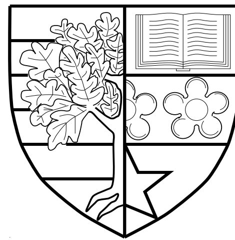
Cost-Driven Autonomous Mobility

by

Xiao Yan Deng



Submitted for the Degree of

Doctor of Philosophy

at Heriot-Watt University

on Completion of Research in the

School of Mathematical and Computer Sciences

June 2007

This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that the copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author or the university (as may be appropriate).

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

____

Xiao Yan Deng (Candidate)

____

Greg Michaelson and Phil Trinder (Supervisor)

____

Date

To My Parents

Deng Jin Shan and Yu Yi

Abstract

Developments in distributed system technology facilitate the sharing of resources, even at a global level. This thesis explores sharing computational resources using mobile computations, agents, and autonomic techniques. We propose *autonomous mobile programs* (AMPs) which are aware of their resource needs and sensitive to the environment in which they execute. AMPs periodically use a cost model to decide where to execute in a network. Unusually this form of autonomous mobility affects only where the program executes and not what it does. We present a generic AMP cost model, together with a validated instantiation and comparative performance results for four AMPs. We demonstrate that AMPs are able to dynamically relocate themselves to minimise execution time in the presence of varying network resources.

Collections of AMPs effectively perform decentralised dynamic load balancing. Experiments on small LANs show that collections of AMPs quickly obtain and maintain optimal or near-optimal balance. The advantages of our decentralised approach are that it has the potential to scale to very large and dynamic networks, and to achieve improved balance, and offers guarantees to limit overheads under reasonable assumptions.

In an autonomous mobile program, the program must contain explicit control of self-aware mobile coordination. To encapsulate this for common patterns of computation over collections, *autonomous mobility skeletons* (AMSs) are proposed. These are akin to algorithmic skeletons in being polymorphic higher order functions, but where algorithmic skeletons abstract over parallel coordination, AMSs abstract over autonomous mobile coordination. AMS cost models have been built over collection iterations. The `automap`, `autofold` and `AutoIterator` AMSs are presented, together with performance measurements for Jocaml, Java Voyager, and JavaGo implementations on LANs.

An AMS considers only the cost of the current collective computation, but it is more useful to know the cost of the entire program. We have extended our AMS

cost models to be parameterised on the cost of the remainder of the program. A cost calculus to estimate the costs for the remainder of a computation at arbitrary points has been built. An automatic Jocaml cost analyser based on the calculus produces cost equations parameterised on program variables in context, and may be used to find both cost in higher-order functions and the cost for the remainder of the program. Costed autonomous mobility skeletons (CAMSs) have been built, which not only encapsulate common patterns of autonomous mobility but take additional cost parameters to provide costs for the remainder of the program. Potential performance improvements are assessed by comparing CAMS to AMS programs. The results show that CAMS programs perform more effectively than AMS programs, because they have more accurate cost information. Hence a CAMS program may move to a faster location when the corresponding AMS program does not.

Acknowledgement

First and foremost I would like to thank my family for supporting me whenever I need help and for trusting me alway even when I want to give up.

I am indebted to Greg Michaelson and Phil Trinder, not only as my supervisors, for guiding my research over the course of my PhD and for helping me in avoiding the pitfalls on the way to obtaining a PhD degree, but also as my friends, for providing invaluable encouragement and sound advice.

I would like to thank all my friends and colleagues in Heriot-Watt University for their help. Finally, I am grateful to Heriot-Watt University and School of Mathematical and Computer Sciences for offering me an opportunity of doing research and for the scholarships that made this degree possible.

