfoFW \mid AgeFW, FW)$

- if one node is classified as NoWork and the other as Recp, then no work transfer among them is desirable; hence, **FWRatio** must be aEqualb;

- if one node is classified as Forn1 and the other as Forn2, then no work transfer among them is desirable; hence, **FWRatio** must be aEqualb;

The State Transition Model

The CPT associated with node **NewBalance**

$$\mathbf{CPT}(NewBalance \mid IrRatio, FWRatio, Transfer)$$

$\mathbf{CPT}(FWRatio \vert FWa, FWb)$		*FWRatio*				
FWa	*FWb*	aMHigherb	aHigherb	aEqualb	aLowerb	aMLowerb
NoWork	NoWork	0	0	1	0	0
	Recp	0	0	1	0	0
	Forn1	0	0	0	0	1
	Forn2	0	0	0	0	1
Recp	NoWork	0	0	1	0	0
	Recp	0	0	1	0	0
	Forn1	0	0	0	1	0
	Forn2	0	0	0	0	1
Forn1	NoWork	1	0	0	0	0
	Recp	0	1	0	0	0
	Forn1	0	0	1	0	0
	Forn2	0	0	1	0	0
Forn2	NoWork	1	0	0	0	0
	Recp	1	0	0	0	0
	Forn1	0	0	1	0	0
	Forn2	0	0	1	0	0

Table 8.6: $\mathbf{CPT}(FWRatio|FWa, FWb)$

constitutes the environment's state transition model. Since

$$\begin{array}{lclr} \#D_{IrRatio} & = & 5 & \#D_{FWRatio} & = & 5 \\ \#D_{Transfer} & = & 7 & \#D_{NewBalance} & = & 3 \end{array}$$

this CPT requires the assessment of $5 * 5 * 7 * (3 - 1) = 350$ independent probabilities. To simplify this assessment some rules must be defined:

- **FWRatio** determines the direction of work transfer, hence:
  - if **FWRatio** = aMHigherb or **FWRatio** = aHigherb, then work is transferred from node a to node b; work transfers from b to a, represented by decisions b2a25, b2a50 and b2a75, must be rejected, independently of **IrRatio**; this is achieved by assigning an absolute certainty to the proposition **NewBalance**=Bad for these decisions; the converse is also true, when b presents higher foreground workload than a; this rule reduces in $3 * 5 * 4 * (3 - 1) = 120$ the number of independent values to be assessed;
  - if **FWRatio** = aEqualb, then some work transfer, from the node exhibiting lower intersection rate to the node with higher intersection rate, may still be desirable; transfers in the opposite direction, however, must be avoided; this rule reduces in $4 * 3 * (3 - 1) = 24$ the number of independent values to be

assessed; the distribution of the $5 * 7 * (3 - 1) = 70$ stochastic independent values associated with **FWRatio** = aEqualb are presented in table 8.7;

- **FWRatio** (**FWR**) and **IrRatio** determine the amount of work to transfer among the two nodes; larger ratios of workload and smaller ratios of intersection rate among the sender and the receiver (as determined by the previous rule), should correspond to larger amounts of transferred work; figures 8.12 and 8.13 present the probability distribution over **NewBalance**'s states, for **FWR**=aMHigherb and **FWR**=aHigherb, respectively, for all values of **IrRatio** and for **Transfer** $\in$ {a2b75, a2b50, a2b25, NT}, where NT stands for NoTransfer; the probabilities for **FWR** = aLowerb and **FWR** = aMLowerb are the converse of these ones with **Transfer** $\in$ {b2a75, b2a50, b2a25, NT}; in order to better understand these graphics two further remarks must be done:
  - the rational behind $\mathbf{CPT}(NewBalance|Transfer = NoTransfer)$ is that not transferring workload among a potential supplier/receiver pair leaves the environment on an undesirable state, and this situation worsens with the increase of the potential receiver's intersection rate;
  - the rational behind

    $$\mathbf{CPT}(NewBalance|IrRatio, FWRatio = aHigherb, Transfer)$$

    which is built from

    $$\mathbf{CPT}(NewBalance|IrRatio, FWRatio = aMHigherb, Transfer)$$

    is that, since the supplier's foreground workload is lighter, transferences are not so profitable, therefore all the probabilities are slightly shifted towards worst values of **NewBalance**; for **Transfer** = NoTransfer, $P(NewBalance = Good)$ is slightly increased.

Figures 8.12 and 8.13 present the remaining 80 rows (160 stochastic independent values) of $\mathbf{CPT}(NewBalance|IrRatio, FWRatio, Transfer)$. These graphs can be read across 3 dimensions:

- reading each graph across the horizontal illustrates how the probability distribution over the values of **NewBalance** changes as the sender's intersection rate increases, for the same action;

- reading each graph across the vertical illustrates how the probability distribution over the values of **NewBalance** changes as a function of the amount of work to transfer, for the same intersection rate ratio;

		$NewBalance$		
$IrRatio$	$Transfer$	VGood	Good	Bad
aMHigherb	a2b75	0.00	0.00	1.00
	a2b50	0.00	0.00	1.00
	a2b25	0.00	0.00	1.00
	NoTransfer	0.60	0.40	0.00
	b2a25	0.30	0.60	0.10
	b2a50	0.00	0.70	0.30
	b2a75	0.00	0.20	0.80
aHigherb	a2b75	0.00	0.00	1.00
	a2b50	0.00	0.00	1.00
	a2b25	0.00	0.00	1.00
	NoTransfer	0.70	0.30	0.00
	b2a25	0.10	0.80	0.10
	b2a50	0.00	0.50	0.50
	b2a75	0.00	0.10	0.90
aEqualb	a2b75	0.00	0.00	1.00
	a2b50	0.00	0.00	1.00
	a2b25	0.00	0.20	0.80
	NoTransfer	1.00	0.00	0.00
	b2a25	0.00	0.20	0.80
	b2a50	0.00	0.00	1.00
	b2a75	0.00	0.00	1.00
aLowerb	a2b75	0.00	0.10	0.90
	a2b50	0.00	0.50	0.50
	a2b25	0.10	0.80	0.10
	NoTransfer	0.70	0.30	0.00
	b2a25	0.00	0.00	1.00
	b2a50	0.00	0.00	1.00
	b2a75	0.00	0.00	1.00
aMLowerb	a2b75	0.00	0.20	0.80
	a2b50	0.00	0.70	0.30
	a2b25	0.30	0.60	0.10
	NoTransfer	0.60	0.40	0.00
	b2a25	0.00	0.00	1.00
	b2a50	0.00	0.00	1.00
	b2a75	0.00	0.00	1.00

Table 8.7: $\mathbf{CPT}(NB|FWRatio = aEqualb, IrRatio, Transfer)$

- comparing two identically positioned blocks of both graphs illustrates how the probability distribution over the values of **NewBalance** changes as a function of the foreground workload ratio, for the same intersection rate ratio and amount of transferred work.

8.5.3 Assign Utilities

An utility function must be specified which translates each action's outcome, probabilistically encoded in **NewBalance**, into a real number that expresses that action's desirability. Since the next state is described by a single variable, the required utility function is a single attribute function. Each possible value of **NewBalance** is given a certain utility. Expected utility is computed by weighing these utilities with the probability that each particular value of **NewBalance** occurs.

Utilities are normalised, therefore the most desirable state, **NewBalance** = VGood, is given an utility of 1, and the least desirable state, **NewBalance** = Bad, is given an utility of 0. Since **NewBalance** = Good is conceptually closer to VGood than to Bad, as can be seen by carefully analysing figures 8.12 and 8.13, it was given an utility of 0.6. Therefore, each action a expected utility given the available evidence E, is computed as follows

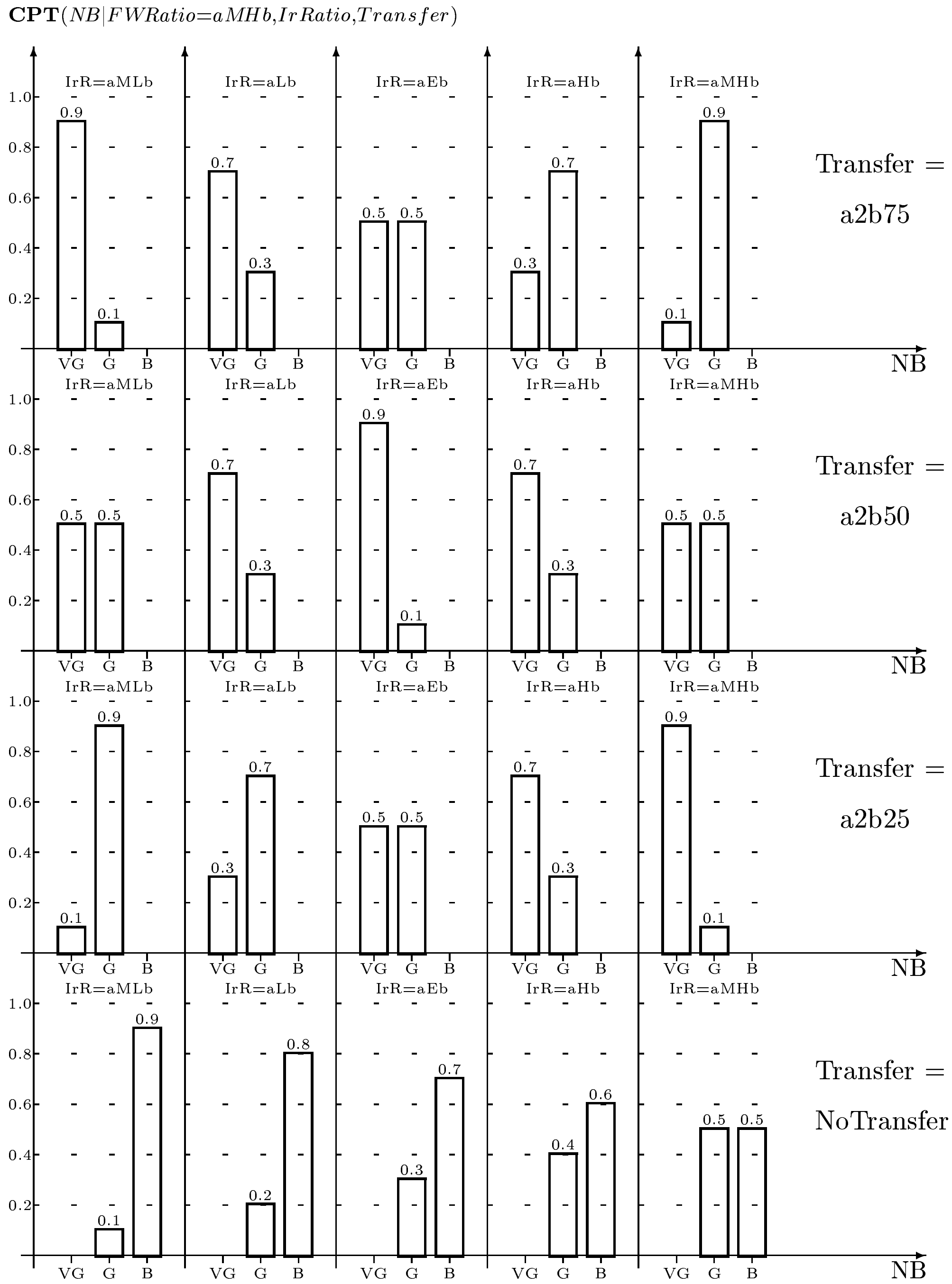
$$\begin{aligned}
U(NB = VGood) &= 1 \\
U(NB = Good) &= 0.6 \\
U(NB = Bad) &= 0 \\
EU(a|\mathbf{E}) &= P(NB = VGood|\mathbf{E}, Do(a)) * U(NB = VGood) + \\
&\quad P(NB = Good|\mathbf{E}, Do(a)) * U(NB = Good) + \\
&\quad P(NB = Bad|\mathbf{E}, Do(a)) * U(NB = Bad)
\end{aligned}$$

where NB stands for **NewBalance**.

8.5.4 Sensitivity Analysis

Sensitivity analysis is used to determine which of the model's qualitative or quantitative parameters are more relevant to the system's behaviour and aims to increase confidence on the agent's decisions. Most of the model's parameters can be analysed, by introducing small variations and verifying if the best decision changes with them. Since there is a large number of different combinations of the various parameters of the model, a tradeoff must be found between the effort spent in sensitivity analysis and the confidence on the agent's decisions.

The tests performed to the decision network just described include the analysis of:



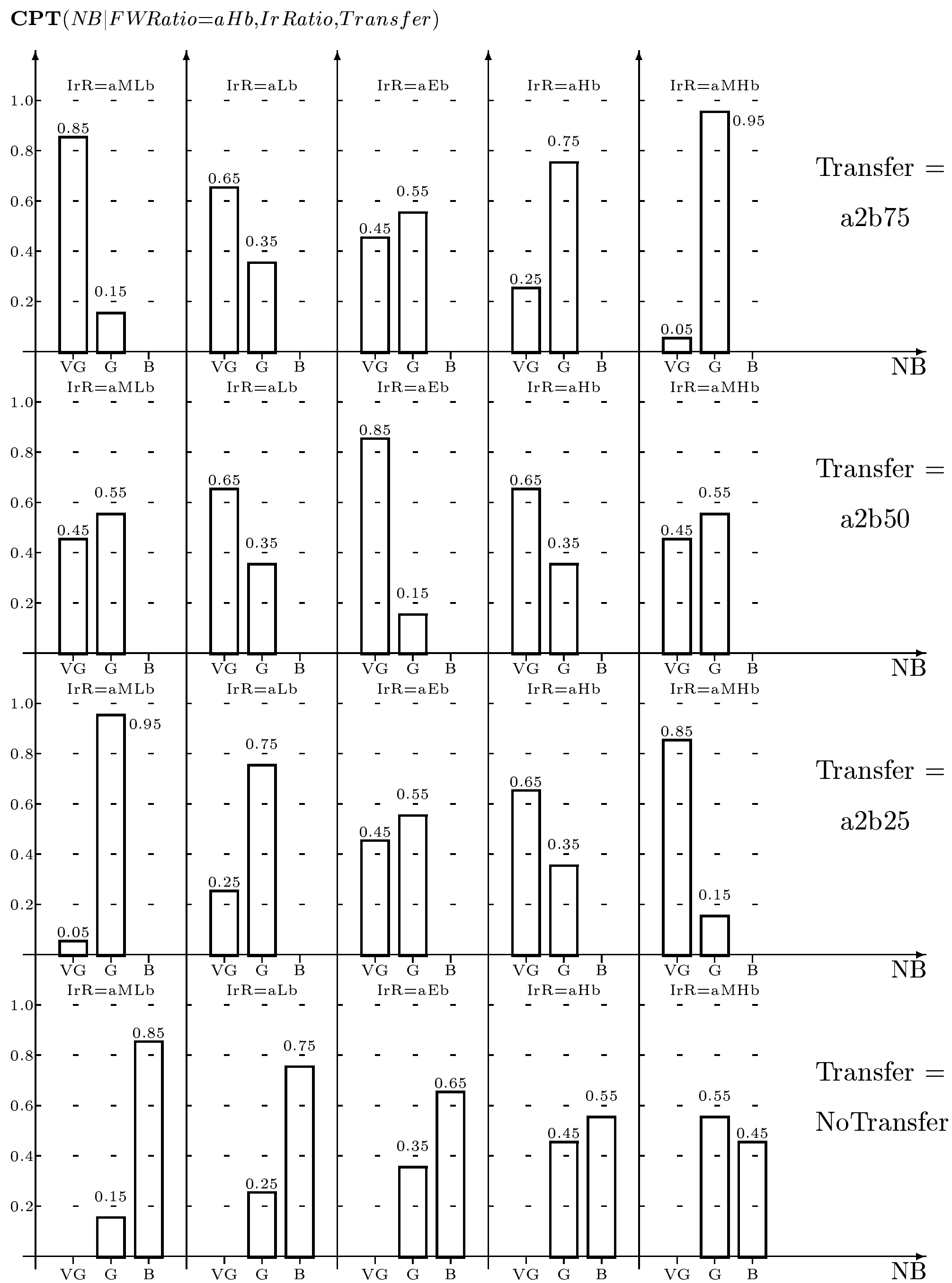
where:

$IrR \equiv IrRatio$; $NB \equiv NewBalance$

$D_{NewBalance} = \{VGood, Good, Bad\} \equiv \{VG, G, B\}$

$D_{FWRatio} = D_{IrRatio} = \{aMHigherb, aHigherb, aEqualb, aLowerb, aMLowerb\} \equiv \{aMHb, aHb, aEb, aLb, aMLb\}$

Figure 8.12: NewBalance's belief distribution given FWRatio=aMHigherb



where:

$IrR \equiv IrRatio$; $NB \equiv NewBalance$

$D_{NewBalance} = \{VGood, Good, Bad\} \equiv \{VG, G, B\}$

$D_{FWRatio} = D_{IrRatio} = \{aMHigherb, aHigherb, aEqualb, aLowerb, aMLowerb\} \equiv \{aMHb, aHb, aEb, aLb, aMLb\}$

Figure 8.13: NewBalance's belief distribution given FWRatio=aHigherb

- the sensitivity of the agent's best decision to the sensors' variables domain cardinality;
- the influence of the age of information variables on the belief distribution over the associated metrics.

Sensors' variables domain cardinality

A complete list of the best decisions for all possible combinations of the evidence variables showed that:

- the recommended decision changes with variations in the the values of both **InfoIra** and **InfoIrb**; the domains of these variables are, therefore, maintained;
- the recommended decision is insensitive to whether **InfoFW** is equal to **NoWork** or **Recp**; the same applies to whether **InfoFW** is equal to **Forn1** or **Forn2**; consequently, these variables' domains are redefined to include only two states:

$$D_{InfoFWa} = D_{InfoFWb} = \{Recp, Forn\}$$

As a consequence of the latter item, the domains of **FWa** and **FWb** must be redefined to the same set of values as **InfoFWa**'s domain.

The foreground workload sensor model must also be adjusted, as well as **FWRatio**'s domain. $\mathbf{CPT}(FWRatio|FWa, FWb)$ is now given by table 8.8.

$\mathbf{CPT}(FWRatio\|FWa, FWb)$		*FWRatio*		
FWa	*FWb*	aMHigherb	aEqualb	aMLowerb
Recp	Recp	0	1	0
Recp	Forn	0	0	1
Forn	Recp	1	0	0
Forn	Forn	0	1	0

Table 8.8: $\mathbf{CPT}(FWRatio|FWa, FWb)$

Since **FWRatio**'s domain now has only three elements, the entries of the state transition model, $\mathbf{CPT}(NewBalance|IrRatio, FWRatio, Transfer)$ for **FWRatio = aHigherb** and **FWRatio = aLowerb** no longer exist. This conditional probability table (CPT) now requires the assessment of $5 * 3 * 7 * (3 - 1) = 210$ independent probabilities, which are equal to those presented in section 8.5.2.

Sensitivity to the age of information

The accuracy of the sensors' models

$$\mathbf{CPT}(InfoFW|AgeFW, FW)$$

$$\mathbf{CPT}(InfoIr|AgeIr, Ir)$$

determines whether the age of the information, **AgeFW** and **AgeIr**, and the prior probabilities of the metrics, $\mathbf{P}(FW)$ and $\mathbf{P}(Ir)$, have any influence on the belief on these metrics, $\mathbf{BEL}(FW)$ and $\mathbf{BEL}(Ir)$, given the available evidence.

The more accurate the sensors, the less these beliefs are influenced by the age of information and by the prior probabilities, since the uncertainty about the sensor's readings decreases. On the case of a perfect sensor, represented by a deterministic probability table, there is no influence from these quantities, since there is no uncertainty associated with the sensor's readings. Real sensors, however, are not perfect, hence the belief on the non–observable metrics should be influenced by the age of information and by the prior probabilities of these metrics.

The accuracy of the sensors is modelled in the CPTs associated with the variables that represent the sensors' readings: **InfoIr** and **InfoFW**. In order to adjust these models, so that the age of information influences $\mathbf{BEL}(FW)$ and $\mathbf{BEL}(Ir)$, two different sets of CPTs were tried. Two indexing variables, one for each environment metric, were added to the decision network. These are **IrS** and **FWS**, and allow the selection between two sensor models: **Model1** models an accurate sensor, with low uncertainty about the sensors' readings, while **Model2** represents a more inaccurate sensor, increasing the agent's uncertainty about its readings. If the indexing variable is set to **Model1** then the accurate sensor model is selected, otherwise it must be set to **Model2** and the inaccurate model is selected. Figure 8.14 depicts the decision network with the indexing variables, and tables 8.9 and 8.10 present the alternative conditional probability tables for the intersection rate and the foreground workload sensors, respectively.

The values obtained for $\mathbf{BEL}(Ir)$ and $\mathbf{BEL}(FW)$, for all possible combinations of the evidence and indexing variables, have shown that only with **Model2** do these beliefs exhibit some sensitivity to the information's age. Furthermore, only these sets of conditional probabilities allow the beliefs on these metrics to be influenced by the prior probabilities of the intersection rate and foreground workload, $\mathbf{P}(Ir)$ and $\mathbf{P}(FW)$. Hence, the sensors models which include more uncertainty about the sensors' readings were selected for the current experiment. The final decision network is thus obtained by removing the indexing variables, **IrS** and **FWS**, and using the conditional probability tables associated with **Model2**.

8.6 Results' Analysis

This section presents and discusses the results obtained with the schedulers introduced in the previous sections. The numerical results are presented in appendix D. Section

$\mathbf{CPT}(InfoIr \mid AgeIr, Ir, IrS)$			$InfoIr$				
IrS	Ir	$AgeIr$	VeryLow	Low	Medium	High	VeryHigh
Model1	VeryLow	Current	0.950	0.050	0.000	0.000	0.000
		Recent	0.900	0.100	0.000	0.000	0.000
		OutDated	0.850	0.150	0.000	0.000	0.000
	Low	Current	0.050	0.900	0.050	0.000	0.000
		Recent	0.100	0.800	0.100	0.000	0.000
		OutDated	0.150	0.700	0.150	0.000	0.000
	Medium	Current	0.000	0.050	0.900	0.050	0.000
		Recent	0.000	0.100	0.800	0.100	0.000
		OutDated	0.000	0.150	0.700	0.150	0.000
	High	Current	0.000	0.000	0.050	0.900	0.050
		Recent	0.000	0.000	0.100	0.800	0.100
		OutDated	0.000	0.000	0.150	0.700	0.150
	VeryHigh	Current	0.000	0.000	0.000	0.050	0.950
		Recent	0.000	0.000	0.000	0.100	0.900
		OutDated	0.000	0.000	0.000	0.150	0.850
Model2	VeryLow	Current	0.950	0.030	0.015	0.005	0.000
		Recent	0.850	0.100	0.030	0.020	0.000
		OutDated	0.750	0.150	0.070	0.020	0.010
	Low	Current	0.050	0.900	0.030	0.015	0.005
		Recent	0.150	0.700	0.100	0.030	0.020
		OutDated	0.250	0.500	0.150	0.030	0.020
	Medium	Current	0.020	0.030	0.900	0.030	0.020
		Recent	0.050	0.100	0.700	0.100	0.050
		OutDated	0.100	0.150	0.500	0.150	0.100
	High	Current	0.005	0.015	0.030	0.900	0.050
		Recent	0.020	0.030	0.100	0.700	0.150
		OutDated	0.030	0.070	0.150	0.500	0.250
	VeryHigh	Current	0.000	0.005	0.015	0.030	0.950
		Recent	0.000	0.020	0.030	0.100	0.850
		OutDated	0.010	0.020	0.070	0.150	0.750

Table 8.9: $\mathbf{CPT}(InfoIr \mid AgeIr, Ir, IrS)$ — sensitivity analysis

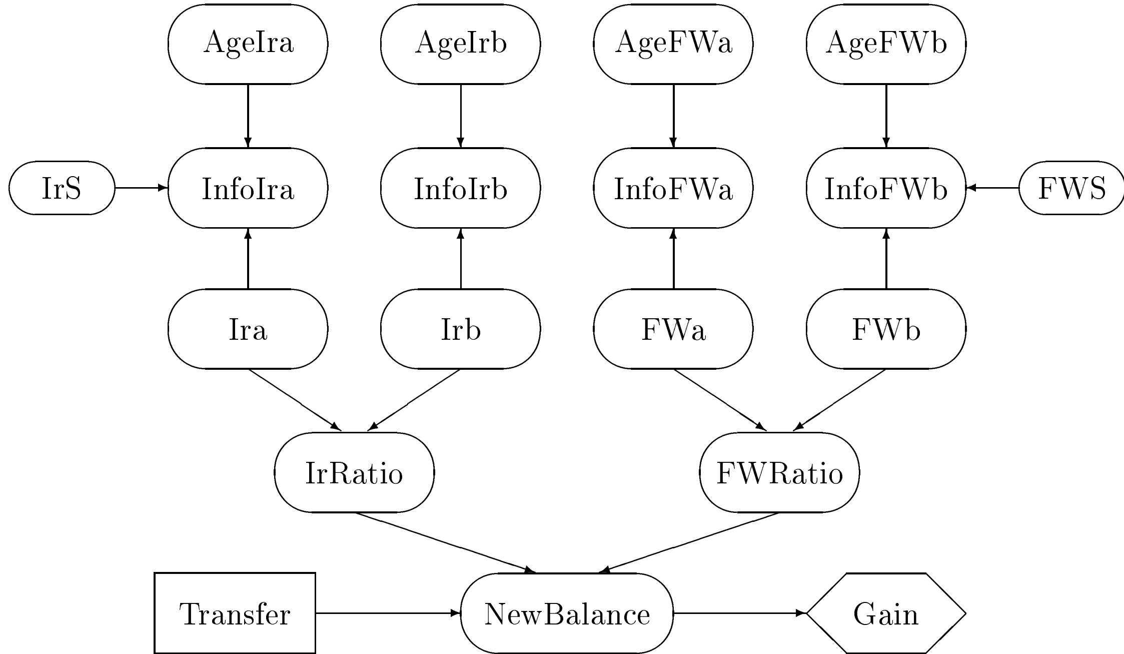


Figure 8.14: Sensitivity analysis

8.6.1 discusses the results obtained in dedicated mode, while throughout section 8.6.2 the distributed system is submitted to different background workload patterns. In section 8.6.3 the stochastic scheduler learns some of the model's probabilities, using a process known as sequential update, in an attempt to both correct eventual errors in the initial assessment of these parameters and to adapt to changes in the dynamics of the environment. Finally, section 8.6.4 discusses and summarises the obtained results.