Contents

1 Introduction **1**

1.1 Context 1

1.2 Thesis Contributions 2

1.3 Thesis Outline 4

1.4 Publications 5

2 Background **7**

2.1 Mobility & Mobile Languages 7

2.1.1 Mobility Taxonomy 7

2.1.2 Calculi with Mobility 8

2.1.3 Properties of Mobile Languages 11

2.1.4 Java Voyager 13

2.1.5 JavaGo 14

2.1.6 Other Imperative Languages 17

2.1.7 MobileML 18

2.1.8 Nomadic Pict 19

2.1.9 Mobile Haskell 21

2.1.10 Jocaml 23

2.1.11 Mobile Language Summary 27

2.2 Agents & Autonomous Systems 28

2.2.1 Agents 28

2.2.2 Autonomous Systems 30

2.2.3 Discussion 32

2.3 Load Management Systems 33
2.3.1 Static/Dynamic Load Management 33
2.3.2 Centralised & Decentralised Load Management 36
2.3.3 Push & Pull Policies 37
2.3.4 Preemptive & Non-Preemptive Load Management 37
2.3.5 Dynamic Load Management Systems 38
2.4 Cost Models 40
2.4.1 Computation Cost Models 40
2.4.2 Parallel Coordination Models 42
2.4.3 Cost Models for Algorithmic Skeletons 43
2.4.4 Cost Model Summary 47
2.5 Summary 47

3 Autonomous Mobile Programs 49
3.1 Introduction 49
3.2 Related Work 51
3.3 Cost Models 52
3.3.1 Traditional Systems Costs 52
3.3.2 AMPs Costs 53
3.4 AMP Implementation 55
3.4.1 Mobile Programming Languages Choice 55
3.4.2 Matrix Multiplication Cost Model 56
3.4.3 Validating Matrix Multiplication Cost Model 58
3.4.4 Matrix Multiplication AMP Speed Up Measurement 61
3.5 AMP Movement Measurement 62
3.6 Summary 63

4 Autonomous Mobility Skeletons 65
4.1 Introduction 65
4.2 Autonomous Mobility Map (`automap`) 68
4.2.1 Collection Iteration Cost Model 68
4.2.2 Auxiliary Functions for AMS in Jocaml 70

4.2.3 Jocaml `automap` Design and Implementation 72
4.2.4 Jocaml `automap` Cost Model Validation 72
4.2.5 Jocaml `automap` AMPs Speed Up Measurement 76
4.2.6 Java Voyager `automap` Design and Implementation 77
4.2.7 Java Voyager `automap` Cost Model Validation 78
4.2.8 Java Voyager `automap` AMPs Speed Up Measurements . . 81
4.3 Autonomous Mobility Fold (`autofold`) 82
4.3.1 Jocaml `autofold` Design and Implementation 83
4.3.2 Jocaml `autofold` Cost Model Validation 83
4.3.3 Jocaml `autofold` AMPs Speed Up Measurements 86
4.3.4 Java Voyager `autofold` Design and Implementation 87
4.3.5 Java Voyager `autofold` Cost Model Validation 87
4.3.6 Java Voyager `autofold` AMPs Speed Up Measurements . . 89
4.4 An Autonomous Mobile Iterator 89
4.4.1 `AutoIterator` & JavaGo 89
4.4.2 `AutoIterator` Implementation 91
4.4.3 `AutoIterator` AMPs Speed Up Measurements 92
4.5 AMP with AMS Movement Measurement 92
4.6 Jocaml and Java Voyager Comparison 95
4.6.1 Mobility Behaviour Comparison 95
4.6.2 Computation Time Comparison 97
4.6.3 Communication Time Comparison 98
4.6.4 Coordination Time Comparison 99
4.7 Other Skeletons 99
4.7.1 `PIPE` Skeleton 99
4.7.2 `FARM` Skeleton 100
4.7.3 `DC` Skeleton 100
4.7.4 `RaMP` Skeleton 101
4.7.5 Discussion 101
4.8 Summary 101