8.6.1 Dedicated Mode

Figure 8.15 presents the execution time for all four scenes and scheduling strategies, while figure 8.16 presents the performance improvement obtained by using each dynamic scheduling strategy relative to the static uniform workload distribution. The performance improvement is computed as

$$\frac{T_{unif} - T_{sched}}{T_{unif}} * 100$$

where *sched* refers to each scheduling strategy and *unif* refers to the uniform work distribution [101]. These results were obtained using the distributed system in dedicated mode, i.e., without any additional background workload, as described in section 6.6.

Figure 8.15 aims to illustrate the system's behaviour as a function of the number of nodes and scheduling strategy. Since the execution time with the uniform allocation strategy is

$\mathbf{CPT}(InfoFW \mid AgeFW, FW, FWS)$			$InfoFW$	
FWS	FW	$AgeFW$	Recp	Forn
Model1	Recp	Current	0.950	0.050
		Recent	0.900	0.100
		OutDated	0.850	0.150
	Forn	Current	0.050	0.950
		Recent	0.100	0.900
		OutDated	0.150	0.850
Model2	Recp	Current	0.900	0.100
		Recent	0.825	0.175
		OutDated	0.750	0.250
	Forn	Current	0.100	0.900
		Recent	0.175	0.825
		OutDated	0.250	0.750

Table 8.10: $\mathbf{CPT}(InfoFW \mid AgeFW, FW, FWS)$ — sensitivity analysis

significantly larger than with the dynamic strategies, the differences among these are not easy to read from these graphics. Figure 8.16 aims to clearly show these differences.

Some comments can be made by carefully interpreting these last graphics. Significant gains are obtained by using dynamic scheduling strategies, rather than a static uniform workload distribution. The performance improvement can be as high as 75%, for balls4pv and a system with 5 processing nodes. The only exception is with scene balls3 and 2 processing nodes, where the uniform distribution is more effective than some of the dynamic ones, due to a balanced distribution of work between the two nodes. However, if different background workloads were imposed on these nodes, the static scheduling strategy would not be able to redistribute the workload and, hence, would fail to achieve better results than the dynamic ones.

The performance improvement increases with the scenes' complexity: balls4pv is more complex than teapot9, which is more complex than balls3c, which is more complex than balls3; the improvements achieved follow exactly the same order. This is expected, since more complex workloads, both on required computational power and remote data fetching, provide the scheduler with more opportunities to intervene. However, the scheduling agent's execution model and available information must be adequate, otherwise it may fail to decide correctly.

The performance improvement tends to decrease as the number of nodes increases. This is true for all scenes, when the number of nodes increases from 6 to 7. This trend was not yet confirmed with a larger number of nodes; the cluster used as the target platform will have additional nodes soon, and evaluation on these larger system is already planned.

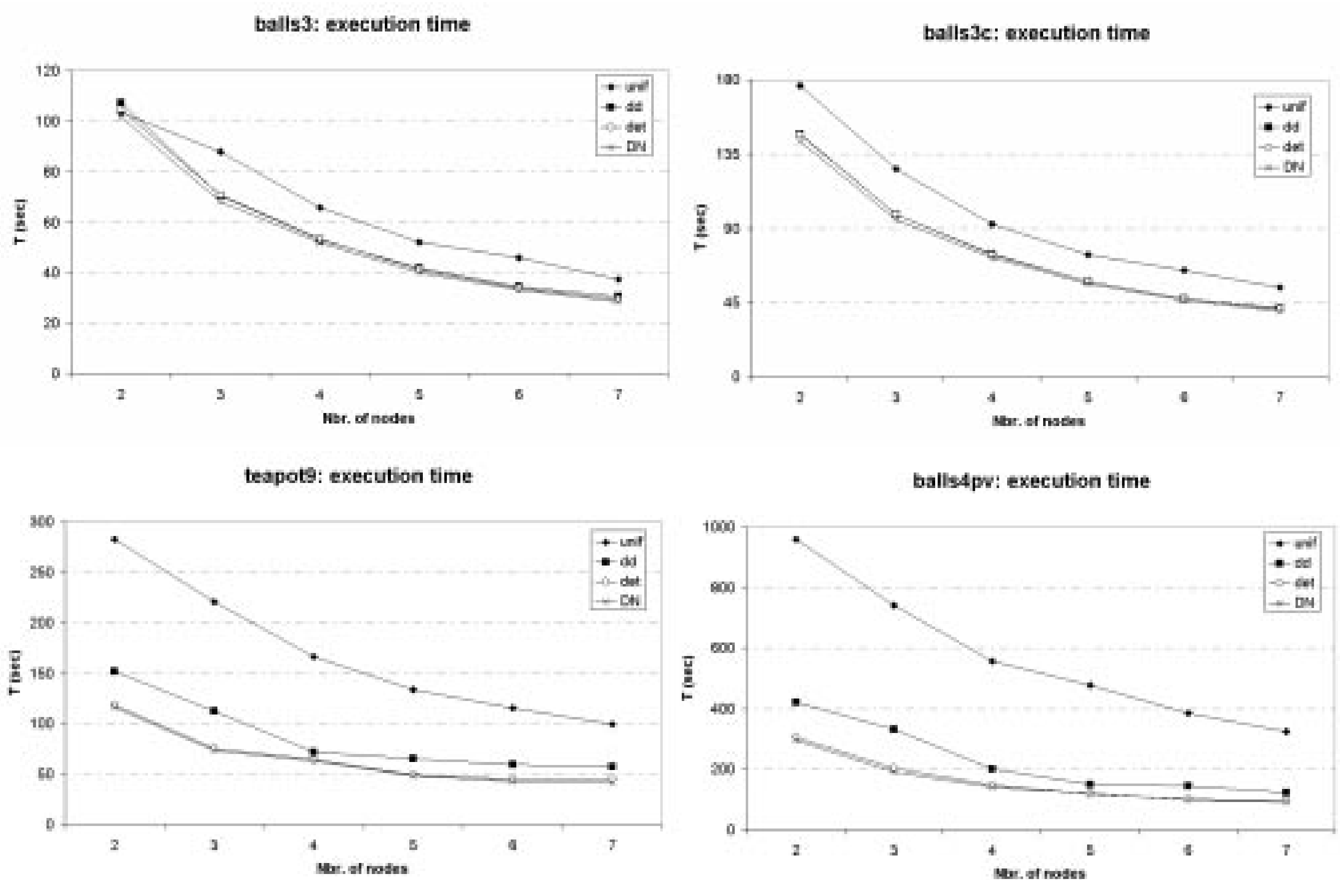


Figure 8.15: Execution time with different scheduling strategies

This is a predictable behaviour, since, as the number of nodes increases, the static uniform scheduling agent divides the image in an increasing number of sub–regions, increasing the probability that the workload gets evenly distributed across all nodes. However, if the nodes' background workloads change in runtime the opposite behaviour is expected, since the static scheduling agent is not able to redistribute the workload according to the variations in the nodes' computing throughputs. This issue will be addressed in section 8.6.2 (see figure 8.22).

The reduction in performance improvement with the number of nodes must not be confused with a scalability problem. Figure 8.15 clearly shows that execution times decrease as the number of nodes increases.

The sensor based dynamic scheduling strategies achieve consistently better results than the demand–driven one. This is specially true for the more complex scenes. It can be concluded that the time spent collecting detailed information about the environment's state and using a more complex decision making mechanism is worthwhile: the better quality of the generated schedules overcomes the overheads associated with these strategies. The higher performance level is also partially due to the more wide range of actions available to

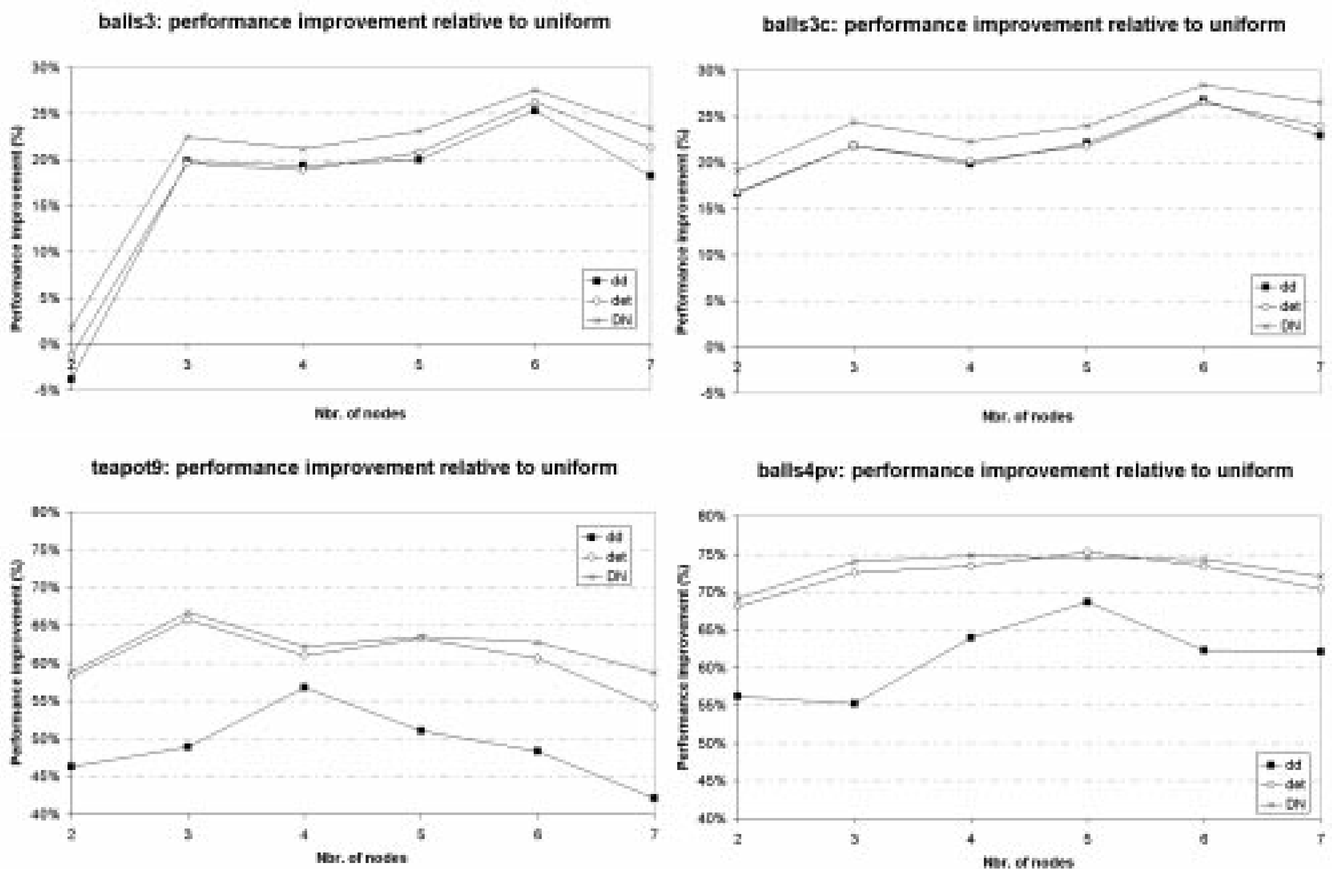


Figure 8.16: Performance improvements with different scheduling strategies.
(Note: the upper and lower graphics' axis have different scales)

the schedulers. These are not restricted to do one–time assignments of tasks to processing nodes, they can migrate tasks by taking advantage of their divisibility property.

The improvements obtained with the stochastic scheduler may not seem significantly larger than those obtained with the deterministic scheduling agent; however, figure 8.17, which plots the performance improvement achieved with DN relative to det, computed as

$$\frac{T_{det} - T_{DN}}{T_{det}} * 100$$

shows that the gains achieved with the stochastic approach increase, although not monotonously, with both the number of nodes and the scenes' complexity. The main exception happens for scenes teapot9 and balls4pv, with 5 nodes, because the deterministic scheduler performs specially well in these two cases. From 6 to 7 nodes, this difference in gain increases significantly. This suggests that this approach can be more rewarding on larger distributed systems.

Figure 8.18 shows indirect costs associated with scenes teapot9 and balls4pv, using the deterministic and stochastic scheduling strategies. It includes the percentage of resource idle times (TTidle%) and the percentage of remote data fetching times (TTdata%) com-

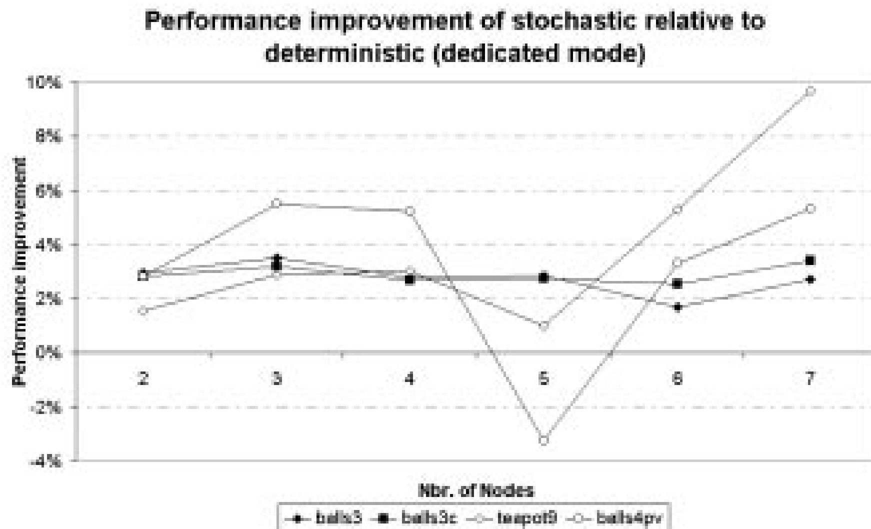


Figure 8.17: Performance improvement of DN relative to det

pared to the aggregated execution time; the number of tasks, which is a direct cost, is also shown. The replication penalty is not included, but it is strongly related to the number of tasks. On each group of bars the left–hand side bar represents the deterministic results (det) and the right–hand side one the decision network (DN) results. DN consistently presents lower indirect costs than det. The improvements achieved with DN are, probably, a consequence of these reduced indirect overheads. Furthermore, these results are achieved with a smaller number of tasks, which further contributes to reduce overheads associated with tasks migrations and work replication penalty.

TTdata is not significant for teapot9, but it represents a large overhead for balls4pv. None of the scheduling agents makes any effort to reduce TTdata, since it is not included on their execution model. However, TTdata is slightly lower with the DN scheduling strategy than with the det one. This is probably due to the lower number of tasks. Since each task is a sub–region of the image and because rays within the same neighbourhood tend to access the same set of objects due to image coherency, it is probable that after processing a few rays most of the objects required to render that region are stored on the processing node local cache. When a task is divided into two sub–regions and one of them is assigned to another node, this node will have to fill its local cache, thereby increasing remote data fetching time.

This graphic also shows that, for balls4pv, TTdata% tends to decrease as the number of nodes increases. Data items are evenly distributed across all processing nodes and the number of data items residing in each node is independent on the number of nodes. As the number of nodes increases, data requests are more evenly spread around all nodes. Hence, each node receives less data requests, which reduces contention on the process servicing these requests. These are satisfied faster, reducing each node's remote data fetching time.

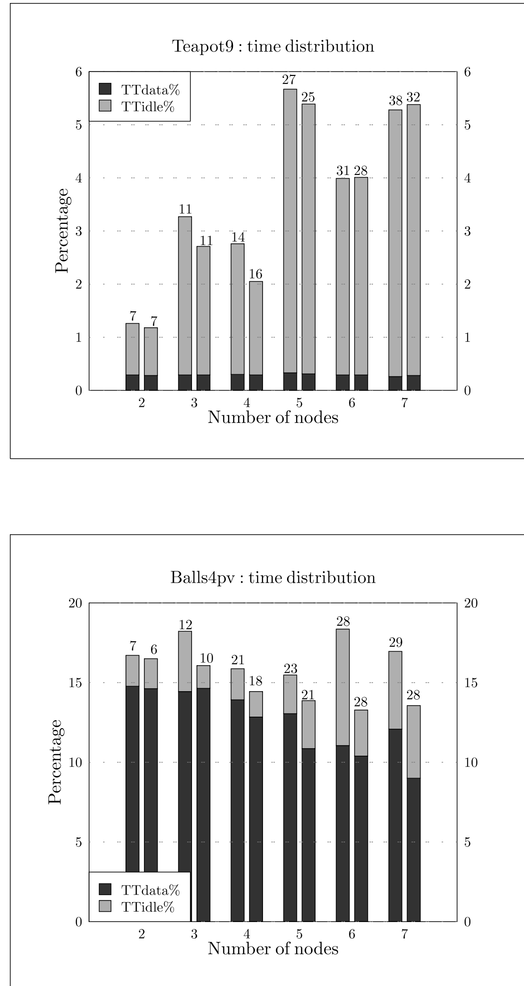


Figure 8.18: TTidle% and TTdata% with *det* and DN scheduling strategies.
On each group of bars *det* is the left hand–side bar and DN the right hand-side one.
The labels above the bars represent the number of tasks.
(Note: the two graphics' axis have different scales)

8.6.2 Background Workload

Figure 8.19 presents the execution time, in a system with 7 nodes, for the different scheduling strategies and background workload patterns described in section 6.6, while figure 8.20 presents the respective performance improvement relative to the uniform work distribution.

A few remarks can be made from this figure:

- the performance improvement increases with the weigh of the background workload; the static uniform work distribution strategy gets more ineffective as the heterogeneity of the nodes' computing throughput increases, while the dynamic schedulers can redistribute the workload in runtime, migrating tasks from overloaded nodes to

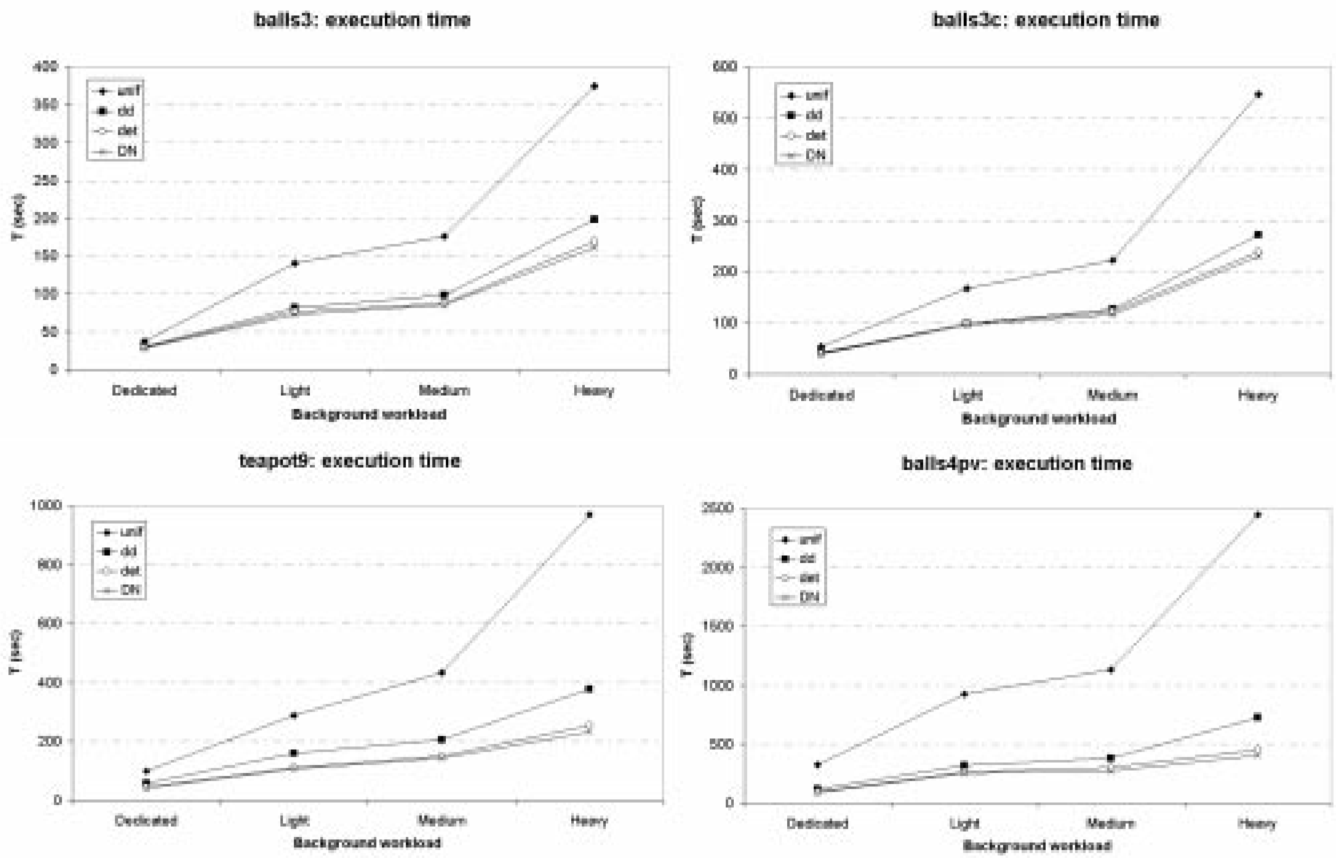


Figure 8.19: Execution time with different background workloads (7 nodes)

underloaded ones;

- the sensor based dynamic strategies get more effective than the demand–driven approach as the background workload increases; this confirms and reinforces the previous conclusion that the additional costs of collecting detailed information about the environment's state are overcome by the improved quality of the generated schedule;
- the performance improvement obtained with the stochastic strategy is larger than that obtained with the deterministic strategy, for all scenes and background workloads; furthermore, the performance improvement obtained with the stochastic scheduler relative to the deterministic scheduler, computed as

$$\frac{T_{det} - T_{DN}}{T_{det}} * 100$$

increases with the background workload weigh and the scene's complexity, as illustrated by figure 8.21.

Figure 8.22 presents the performance improvement obtained with scene teapot9 and the heavy synthetic background workload for different number of nodes. Clearly, the performance improvement increases with the number of nodes. The only exceptions are with 3

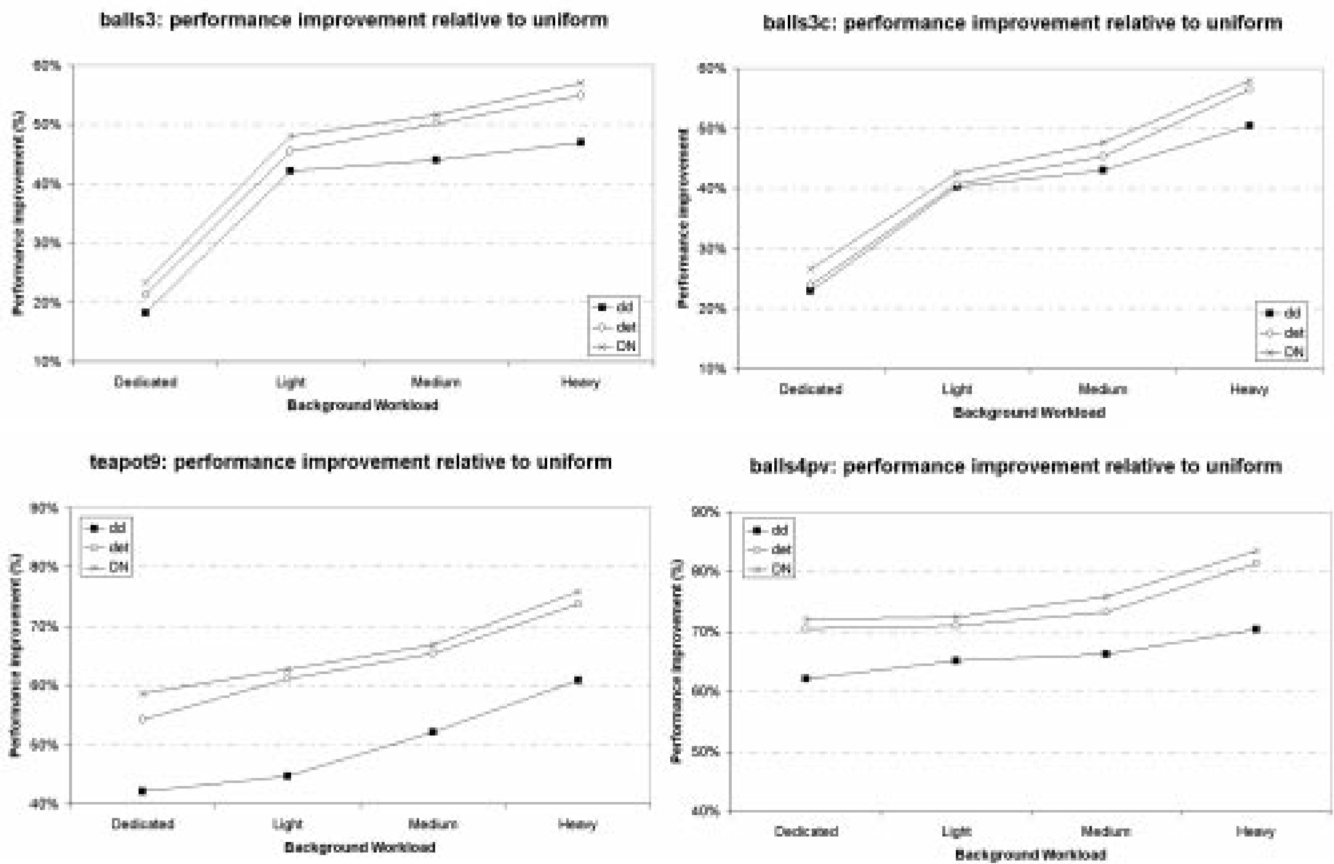


Figure 8.20: Performance improvement with different background workloads (7 nodes)
(Note: the upper and lower graphics' axis have different scales)

nodes, where the uniform scheduler performs poorly, and with 5 nodes, where it performs particularly well.

Figure 8.23 shows indirect costs and the number of tasks, for scenes teapot9 and balls4pv, using the deterministic and stochastic scheduling strategies, on the 7 nodes distributed system, with different background workloads. The stochastic scheduler, once again, incurs less indirect overheads and requires less task migrations than the deterministic scheduler. Remote data fetching times grow very significantly as the background workload increases. Since processors are busier, the time to reply to data items requests is longer. These results suggest that including remote data fetching overheads on the execution model, so that the scheduling agent tries to minimise them by exploiting data locality, can increase the scheduler's performance when the background workload is significantly heavy.

This graphic corroborates the observations made about figure 8.18. The stochastic scheduling strategy consistently incurs less indirect overheads than the deterministic one, reducing both resources' idle times and the time spent waiting for remote data. The number of tasks generated by this scheduler is also smaller.

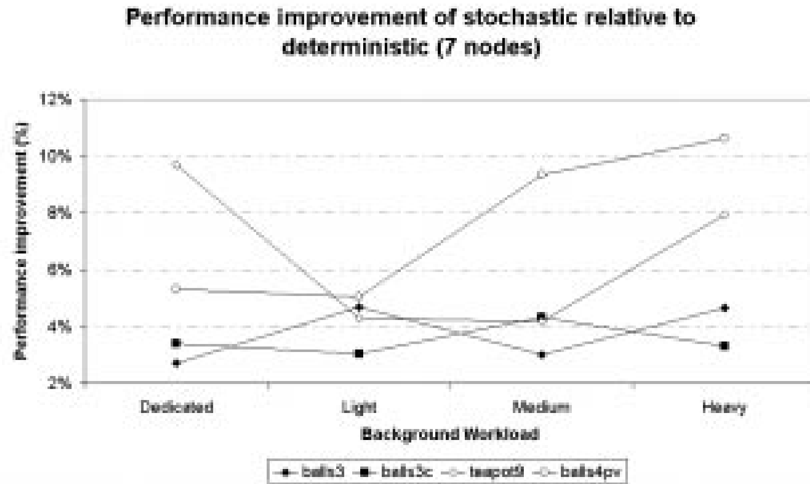


Figure 8.21: Performance improvement of DN relative to det (7 nodes).

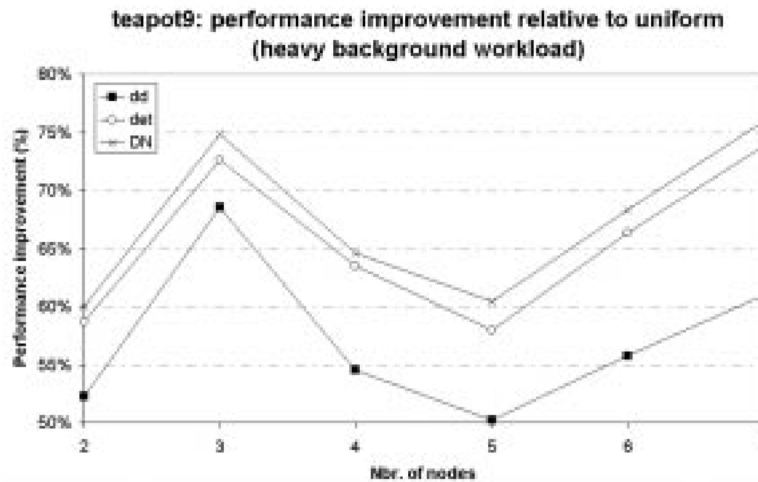


Figure 8.22: Teapot9: performance improvement with heavy background workload

8.6.3 Using Previous Knowledge about the Background Workload

The decision network used so far does not include any previous knowledge about the nodes' actual intersection rates. In fact, the prior probabilities of these variables are assigned uniform distributions, $\mathbf{P}(Ir) = \{0.2, 0.2, 0.2, 0.2, 0.2\}$. This assignment does not mean that it is believed that all nodes have an equal probability of being in any state with respect to their computing throughput. Rather, it is a consequence of the designer's ignorance about the nodes' background workloads and computing throughput. This ignorance prevents the designer from making more accurate guesses about the nodes' intersection rates and forces him to rely solely on the sensors' inputs.

The imprecisions on the prior probabilities $\mathbf{P}(Ir)$ can be corrected, by updating them

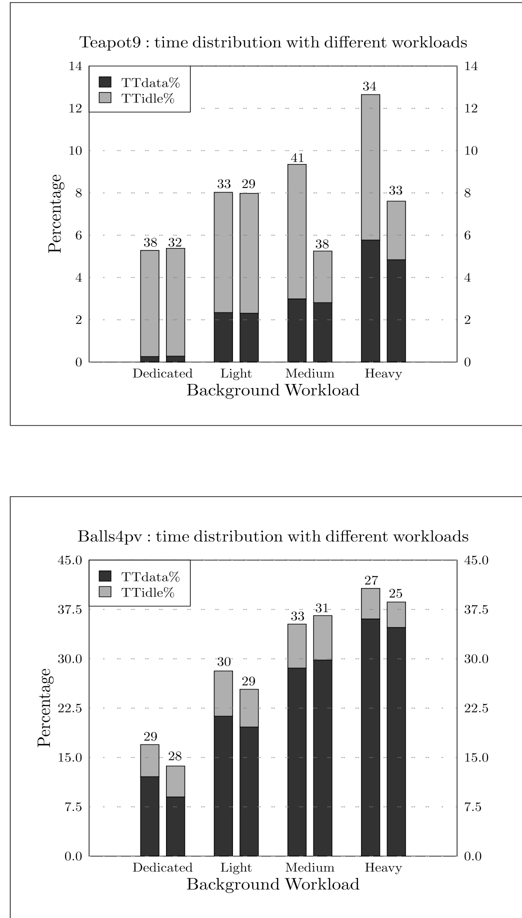


Figure 8.23: TTidle% and TTdata% with *det* and DN scheduling strategies and different background workloads.
On each group of bars *det* is the left hand–side bar and DN the right hand-side one.
The labels above the bars represent the number of tasks.
(Note: the two graphics' axis have different scales)

whenever new data is available. At each inference step, hereby designated as iteration, the scheduling agent observes the environment's state as perceived by its sensors. The observed variables are represented by the evidence vector $\mathbf{E}$, which includes the intersection rates' sensors readings. Using this evidence the agent infers the expected utility for each action, and selects the one which maximises it. Since the used inference algorithm updates the belief on all the decision network's variables, the posterior probabilities of the intersection rate variables given the available evidence, $\mathbf{P}(Ira|\mathbf{E})$ and $\mathbf{P}(Irb|\mathbf{E})$, are also computed. Hence, these values, which are available without any additional computational effort, can be used to update the prior probabilities $\mathbf{P}(Ira)$ and $\mathbf{P}(Irb)$. This can be achieved by using an exponentially recency–weighed average at each iteration, as discussed in section 8.3.2, with $\alpha = 0.5$,

$$\mathbf{P}_{n+1}(Ir) = \mathbf{P}_n(Ir) + \alpha * [\mathbf{P}_n(Ir|\mathbf{E}) - \mathbf{P}_n(Ir)] \tag{8.11}$$

This process is known as sequential updating of numerical parameters on a fixed structure, since the network's topology is kept constant. Sequential update is an online learning problem. At each iteration n, the Bayesian network B_n receives new data u_n and produces the next hypothesis B_{n+1} [55, 66, 164]. Sequential update is a crucial capability, allowing the development of adaptive systems that can overcome errors in their initial model and adapt to changes in the dynamics of the environment being modelled. It requires a conceptual change on how probabilities are understood; these are no longer restricted to represent the degree of belief on a given proposition, as proposed by the subjectivist school, but also encode the frequency with which a given event occurs.

A new version of the decision network based scheduler was developed, which sequentially updates the nodes' intersection rates prior probabilities at each inference step, using equation 8.11. This is referred to as the adaptive stochastic scheduler. The proposed decision network evaluates the best action to take, based on pairs of processing nodes, instantiating the variables subscripted with a and b with data from the appropriate nodes. Therefore, the posterior probabilities $\mathbf{P}(Ir|\mathbf{E})$ and the prior probabilities $\mathbf{P}(Ir)$ must be computed separately for each processing node.

Figure 8.24 a) presents the performance improvement of the adaptive stochastic scheduler relative to the previous stochastic scheduler. The results were obtained with the four different scenes and synthetic background workloads, for a system with 7 nodes.

The improvements in execution time go as high as 3.7%. These improvements grow with the background workload weigh and the scene's complexity. The improvement growth with the background workload weigh shows that as the heterogeneity among the processing nodes' intersection rates increases, the information encoded on the prior probabilities $\mathbf{P}(Ir)$ gets more valuable, allowing the scheduling agent to make more correct decisions with respect to the nodes' future behaviour.

Figure 8.24 b) plots the performance improvement obtained with the adaptive stochastic scheduler relative to the deterministic scheduler, computed as

$$\frac{T_{det} - T_{SeqUpd}}{T_{det}} * 100$$

This improvement is significant and grows, although not monotonously, with the scene's complexity and the background workload weigh. It goes up to 13.8% for balls4pv and a heavy background workload, corroborating the hypothesis that the decision network based scheduler can be more effective than an equivalent deterministic one, by explicitly including on its execution model and decision making mechanism the uncertainty about the environment.

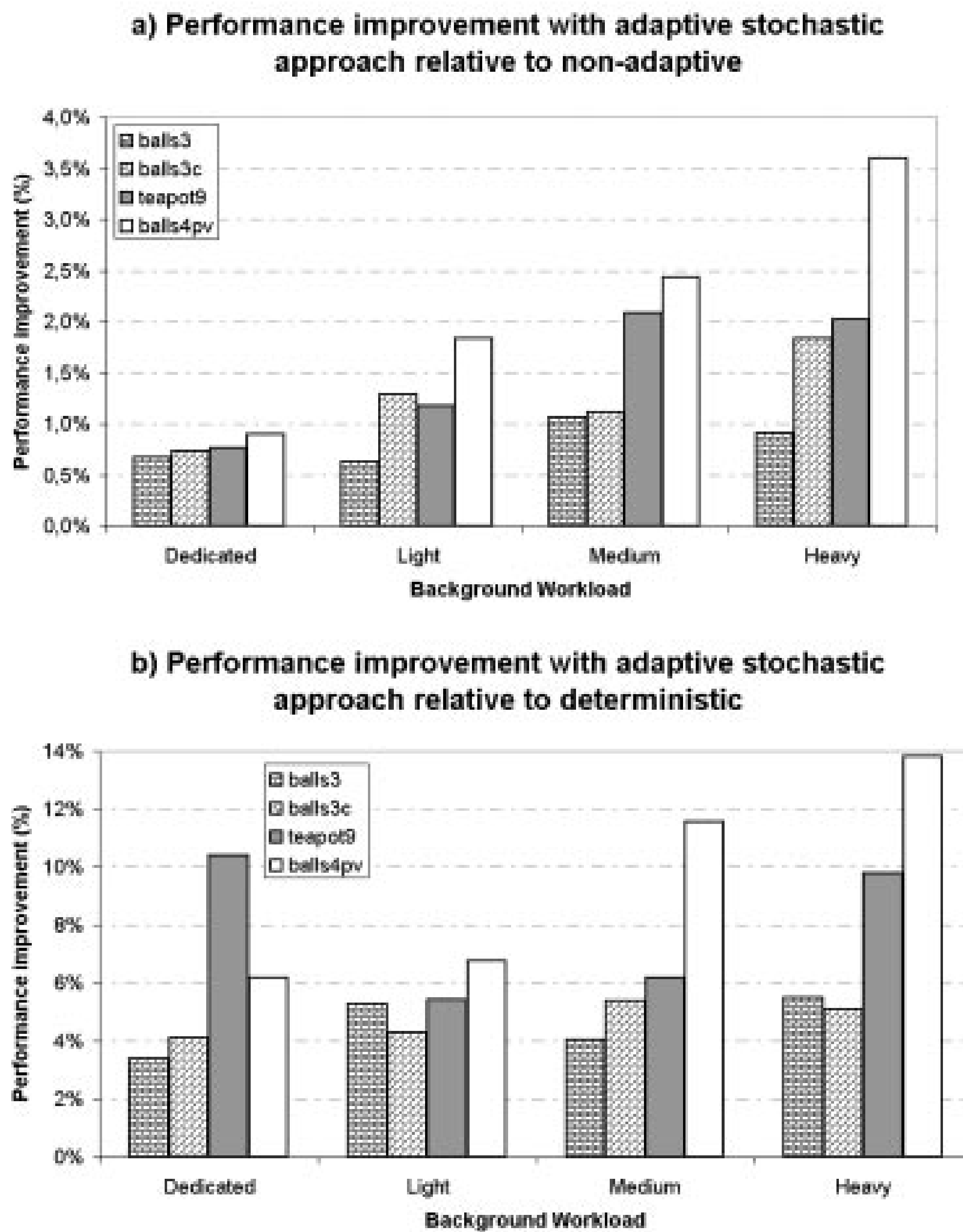


Figure 8.24: Performance improvements with adaptive stochastic approach
a) relative to non–adaptive
b) relative to deterministic

8.6.4 Discussion

The results presented throughout this chapter show that significant performance improvements are obtained by using sensor based dynamic scheduling strategies, rather than a static uniform workload distribution or even a dynamic demand–driven approach. This strategy is a very common approach to the scheduling problem, due to its simplicity. Figure 8.25 shows that the performance improvements obtained with both the determinis-

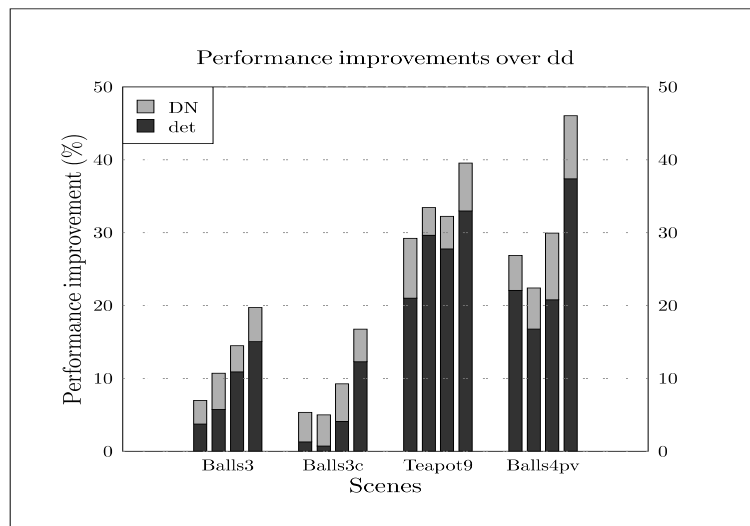


Figure 8.25: Performance improvements relative to demand driven with 7 nodes
For each scene the bars represent, from left to right, dedicated mode, light, medium and heavy background workloads

tic and the adaptive stochastic sensor based schedulers relatively to the demand–driven one increase, although not monotonously, with the complexity of the data set being processed and the background workload weigh, and that the stochastic scheduler is more effective than the deterministic one. The additional costs imposed upon the system to gather more detailed system's state data and by a more sophisticated decision making mechanism, are largely overcome by the better quality of the generated schedule. This conclusion contradicts some authors that claim that sensor based scheduling strategies are not likely to offer much improvements over very simple ones [25, 39, 182] (section 3.2). This statement is based on the fact that sensor based strategies rely on detailed information about the environment's state, which may be inaccurate and expensive to gather, and on the potential for instability. These authors, however, only consider small, homogeneous and dedicated distributed systems. Becker [10, 12], on the other hand, argues that sophisticated scheduling is more effective than simply assigning tasks to the least loaded resources, when the environment being managed is complex enough, which is confirmed by the results just presented. The efficiency of a sophisticated scheduling strategy may, however, fall below

acceptable levels if the overheads of detailed information collection and decision making overcome the associated benefits. The information policy and the decision making mechanism may not scale with the environment's heterogeneity and size. Different organisations and levels of sophistication may be required for these two components of the scheduler, depending on the environment's properties.

The performance improvements obtained with the stochastic scheduler on small dedicated distributed systems do not seem significant when compared with those obtained with the deterministic scheduler. Since their respective performance levels are similar, a straightforward conclusion is that the design of decision network based schedulers may not be worthwhile: although they are able to meet their performance requirements to the same degree as the deterministic ones, they require the assessment of a large number of numerical parameters (conditional probabilities and utilities), which the scheduler's designer, probably more used to build deterministic models, can find difficult to estimate.

The advantages of the stochastic scheduler become clear as soon as the environment gets more complex. The improvements grow with the scene's complexity, the background workload weigh and the size — number of nodes — of the distributed system (section 8.6.2). Furthermore, they are obtained by reducing both resources' idle times and the number of migrations triggered by the scheduler. This result is very encouraging, since it suggests that the decision network approach may be specially effective in larger and more complex environments, such as distributed environments on a wide Internet scale and the newly emerging computational Grid [50]. Applying this paradigm to these complex environments would require an adequate execution model and information policy. The additional communication costs — due to a high latency/low bandwidth network — and complexity — due to issues like heterogeneity and ownership — of such an environment would have to be taken into consideration; however, decision networks seem to be an appropriate alternative to deterministic models.

The measured values of resources' idle times and remote data access overheads suggest that there are still opportunities to further increase the overall scheduler's effectiveness (figure 8.23):

- the execution models used throughout these experiments do not explicitly handle communications costs, focusing on computation costs; however, indirect overheads due to remote data accesses increase significantly with the weigh of the background workload; further performance improvements may be obtained on heavily loaded systems, by including in the decision network the communication costs due to remote data accesses, to reduce these costs by increasing data locality;
- the schedulers used throughout this work do not make any effort to generate an optimised initial distribution of work, relying on runtime task migrations to prop-

erly redistribute the workload among the resources; an optimised initial distribution of work could reduce the number of required task migrations, decreasing direct scheduling overheads; this approach's effectiveness depends on the dynamics of the environment;

- the execution model may require further tuning, to more adequately represent the environment's behaviour; imprecisions may be present either in the model's qualitative component (network topology, random variables and respective domain, available actions) or in its quantitative component, i.e., conditional probability tables and utility function.

These issues are discussed in detail in chapter 9. The latter, however, has been partially tackled by using sequential update to recompute some of the decision network's probabilities. The assessment of the conditional probabilities can be particularly difficult, since humans are generally bad numerical estimators. These numbers often need only to be specified approximately. As long as the ratio between the probability of an event occurring and not occurring, given the same evidence, roughly reflects genuine experience, valid conclusions will still be reached [127]. If the network topology is built following a causal ordering, the required probabilities are psychological meaningful and this topology constitutes a qualitative knowledge base which is maintained independently of the numerical parameters, rendering the conclusions more dependable. Nevertheless, the required probabilities' correctness can be increased by updating them whenever new data is available (section 8.6.3). This process, known as sequential updating, has the additional capability of adapting the decision network's numerical parameters to changes in the environment's dynamics.

An adaptive version of the stochastic scheduler has been developed, which updates the prior probabilities $\mathbf{P}(Ir)$ whenever new data is available. The results achieved with this adaptive version are significantly better than those achieved with the deterministic one. This corroborates the hypothesis that the the decision network approach effectiveness can be increased by fine tuning the execution model's numerical parameters.

8.7 Summary

Chapter 6 defined a methodology to validate the hypothesis put forward by this thesis. A stochastic scheduling strategy was presented, and further refined in this chapter, to include on the scheduler's execution model the uncertainty about the environment's state and behaviour. It was shown that the scheduler's execution model successfully benefited from the use of Bayesian decision networks.

The final results also show that, by updating some of the model's numerical parameters based on past observations, significant improvements are achieved with the stochastic scheduler, when compared to a deterministic one with an equivalent level of complexity. Furthermore, these improvements grow with the environment's complexity.

Since the stochastic scheduler achieves better results than those of the deterministic scheduler, the hypothesis put forward in the first part of this thesis is corroborated and can not be rejected. Decision networks do constitute a suitable approach to the scheduling problem, particularly for highly parallel irregular applications, exhibiting low interaction among arbitrarily divisible tasks, and executing on distributed shared systems.

Chapter 9

Conclusions

Contents

9.1 Discussion 207
9.2 Future Work 210

9.1 Discussion

This thesis addresses application level scheduling in distributed dynamically shared systems. In such environments the scheduling agent must decide and act under conditions of uncertainty about the application's requirements, the distributed system's state and the consequences of its actions. Distributed shared systems are often dynamic, non–deterministic and partially inaccessible, preventing the scheduling agent from having a complete, accurate and updated image of the environment's state at any instant and from being completely sure about its actions outcomes. The problem tackled by this thesis is: should the scheduling agent explicitly include the uncertainty and incompleteness of information it has about the environment on its internal model of the world and on its decision making mechanism, in order to more tightly meet its performance requirements?

To improve the scheduler's effectiveness, decision networks were applied to the scheduling agent's execution model and decision making mechanism. Decision networks are one of the tools proposed by decision theory to enable automated rational decision making under conditions of uncertainty.

A generic structure for this decision network is proposed, which includes the entities that play a relevant role on the scheduling process: the environment's current state description — composed by the tasks' requirements, the resources' capacity and the scheduling overheads — the alternative actions available to the agent, the estimated next state and the expected utility for each action.

A subset of the proposed structure is evaluated, by comparing both its effectiveness and efficiency with those of a deterministic scheduling strategy of identical complexity. To perform this evaluation, a parallel ray tracer is used, representative of a broader class of parallel applications, exhibiting low interaction among tasks, high parallelism, irregular workload and decomposed across data space, which provides arbitrarily divisible tasks. The distributed computing system used as the target platform is a small, heterogeneous cluster. To simulate the workload generated by several simultaneous users, some predefined synthetic background workload patterns are applied.

An adaptive version of the stochastic scheduler is also evaluated, which uses an online learning process, referred to as sequential update, to learn some of the probabilistic model numerical parameters. This process allows the development of adaptive systems that can overcome errors in their initial model and adapt to changes in the dynamics of the environment being modelled.

The results presented throughout this thesis show that sensor based dynamic scheduling strategies, that include a rather sophisticated execution model of the environment being managed and that use dynamically gathered data about the state of this environment, obtain significant performance improvements when compared with simpler approaches, such as a dynamic demand driven strategy. The additional costs imposed upon the system by detailed information collection and a more sophisticated decision making mechanism are largely overcome by the better quality of the generated schedule.

The execution models used by both the deterministic and the stochastic schedulers do not explicitly handle communication costs. The tasks' requirements are characterised in terms of estimated time to completion, without distinguishing between computation and communication requirements, the data access patterns are not considered and the resources' capacity is measured in terms of computing throughput only. However, the results presented show that indirect overheads caused by remote data accesses increase significantly with the background workload weigh. This suggests that these overheads should be considered on the schedulers' execution models, in an attempt to reduce them by exploiting data locality.

The performance improvements obtained with the stochastic scheduler relative to the deterministic one, and in particular the adaptive scheduler, become increasingly significant as the environment's complexity grows. The complexity of the data set, the background workload weigh and the size of the distributed system contribute to increase the environment's complexity, providing the scheduler with more opportunities to intervene. The adaptive stochastic scheduler is able to explore these opportunities more effectively than the deterministic one. These additional improvements are obtained by reducing the number of task migrations, and by reducing indirect overheads, including resources' idle times and remote data fetching times.

When the stochastic scheduler is used to manage small, dedicated distributed systems the performance improvements over the deterministic one are not significant. The additional efforts required to assemble a probabilistic model, particularly the assessment of the model's numerical parameters, seem worthless in these simple environments. However, the improvements obtained in more complex, shared environments, with complex data sets and dynamic background workloads, do justify these efforts. The decision network approach gets more effective as the level of system sharing and the application's workload irregularity increase, as illustrated by figure 9.1. These results corroborate the hypothesis that the explicit representation of uncertainty on the scheduler's execution model and decision making mechanism, using decision networks, increases its effectiveness by overcoming the problems caused by this uncertainty. This was verified by the class of applications and distributed computing systems considered throughout this thesis.

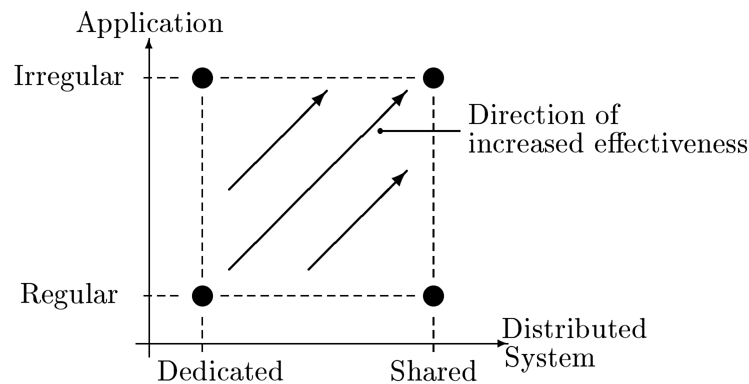


Figure 9.1: The decision network approach effectiveness

The effectiveness of the stochastic scheduler, particularly of the adaptive one, increases with the complexity of the environment being managed. This is an encouraging result, since it suggests that the decision network approach may be specially effective in complex and large environments, such as distributed environments on a wide Internet scale and the newly emerging computational Grid. These environments present new challenges that have to be properly handled; these include a large number of heterogeneous resources, a high latency/low bandwidth communication medium and a number of new issues like ownership and accounting. The centralised approach used throughout this thesis is probably inadequate for these environments. The large number of different and autonomous organisations involved prevents the utilisation of a single point of control, which could also become a bottleneck, compromising the system's scalability. Design choices like the decentralisation of the scheduling agent and the implementation of a decision making policy with adaptive complexity have to be considered. In order to keep the relationship between scheduling overheads and profits under control, the schedulers operating on more complex environments may perform a more sophisticated profitability determination analysis. This may be achieved by explicitly representing scheduling overheads in the execution model,

as proposed in section 5.7. With this extended execution model the scheduling agent will be able to estimate the scheduling overheads associated with a given action and weigh it against the potential gain obtainable with the same action. The relationship between cost and benefit could, thus, be automatically optimised [10, 12].

The main difficulty involved on the development of decision networks lies on the assessment of the numerical parameters, both probabilities and utilities. Identifying the relevant variables to represent in the model and the direct causal relationships holding among them is a fairly simpler task [127]. The absolute estimates of the probabilistic values, however, are unimportant, as long as their ratio reflects the conditions holding on the real world. Nevertheless, techniques developed by decision analysts can help the expert on this task. Sensitivity analysis, for example, helps in verifying how sensitive is the system to small variations in these values. This task may be made easier by using automated learning mechanisms, like sequential update or reinforcement learning, that generate new estimates by observing the system's behaviour.

9.2 Future Work

The approach to the scheduling problem proposed throughout this thesis has been applied to a single class of parallel applications. It remains an open question whether it can be successfully applied to different applications — exhibiting high interactions among parallel tasks, fixed grain size or hybrid functional plus data domain decomposition — or to problems with different performance requirements.

The initial allocation of work to the distributed system's resources has been neglected throughout this work, on the basis that the application's requirements and the system's behaviour are highly unpredictable and dynamic. If these can be modelled to some extent [8], then techniques from static scheduling can be used to generate an optimised initial schedule, which may require less task migrations, thereby reducing direct scheduling overheads.

The performance improvements achieved with the stochastic scheduler grow with the environment's complexity. This conclusion must be further validated by using more complex distributed computing systems. Experiments are already planned in a cluster with more processing nodes. The presented results suggest that the decision network approach would be more effective if used on a highly heterogeneous system, distributed in a wide Internet scale, probably composed of physically distributed clusters, owned, administered and used by different groups of people. These resources could be dynamically joined together on a virtual machine, using software like PVM or Legion [28, 62, 63, 177], and made available to the users on–demand or transparently. A stochastic scheduler would, hopefully, be able to

manage effectively this pool of heterogeneous resources, workloads and performance goals.

A centralised architecture is probably inadequate to larger environments. The central scheduling agent may represent a bottleneck, compromising the system's scalability. Distributed approaches must be investigated, since they scale better with system's size and activity. These can be either completely distributed or partition the system into domains. Inside each domain centralised scheduling strategies may be applied, while scheduling among domains can be done hierarchically or on a neighbourhood basis (sections 3.5 and 4.2.6). By applying decision networks to these distributed scheduling agents their performance goals may be more tightly met.

In such environments the scheduling overheads can be far more significant than in the environments used throughout this thesis; the scheduler must include in its execution model a profitability determination component, that weighs each action's benefits with the expected overheads. This can be achieved by explicitly including the scheduling overheads on the decision network. Furthermore, the environment's state is better described by including a more comprehensive set of metrics, such as the communication's network bandwidth and latency. This also applies to the parallel ray tracer, as discussed in section 8.6.4.

The stochastic scheduler used throughout this work is adaptive with respect to the node's computing throughputs, since their prior probabilities are dynamically computed using a sequential update technique. However, there are several different opportunities for adaptation for a dynamic scheduler, as identified by Becker and Waldmann [12] and discussed in section 2.3:

- correction of profile and load predictions;
- determination of diffuse factors in the execution model;
- control of the relationship between overhead and profit.

The first of these opportunities has been addressed by dynamically updating the nodes' computing throughputs prior probabilities. The third one can be addressed by including the expected scheduling overheads on the decision network and by dynamically updating some of the associated probabilities. The determination of diffuse factors will depend on the execution model being used. Although the agent may be given the ability to adapt many of its parameters, such as the Bayesian decision network's structure itself [18, 54, 56, 84, 91, 97, 98], the state transition model is an obvious candidate to adaptation.

The state transition model is a conditional probability table (CPT), that relates the environment's next state to its current state and the action selected by the agent. Since this CPT is initially assessed by the agent's designer, it may contain numerical imprecisions. These can be corrected by updating the conditional probabilities whenever new data is

available. This is a learning process, since it involves acquiring new knowledge from the environment, or reorganising current knowledge, in order to improve the agent's performance [35]. Furthermore, this is an unsupervised learning process, since no entity can tell the agent which is the best action to take at each instant. Either Bayesian learning techniques or reinforcement learning algorithms [166, 170, 183] can be applied to improve the state transition model accuracy.

Reinforcement learning algorithms use a reward–penalty scheme to update each action's probabilities; if, after selecting an action, the agent receives a feedback signal from the environment indicating that the action was successful, then increase the appropriate values in the CPT to increase the probability of this action being selected the next time the environment is in the same state; otherwise, decrease them. Reward–penalty schemes are presented in section 4.2.7. This is a promising technique, whose suitability to increase the scheduling agent's effectiveness deserves further investigation.

The decision networks and the probabilistic inference algorithm used throughout this work, require that all the model's variables are discrete random variables. However, some of the metrics are continuous values; representing them with discrete variables requires the discretisation of their domain. This process can lead to imprecisions on the model, particularly if the range of values each variable can take on is infinite or can not be previously determined. Representing these metrics as continuous random variables provides a natural way of expressing them, leading to models in better accordance with reality, which increases the models' clarity.

Several inference algorithms for decision networks with mixtures of discrete and continuous random variables have been developed [5, 88, 99, 119, 123, 131, 149]; unfortunately, exact probabilistic inference is only possible when all the continuous variables are normally distributed and have no discrete children. Directed graphical probabilistic models require the specification of the conditional distribution of each node, given its parents. For discrete variables with discrete parents, this distribution is usually represented using a conditional probability table. For continuous variables with continuous parents, the variable's distribution is given by $N(\mu + \beta Y, \sigma^2)$, where μ is the normal distribution mean, Y is the vector of states of the continuous parents, β is a vector of weighs and σ is the normal distribution variance. If the continuous variable also has discrete parents, then a distribution $N(\mu_i + \beta_i Y, \sigma_i^2)$ must be specified for each configuration i of the discrete parents. Representing the environment's continuous metrics as continuous variables, rather than discrete ones, increases the probabilistic model's clarity and capacity of expression, which can contribute to increase the scheduling agent's effectiveness. Furthermore, continuous models require less numerical parameters and these are more meaningful, which allows a more accurate initial assessment of these numbers and facilitates automated learning based on dynamically available data.

Appendices

Appendix A

Glossary

Contents

A.1 Decision Theory 215
A.2 Load Management 218

This appendix presents a brief definition of the terms most frequently used throughout this thesis. These are subdivided into three sections, according to the subject with which they are related.

A.1 Decision Theory

Bayes' rule — This rule states that the belief on an hypothesis, or cause, H, given the available evidence, or effects of this cause, $\mathbf{E}$, can be computed by multiplying the previous belief on the hypothesis $\mathbf{P}(H)$ by the likelihood that $\mathbf{E}$ will materialise given the hypothesis $\mathbf{P}(\mathbf{E}|H)$. This is given by

$$\mathbf{P}(H|\mathbf{E}) = \frac{\mathbf{P}(\mathbf{E}|H)\mathbf{P}(H)}{\mathbf{P}(\mathbf{E})}$$

The heart of Bayesian inference techniques lies in this inversion formula, which enables diagnostic inferences, from effects to causes, on Bayesian networks where the available information is causal, i.e., from causes to effects.

bayesian network — Bayesian networks are a tool proposed by decision theory to probabilistically model real world problems. The network's nodes represent the aspects of the real world that are being modelled, while the network's directed links represent direct causal relationships among them. The network's nodes are associated with stochastic variables U_i, which describe the network's current belief on the state of

each of the world's relevant aspects. The network's links are associated with conditional probability tables, which represent the strength of the influence of a variable's state on its successors' states. The network topology, once built, displays a consistent set of direct and indirect dependencies, and preserves them as a stable qualitative characteristic of the model, independently of any particular assignment of quantitative information, namely conditional probabilities. The network topology can be thought of as an abstract knowledge base, that holds in a variety of settings, representing the domain's general structure of causal relationships [69, 127, 143]. Evidence about the current state of one or more variables can be entered into the network, and a probabilistic inference mechanism can be triggered to compute the new probability distribution over the remaining variables.

Formally, a Bayesian network is an annotated directed acyclic graph, constituted by the pair $B = (G, \Theta)$, that encodes a joint probability distribution of a set of random variables $\mathbf{U}$. G is a directed acyclic graph whose vertices correspond to the random variables $U_1, \ldots, U_n$ and whose edges represent direct dependencies among the variables. Θ represents the set of numerical parameters that quantify the network. It contains parameters $\theta_{u_i|Parents(U_i)} = P_B(u_i|Parents(U_i)), \forall_{u_i \in D_{U_i}, U_i \in U}$, where $Parents(U_i)$ denotes the set of parents of U_i in G. A Bayesian network B defines a unique joint probability distribution over $\mathbf{U}$ given by $P_B(U_1, \ldots, U_n) = \prod_{i=1}^{n} \mathbf{P_B}(U_i|Parents(U_i))$ [55].

d–separation — Stands for direction–dependent separation and provides a mechanism for directly reading from a bayesian network whether or not two variables are conditionally independent, given the available evidence [77, 84, 127, 143]. Two variables X and Y in a belief network are said to be d–separated, if for all paths between X and Y, there is an intermediate variable V, such that:

1. the connection is serial, $\rightarrow V \rightarrow$, or diverging, $\leftarrow V \rightarrow$, and the state of V is known;
2. the connection is converging, $\rightarrow V \leftarrow$, and neither V nor any of its descendants are known.

If, given all the evidence available $\mathbf{E}$, two variables X and Y are d–separated, then changes on the belief of X have no impact on the belief of Y, and conversely. Therefore, if X and Y are d–separated by $\mathbf{E}$, then they are conditionally independent of each other given $\mathbf{E}$, i.e.,

$$\mathbf{P}(X|\mathbf{E}, Y) = \mathbf{P}(X|\mathbf{E})$$
$$\mathbf{P}(Y|\mathbf{E}, X) = \mathbf{P}(Y|\mathbf{E})$$

decision networks — Decision networks are a tool proposed by decision theory for rational decision making under conditions of uncertainty. These networks combine belief

networks with additional node types for actions and utilities. The set of actions available to the agent at any given instant can be represented by decision variables that are under the full control of the decision making agent. Selecting an action amounts to impose the value of a decision variable, rather than determine it probabilistically. This setting alters the probability distribution of another set of stochastic variables in the network, known as the consequences of the decision variable. The utility function can then be evaluated, taking into account the probability distribution over those variables that directly affect utility. A rational agent should assert the decision variables to all possible combinations, and select the sequence that maximises its expected utility.

joint distribution — A probabilistic model consists of a set of stochastic variables that can take particular values with certain probabilities. Each variable represents some important aspect of the world being modelled. The values each variable can take on are the variable's domain. The variable's probability distribution over its domain specifies the designer's degree of belief that the variable will take on a particular value. An atomic event is an assignment of particular values to all the variables in the model. The joint distribution assigns probabilities to all possible atomic events, and it allows direct access to the probability of any atomic event. The cardinality of the joint distributions is given by

$$\prod_{i=1}^{N} \#D_{V_i}$$

where N is the number of stochastic variables in the model and $\#D_{V_i}$ is the cardinality of V_i's domain.

sequential updating — Sequential updating is an online learning process. At each iteration n the Bayesian network B_n receives new data u_n and produces the next hypothesis B_{n+1}, which is then used to predict u_{n+1}. In practice, at each iteration n each variable of interest X_i is assigned a given prior probability $\mathbf{P}_n(X_i)$. Upon reception of new evidence $\mathbf{E}$, the posterior probabilities $\mathbf{P}_n(X_i|\mathbf{E})$ are inferred. The prior probabilities for each variable X_i can be updated using the old estimate $\mathbf{P}_n(X_i)$ and the newly computed posterior probabilities $\mathbf{P}_n(X_i|\mathbf{E})$ by using the exponentially recency-weighed average, as given by

$$\mathbf{P}_{n+1}(X_i) = \mathbf{P}_n(X_i) + \alpha * [\mathbf{P}_n(X_i|\mathbf{E}) - \mathbf{P}_n(X_i)]$$

This is a crucial capability for building adaptive models that can overcome errors in the initial model numerical parameters, and that can adapt to the dynamics of the underlying system [55, 164].

utility function — Utility theory is used to represent and reason about preferences. It says that each state has an utility for an agent, and that the agent will prefer states

with higher utility. Utility is a function that maps system's states to real numbers [77, 127, 143]. The agent's preferences among different states are captured by this function, which assigns a single number to express the state's desirability. Utility imposes a preferential ordering on the system's states. Every utility function can be normalised, such that the most preferable state has an utility $U(S) = 1$, and the least preferable one has an utility $U(S) = 0$. Most real problems require the system's state to be characterised by many different variables, or attributes. In such cases, it is necessary to resort to multiattribute utility theory, in order to specify the utility function. If the system's state is described by variables $X_1, \ldots, X_n$, the multiattribute utility function is usually an additive value function

$$U(S) = \sum_{i=1}^{n} \alpha_i X_i$$

Additive functions are a natural way of expressing an agent's preferences, and are valid in many real world problems. These functions can be safely used when the attributes exhibit mutual preference independence, i.e., when each attribute does not affect the way in which the agent trades off the other attributes against each other. Two attributes X_1 and X_2 are preferentially independent of a third attribute X_3, if the preference among the outcomes (x_1, x_2, x_3) and (x'_1, x'_2, x_3) does not depend on the particular value x_3 for attribute X_3. However, when mutual preference independence does not strictly hold, an additive function can still be a good approximation to the agent's preferences.

A.2 Load Management

application–dependent metrics — Application–dependent metrics are those which use the actual work performed by the application to measure some particular aspect of the environment. On an image processing or graphic application, for instance, this could be the number of pixels processed by unit of time. Application–dependent metrics usually convey more information than application–independent ones. To avoid dependencies among the actual data being processed by each node and the metric's values, data templates, equal in all nodes, can be used to compute the metric. This approach will, however, impose an additional overhead, since computing the metric does not directly contribute to finish the task in hand. The benefits of using an application–dependent metric must be carefully weighed with the overheads of computing it.

application–independent metrics — Application–independent metrics refer to characteristics of the system, like each resource's waiting queue or the network's latency, rather than to the application's characteristics. These can be computed either by

the applications themselves or by some system specific software, eventually at the operating system level.

application level scheduling — Scheduling performed by the application itself in order to meet its own performance objectives, while in competition with any other applications that may be sharing the same distributed system [14, 19]. Effective application level scheduling involves the integration of application-specific and system-specific data, and it depends on the dynamic interactions between the application and the computing system. The scheduler, embedded within each application, must effectively combine the data it receives about the distributed system and the tasks' states in order to meet its performance requirements.

background workload — The workload associated with all those processes that are not under the application level scheduling agent's control, like other applications, eventually belonging to different users, and operating system processes.

congestion problem — Formation of regions of nodes within the distributed system with an excessive workload, when compared with the remaining nodes. This problem may be difficult to solve, specially with schedulers using local information and migration policies, like nearest–neighbour ones.

coordination problem — Occurs when several overloaded nodes simultaneously select the same underloaded node as their migration target, overflowing it with tasks.

data locality — Addresses the fact that tasks, being executed by the distributed system's nodes, require data, which can be either located in the node's local memory, or located in remote memories. If the access times to remote memories are significant, the execution time may be severely impaired if data locality is low, due to the increased number of remote data fetches.

direct costs — Direct scheduling overheads represent the resources the scheduler consumes, since they result directly from the scheduler's activity. They depend on the scheduler's strategy complexity and on the frequency with which it triggers its various mechanisms. These include metrics' estimation, information and control messages, selection of the most adequate action to take at each instant and execution of the selected actions.

divisible loads — If data domain decomposition has been used to parallelise an application, then the application's tasks are defined by the subset of data that each task must process. If the data set can be arbitrarily subdivided at run–time into smaller regions, therefore originating an arbitrary number of tasks, then the workload is classified as an arbitrarily divisible load.

dynamic scheduling — Dynamic scheduling policies generate the schedule at run time, using a set of rules to specify correcting actions that redistribute the workload over the system. Among dynamic policies three different approaches can be distinguished:

- those that do not consider the environment's state at each instant, deciding as if they were blind;
- those that use environment's state information as input to their set of rules, hoping to make better decisions;
- those that go a step beyond, by using environment's state inputs to modify either its rules' parameters or the rules themselves, i.e., the scheduling agent's execution model is modified in run–time in order to better represent the external world; these are usually classified as adaptive policies.

Those that use environment's state information on their decision making process are also referred to as sensor based dynamic scheduling policies, since they use data gathered through the scheduler's sensors.

environment metrics — Metrics used to update the image the scheduler has about the environment's current state. These may include the distributed system's state, the application's current state of execution and workload profiles. These quantities are, usually, used directly by the agent's decision making mechanism. Environment metrics can be further subdivided into foreground workload metrics and resources' capacity metrics.

execution model — To be able to generate an effective schedule, the scheduler must have an internal execution model of the world, that adequately represents the distributed system and the workload's most relevant aspects and respective interrelationships. This execution model is used to generate estimates of future behaviours that are used as inputs to the scheduler's decision making mechanism. Being a computational representation of a real world problem, the execution model must be a simplification of the objects and relations that hold on the universe being considered. This simplification may result in inaccuracies on the generated estimates, that must be properly handled by the scheduler, so that its effectiveness does not get compromised.

foreground workload — The workload associated with all those processes that are under the application level scheduling agent's control.

foreground workload metrics — Metrics used to measure each resource's current workload that was directly assigned by the application level scheduler. They must correlate well with tasks' response times, since they are used to predict a task's performance if executed at a given resource.

indirect costs — Indirect scheduling costs are consequences of the scheduler's selected actions. By changing the resources allocated to the application's tasks, the scheduling agent modifies both the application's course of action and the pattern of resources' utilisation. These changes on the environment's behaviour may cause additional overheads that reduce the scheduler's efficiency. Indirect costs depend on the application and distributed system being managed. Three different kinds of indirect costs, however, occur for many applications, in distributed memory parallel systems: work replication, resources' idle times and remote data access overheads.

information aging — Since the environment's state changes continuously, the image an agent has about it gets obsolete with time. This phenomenon is known as information aging. Increasing the rate at which information is acquired can be prohibitively expensive, and the problem can never be completely solved since the environment's state changes while the information is being transmitted.

information policy — Set of rules that determine when, where and what information must be collected, through the scheduling agent's sensors, about the system's resources current workload and tasks' workload profiles.

location policy — Set of rules that identify the various partners enrolled on a given scheduling action, e.g., workload transfer.

measurability — The environment's measurability is a function of the total set of information available to the scheduling agent and the cost of acquiring this information. The more information available, the more measurable the environment is.

performance metrics — Metrics used to evaluate the scheduler's performance goals degree of achievement, or effectiveness.

performance model — In order to both make decisions and assess its efficiency and effectiveness, the scheduling agent must collect a set of metrics. This set of metrics is referred to as the scheduler's performance model. These metrics are acquired through the agent's sensors, and constitute all the dynamic external information it has about the environment's state and the quality of the schedule it is generating. They can be subdivided into three groups: performance, environment and scheduling overheads metrics.

profitability determination — Profitability determination is the assessment of whether or not a given workload transfer among resources should take place, by weighing the expected benefits with the overheads incurred with this action.

resources' capacity metrics — Metrics used to assess the background workload on each resource, in contrast to load metrics, which measure the workload directly assigned by the application level scheduling agent. These values must correlate with

the performance degradation that results from sharing the resource with several processes.

scheduling effectiveness — The effectiveness is the scheduler's performance goals degree of achievement. This is closely related to the quality of its decisions, and can be described as the application's satisfaction with how well the scheduler manages the resources in question. The effectiveness of the scheduler's decisions can only be evaluated by using suitable *performance metrics*.

scheduling efficiency — Efficiency is a measure of the scheduling overheads. It is related to the application's satisfaction in terms of how costly it is to be serviced by the scheduler and to the level of intrusion that it imposes on the system.

scheduling overhead metrics — Metrics used to quantify the overheads, both direct and indirect, imposed by the scheduler upon the distributed system; they can be used in two different ways: either by the scheduler's designer to analyse *a posteriori* the scheduler's efficiency and eventually change its algorithm, or by the scheduling agent itself to automatically adapt its strategy in order to minimise overheads and maximise benefits, i.e., to optimise its efficiency.

scheduling problem — The scheduling problem is the problem of mapping a set of tasks onto a set of resources. Local, or intranode, scheduling is concerned with scheduling within a single node. Global, or internode, scheduling is a level above this, and is concerned with scheduling among nodes. The scheduling problem has five components: the tasks (or relevant events related to them), the distributed system, the performance requirements, the schedule and the scheduler. The tasks' events, system's characteristics and performance requirements are the input to the scheduler, or load manager, and the mapping, or schedule, is its output.

scalability — Scalability can be defined as a system's ability to achieve a performance which is proportional to its hardware and software resources' capabilities. According to Hwang [73], a computer system is scalable if it can scale up (improve its resources) to accommodate ever increasing performance and functionality requirements and/or scale down (decrease its resources) to reduce costs.

selection policy — Set of rules that decide which work must be transferred among the partners identified by the scheduler's location policy.

stability — Stability is the scheduler's ability to detect when the effects of further actions (which consume the resources being scheduled) will not improve the environment's state. A stable algorithm will return the environment to a state of equilibrium after a perturbation from this equilibrium and, in the absence of further input, ceases to take actions which cause changes in environment's state in finite time. In the context of

load management, a perturbation is caused by a sudden change in the environment's behaviour, due either to the arrival or removal of tasks or background workload, or to modifications on the tasks' activity, which may cause imbalances between the nodes' loads. Instability relates to the amount of schedulable resources being consumed by the scheduler, while the environment's state is changing, but not moving towards a more stable state.

static scheduling — Static scheduling policies generate the schedule before execution time, based on the system's properties and on the tasks' requirements. This is also called the mapping problem, because a mapping function must be defined, which assigns tasks to resources before the execution begins.

transfer policy — Set of rules that determine whether a given resource is in a suitable state to participate on a workload transfer, either as a sender or a receiver of work.

Appendix B

Propagation Rules for Bayesian Networks

Contents

B.1 Notation 225
B.2 Propagation Rules for Chains 227
B.3 Propagation Rules for Trees 228
B.4 Propagation Rules for Polytrees 231
B.5 An Example: The Burglary Alarm 233

This appendix presents the algorithm proposed by Judea Pearl [127] for probabilistic inference on Bayesian Belief Networks. This algorithm is based on local message passing between neighbouring nodes. It is suitable only for polytrees, i.e., networks which do not contain undirected cycles. If the network has undirected cycles, local message passing algorithms run the risk of double counting information arriving from a common source. Furthermore, all the model's variables must be discrete stochastic variables.

It introduces all the relevant notation and presents the propagation rules for chains, trees and polytrees. Section B.5 finishes with an example that illustrates the algorithm with a simple network.

B.1 Notation

Discrete random variables are denoted by capital letters (e.g X, Y, Z), whereas specific values taken by these variables are represented by lowercase letters (e.g. x, y, z). A random variable X may take on values from a finite domain D_X. The number of different values a variable X can take, i.e., the cardinality of its domain D_X, is labelled by $\#D_X$.

Sets of variables are denoted by boldfaced uppercase letters (e.g. $\mathbf{U}$), and assignments of values to these variables are denoted by boldfaced lowercase letters (e.g. $\mathbf{u}$). If $\mathbf{Z}$ stands for the set of variables $\{X, Y\}$, then $\mathbf{z}$ represents the assignment $\{x, y\} : x \in D_X, y \in D_Y$.

$P(x)$ is used as a short notation for the probability $P(X = x), x \in D_X$. $P(\mathbf{z})$, for the set $\mathbf{Z} = \{X, Y\}$, means

$$P(\mathbf{Z} = \mathbf{z}) = P(X = x, Y = y), x \in D_X, y \in D_Y$$

i.e., the probability that $X = x$ **and** $Y = y$.

The probability distribution of a variable X over its domain $D_X = \{x_1, x_2, \ldots, x_n\}$ is denoted by the boldfaced operator $\mathbf{P}(X)$, representing the vector

$$\mathbf{P}(X) = \{P(X = x_1), P(X = x_2), \ldots, P(X = x_n)\}, \sum_i P(X = x_i) = 1$$

To distinguish between the fixed conditional probability table associated with each of the network's links (that describes the variable X probability distribution as a tabular function of all possible combinations of its parents) and a variable's probability distribution given what is known, the first is referred to as $\mathbf{CPT}(X|Parents(X))$ and the latter as $\mathbf{P}(X|Y, Z, \ldots)$. Moreover, the dynamic values of the updated nodes' probabilities, inferred using the algorithm described throughout this appendix, that reflects the belief distribution accorded to X by all existing evidence $\mathbf{E}$, is referred to as $\mathbf{BEL}(X)$. Thus

$$\mathbf{BEL}(X) = \mathbf{P}(X|\mathbf{E})$$

The conditional probability $\mathbf{CPT}(X|Parents(X))$, with $Parents(X) = Y$, is represented by a two–dimensional matrix with $\#D_Y$ rows and $\#D_X$ columns. If $D_X = \{x_1, x_2\}$ and $D_Y = \{y_1, y_2, y_3\}$ then,

$$\mathbf{CPT}(X|Y) = \begin{bmatrix} P(x_1|y_1) & P(x_2|y_1) \\ P(x_1|y_2) & P(x_2|y_2) \\ P(x_1|y_3) & P(x_2|y_3) \end{bmatrix}$$

where the values across each row add up to one, i.e., $\sum_{i=1}^{2} P(x_i|y_j) = 1, \forall y_j \in D_Y$.

The symbol α denotes a normalizing constant, e.g.,

$$\alpha(2, 2, 4) = (0.25, 0.25, 0.5)$$

All incoming evidence is represented by $\mathbf{e}$ and is regarded as coming from a set $\mathbf{E}$ of variables whose value is known. $\mathbf{E}_X^+$ and $\mathbf{E}_X^-$ label evidence as arriving from, respectively,

ancestor and descendant variables of X. $\mathbf{E} = \mathbf{E}_X^+ \cup \mathbf{E}_X^-$ and X separates $\mathbf{E}_X^+$ from $\mathbf{E}_X^-$, therefore

$$\begin{aligned} \mathbf{P}(X|\mathbf{e}) &= \mathbf{P}(X|\mathbf{e}_X^+, \mathbf{e}_X^-) = \alpha \mathbf{P}(\mathbf{e}_X^-|X, \mathbf{e}_X^+)\mathbf{P}(X|\mathbf{e}_X^+) \\ &= \alpha \mathbf{P}(\mathbf{e}_X^-|X)\mathbf{P}(X|\mathbf{e}_X^+) \end{aligned}$$

The symbol λ represents the diagnostic support that a descendant node attributes to its parent. Therefore, $\lambda(x) = P(\mathbf{e}_X^-|x)$. This vector is usually referred to as the likelihood vector. On a network with two nodes and a single link $X \rightarrow Y$, the belief distribution over X is given by

$$\begin{aligned} \mathbf{BEL}(X) &= \alpha \mathbf{P}(X)\lambda(x) \\ \lambda(x) &= \mathbf{CPT}(Y|X)\lambda(y) \end{aligned}$$

$\lambda(x)$ is computed at node Y and transmitted to node X. Since Y has no descendants its likelihood vector $\lambda(y)$ is set to whatever is known about Y. If $D_Y = \{y_1, y_2, y_3\}$ and nothing is known about Y, then $\lambda(y) = (1, 1, 1) = \mathbf{1}$. But if it is known for sure that $Y = y_2$, then $\lambda(y) = (0, 1, 0)$.

The symbol π represents the causal support that a parent node attributes to its child. Therefore, $\pi(x) = P(x|\mathbf{e}_X^+)$. Considering again a simple network of two nodes and a single link $X \rightarrow Y$, the belief distribution is given by

$$\begin{aligned} \mathbf{BEL}(X) &= P(X|\mathbf{e}_X^+, \mathbf{e}_X^-) = \alpha P(X|\mathbf{e}_X^+)P(\mathbf{e}_X^-|X) \\ &= \alpha \pi(x)\lambda(x) \\ \mathbf{BEL}(Y) &= P(Y|\mathbf{e}_Y^+, \mathbf{e}_Y^-) = \alpha P(Y|\mathbf{e}_Y^+)P(\mathbf{e}_Y^-|Y) \\ &= \alpha \pi(y)\lambda(y) \\ \pi(y) &= \pi(x)\mathbf{CPT}(Y|X) \end{aligned}$$

Since X has no parents, $\pi(x)$ is set to the prior probabilities of X, i.e., $\pi(x) = \mathbf{P}(X)$.

B.2 Propagation Rules for Chains

A chain is a network where each node has exactly one parent and one child, except for the head and tail nodes which have, respectively, only one parent and one child (see figure B.1).

Local belief updating can be done by each node in three steps which can be executed in any order. The method for belief updating is triggered whenever a message is received from any of the node's direct neighbours.

Step 1 — Belief updating: Node X might receive either an update vector $\lambda(x)$ from its descendant Y, or an update vector $\pi(u)$ from its parent U. If $\pi(u)$ is received, then

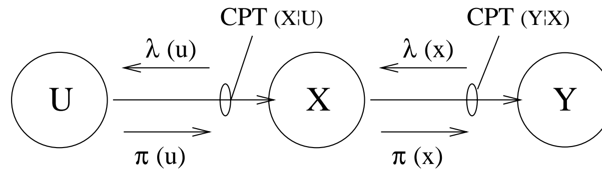


Figure B.1: A Bayesian network with a chain topology

$\pi(x)$ must be computed,

$$\pi(x) = \pi(u)\mathbf{CPT}(X|U) \tag{B.1}$$

Belief updating can now take place

$$\mathbf{BEL}(X) = \alpha\pi(x)\lambda(x) \tag{B.2}$$

Step 2 — Bottom-up propagation: When a $\lambda(x)$ message is received from the node's child, a new message $\lambda(u)$ must be computed and sent to the parent U:

$$\lambda(u) = \mathbf{CPT}(X|U)\lambda(x) \tag{B.3}$$

Step 3 — Top-down propagation: When a $\pi(u)$ message is received from the node's parent, $\pi(x)$ must be computed, using equation B.1, and sent to the child Y.

Figure B.2 illustrates an individual node's structure.

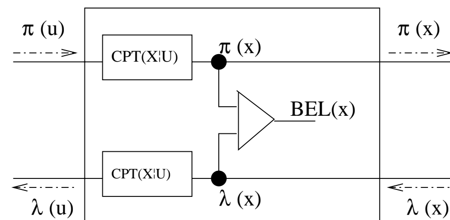


Figure B.2: Structure of an individual node on a chain network

B.3 Propagation Rules for Trees

On tree structured networks each node might have several children and only one parent (figure B.3). Each node must combine the impacts of λ messages obtained from several children, and must distribute separate π messages to each of its children.

The belief distribution over X depends on two distinct sets of evidence: diagnostic evidence from the sub–tree rooted at X and causal evidence from the remainder of the tree. The former evidence is obtained from each of X's children in the form of likelihood vectors $\lambda_{Y_i}(x) = \mathbf{P}(\mathbf{e}^-_{Y_i}|X)$, while the latter is obtained from U as the causal support vector $\pi_X(u) = \mathbf{P}(X|\mathbf{e}^+_X)$.

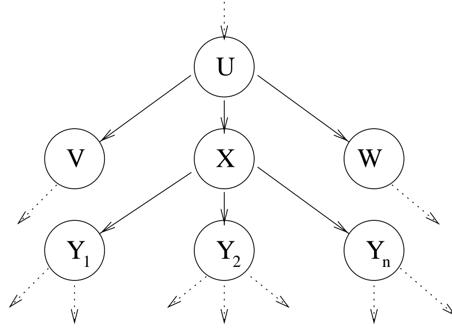


Figure B.3: A Bayesian network with a tree topology

Data Fusion

To compute the belief induced on X by some evidence $\mathbf{e} = \mathbf{e}_X^+ \cup \mathbf{e}_X^-$, $\lambda(x)$ and $\pi(x)$ must be calculated

$$\begin{aligned}\lambda(x) &= \mathbf{P}(\mathbf{e}_X^- | X) \\ \pi(x) &= \mathbf{P}(X | \mathbf{e}_X^+)\end{aligned}$$

and combined together

$$\mathbf{BEL}(X) = \alpha\lambda(x)\pi(x)$$

Since there is only one parent, the causal support is computed exactly the same way as for chain networks, thus

$$\pi(x) = \pi(u)\mathbf{CPT}(X|U)$$

The information arriving from the various descendants of X $(\lambda_{Y_1}(x), \ldots, \lambda_{Y_n}(x))$ must be combined. The sub–tree rooted at X can be partitioned into the root X and one sub–tree for each of its children. If X itself is not initialised with evidence, then $\mathbf{e}_X^- = \mathbf{e}_{Y_1}^- \cup \ldots \cup \mathbf{e}_{Y_n}^-$, and since X separates its children

$$\begin{aligned}\lambda(x) &= \mathbf{P}(\mathbf{e}_X^- | x) \\ &= \mathbf{P}(\mathbf{e}_{Y_1}^-, \ldots, \mathbf{e}_{Y_n}^- | x) \\ &= \mathbf{P}(\mathbf{e}_{Y_1}^- | x) * \ldots * \mathbf{P}(\mathbf{e}_{Y_n}^- | x) \\ &= \lambda_{Y_1}(x) * \ldots * \lambda_{Y_n}(x) \\ &= \prod_i \lambda_{Y_i}(x)\end{aligned}$$

Propagation Mechanism

The message that X sends to its parent must include all the diagnostic support given by the sub–tree rooted at X, i. e.,

$$\lambda_X(u) = \mathbf{CPT}(X|U)\lambda(x)$$

The message that X sends to its descendant Y_i must include all the causal support given by X's parent and all diagnostic support induced by all descendants of X except Y_i. This last condition prevents double counting of evidence. Pearl [127] shows that this can be computed as

$$\pi_{Y_i}(x) = \alpha \frac{\mathbf{BEL}(X)}{\lambda_{Y_i}(x)}$$

There is no need to normalise π or λ messages, only $\mathbf{BEL}()$ requires normalisation. The sole purpose of the normalisation constant α is to preserve the probabilistic meaning of the messages. It is a good practice, however, to encode messages so that the smallest element of the vector is 1.

Summary of Propagation Rules

Local belief updating can be done by each node in three steps which can be executed in any order. The method for belief updating is triggered whenever a message is received from any of the node direct neighbours.

Step 1 — Belief updating: Node X might receive either update vectors $\lambda_{Y_i}(x)$ from its descendants, or an update vector $\pi(u)$ from its parent U. It can then update its belief distribution

$$\mathbf{BEL}(X) = \alpha \pi(x) \lambda(x)$$

where

$$\begin{aligned} \pi(x) &= \pi(u)\mathbf{CPT}(X|U) \\ \lambda(x) &= \prod_i \lambda_{Y_i}(x) \end{aligned} \tag{B.4}$$

Step 2 — Bottom-up propagation: Using $\lambda(x)$ a new message $\lambda(u)$ must be computed and sent to the parent U:

$$\lambda(u) = \mathbf{CPT}(X|U)\lambda(x)$$

Step 3 — Top-down propagation: X must compute new π messages to send to its descendants Y_i:

$$\pi_{Y_i}(x) = \alpha \frac{\mathbf{BEL}(X)}{\lambda_{Y_i}(x)} \tag{B.5}$$

Figure B.4 illustrates an individual node's structure.

Root, terminal and evidence nodes require special treatment:

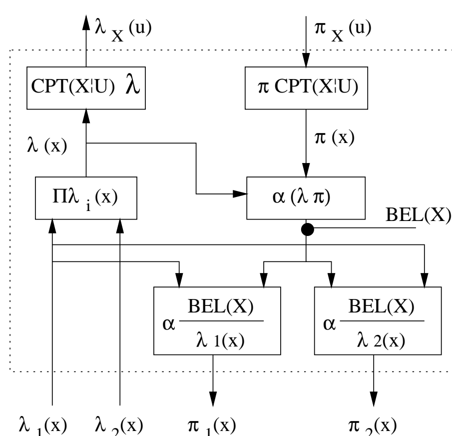


Figure B.4: Structure of an individual node on a tree network

Anticipatory node – a leaf node that has not been initialised. **BEL** must be equal to π, so $\lambda = (1, 1, \ldots, 1)$;

Evidence node – if a variable is initialised with a value, then λ must be set with a 1 at that value's position (e.g. $\lambda = (0, 1, 0, \ldots, 0)$);

Root node – π of the root node must be set equal to this variable's prior probabilities.

B.4 Propagation Rules for Polytrees

A polytree is a singly connected network, where each node might have several parents and children, but no more than one undirected path might exist between any two nodes (figure B.5).

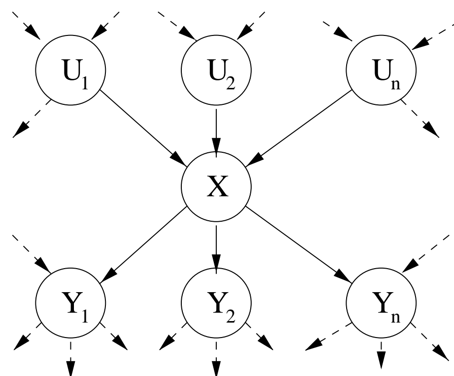


Figure B.5: A Bayesian network with a polytree topology

The belief distribution over X depends on two distinct sets of evidence: diagnostic evidence from the sub–tree rooted at X and causal evidence from the remainder of the polytree. The former evidence is obtained from each of X's children in the form of likelihood vectors

$\lambda_{Y_i}(x) = \mathbf{P}(\mathbf{e}^-_{Y_i}|X)$, while the latter is obtained from each of the parents U_i as causal support vectors $\pi_X(u_i) = \mathbf{P}(X|\mathbf{e}^+_{U_i})$.

Data Fusion

The information from the various descendants of X $(\lambda_{Y_1}(x), \ldots, \lambda_{Y_n}(x))$ is combined exactly the same way as for trees,

$$\lambda(x) = \prod_i \lambda_{Y_i}(x)$$

The information from the various parents of X $(\pi_X(u_i))$ must also be combined on $\pi(x)$. Pearl [127] shows that

$$\pi(x) = \prod_i \pi_X(u_i)\mathbf{CPT}(X|\mathbf{U}) \tag{B.6}$$

The belief distribution over X can be calculated as usually by

$$\mathbf{BEL}(X) = \alpha\lambda(x)\pi(x)$$

Propagation Mechanism

The message $\lambda_X(u_i)$ that X sends to its parent U_i must include all diagnostic support induced by the sub–tree rooted at X (given by $\lambda(x)$) and all causal support induced by all its other parents except U_i. Pearl [127] shows that

$$\lambda_X(u_i) = \alpha \sum_x [\lambda(x)(\sum_{u_k:k\neq i} \mathbf{CPT}(X|\mathbf{U}) \prod_{k\neq i} \pi_x(u_k))] \tag{B.7}$$

The message that X sends to its descendant Y_i is computed as in the previous case,

$$\pi_{Y_i}(x) = \alpha\frac{\mathbf{BEL}(X)}{\lambda_{Y_i}(x)}$$

Summary of Propagation Rules

Local belief updating can be done by each node in three steps which are executed in any order. The method for belief updating is triggered whenever a message is received from any of the node's direct neighbours.

Step 1 — Belief updating: Node X might receive either update vectors $\lambda_{Y_i}(x)$ from its descendants, or update vectors $\pi_X(u_i)$ from its parents U_i. It can then update its belief distribution

$$\mathbf{BEL}(X) = \alpha\pi(x)\lambda(x)$$

where

$$\pi(x) = \prod_i \pi_X(u_i)\mathbf{CPT}(X|\mathbf{U})$$
$$\lambda(x) = \prod_i \lambda_{Y_i}(x)$$

Step 2 — Bottom-up propagation: A new message $\lambda_X(u_i)$ must be computed and sent to each parent U_i, according to equation B.7:

$$\lambda_X(u_i) = \alpha \sum_x [\lambda(x)(\sum_{u_k:k\neq i} \mathbf{CPT}(X|\mathbf{U}) \prod_{k\neq i} \pi_x(u_k))]$$

Step 3 — Top-down propagation: X must compute new π messages to send to its descendants Y_i:

$$\pi_{Y_i}(x) = \alpha \frac{\mathbf{BEL}(X)}{\lambda_{Y_i}(x)}$$

The special treatment required by root, terminal and evidence nodes is the same as for tree networks.

B.5 An Example: The Burglary Alarm

B.5.1 The Probabilistic Model

Judea Pearl [127] presents a probabilistic model of a burglary alarm, which is used to exemplify the proposed algorithm. This alarm can go off due to a burglary or due to an earthquake. Three random variables are identified, namely, Burglary (B), Earthquake (E) and Alarm (A). B and E directly influence A, so $Parents(A) = \{B, E\}$. B and E do not bear any directly influence on each other, at least on the context of this example.

The alarm's owner (Mr. Watson), when away from home, might be warned by any of his two neighbours ($N1$ and $N2$), that usually call him when they hear the alarm. The problem is that they are quite old, and some times they think they heard the alarm, when, in fact, it did not go off. Furthermore, these two neighbours have no possibility of knowing whether or not a burglary or an earthquake occurred, so the event "Call the alarms owner" does not depend on any direct observation of these two events, but only on whether or not each of them thinks he heard the alarm.

Let

$$D_B = \{(T)rue, (F)alse\}$$

$$\begin{aligned} D_E &= \{(S)trong, (M)ild, (F)alse\} \\ D_A &= \{(T)rue, (F)alse\} \\ D_{N1} &= \{(T)rue, (F)alse\} \\ D_{N2} &= \{(T)rue, (F)alse\} \end{aligned}$$

The Bayesian network that represents this model is presented on figure B.6.

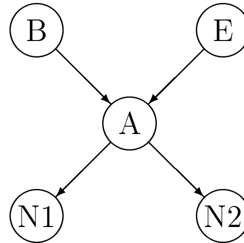


Figure B.6: The burglary alarm's Bayesian network

The prior probabilities of B (Burglary) and E (Earthquake), for the particular region where Mr. Watson lives, are given by table B.1

$\mathbf{P}(B)$	T	F
	0.40	0.60

$\mathbf{P}(E)$	S	M	F
	0.01	0.07	0.92

Table B.1: Burglary Alarm Example: $\mathbf{P}(B)$ and $\mathbf{P}(E)$

The direct influence of A's parents on it, quantified by $\mathbf{CPT}(A|B,E)$, is given by table B.2.

$\mathbf{CPT}(A\|B,E)$		A	
B	E	T	F
T	S	0.98	0.02
T	M	0.95	0.05
T	F	0.90	0.10
F	S	0.30	0.70
F	M	0.20	0.80
F	F	0.01	0.99

Table B.2: Burglary Alarm Example: $\mathbf{CPT}(A|B,E)$

The conditional probabilities $\mathbf{CPT}(N1|A)$ and $\mathbf{CPT}(N2|A)$ are given by table B.3

$\mathbf{CPT}(N1\|A)$	$N1$	
A	T	F
T	0.95	0.05
F	0.10	0.90

$\mathbf{CPT}(N2\|A)$	$N2$	
A	T	F
T	0.99	0.01
F	0.20	0.80

Table B.3: Burglary Alarm Example: $\mathbf{CPT}(N1|A)$ and $\mathbf{CPT}(N2|A)$

B.5.2 Inference Algorithm without Evidence

Initially no evidence is available about any variable, so

$$\lambda(b) = \lambda(a) = \lambda(n1) = \lambda(n2) = \{1, 1\}$$

$$\lambda(e) = \{1, 1, 1\}$$

B and E are root nodes, therefore,

$$\pi(b) = \mathbf{P}(B) = \{0.4, 0.6\}$$

$$\pi(e) = \mathbf{P}(E) = \{0.01, 0.07, 0.92\}$$

From equation B.2

$$\mathbf{BEL}(B) = \alpha\pi(b)\lambda(b) = \{0.4, 0.6\}$$

$$\mathbf{BEL}(E) = \alpha\pi(e)\lambda(e) = \{0.01, 0.07, 0.92\}$$

B and E must compute $\pi_A(b)$ and $\pi_A(e)$, respectively. Since both of them only have only one child, $\pi_A(b) = \pi(b)$ and $\pi_A(e) = \pi(e)$.

After receiving these two vectors, node A must compute $\pi(a)$, using equation B.6.

$$\begin{aligned}
\pi(a) &= \textstyle\prod_i \pi_A(u_i)\mathbf{CPT}(A|B,E) \\
&= \pi_A(b)\pi_A(e)\mathbf{CPT}(A|B,E) \\
&= \{0.4, 0.6\}\{0.01, 0.07, 0.92\}\mathbf{CPT}(A|B,E) \\
&= \{0.004, 0.028, 0.368, 0.006, 0.042, 0.552\}\mathbf{CPT}(A|B,E) \\
&= \{0.37744, 0.62256\}
\end{aligned}$$

Using equation B.2

$$\mathbf{BEL}(A) = \alpha\pi(a)\lambda(a) = \{0.37744, 0.62256\}$$

A must now compute $\pi_{N1}(a)$ and $\pi_{N2}(a)$, to send to $N1$ and $N2$, respectively. Using equation B.5

$$\begin{aligned}\pi_{N1}(a) &= \alpha\frac{\mathbf{BEL}(A)}{\lambda_{N1}(a)} \\ &= \{0.37744, 0.62256\} \\ \pi_{N2}(a) &= \pi_{N1}(a)\end{aligned}$$

After receiving $\pi_{N1}(a)$, node $N1$ must compute $\pi(n1)$ using equation B.1

$$\pi(n1) = \pi_{N1}(a)\mathbf{CPT}(N1|A) = \{0.420824, 0.579176\}$$

Therefore

$$\mathbf{BEL}(N1) = \alpha\pi(n1)\lambda(n1) = \{0.420824, 0.579176\}$$

Node $N2$ must proceed the same way, yielding

$$\pi(n2) = \pi_{N2}(a)\mathbf{CPT}(N2|A) = \{0.4981776, 0.5018224\}$$

Therefore

$$\mathbf{BEL}(N2) = \alpha\pi(n2)\lambda(n2) = \{0.4981776, 0.5018224\}$$

B.5.3 Belief Distribution with Evidence

If neighbour N2 calls, then $\lambda(n2) = \{1, 0\}$, and the belief distribution over $N2$ becomes

$$\mathbf{BEL}(N2) = \alpha\pi(n2)\lambda(n2) = \{1, 0\}$$

Node N2 must compute $\lambda_{N2}(a)$ and send it to node A, using equation B.3

$$\lambda_{N2}(a) = \mathbf{CPT}(N2|A)\lambda(N2) = \{0.99, 0.20\}$$

Upon reception of $\lambda_{N2}(a)$, node A computes $\lambda(a)$, given by equation B.4

$$\lambda(a) = \lambda_{N1}(a)\lambda_{N2}(a) = \{1, 1\}\{0.99, 0.20\} = \{0.99, 0.20\}$$

The new belief distribution over A is thus given by

$$\mathbf{BEL}(A) = \alpha\pi(a)\lambda(a) = \alpha\{0.37744, 0.62256\}\{0.99, 0.20\} = \{0.75007, 0.24993\}$$

As expected the belief that the alarm has actually gone off, increased from 0.37744 to 0.75007 with N2's phone call.

Node A must send $\pi_{N1}(a)$ to N1, $\lambda_A(b)$ to B and $\lambda_A(e)$ to E.

$$\pi_{N1}(a) = \alpha\frac{\mathbf{BEL}(A)}{\lambda_{N1}(a)} = \{0.75007, 0.24993\}$$

Node N1 can now estimate its belief distribution over *N1*

$$\pi(n1) = \pi_{N1}(a)\mathbf{CPT}(N1|A) = \{0.73756, 0.26244\}$$

$$\mathbf{BEL}(N1) = \alpha\pi(n1)\lambda(n1) = \{0.73756, 0.26244\}$$

Mr. Watson's belief that neighbour N1 is going to phone, increased with N2 phone call, as expected.

The diagnostic support induced by A on B and E must be computed using equation B.7

$$\begin{aligned}
\lambda_A(b) &= \sum_a [\lambda(a)(\sum_e \mathbf{CPT}(A|B,E)\pi_A(e))] \\
&= \sum_a [\lambda(a)(\sum_e \begin{bmatrix} 0.98 & 0.02 \\ 0.95 & 0.05 \\ 0.90 & 0.10 \\ 0.30 & 0.70 \\ 0.20 & 0.80 \\ 0.01 & 0.99 \end{bmatrix} \{0.01, 0.07, 0.92\})] \\
&= \sum_a [\{0.99, 0.20\} \begin{bmatrix} 0.9043 & 0.0957 \\ 0.0262 & 0.9738 \end{bmatrix}] \\
&= \{0.914397, 0.220698\}
\end{aligned}$$

$$\begin{aligned}
\lambda_A(e) &= \sum_a [\lambda(a)(\sum_b \mathbf{CPT}(A|B,E)\pi_A(b))] \\
&= \sum_a [\lambda(a)(\sum_b \begin{bmatrix} 0.98 & 0.02 \\ 0.95 & 0.05 \\ 0.90 & 0.10 \\ 0.30 & 0.70 \\ 0.20 & 0.80 \\ 0.01 & 0.99 \end{bmatrix} \{0.40, 0.60\})] \\
&= \sum_a [\{0.99, 0.20\} \begin{bmatrix} 0.572 & 0.428 \\ 0.500 & 0.500 \\ 0.366 & 0.634 \end{bmatrix}] \\
&= \{0.65188, 0.59500, 0.48914\}
\end{aligned}$$

Nodes B and E can now compute their respective belief distribution

$$\mathbf{BEL}(B) = \alpha\pi(b)\lambda(b) = \{0.73419, 0.26581\}$$

$$\mathbf{BEL}(E) = \alpha\pi(e)\lambda(e) = \{0.01309, 0.08360, 0.90332\}$$

Mr. Watson's belief on both events increases, but the increase on burglary's belief is larger than on earthquakes, since burglaries are more likely and the alarm is more sensible to them.

Appendix C

PaRT 2.1 : User's Manual

Contents

C.1 Introduction **239**

C.2 Illumination Model **240**

C.3 Supported Primitives **242**

C.4 Neutral File Format **242**

C.5 PaRT 2.1 Extensions to the Neutral File Format **246**

C.6 PaRT 2.1 Usage **247**

C.1 Introduction

PaRT (Parallel Ray Tracer) is a fairly simple ray tracer, designed with the purpose of developing an adaptive data and load management strategy, which can learn how to achieve better effectiveness. Its sources are available and are quite simple to understand. PaRT 2.1 is a parallel version, running over PVM 3.3 or later.

It supports spheres, polygons, polygonal patches, cylinders, cones and axis aligned boxes. The illumination model includes both local and global illumination. The local term uses the Phong model with four terms: ambient, diffuse reflection, specular reflection and specular transmission. The global term includes global specular reflection and transmitted rays.

Three intensity equations (Red, Green and Blue) are used to simulate colour.

Rays are shooted through the pixels' corners (meaning that 513 x 513 primary rays are created, for a 512x512 resolution). The four corner contributions are averaged to arrive at a pixel value. Adaptive supersampling is possible, consisting on shooting a fifth ray through the pixel's centre if any two corners differ more than by some pre-established amount. The

value of this ray is averaged (with weight = 0.5) with the four corner contributions. This feature can be turned on or off through a command line switch.

Some acceleration techniques have been included to improve the efficiency over exhaustive ray tracing. These include adaptive depth control and nonuniform space partitioning. These features can be turned on or off using command line switches.

PaRT accepts scene descriptions written in Neutral File Format (NFF) [65]. A few extensions to NFF are included (see C.5), so that all of the illumination model's parameters can be set.

PaRT 2.1 includes five different scheduling strategies. These include: static uniform load distribution, demand-driven work assignment, a complex deterministic scheduling policy and two stochastic approaches to load management. The scheduling strategy is selected by a command line switch.

PaRT 2.1 can interactively display the image being generated. To use this capability XWindows and Tcl/Tk must be installed in the root node. If these packages are not available, PaRT 2.1 can still be executed , but the -X switches can not be used.

C.2 Illumination Model

The illumination model includes both local and global illumination. Local or direct illumination is light incident on a surface directly from the light sources. If light is incident on a surface after interaction with another object then that illumination is categorised as global. Global illumination arises from the interaction of direct light with reflective or transparent objects [48, 59, 173].

To calculate colour three intensity equations are used, for Red, Green and Blue. This implies having different coefficients for each light frequency.

The local component is calculated using the Phong reflection model, extended by another term to account for transparent objects. The model is a linear combination of four terms: diffuse reflection, ambient light, specular reflection and specular transmission, which are described in section 7.2.1.

From each intersection point reflected rays and transmitted rays are conditionally shoot to account for global illumination. A reflected ray is shoot if the material's global specular reflection coefficient k_{gs} is greater than zero, whereas a transmitted ray is shoot if the material's global transparency coefficient k_{gt} is greater than zero. These new rays are then traced as if they were primary rays. These tree of rays is recursively traced until a given maximum depth is reached or no additional rays need to be spawn. Total internal reflection

is not implemented on this version of PaRT. Details can be found in section 7.2.2.

Most ray tracers spawn shadow rays from the intersection point towards each light source to determine whether or not that light source is visible from that intersection point. If any object is found in the path towards the light, then this one is considered occluded and will not contribute to that point illumination.

PaRT takes a different approach, in order to account for transparent objects. The local transparency coefficients of all objects intersected by the shadow ray are multiplied by the light source intensity for each wavelength. If the product of these constants is zero then the intersection process terminates because that light source won't contribute to that point illumination.

C.2.1 Rendering Equation

The final equations used by PaRT are

$$I_R = I_{l,R} + k_{gs} I_{s,R} + k_{gt} I_{t,R} \tag{C.1}$$

$$I_G = I_{l,G} + k_{gs} I_{s,G} + k_{gt} I_{t,G} \tag{C.2}$$

$$I_B = I_{l,B} + k_{gs} I_{s,B} + k_{gt} I_{t,B} \tag{C.3}$$

The following parameters are needed to describe each material properties:

Symbol	Description
$k_{d,R}$, $k_{d,G}$, $k_{d,B}$	Diffuse reflection coefficients
$k_{a,R}$, $k_{a,G}$, $k_{a,B}$	Ambient reflection coefficients
k_s	Specular reflection coefficient
ns	Specular reflection exponent
$k_{t,R}$, $k_{t,G}$, $k_{t,B}$	Local transparency coefficients
nt	Local transparency exponent
η	Index of refraction
k_{gs}	Global specular reflection coefficient
k_{gt}	Global transparency coefficient

Table C.1: Set of parameters describing the material properties

Additionally the position and wavelengths of each light source must also be described. The ambient term and the background colour require three additional parameters each (one per wavelength).

C.3 Supported Primitives

PaRT 2.1 supports spheres, polygons, polygonal patches, cylinders, cones and axis aligned boxes. It also supports texture mapping on 4–sided polygons.

Spheres

A sphere takes 4 parameters, namely, its centre coordinates (X, Y, Z) and its radius.

Polygons

Polygons can have up to a maximum number of vertices (200 on the actual version). All vertices must be coplanar.

Polygonal Patches

A polygonal patch is a polygon, but the surface normal at each edge must also be supplied. During the rendering process the surface normal at each intersection point is computed based on the distance between this point and each of the vertices.

Cylinders and Cones

Both cylinders and cones require the specification of the bases coordinates and radius. If one of the bases has a radius of 0, then it is a perfect cone. Hence, it requires 8 parameters, namely, the first and second bases coordinates and radius (X1, Y1, Z1, R1) and (X2, Y2, Z2, R2).

Axis Aligned Boxes

These are rectangular boxes whose edges are aligned with the world coordinates axis. To specify an axis aligned box only the coordinates of two opposed vertices are required.

C.4 Neutral File Format

The NFF (Neutral File Format) is designed as a minimal scene description language. The language was designed in order to test various rendering algorithms and efficiency schemes. It is meant to describe the geometry and basic surface characteristics of objects, the placement of lights, and the viewing frustum for the eye. Some additional information is provided for aesthetic reasons (such as the colour of the objects, which is not strictly necessary for testing the efficiency of rendering algorithms). At present the NFF file format is used in conjunction with the SPD (Standard Procedural Database) software, a package designed to create a variety of databases for testing rendering schemes [65].

Files are constituted by lines of text. For each entity, the first field defines its type. The

remainder of the line and possibly other lines contain further information about the entity. Entities include:

"v" viewing vectors and angles

"b" background colour

"l" positional light location

"f" object material properties

"c" cone or cylinder primitive

"s" sphere primitive

"p" polygon primitive

"pp" polygonal patch primitive

Viewpoint Location

"v"
"from" Fx Fy Fz
"at" Ax Ay Az
"up" Ux Uy Uz
"angle" angle
"hither" hither
"resolution" xres yres

Format:

v
from float float float
at float float float
up float float float
angle float
hither float
resolution int int

The parameters are:

From the eye location in XYZ.

At a position to be at the centre of the image, in XYZ world coordinates. A.k.a. "lookat".

Up a vector defining which direction is up, as an XYZ vector.

Angle in degrees, defined as from the centre of top pixel row to bottom pixel row and left column to right column.

Hither distance of the hither plane (if any) from the eye. Mostly needed for hidden surface algorithms. Not supported by PaRT 2.1.

Resolution in pixels, in x and y.

No assumptions are made about data normalisation (e.g. the from-at distance does not have to be 1). Also, vectors are not required to be perpendicular to each other.

A view entity must be defined before any objects are defined (this requirement is so that NFF files can be displayed on the fly by hidden surface machines).

Background Colour

A colour is simply RGB with values between 0 and 1:

"b" R G B

Format:

b float float float

If no background colour is set, assume RGB = 0,0,0.

Positional light

A light is defined by XYZ position and RGB intensity.

"l" X Y Z [R G B]

Format:

l float float float [float float float]

All light entities must be defined before any objects are defined (this requirement is so that NFF files can be used by hidden surface machines). If no RGB intensity is given then the light is assumed to be white (RGB = 1, 1, 1).

Shading Parameters

"f" $k_{a,R}$ $k_{a,G}$ $k_{a,B}$ k_d k_s n k_{gt} η

Format:

f float float float float float float float float

$k_{a,R}$, $k_{a,G}$, $k_{a,B}$ are the material ambient reflection coefficients.

k_d PaRT calculates the material diffuse reflection coefficients as

$$k_{d,R} = k_d * k_{a,R}$$

$$k_{d,B} = k_d * k_{a,B}$$

$$k_{d,G} = k_d * k_{a,G}$$

k_s is used both as the material's local and global specular reflection coefficient.

n is used both as the material's specular and transparency exponent.

k_{gt} is the material's local and global transparency coefficient.

η index of refraction.

The shading parameters are used to shade the objects following it until a new set of parameters is assigned.

Sphere

A sphere is defined by a radius and centre position:

"s" center.x center.y center.z radius

Format:

s float float float float

Cylinders and Cones

Cylinders and cones are defined by the coordinates and radius of both bases. A base of 0 defines a perfect cone.

"c" base1.x base1.y base1.z base1.radius base2.x base2.y base2.z base2.radius

Format:

c float float float float float float float float

Polygon

A polygon is defined by a set of coplanar vertices.

"p" totalVertices
vert1.x vert1.y vert1.z
(etc. for total vertices)

Format:

p int
float float float
(etc. for total vertices)

Polygonal Patch

A patch is defined by a set of coplanar vertices and their normals.

"pp" totalVertices
vert1.x vert1.y vert1.z norm1.x norm1.y norm1.z
(etc. for total vertices)

Format:

pp int
float float float float float float (etc. for total vertices)

Comment

Format:

string

As soon as a "#" character is detected, the remainder of the line is considered a comment.

C.5 PaRT 2.1 Extensions to the Neutral File Format

PaRT 2.1 extends NFF by recognizing 4 additional commands:

Amb - Ambient light;

Myf - Extended set of shading parameters;

AlignB - Axis aligned box;

Tmap - Texture to map on the next 4-sided polygon.

Ambient Light

The ambient light is RGB with values between 0 and 1:

"Amb" R G B

Format:

Amb float float float

If no ambient light is set, assume RGB = 0,0,0.

Extended Set of Shading Parameters

This command allows the specification of all the parameters used by the illumination model as described in table C.1

"Myf" $k_{a,R}$ $k_{a,G}$ $k_{a,B}$ $k_{d,R}$ $k_{d,G}$ $k_{d,B}$ k_s ns $k_{t,R}$ $k_{t,G}$ $k_{t,B}$ nt η k_{gs} k_{gt}

Format:

Myf float float float float float float float float float float float float float float float

Axis Aligned Box

An axis aligned box is defined by the coordinates of two opposite vertices.

"AlignB" vert1.x vert1.y vert1.z
vert2.x vert2.y vert2.z

Format:

AlignB float float float
float float float

Texture

A bitmap figure can be mapped onto a 4-sided polygon. The Tmap primitive specifies the path for a Targa file – tga extension. This figure is then mapped on the polygon specified after the Tmap primitive. This must be a rectangle.

Format:

Tmap drive:pathname

C.6 PaRT 2.1 Usage

C.6.1 Installation and Requirements

PaRT 2.1 requires PVM 3.3 or greater installed on all computers that will be part of the parallel virtual machine.

To install PaRT 2.1 just type 'aimk'. This command will read Makefile.aimk. This file must be edited in order to correct the pathnames to each particular machine. The executables (part, ap, lm, dm, MyXProc) are copied to the appropriate PVM binary directory (usually $HOME/pvm3/bin/$PVM_ARCH).

In order to visualise the images being generated MyXProc needs to be built. This requires that both XWindows and Tcl/Tk are installed on the root computer. If this is not the case, then PaRT 2.1 can still be used , but the -X switches can not be used. MyXProc is required only on the root computer, i.e., the one where PIRT is started. If MyXProc is succesfully built on one machine, then the -X switches can be used if PaRT is started on that machine.

C.6.2 Usage

The first step is to start a Parallel Virtual Machine. Then PaRT can be executed. It will spawn processes on all nodes of the Virtual Machine.

```
PaRT <input file> [<output file>] [-<switch>]
```

If no output file is given, data is output to a file with the same name as input with extension .tga

Supported switches:

-p If present the percentage of work already finished is displayed;

-adap float If present turns on adaptive depth control: the parameter is the minimum contribution of a ray for it to be spawned;

-nopart If present turns off hierarchical space partitioning;

-adsamp If present turns on adaptive sampling;

-raydepth int If present the rays tree maximum depth will be equal to the parameter, otherwise it is some default value;

-unif Uniform data distribution. The image is split on as many regions as nodes in the system. Each of these images will be processed on one of the nodes;

-dd Demand driven load distribution. The image is subdivided into regions according to the following rule: if there are 4 or less nodes split into 25 regions, if there are between 5 and 8 nodes split into 64 regions, otherwise split into 100 regions. These tasks are then sent to the nodes on demand. When one AP has no work, it requests a task to its local load manager, which, in turn, requests a task to the root load manager. The local load managers usually fetch one task in advance, so that they can quickly satisfy the AP request. Processing finishes when there are no tasks left;

-det Deterministic complex scheduling strategy;

-bayes Bayesian network based scheduler;

-sequpd Bayesian network based scheduler, with sequential updating of prior probabilities;

-Xasap Display the image on a X window. The regions of the image are displayed as soon as processed;

-Xc Display the image on a X window. The image is displayed only when all its subregions have been processed.

Appendix D

Results

This appendix presents the results obtained with the experiments described in chapter 8. These values were obtained using several different scheduling policies: uniform work distribution (unif), demand–driven work allocation (dd), the deterministic scheduling strategy (det1), the deterministic scheduling strategy with extended information policy (det), the decision network approach (dn) and the decision network with updated probabilities (dn–upd). det1's execution times were measured only for balls4pv and teapot9. The columns' labels are described in table D.1.

Label	Comment
N	Number of nodes
Sched	Scheduling strategy
T_{exec}	Execution time
#T	Number of tasks
#TS	Number of information messages sent by the application processes
Pen%	Replication penalty
TTidle	Total time spent waiting for tasks
TTidle%	Percentage of (execution time * N)
StdDev	Standard deviation
TTdata	Total time spent waiting for remote data
TTdata%	Percentage of (execution time * N)

Table D.1: Columns's Labels

Tables D.2, D.3, D.4 and D.5 present the results obtained using the distributed system in dedicated mode, i.e., without any additional users. All available processing power and communication bandwidth are dedicated to the ray tracer and operating system tasks. Tables D.6, D.7, D.8 and D.9 present the results obtained using the distributed system with 7 nodes and different synthetic background workloads, as described in section 6.6.

N	Sched	T_{exec}	#T	#TS	Pen%	TTidle	%	StdDev
2	unif	103.40	2	—	0.0%	2.95	1.43%	1.40
2	dd	107.31	25	—	1.4%	7.82	3.64%	3.65
2	det	104.69	3	39	0.1%	0.91	0.44%	0.18
2	dn	101.60	3	40	0.1%	0.69	0.34%	0.20
2	dn–upd	101.84	3	39	0.1%	0.55	0.27%	0.07
3	unif	87.98	3	—	0.0%	59.77	22.64%	13.94
3	dd	70.53	25	—	1.2%	4.63	2.19%	0.94
3	det	70.74	8	73	0.3%	2.12	1.00%	0.38
3	dn	68.25	7	71	0.3%	0.89	0.43%	0.09
3	dn–upd	67.98	7	68	0.3%	0.79	0.39%	0.08
4	unif	65.88	4	—	0.0%	58.75	22.30%	12.40
4	dd	53.14	64	—	2.2%	3.19	1.50%	0.63
4	det	53.43	13	100	0.4%	3.04	1.42%	0.37
4	dn	51.93	10	91	0.2%	2.89	1.39%	0.45
4	dn–upd	50.51	10	89	0.3%	2.75	1.36%	0.55
5	unif	52.18	5	—	0.0%	59.33	22.74%	10.70
5	dd	41.75	64	—	2.0%	5.31	2.54%	0.71
5	det	41.31	15	121	0.4%	1.31	0.64%	0.07
5	dn	40.13	16	126	0.4%	1.24	0.62%	0.09
5	dn–upd	39.75	13	118	0.3%	1.14	0.58%	0.09
6	unif	46.05	6	—	0.0%	77.91	28.20%	9.24
6	dd	34.39	64	—	1.8%	6.38	3.09%	0.83
6	det	33.96	17	145	0.4%	1.86	0.91%	0.12
6	dn	33.37	15	139	0.3%	1.70	0.85%	0.29
6	dn–upd	33.14	15	138	0.3%	1.66	0.83%	0.30
7	unif	37.36	7	—	0.0%	63.81	24.40%	7.09
7	dd	30.54	64	—	1.6%	12.56	5.87%	1.10
7	det	29.40	27	192	0.6%	2.05	1.00%	0.10
7	dn	28.60	20	172	0.4%	1.87	0.93%	0.14
7	dn–upd	28.41	21	176	0.4%	1.69	0.85%	0.14

Table D.2: Balls3: dedicated mode results

N	Sched	T_{exec}	#T	#TS	Pen%	TTidle	%	StdDev
2	unif	176.18	2	—	0.0%	65.71	18.65%	32.69
2	dd	146.88	25	—	1.4%	2.80	0.95%	1.15
2	det	146.65	5	34	0.3%	0.67	0.23%	0.02
2	dn	142.44	5	34	0.3%	0.63	0.22%	0.15
2	dn–upd	142.41	5	34	0.3%	0.60	0.21%	0.14
3	unif	125.84	3	—	0.0%	89.95	23.83%	23.04
3	dd	98.36	25	—	1.2%	3.99	1.35%	0.94
3	det	98.37	8	53	0.3%	1.63	0.55%	0.31
3	dn	95.23	8	53	0.3%	0.88	0.31%	0.10
3	dn–upd	94.58	8	53	0.3%	0.85	0.30%	0.09
4	unif	92.69	4	—	0.0%	82.78	22.33%	16.51
4	dd	74.32	64	—	2.2%	2.70	0.91%	0.35
4	det	73.99	13	77	0.4%	1.53	0.52%	0.19
4	dn	71.99	13	78	0.5%	1.34	0.46%	0.15
4	dn–upd	69.97	12	71	0.4%	1.17	0.42%	0.12
5	unif	74.02	5	—	0.0%	87.25	23.57%	15.08
5	dd	57.63	64	—	2.0%	1.62	0.56%	0.20
5	det	57.85	18	98	0.5%	1.92	0.66%	0.16
5	dn	56.26	17	91	0.5%	1.76	0.62%	0.19
5	dn–upd	56.15	17	97	0.5%	1.67	0.59%	0.18
6	unif	64.64	6	—	0.0%	107.29	27.66%	13.71
6	dd	47.37	64	—	1.8%	2.81	0.99%	0.37
6	det	47.51	20	111	0.5%	2.31	0.81%	0.20
6	dn	46.30	22	117	0.5%	2.35	0.84%	0.21
6	dn–upd	45.97	21	116	0.5%	1.99	0.72%	0.20
7	unif	54.04	7	—	0.0%	95.85	25.34%	11.21
7	dd	41.64	64	—	1.6%	8.17	2.80%	0.60
7	det	41.10	23	138	0.4%	2.42	0.84%	0.14
7	dn	39.70	20	128	0.4%	2.10	0.75%	0.12
7	dn–upd	39.41	20	127	0.4%	1.80	0.65%	0.15

Table D.3: Balls3c: dedicated mode results

N	Sched	T_{exec}	#T	#TS	Pen%	TTidle	%	StdDev	TTdata	%
2	unif	282.54	2	—	0.0%	204.07	36.11%	101.90	0.57	0.10%
2	dd	151.87	25	—	1.4%	62.37	20.54%	31.10	0.57	0.19%
2	det1	138.23	6	81	0.4%	28.96	10.5%	14.36	0.62	0.20%
2	det	118.19	7	90	0.5%	2.30	0.97%	1.03	0.69	0.29%
2	dn	116.35	7	89	0.5%	2.09	0.90%	1.01	0.66	0.28%
2	dn–upd	114.80	7	90	0.5%	1.81	0.79%	0.89	0.66	0.29%
3	unif	220.83	3	—	0.0%	291.84	44.05%	82.71	0.64	0.10%
3	dd	112.84	25	—	1.2%	21.46	6.34%	7.93	0.92	0.27%
3	det1	85.30	12	134	0.6%	41.71	16.30%	9.74	0.72	0.30%
3	det	75.60	11	140	0.5%	6.75	2.98%	1.82	0.66	0.29%
3	dn	73.41	11	140	0.5%	5.34	2.42%	1.64	0.63	0.29%
3	dn–upd	71.96	9	134	0.4%	4.15	1.92%	1.14	0.65	0.30%
4	unif	166.40	4	—	0.0%	266.26	40.0%	65.48	0.47	0.07%
4	dd	71.96	64	—	2.2%	27.43	9.53%	5.27	1.01	0.35%
4	det1	65.57	21	178	0.8%	17.32	6.60%	2.93	0.72	0.30%
4	det	64.83	14	167	0.5%	6.37	2.46%	1.32	0.78	0.30%
4	dn	62.86	16	178	0.6%	4.44	1.76%	0.66	0.72	0.29%
4	dn–upd	59.68	16	175	0.6%	3.97	1.66%	0.61	0.74	0.31%
5	unif	133.58	5	—	0.0%	266.97	39.97%	46.66	0.47	0.07%
5	dd	65.35	64	—	2.0%	50.06	15.32%	5.72	0.83	0.25%
5	det1	51.86	30	253	1.00%	24.12	9.30%	2.41	0.68	0.30%
5	det	49.16	27	242	0.9%	13.13	5.34%	1.11	0.82	0.33%
5	dn	48.66	25	242	0.8%	12.36	5.08%	1.53	0.77	0.31%
5	dn–upd	46.95	24	248	0.7%	8.44	3.60%	0.78	0.80	0.34%
6	unif	115.24	6	—	0.0%	289.95	41.94%	38.87	0.60	0.09%
6	dd	59.59	64	—	1.8%	43.40	12.14%	6.05	1.15	0.32%
6	det1	52.47	20	252	0.50%	45.94	14.60%	4.94	0.85	0.30%
6	det	45.34	31	298	0.8%	10.07	3.7%	1.01	0.79	0.29%
6	dn	42.94	28	282	0.7%	9.57	3.72%	1.07	0.74	0.29%
6	dn–upd	42.67	26	268	0.6%	6.79	2.65%	0.58	0.67	0.26%
7	unif	99.57	7	—	0.0%	297.84	42.73%	31.75	0.61	0.09%
7	dd	57.67	64	—	1.6%	77.85	19.29%	5.68	0.94	0.23%
7	det1	47.89	22	264	0.6%	23.41	7.00%	4.5	0.39	0.80%
7	det	45.55	38	337	0.9%	16.02	5.02%	1.30	0.83	0.26%
7	dn	41.13	32	315	0.7%	14.68	5.10%	1.47	0.80	0.28%
7	dn–upd	40.82	28	305	0.6%	14.28	5.00%	1.40	0.84	0.30%

Table D.4: Teapot9: dedicated mode results

N	Sched	T_{exec}	#T	#TS	Pen%	TTidle	%	StdDev	TTdata	%
2	unif	958.03	2	—	0.0%	791.41	41.30%	395.56	197.01	10.28%
2	dd	420.17	25	—	2.7%	51.78	6.16%	25.78	134.02	15.95%
2	det1	314.30	8	37	1.2%	22.84	3.6%	11.32	88.70	14.10%
2	det	305.69	7	35	1.0%	11.87	1.94%	5.83	90.29	14.77%
2	dn	297.05	6	33	0.8%	11.17	1.88%	5.79	86.84	14.62%
2	dn–upd	295.97	6	33	0.8%	10.32	1.74%	5.26	86.68	14.64%
3	unif	740.03	3	—	0.0%	975.27	43.93%	285.38	213.19	9.60%
3	dd	331.82	25	—	2.3%	27.38	2.75%	7.55	131.13	13.17%
3	det1	218.25	12	61	1.2%	77.29	11.80%	18.24	85.76	13.10%
3	det	203.39	12	61	1.2%	23.08	3.78%	5.54	88.14	14.44%
3	dn	192.19	10	57	0.9%	8.25	1.43%	2.71	84.40	14.64%
3	dn–upd	191.85	10	56	0.9%	7.71	1.34%	2.36	84.02	14.60%
4	unif	557.61	4	—	0.0%	1061.92	47.61%	210.87	191.96	8.61%
4	dd	201.23	64	—	4.3%	2.98	0.37%	0.63	92.77	11.52%
4	det1	191.32	17	80	1.3%	221.28	28.90%	31.98	72.03	9.40%
4	det	148.35	21	99	1.7%	11.54	1.95%	2.45	82.59	13.92%
4	dn	140.60	18	89	1.4%	8.98	1.60%	0.56	72.20	12.84%
4	dn–upd	138.87	18	88	1.4%	8.79	1.58%	0.51	69.21	12.46%
5	unif	477.37	5	—	0.0%	1157.75	48.51%	170.55	176.31	7.39%
5	dd	149.58	64	—	3.9%	13.90	1.86%	1.39	95.07	12.71%
5	det1	124.10	25	115	1.6%	54.65	8.80%	5.79	70.61	11.40%
5	det	118.11	23	113	1.4%	14.34	2.43%	1.63	77.09	13.05%
5	dn	121.95	21	108	1.2%	18.38	3.01%	5.30	66.21	10.86%
5	dn–upd	119.72	21	108	1.2%	15.31	2.56%	4.48	78.98	12.35%
6	unif	384.94	6	—	0.0%	1145.50	49.60%	138.71	158.60	6.87%
6	dd	145.16	64	—	3.5%	41.80	4.80%	3.12	125.50	14.41%
6	det1	114.36	36	151	2.00%	63.63	9.30%	8.60	81.31	11.9%
6	det	102.47	28	139	1.4%	44.94	7.31%	4.56	67.94	11.05%
6	dn	99.09	28	137	1.4%	17.17	2.89%	1.81	61.80	10.39%
6	dn–upd	98.94	27	142	1.4%	14.82	2.50%	1.59	59.20	9.97%
7	unif	325.13	7	—	0.0%	1107.59	48.67%	119.10	156.54	6.88%
7	dd	123.33	64	—	3.1%	82.29	9.53%	5.08	93.31	10.81%
7	det1	105.36	38	176	2.0%	71.12	9.6%	7.91	82.03	11.10%
7	det	96.11	29	150	1.2%	32.85	4.88%	2.90	81.25	12.08%
7	dn	91.00	28	204	1.2%	29.92	4.56%	2.91	57.35	9.00%
7	dn–upd	90.18	29	178	1.3%	27.28	4.32%	2.85	55.01	8.71%

Table D.5: Balls4pv: dedicated mode results

Workload	Sched	T_{exec}	#T	#TS	Pen%	TTidle	%	StdDev
Dedicated	unif	37.36	7	—	0.0%	63.81	24.40%	7.09
	dd	30.54	64	—	1.6%	12.56	5.87%	1.10
	det	29.40	27	192	0.6%	2.05	1.00%	0.10
	dn	28.60	20	172	0.4%	1.87	0.93%	0.14
	dn–upd	28.41	21	176	0.4%	1.69	0.85%	0.14
Light	unif	140.79	7	—	0.0%	423.94	43.02%	33.94
	dd	81.32	64	—	1.6%	52.17	9.16%	4.12
	det	76.65	33	221	0.8%	21.10	3.93%	1.24
	dn	73.06	22	197	0.4%	12.56	2.46%	0.91
	dn–upd	72.60	21	178	0.4%	7.86	1.55%	0.74
Medium	unif	175.91	7	—	0.0%	542.25	44.04%	36.32
	dd	98.37	64	—	1.6%	110.51	16.05%	7.54
	det	87.65	24	190	0.4%	50.53	8.24%	4.41
	dn	85.02	22	213	0.3%	25.15	4.23%	2.00
	dn–upd	84.12	23	195	0.3%	20.99	3.56%	1.52
Heavy	unif	374.83	7	—	0.0%	870.73	33.19%	71.83
	dd	198.65	64	—	1.6%	132.51	9.53%	8.82
	det	168.75	21	187	0.4%	31.22	2.64%	2.04
	dn	160.91	20	188	0.4%	24.28	2.16%	1.98
	dn–upd	159.46	22	195	0.5%	19.44	1.74%	1.45

Table D.6: Balls3: results with different background workloads (7 nodes)

Workload	Sched	T_{exec}	#T	#TS	Pen%	TTidle	%	StdDev
Dedicated	unif	54.05	7	—	0.0%	95.85	25.34%	11.21
	dd	41.64	64	—	1.6%	8.17	2.80%	0.60
	det	41.10	23	138	0.4%	2.42	0.84%	0.14
	dn	39.70	20	128	0.4%	2.10	0.75%	0.12
	dn–upd	39.41	20	127	0.4%	1.80	0.65%	0.15
Light	unif	166.46	7	—	0.0%	398.32	34.18%	45.95
	dd	99.43	64	—	1.6%	12.66	1.82%	1.04
	det	98.71	22	152	0.4%	4.68	0.68%	0.16
	dn	95.69	24	148	0.5%	4.01	0.60%	0.24
	dn–upd	94.46	23	149	0.4%	3.48	0.53%	0.25
Medium	unif	222.84	7	—	0.0%	544.24	34.89%	54.76
	dd	127.04	64	—	1.6%	20.78	2.34%	1.75
	det	121.83	25	156	0.5%	11.61	1.36%	0.64
	dn	116.57	24	156	0.5%	7.13	0.87%	0.55
	dn–upd	115.27	24	160	0.5%	6.66	0.83%	0.38
Heavy	unif	546.89	7	—	0.0%	1355.99	35.42%	96.25
	dd	271.08	64	—	1.6%	102.01	5.38%	7.19
	det	237.76	21	150	0.4%	18.99	1.14%	0.58
	dn	229.87	19	153	0.5%	16.61	1.03%	0.51
	dn–upd	225.65	19	154	0.5%	14.60	0.92%	0.53

Table D.7: Balls3c: results with different background workloads (7 nodes)

Workload	Sched	T_{exec}	#T	#TS	Pen%	TTidle	%	StdDev	TTdata	%
Dedicated	unif	99.57	7	—	0.0%	297.84	42.73%	31.75	0.61	0.09%
	dd	57.67	64	—	1.6%	77.85	19.29%	5.68	0.94	0.23%
	det	45.55	38	337	0.9%	16.02	5.02%	1.30	0.83	0.26%
	dn	41.13	32	315	0.7%	14.68	5.10%	1.47	0.80	0.28%
	dn–upd	40.82	28	305	0.6%	14.28	5.00%	1.40	0.84	0.30%
Light	unif	288.70	7	—	0.0%	817.80	40.47%	100.99	16.27	0.81%
	dd	159.67	64	—	1.6%	208.99	18.70%	18.35	14.80	1.32%
	det	112.35	33	325	0.7%	44.71	5.69%	3.53	18.41	2.34%
	dn	107.53	29	315	0.9%	42.70	5.67%	3.29	17.39	2.31%
	dn–upd	106.27	29	317	0.9%	40.89	5.50%	2.93	17.25	2.32%
Medium	unif	432.49	7	—	0.0%	1562.94	51.63%	121.29	20.86	0.69%
	dd	206.93	64	—	1.6%	413.07	28.52%	39.95	25.61	1.77%
	det	149.46	41	381	0.9%	66.58	6.36%	5.74	31.28	2.99%
	dn	143.20	38	393	0.9%	24.46	2.44%	2.63	28.17	2.81%
	dn–upd	140.21	36	374	0.9%	20.90	2.13%	2.14	27.75	2.83%
Heavy	unif	967.32	7	—	0.0%	3377.43	49.88%	291.26	69.37	1.02%
	dd	377.85	64	—	1.6%	512.19	19.36%	47.04	80.88	3.06%
	det	253.24	34	352	0.8%	122.03	6.88%	8.46	102.20	5.77%
	dn	233.13	33	357	0.7%	46.14	2.77%	4.34	80.58	4.84%
	dn–upd	228.43	33	361	0.7%	43.77	2.71%	4.25	76.84	4.76%

Table D.8: Teapot9: results with different background workloads (7 nodes)

Workload	Sched	T_{exec}	#T	#TS	Pen%	TTidle	%	StdDev	TTdata	%
Dedicated	unif	325.13	7	—	0.0%	1107.59	48.67%	119.10	156.54	6.88%
	dd	123.33	64	—	3.1%	82.29	9.53%	5.08	93.31	10.81%
	det	96.11	29	150	1.2%	32.85	4.88%	2.90	81.25	12.08%
	dn	91.00	28	204	1.2%	29.92	4.70%	2.91	57.35	9.00%
	dn–upd	90.18	29	178	1.3%	27.28	4.32%	2.85	55.01	8.71%
Light	unif	926.90	7	—	0.0%	3134.14	48.30%	340.52	983.90	15.16%
	dd	323.39	64	—	3.1%	186.32	8.23%	17.48	526.91	23.28%
	det	269.15	30	151	1.3%	129.47	6.87%	10.13	400.92	21.28%
	dn	255.60	29	154	1.3%	102.62	5.74%	7.98	351.20	19.63%
	dn–upd	250.91	27	149	1.2%	96.86	5.52%	5.94	347.41	19.78%
Medium	unif	1129.35	7	—	0.0%	3432.14	43.41%	408.85	1206.16	15.26%
	dd	381.19	64	—	3.1%	180.34	6.76%	11.32	542.56	20.33%
	det	301.94	33	178	1.5%	141.18	6.68%	6.61	604.24	28.59%
	dn	273.70	31	214	1.4%	129.50	6.76%	5.87	571.20	29.81%
	dn–upd	267.05	30	190	1.4%	92.91	4.97%	5.12	558.72	29.89%
Heavy	unif	2445.71	7	—	0.0%	4001.68	23.37%	1519.93	3771.69	22.03%
	dd	725.29	64	—	3.1%	576.67	11.36%	615.42	2040.86	40.20%
	det	454.10	27	159	1.1%	147.39	4.64%	13.59	1146.36	36.06%
	dn	405.88	25	153	1.0%	109.91	3.87%	8.03	987.54	34.76%
	dn–upd	391.28	25	152	1.0%	106.78	3.90%	7.88	953.66	34.82%

Table D.9: Balls4pv: results with different background workloads (7 nodes)

Bibliography

[1] Ahmad, I., and Ghafoor, A. Semi–Distributed Load Balancing for Massively Parallel Multicomputer Systems. *IEEE Transactions on Software Engineering 17*, 10 (Oct. 1991), 987–1004.

[2] Al-Saqabi, K., Otto, S., and Walpole, J. Gang Scheduling in Heterogeneous Distributed Systems. Tech. Rep. 94–023, Oregon Graduate Institute, 1994.

[3] Almasi, G., and Gottlieb, A. *Highly Parallel Computing*, 2nd ed. Benjamin Cummings Publishing, 1994. ISBN 0-8053-0443-6.

[4] Azevedo, C., and Azevedo, A. *Metodologia Científica*. C. Azevedo, 1994. ISBN 972 9114 10 2.

[5] Azevedo-Filho, A., and Schachter, R. Laplace's Method Approximations for Probabilistic Inference in Belief Networks with Continuous Variables. In *10th Conference on Uncertainty in Artificial Intelligence* (1994), Morgan–Kaufmann.

[6] Bacchus, F., and Grove, A. Graphical Models for Preference and Utility. In *11th Conference on Uncertainty in Artificial Intelligence* (1995), Morgan–Kaufmann, pp. 3–10.

[7] Balter, M., and Downey, A. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (Philadelphia, USA, May 1996), ACM.

[8] Barbosa, J. *Paralelismo em Processamento e Análise de Imagens Médicas*. PhD thesis, Faculdade de Engenharia da Universiade do Porto, Porto, Portugal, July 2000.

[9] Baumgartner, K., and Wah, B. Computer Scheduling Algorithms: Past, Present and Future. *Information Sciences 57, 58* (Dec. 1991), 319–345.

[10] Becker, W. Dynamic Balancing Complex Workload in Workstation Networks - Challenge, Concepts and Experience. In *High Performance Computing and Networking (HPCN95)* (Milan, Italy, May 1995), Springer, pp. 407–412.

[11] Becker, W., and Waldmann, G. Exploiting Inter Task Dependencies for Dynamic Load Balancing. In *IEEE 3rd Int. Symposium on High-Performance Distributed Computing* (San Francisco, California, Aug. 1994).

[12] Becker, W., and Waldmann, G. Adaption in Dynamic Load Balancing: Potential and Techniques. In *Fachtagung Arbeitsplatz-Rechensysteme* (Hanover, Germany, May 1995).

[13] Becker, W., and Zedelmayr, J. Scalability and Potential for Optimization in Dynamic Load Balancing-Centralized and Distributed Structures. In *Parallel Algorithmen und Rechnerstrukturen* (Potsdam, 1994).

[14] Berman, F., Wolski, R., Figueira, S., Schopf, J., and Schao, G. Application Level Scheduling on Distributed Heterogeneous Networks. In *SuperComputing'96* (Pittsburgh, USA, 1996).

[15] Bharadwaj, V., Ghose, D., Mani, V., and Robertazzi, T. *Scheduling Divisible Loads in Parallel and Distributed Systems*, 1st ed. IEEE Computer Society, 1996.

[16] Bischof, S., Ebner, R., and Erlebach, T. Parallel Load Balancing for Problems with Good Bisectors. *Journal of Parallel and Distributed Computing 60*, 9 (Sept. 2000), 1047–1073.

[17] Bozyigit, M., Al-Ghamdi, J., Ghouseuddin, M., and Barada, H. A Load Balanced Distributed Computing System. Tech. rep., Information and Computer Science Department — King Fahd University of Petroleum & Minerals, 1999.

[18] Buntine, W. Operations for Learning with Graphical Models. *Journal of Artificial Intelligence Research 2* (Dec. 1994), 159–225.

[19] Buyya, R. *High Performance Cluster Computing*, 1st ed., vol. 1. Prentice Hall, 1999. ISBN 0-13-013784-7.

[20] CalzaRossa, M., and Massari, L. Measurement–Based Approach to Workload Characterization. In *7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (1994), R. Marie, G. Haring, and G. Kotsis, Eds.

[21] CalzaRossa, M., Massari, L., and Merlo, A. *General Purpose Parallel Computers: Architectures, Programming Environments and Tools*, 1st ed. Edizioni ETS, Consiglio Nazionale delle Ricerche, 1995. chaps. 11,13,14.

[22] CalzaRossa, M., Massari, L., and Tessera, D. Workload Characterization: Issues and Methodology. In *Performance Evaluation – Issues and Methodology*

(2000), G. Haring, C. Lindermann, and M. Reiser, Eds., no. 1769 in Lecture Notes in Computer Science, Springer, pp. 459–484.

[23] CalzaRossa, M., and Serazzi, G. Workload Characterization: A Survey. *Proceedings of the IEEE 81*, 8 (Aug. 1993), 1136–1150.

[24] Casavant, T., and Kuhl, J. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering* (Feb. 1988), 141–154.

[25] Casavant, T., and Kuhl, J. Effects of Response and Stability on Scheduling in Distributed Computing Systems. *IEEE Transactions on Software Engineering* (Nov. 1988), 1578–1588.

[26] Castillo, E., Gutierrez, J., and Hadi, A. *Expert Systems and Probabilistic Network Models*. Springer-Verlag, 1997. ISBN 0-387-94858-9.

[27] Chalmers, A. *A Minimum Path System for Parallel Processing*. PhD thesis, Department of Computer Science, University of Bristol, Apr. 1991.

[28] Chapin, S., Katramatos, D., Karpovich, J., and Grinshaw, A. The Legion Resource Management System. In *Int. Parallel and Distributed Processing Symposium (IPDPS'99) Workshop on Job Scheduling Strategies for Parallel Processing* (San Juan, Puerto Rico, Apr. 1999).

[29] Corradi, A., Leonardi, L., and Zambonelli, F. Diffusive Load Balancing Policies for Dynamic Applications. *IEEE Concurrency: Parallel, Distributed and Mobile Computing* (Jan. 1999), 22–31.

[30] Costa, L., and Neves, J. An Intelligent Self-Organizing System for Dynamic Load Balancing. In *5th IASTED International Conference on Robotics and Manufacturing* (Cancun, Mexico, May 1997).

[31] Cowell, R., Dawid, A., Lauritzen, S., and Spiegelhalter, D. *Probabilistic Networks and Expert Systems*. Springer-Verlag, 1999. ISBN 0-387-98767-3.

[32] Czajkowski, K., Foster, I., Kesselman, C., Martin, S., Smith, W., and Tuecke, S. A Resource Management Architecture for Metacomputing Systems. In *Workshop on Job Scheduling Strategies for Parallel Processing* (1998).

[33] Dandamudi, S. Sensitivity Evaluation of Dynamic Load Sharing in Distributed Systems. *IEEE Concurrency* (July 1998), 62–72.

[34] Dasgupta, P., Majumder, A., and Bhattacharya, P. V_THR: An Adaptive Load Balancing Algorithm. *Journal of Parallel and Distributed Computing*, 42 (1997), 101–108.

[35] Dean, T., Allen, J., and Aloimonos, Y. *Artificial Intelligence: Theory and Practice.* Benjamim Cummings Publishing Company, 1995.

[36] Decker, T. Virtual Data Space — A Universal Load Balancing Scheme. In *Solving Irregular Structured Problems in Parallel (IRREGULAR'97)* (1997), Springer – Lecture Notes in Computer Science 1253, pp. 159–166.

[37] Diekmann, R., Monien, B., and Preis, R. Load Balancing Strategies for Distributed Memory Machines. *World Scientific* (1997).

[38] Downey, A., and Feitelson, D. The Elusive Goal of Workload Characterization. *Performance Evaluation Review 26*, 4 (Mar. 1999), 14–29.

[39] Eager, D., Lazowska, E., and Zahorjan, J. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering* (May 1986), 662–675.

[40] Epema, D., Livny, M., Dantzig, R., Evers, X., and Pruyne, J. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems 12* (1996), 53–65.

[41] Fabero, J., Martin, I., Bautista, A., and Molina, S. Dynamic Load Balancing in a Heterogeneous Environment under PVM. In *4th EuroMicro Workshop on Parallel and Distributed Processing* (Braga, Portugal, Jan. 1996), IEEE Computer Society Press, pp. 414–419.

[42] Feitelson, D. Job Scheduling in Multiprogrammed Parallel Systems. Research Report RC 19790, IBM T.J. Research Center, Aug. 1997.

[43] feitelson, D., and Rudolph, L. Parallel Job Scheduling: Issues and Approaches. In *Job Scheduling Strategies for Parallel Processing* (1995), Springer – Lecture Notes in Computer Science 949, pp. 1–18.

[44] Ferguson, D., Nikolaou, C., Sairamesh, J., and Yemini, Y. Economic Models for Allocating Resources in Computer Systems. Tech. rep., IBM T. J. Watson Research Center and Columbia University, 1995.

[45] Ferguson, D., Nikolaou, C., and Yemini, Y. Microeconomic Algorithms for Dynamic Load Balancing in Distributed Computer Systems. Tech. rep., IBM T. J. Watson Research Center and Columbia University, 1995.

[46] Ferrari, D., and Zhou, S. An Empirical Investigation of Load Indices for Load Balancing Applications. Tech. rep., University of California, Berkeley, 1987.

[47] Ferstl, F. Job and Resource Management Systems in Heterogeneous Clusters. *Future Generation Computer Systems 12* (1996), 39–51.

[48] Foley, J., van Dam, A., Feiner, S., and Hughes, J. *Computer Graphics: Principles and Practice*, 2nd ed. Addison-Wesley, 1990. ISBN 0-201-12110-7.

[49] Foster, I. *Designing and Building Parallel Programs.* Addison-Wesley, 1994. ISBN 0-201-57594-9.

[50] Foster, I., and Kesselman, C., Eds. *The Grid: Blueprint for a New Computing InfraStructure.* Morgan Kaufmann, 1999.

[51] Foster, I., Kesselman, C., Lee, C., Lindell, B., Nahrstedt, K., and Roy, A. A Distributed Resource Management Architecture that Supports Advance Reservations and Co–Allocation. In *Proceedings of the 7th International Workshop on Quality of Service* (1999).

[52] Foster, I., Roy, A., and Sander, V. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proceedings of the 8th International Workshop on Quality of Service* (June 2000), pp. 181–188.

[53] Fox, G., and Coddington, P. Parallel Computers and Complex Systems. Tech. rep., Northeast Parallel Architectures Center, Syracuse University, June 1994.

[54] Friedman, N., and Goldszmidt, M. Discretizing Continuous Variables while Learning Bayesian Networks. In *13th International Conference on Machine Learning* (1996), pp. 157–165.

[55] Friedman, N., and Goldszmidt, M. Sequential Update of Bayesian Network Structure. In *13th Conference on Uncertainty in Artificial Intelligence* (1997), Morgan–Kaufmann, pp. 165–174.

[56] Friedman, N., Goldszmidt, M., Heckerman, D., and Russell, S. Challenge: Where is the Impact of Bayesian Networks in Learning? In *13th International Joint Conference on Artificial Intelligence* (1997).

[57] Gehring, J., and Reinefeld, A. MARS - A framework for minimizing the job execution time in a metacomputing environment. *Future Generation Computer Systems 12* (1996), 88–99.

[58] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*, 1st ed. The MIT Press, 1994.

[59] Glassner, A. *An Introduction to Ray Tracing*, 7th ed. Academic Press, 1997. ISBN 0-12-286160-4.

[60] Grama, A., Gupta, A., and Kumar, V. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel and Distributed Technology* (1993), 12–21.

[61] Green, S. *Parallel Processing for Computer Graphics*, 1st ed. The MIT Press, 1991. ISBN 0-262-57087-4.

[62] Grinshaw, A., and Wulf, W. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM 40*, 1 (Jan. 1997).

[63] Grinshaw, A., Wulf, W., French, J., Weaver, A., and Reynolds, P. A Synopsys of the Legion Project. Tech. rep., University of Virginia, June 1994.

[64] Gustafson, J. Reevaluating Amdahl's Law. *Communications of the ACM* (May 1988), 532–533.

[65] Haines, E. *Neutral File Format*, 1992. ftp.princeton.edu (pub/Graphics/SPD).

[66] Heckerman, D. A Tutorial on Learning with Bayesian Networks. Tech. rep., Microsoft Research — Advanced Technology Division, Mar. 1995. MSR–TR–95–06.

[67] Herbert, S. Features of Generic NQS. http:// www.gnqs.org/ docs/ papers/ gnqs_papers/ gnqs0013.htm, June 1996.

[68] Hockney, R. *The Science of Computer Benchmarking*, 1st ed. SIAM, 1996. ISBN 0 89871 363 3.

[69] Horvitz, E., Breese, J., and Henrion, M. Decision Theory in Expert Systems and Artificial Intelligence. Tech. rep., Palo Alto Laboratory, 1988.

[70] Huang, C., and Darwiche, A. Inference in Belief Networks: A Procedural Guide. *International journal of Approximate Reasoning 11* (1994).

[71] Hui, C., and Chanson, S. Theoretical Analysis of the Heterogeneous Dynamic Load-Balancing Problem Using a HydroDynamic Approach. *Journal of Parallel and Distributed Computing*, 43 (1997), 139–146.

[72] Hui, C., and Chanson, S. Improved Strategies for Dynamic Load Balancing. *IEEE Concurrency* (July 1999), 58–66.

[73] Hwang, K., and Xu, Z. *Scalable Parallel Computing: Technology, Architecture, Programming.* McGraw–Hill, 1998.

[74] International Business Machines (IBM). Enterprise Job Scheduling: why only the toughest survive. http:// www.tivoli.com/, 1998.

[75] JACQMOT, C., AND MILGROM, E. A Systematic Approach to Load Distribution Strategies for Distributed Systems. In *Decentralized and Distributed Systems* (Palma de Mallorca, Spain, Sept. 1993), IFIP, pp. 291–303.

[76] JENSEN, F. Bayesian Networks Basics. *AISB Quarterly 94* (1996), 9–22.

[77] JENSEN, F. *An Introduction to Bayesian Networks.* Springer-Verlag, 1996. ISBN 0-387-91502-8.

[78] JENSEN, F. Bayesian Graphical Models. *Encyclopedia of Environmetrics* (2000).

[79] JENSEN, F. Influence Diagrams. *Encyclopedia of Environmetrics* (2000).

[80] JENSEN, F., AND LAURITZEN, S. Probabilistic Networks. *Handbook of Defeasible and Uncertainty Management Systems: Algorithms for Uncertainty and Defeasible Reasoning 5* (2000), 289–320.

[81] JONES, J. Evaluation of Job Queuing/Scheduling Software: Phase 1 Report. Tech. Rep. NAS-96-009, NAS High Performance Processing Group, NASA Ames Research Center, July 1996.

[82] JONES, J. NAS Requirements Checklist for Job Queuing/Scheduling Software. Tech. Rep. NAS-96-003, NAS High Performance Processing Group, NASA Ames Research Center, Apr. 1996.

[83] JONES, J., AND BRICKELL, C. Second Evaluation of Job Queuing/Scheduling Software: Phase 1 Report. Tech. Rep. NAS-97-013, NAS High Performance Processing Group, NASA Ames Research Center, June 1997.

[84] JORDAN, M. *Learning in Graphical Models.* MIT Press, 1998. ISBN 0-262-60032-3.

[85] KAFIL, M., AND AHMAD, I. Optimal Task Assignment in Heterogeneous Distributed Computing Systems. *IEEE Concurrency* (July 1998), 42–51.

[86] KAJIYA, J. The Rendering Equation. In *Computer Graphics (SIGGRAPH'86 Proceedings)* (New York, Aug. 1986), pp. 143–150.

[87] KAPLAN, J., AND NELSON, M. A Comparison of Queueing, Cluster and Distributed Computing Systems. Technical Memorandum 109025, NASA Langley Research Center, June 1994.

[88] KENLEY, C. *Influence Diagrams Models with Continuous Variables.* PhD thesis, Stanford University, June 1986.

[89] KINGSBURY, B. The Network Queueing System (NQS). http:// www.gnqs.org /docs /papers /mnqs_papers /original_cosmic_nqs_paper.htm, Apr. 1992.

[90] Koche, J. *Fundamentos de Metodologia Científica: Teoria da ciência e prática da pesquisa*, 14 ed. Vozes, Petrópolis, Brasil, 1997. ISBN 85 326 1804 9.

[91] Krause, P. Learning Probabilistic Networks. Tech. rep., Philips Research Laboratories, 1998.

[92] Kremien, O., and Kramer, J. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Parallel and Distributed Systems* (Nov. 1992).

[93] Kremien, O., Kramer, J., and Magee, J. Scalable, Adaptive Load Sharing for Distributed Systems. *IEEE Parallel and Distributed Technology: Systems and Applications* (Aug. 1993), 62–70.

[94] Kropf, P. *Load Balancing.* Short Course on Advanced Parallel Computation, Cosmase - EPFL - Lausanne, Suisse, Mar. 1996.

[95] Kumar, V., and Grama, A. Scalable Load Balancing Techniques for Parallel Computers. Tech. rep., University of Minnesota, 1992.

[96] Kunz, T. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions on Software Engineering* (July 1991), 1327–1341.

[97] Lam, W., and Bacchus, F. Learning Bayesian Belief Networks: An Approach Based on the MDL Principle. *Computational Intelligence 10*, 4 (1994), 269–293.

[98] Lam, W., and Bacchus, F. Using New Data to Refine a Bayesian Network. In *10th Conference on Uncertainty in Artificial Intelligence* (1994), Morgan–Kaufmann, pp. 383–390.

[99] Lauritzen, S., and Jensen, F. Stable Local Computation with Conditional Gaussian Distributions. Tech. rep., Aalborg University — Departement of Mathematical Sciences, Sept. 1999. R–99–2014.

[100] Lin, F., and Keller, R. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering* (1987), 32–38.

[101] Lin, W., Lau, R., Hwang, K., Lin, X., and Cheung, P. Adaptive Parallel Rendering on Multiprocessors and Workstation Clusters. *IEEE Transactions on Parallel and Distributed Systems* (Sept. 2000).

[102] Livney, M., and Melman, M. Load Balancing in Homogeneous Broadcast Distributed Systems. In *Proceedings of the ACM Computer Network Performance Symposium* (1982), pp. 47–55.

[103] LIVNY, M. *High Performance Distributed Computing: Building a Computational Grid*, 1st ed. Morgan Kaufmann, 1997.

[104] LIVNY, M., BASNEY, J., RAMAN, R., AND TANNENBAUM, T. Mechanisms for High Throughput Computing, May 1997.

[105] LOH, P., HSU, W., C. WENTONG, AND SRISKATHAN, N. How Network Topology Affects Dynamic Load Balancing. *IEEE Parallel & Distributed Technology* (Fall 1996), 25—35.

[106] LU, Q., AND LAU, S. A Negotiation Protocol for Dynamic Load Distribution Using Batch Task Assignments. *Journal of Parallel and Distributed Computing*, 55 (1998), 166–191.

[107] LULING, R., AND MONIEN, B. A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance. In *ACM Symposium on Parallel Algorithms and Architectures* (Paderborn, Germany, 1993), ACM Press.

[108] LULING, R., MONIEN, B., AND RAMME, F. A Study on Dynamic Load Balancing Algorithms. Tech. rep., Paderborn Center for Parallel Computing, June 1992.

[109] MAHESWARAN, M., ALI, S., SIEGEL, H., HENSGEN, D., AND FREUND, R. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing 59*, 2 (Nov. 1999), 107–131.

[110] MAUI HIGH PERFORMANCE COMPUTING CENTER. Load Leveler. http:// www.lrz-muenchen.de/ services/ compute/ sp2/ loadleveler/ LoadLeveler.htm, 1997.

[111] MEHRA, P., AND WAH, B. Automated Learning of Workload Measures for Load Balancing on a Distributed System. In *1993 International Conference on Parallel Processing* (Syracuse University, Aug. 1993), CRC Press, Inc., pp. 263–270.

[112] MEHRA, P., AND WAH, B. Synthetic Workload Generation for Load-Balancing Experiments. *IEEE Parallel and Distributed Technology: Systems & Applications* (Oct. 1995), 4–19.

[113] MEHRA, P., AND WAH, B. Automated Learning of Load-Balancing Strategies in MultiProgrammed Distributed Systems. *International Journal of System Sciences* (1997).

[114] MEYER, P. *Probabilidade: Aplicaçoes a Estatistica*, 2nd ed. Livros Tecnicos e Cientificos, 1983.

[115] MINSK, M. Steps Toward Artificial Intelligence. In *Institute of Radio Engineers* (1961), pp. 8–30.

[116] MONIEN, B. Mapping and Load Balancing on Distributed Memory Machines. Paderborn Spring School - Course Notes, Apr. 1995.

[117] MONIEN, B. *Load Balancing Driven Process Migration.* EURO-PAR'96, Lyon, France, Aug. 1996. Tutorial.

[118] MORGAN, M., AND HENRION, M. *Uncertainty: A Guide to Dealing with Uncertainty in Quantitative Risk and Policy Analysis*, 1st ed. Cambridge University Press, 1990. ISBN 0 521 42744 4.

[119] MURPHY, K. A Variational Approximation for Bayesian Networks with Discrete and Continuous Latent Variables. Tech. rep., University of California — Computer Science Division, 1999.

[120] NEVES, J., MACHADO, J., COSTA, L., AND CORTEZ, P. A Software Agent Distributed System for Dynamic Load Balancing. In *Modelling and Simulation ESM'96* (Budapest, Hungary, June 1996).

[121] NI, L., XU, C., AND GENDREAU, T. A Distributed Drafting Algorithm for Load Balancing. *IEEE Transactions on Software Engineering* (Oct. 1985), 1153–1161.

[122] NIEDERMAYER, D. An Introduction to Bayesian Networks and their Contemporary Applications. Tech. rep., Dec. 1998. CS–420.

[123] OLESEN, K. Causal probabilistic Networks with both Discrete and Continuous Variables. *IEEE Transactions on Pattern Analysis and Machine Intelligence 3*, 15 (1993).

[124] OVEREINDER, B., SLOOT, P., HEEDERIK, R., AND HERTZBERGER, L. A Dynamic Load Balancing System for Parallel Cluster Computing. *Future Generation Computer Systems 12* (1996), 101–115.

[125] OZDEN, B., GOLDBERG, A., AND SILBERSCHATZ, A. Scalable and Non-Intrusive Load-Sharing in Owner-Based Distributed Systems. In *5th IEEE Symposium on Parallel and Distributed Processing* (Dallas, TE, Dec. 1993), IEEE, pp. 690–699.

[126] PAPAKHIAN, M. A Comparison of Job Management Systems from the User's Perspective. *IEEE Computational Science & Engineering* (1998).

[127] PEARL, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers, 1988. ISBN 1-55860-479-0.

[128] PEARL, J. *Bayesian Networks.* MIT Press, 1995.

[129] PLATFORM COMPUTING. LSF MultiCluster: Software for Global Load Sharing. http:// www.platform.com/ platform/, Dec. 1996.

[130] Platform Computing. LSF - Product Overview. http:// www.platform.com/ platform/, 1997.

[131] Poland, W., and Schachter, R. Mixtures of Gaussians and Minimum Relative Entropy Techniques for Modeling Continuous Uncertainties. In *9th Conference on Uncertainty in Artificial Intelligence* (1993), Morgan–Kaufmann.

[132] Pozzetti, E., and Vetland, V. *General Purpose Parallel Computers: Architectures, Programming Environments and Tools*, 1st ed. Edizioni ETS, Consiglio Nazionale delle Ricerche, 1995. chap. 12.

[133] Pulidas, S., Towsley, D., and Stankovic, J. Design of Efficient Parameter Estimators for Decentralized Load Balancing Policies. Tech. rep., University of Massachusetts, Aug. 1987.

[134] Ramamritham, K., Stankovic, J., and Zhao, W. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers* (Aug. 1989), 1110–1123.

[135] Reinhard, E. *Scheduling and Data Management for Parallel Ray Tracing*. PhD thesis, Department of Computer Science, University of Bristol, Oct. 1999.

[136] Reinhard, E., Chalmers, A., and Jansen, F. Overview of Parallel Photo–Realistic Graphics. *EuroGraphics'98: State of The Art Report* (Sept. 1998).

[137] Reinhard, E., Kok, A., and Chalmers, A. Cost Distribution for Parallel Ray Tracing. In *2nd EuroGraphics Workshop on Parallel Graphics and Visualization* (Sept. 1998), EuroGraphics, pp. 77–90.

[138] Reinhard, E., Kok, A., and Jansen, F. Cost Prediction in Ray Tracing. In *Rendering Techniques'96* (1996), Springer-Verlag, pp. 41–50.

[139] Riedl, R., and Richter, L. Classification of Load Distribution Algorithms. In *4th EuroMicro Workshop on Parallel and Distributed Processing* (Braga, Portugal, Jan. 1996), IEEE Computer Society Press, pp. 404–413.

[140] Romke, T., Rottger, M., Schroeder, U., and Simon, J. On Efficient Embeddings of Grids into Grids in PARIX. In *International Conference on Parallel Processing (EURO-PAR'95)* (University of Paderborn, 1995).

[141] Rosti, E., Smirni, E., Dowdy, L., Serazzi, G., and Sevcik, K. Processor Saving Scheduling Policies for Multiprocessor Systems. *IEEE Transactions on Computers 47*, 2 (Feb. 1998), 178–189.

[142] Russ, S., Reece, K., Robinson, J., Meyers, B., Rajan, R., Rajagopalan, L., and Tan, C. Sensitivity Evaluation of Dynamic Load Sharing in Distributed Systems. *IEEE Concurrency* (Apr. 1999), 47–55.

[143] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 1995. ISBN 0-13-103805-2.

[144] Ryou, J.-C., and Juang, J.-Y. An Efficient Load Balancing Algorithm in Distributed Computing Systems. In *5th IEEE Symposium on Parallel and Distributed Processing* (Dallas, Texas, Dec. 1993), IEEE Computer Society Press, pp. 233–240.

[145] Santos, L., Chalmers, A., and Proença, A. A Message Density Monitoring Strategy for Distributed Memory Parallel Systems. In *2nd Int. Conference on Software for MultiProcessors and SuperComputers: Theory, Practice and Experience* (Moscow, Russia, Sept. 1994), pp. 288–288.

[146] Santos, L. P. Optimização de Tráfego em Sistemas de Memória Distribuída. Master's thesis, Departamento de Informática, Universidade do Minho, Braga, Portugal, Sept. 1994.

[147] Saphir, W., Tanner, L., and Traversat, B. Job Management Requirements for NAS Parallel Systems and Clusters. Tech. Rep. NAS–95–006, NAS Scientific Computing Branch — NASA Ames Research Center, Feb. 1995.

[148] Schachter, R. Bayes–Ball: The Rational Pastime. In *14th Conference on Uncertainty in Artificial Intelligence* (1998), Morgan–Kaufmann, pp. 480–487.

[149] Schachter, R., and Kenley, R. Gaussian Influence Diagrams. *Management Science 35*, 5 (May 1989), 527–550.

[150] Schaerf, A., Shoham, Y., and Tennenholtz, M. Adaptive Load Balancing: A Study in Multi-Agent Learning. *Journal of Artificial Intelligence Research 2* (May 1995), 475–500.

[151] Scheurer, C., Scheurer, H., and Kropf, P. Load Balancing Driven Process Migration. Tech. rep., University of Berne, June 1995.

[152] Schopf, J. Structural Prediction Models for High–Perfomance Distributed Applications. In *Cluster Computing Conference* (1997).

[153] Schopf, J. A Practical Methodology for Defining Histograms for Predictions and Scheduling. In *ParCo'99* (Aug. 1999).

[154] Schopf, J., and Berman, F. Performance Prediction Using Intervals. Tech. Rep. CS97-541, University of California, San Diego, May 1997.

[155] Schopf, J., and Berman, F. Stochastic Scheduling. In *SuperComputing'99* (Portland, OR, USA, 1999).

[156] Schopf, J., and Berman, F. Using Stochastic Intervals to Predict Application Behavior on Contended Resources. In *WorkShop on Advances in Parallel Computing Models – ISPAN'99* (1999).

[157] Senar, M., Ripoll, A., Cortés, A., and Luque, E. Performance Comparison of Strategies for Static Mapping of Parallel Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 4th European PVM/MPI Users'Group Meeting* (Cracow, Poland, Nov. 1997), no. 1332 in Lecture Notes in Computer Science, Springer, pp. 575–587.

[158] Shin, K. G., and Chang, Y.-C. A Coordinated Location Policy for Load Sharing in HyperCube-Connected Multicomputers. *IEEE Transactions on Computers* (May 1995), 669–682.

[159] Shirley, P. *Physically Based Lighting Calculations for Computer Graphics.* PhD thesis, University of Illinois, Urbana-Champaign, Nov. 1991.

[160] Shivaratri, N., Krueger, P., and Singhal, M. Load Distributing for Locally Distributed Systems. *IEEE Computer* (Dec. 1992), 33–44.

[161] Smith, W., Foster, I., and Taylor, V. Predicting Application Run Times Using Historical information. In *Workshop on Job Scheduling Strategies for Parallel Processing* (1998).

[162] Smith, W., Taylor, V., and Foster, I. Using Runtime Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In *Workshop on Job Scheduling Strategies for Parallel Processing* (1999).

[163] Sohn, A., and Biswas, R. Guest Editors' Introduction. *Journal of Parallel and Distributed Computing – Special Issue on Dynamic Load Balancing*, 47 (1997), 99–101.

[164] Spiegelhalter, D., and Lauritzen, S. Sequential Updating of Conditional Probabilities on Directed Graphical Structures. *Networks 20* (1990), 579–605.

[165] Stankovic, J. An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling. *IEEE Transactions on Computers C-34*, 2 (Feb. 1985), 117–129.

[166] Stankovic, J. Stability and Distributed Scheduling Algorithms. *IEEE Transactions on Software Engineering* (Oct. 1985), 1141–1152.

[167] Stankovic, J., Ramamritham, K., and Cheng, S. Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems. *IEEE Transactions on Computers* (Dec. 1985), 1130–1143.

[168] Stankovic, J., Spuri, M., Natale, M. D., and Buttazzo, G. Implications of Classical Scheduling Results for Real–Time Systems. *IEEE Computer 28*, 5 (1995), 16–25.

[169] Suen, T., and Wong, J. Efficient Task Migration Algorithm for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems 3*, 4 (July 1992), 488–499.

[170] Sutton, R., and Barto, A. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[171] Theimer, M., and Lantz, K. Finding Idle Machines in a Workstation Based Distributed System. *IEEE Transactions on Software Engineering* (Nov. 1989), 1444–1458.

[172] Wang, Y., and Morris, R. Load Sharing in Distributed Systems. *IEEE Transactions on Computers C-34*, 2 (Mar. 1985), 204–217.

[173] Watt, A. *3D Computer Graphics*, 2nd ed. Addison-Wesley, 1993. ISBN 0-201-63186.

[174] Watt, A., and Watt, M. *Advanced Animation and Rendering Techniques: Theory and Practice*, 1st ed. Addison-Wesley, 1998. ISBN 0-201-54412-1.

[175] Watts, J., and Taylor, S. A Practical Approach to Dynamic Load Balancing. *IEEE Transactions on Parallel and Distributed Systems 9*, 3 (Mar. 1998), 235–248.

[176] Weiss, G. *MultiAgent Systems: A Modern Approach to Distributed Artificial Intelligence*, 1st ed. The MIT Press, 1999.

[177] White, B., Grinshaw, A., and Nguyen-Tuong, A. Grid–Based File Access: The Legion I/O Model. In *9th IEEE International Symposium on High Performance Distributed Computing* (Pennsylvania, U.S.A., Aug. 2000), IEEE Computer Society Press.

[178] Willebeek-LeMair, M., and Reeves, A. Strategies for Dynamic Load Balancing on Highly Paralel Computers. *IEEE Transactions on Parallel and Distributed Systems 4* (Sept. 1993), 979–993.

[179] Xu, C., and Lau, F. *Load Balancing in Parallel Computers: Theory and Practice.* Kluwer Academic Publishers, 1997.

[180] XU, C., LULING, R., MONIEN, B., AND LAU, F. C. M. An Analytical Comparison of Nearest Neighbours Algorithms for Load Balancing in Parallel Computers. In *9th International Parallel Processing Symposium* (Paderborn, Germany, Apr. 1995).

[181] ZAKI, M., LI, W., AND PARTHASARATHY, S. Customized Dynamic Load Balancing for a Network of Workstations. *Journal of Parallel and Distributed Computing 43* (1997), 156–162.

[182] ZHOU, S. A Trace Driven Simulation Study of Dynamic Load Balancing. *IEEE Transactions on Software Engineering* (Sept. 1988), 1327–1341.

[183] ZOMAYA, A., CLEMENTS, M., AND OLARIU, S. A Framework for Reinforcement–Based Scheduling in Parallel Processor Systems. *IEEE Transactions on Parallel and Distributed Systems 9*, 3 (Mar. 1998), 249–259.

Index

average, recency–weighed, **161**, 200

Bayes' rule, 21, **89**
bayesian networks, **85**
bidding algorithm, 35, **67**

certainty factors, **20**
congestion problem, **33**
coordination problem, **33**, 41, 56
credit–assignment problem, **48**

d–separation, **88**, 176
data locality, **41**, 119
decision networks, **94**
decomposition
  domain, **114**, 144
  functional, **114**, 144
default reasoning, **20**
degree of balancing, **26**
Dempster-Shafer theory, **20**
Distributed Job Management System, 14, **72**
distributed shared memory, *see* parallel architectures, DSM
distributed shared system, **12**, 13
divisible loads, **40**, 115
drafting algorithm, **67**

environment, 10
  measurability, **15**
  properties, **15**
execution model, **11**, 14
  structural, **21**, 31
exploitation, **70**
exploration, **70**

fuzzy logic, **20**

High Throughput Computing, **73**

idle–while–waiting condition, **9**
illumination model, **135**
  ambient light, **137**
  diffuse reflection, 135, **136**
  diffuse transmission, **135**
  filtered transparency, **133**
  global, 135, **139**
  index of refraction, 138, **139**
  Lambertian reflection, *see* diffuse reflection
  local, **135**
  Phong model, **136**
  refraction, **133**
  Snell's Law, **139**
  specular reflection, 135, **137**
  specular transmission, 135, **138**
information aging, **16**, 18, 35

joint distribution, 20, **80**

knowledge engineering, **95**, 169

load balancing, **26**
load sharing, **26**

mapping problem, **28**
Maximum Expected Utility, 21, **93**
measurability, *see* environment, measurability
metrics
  application–dependent, **37**, 117
  application–independent, **37**, 117

environment, 10, **116**, 159
foreground workload, **116**, 159
performance, 42, **116**, 159
resources' capacity, **117**, 160
scheduling overhead, 42, **117**, 161
multithreading, **149**

nearest–neighbour policy, 33, **62**
diffusion, **63**
dimension exchange, **62**
gradient, **63**
Neutral File Format, **146**
NFF, *see* Neutral File Format
NP–complete, 2, **28**

parallel application characteristics, **113**
parallel architectures
memory organisation
CC–NUMA, **112**
COMA, **112**
DSM, **112**, 115, 148
NORMA, **112**
NUMA, **112**, 115, 148
UMA, **112**
shared–disk, **112**
shared–memory, **112**
shared–nothing, **112**
SMP, **112**
performance model, 41, **116**, 159
probability theory, 20, **79**
probing algorithm, **65**
process checkpointing, **73**
processor farm, **61**, 122
processor trashing, **26**, 46
profitability determination, **26**, 42

ray tracing, 114, **131**
3D space partitioning, **144**, 146
adaptive depth control, **142**, 146
adaptive supersampling, **141**, 146
coherence, **143**
occluding object buffer, **143**, 146
primary rays, **133**
rendering equation, *see* rendering equation
secondary rays, **139**
shadow rays, **133**
reinforcement learning, 21, **69**
rendering equation, **132**, 140

scalability, **45**, 193
scheduling
agent, **11**
application level, **13**
requirements, 14
centralised, **32**, 61
direct costs, **43**, 118, 162
distributed, **32**
domains, **33**
effectiveness, **42**
efficiency, **42**
indirect costs, **43**, 118, 162
overheads, **42**
performance, **42**
real–time systems, **12**
system level, **14**, 72
scheduling policies
adaptive, **17**, 30
opportunities, 17
classification scheme, 52
Casavant's taxonomy, **53**
decision × migration space, **52**
ESR, **57**, 125
Families of strategies, **54**
Load, action and solution models, **56**
components, **34**
information policy, **35**
location policy, **41**
selection policy, **39**
transfer policy, **38**

dynamic, **16**, 28
sensor based, **17**, 30
static, **16**, 28
scheduling problem, **10**
global, **10**
local, **10**
sequential update, **201**
simulated annealing, **31**, 71
stability, **46**
stochastic learning automata, 21, **69**

task, **9**

uncertainty, 18, **77**
reasons, 18
utility, **92**
function, **92**, 185

variable
decision, **94**, 175
random, **86**, 168, 171, 175
utility, **94**, 175

workload
background, 115, 117, **125**
foreground, **126**
synthetic, **126**