5 Autonomous Load Management 103

5.1 Introduction . . . 104

5.1.1 Autonomous Load Management . . . 104

5.1.2 Autonomous Load Management Activities . . . 104

5.1.3 Autonomous Load Management Policies . . . 105

5.2 Load Server Architecture . . . 106

5.2.1 Load Server and Non-Load Server System Structure . . . 107

5.2.2 Information Renewal Time . . . 108

5.2.3 AMP Coordination Time . . . 110

5.3 Collections of AMPs Measurements . . . 111

5.3.1 Collections of AMPs on Homogeneous Networks . . . 111

5.3.2 Collections of AMP on Heterogeneous Network . . . 115

5.4 Summary . . . 119

6 Automatic Continuation Cost Analysis 121

6.1 Introduction . . . 122

6.2 Syntax of Language $\mathcal{J}'$ . . . 123

6.3 Cost Calculus for $\mathcal{J}'$ . . . 124

6.3.1 Index Semantics for $\mathcal{J}'$ . . . 125

6.3.2 Cost Semantics for $\mathcal{J}'$ . . . 127

6.3.3 Costafter Semantics for $\mathcal{J}'$ . . . 130

6.4 Costed Autonomous Mobility Skeletons . . . 136

6.4.1 Cost Model for Costed Autonomous Mobility Skeletons . . . 136

6.4.2 An Implementation of Costed Autonomous Mobility Skeletons . . . 138

6.5 Evaluating Costed Autonomous Mobility Skeletons . . . 139

6.5.1 Single Higher Order Function Examples . . . 139

6.5.2 Sequential Composed Higher Order Function Examples . . . 141

6.5.3 Performance Comparison with Changing Load . . . 146

6.5.4 Conclusion . . . 147

6.6 Automatic Continuation Cost Analyser . . . 148

6.6.1 Structure of the Automatic Continuation Cost Analyser . . 148
6.6.2 An Implementation of the Cost Calculus 149
6.6.3 Generator: Generating Autonomous Mobility Skeletons . . 151
6.6.4 Matrix Multiplication Example 152
6.6.5 Comparing Automatic and Hand Analysis 153
6.7 Summary 154

7 Conclusion and Future Work 157
7.1 Summary 157
7.2 Contributions and Further Work 159
7.2.1 AMPs on Large Scale Networks 160
7.2.2 Autonomous Mobility for Irregular Computation 163
7.2.3 Resource Driven Mobility 164
7.2.4 Automatic Continuation Cost Analysis for Java 165

A Cost Calculus for $\mathcal{J}$ 167
A.1 Syntax of $\mathcal{J}$ 167
A.2 Cost Calculus for $\mathcal{J}$ 167
A.2.1 Index Semantics 167
A.2.2 Cost Semantics 170
A.2.3 Costafter Semantics 175

B Autonomous Mobility Skeletons Code 184
B.1 Jocaml Autonomous Mobility Skeletons 184
B.2 Voyager Autonomous Mobility Skeletons 192
B.2.1 `Auto` Class Implementation 192
B.2.2 Load Server Implementation 201
B.3 JavaGo Autonomous Mobile Iterator 209

C Balance Status of Collections of AMPs 214

D Automatic Cost Analyser Validation 217

Bibliography **217**

List of Tables

2.1 Summary of Calculi 11

2.2 Summary of Languages 27

3.3 Jocaml AMP Matrix Multiplication Computation Time Validation 59

3.4 Jocaml AMP Matrix Multiplication Communication Time Calculation 60

3.5 Jocaml Coordination Time Calculation 61

4.6 Matrix Multiplication Computation Time Validation (Jocaml) . . 73

4.7 Ray Tracing Computation Time Validation (Jocaml) 74

4.8 Jocaml AMS Matrix Multiplication Communication Time Validation 75

4.9 Jocaml AMS Ray Tracing Communication Time Validation 75

4.10 Jocaml Coordination Time Validation 76

4.11 Matrix Multiplication Computation Time Validation (Voyager) . . 80

4.12 Ray Tracing Computation Time Validation (Voyager) 80

4.13 Voyager AMS Matrix Multiplication Communication Time Validation 81

4.14 Voyager AMS Ray Tracing Communication Time Validation . . . 81

4.15 Voyager Coordination Time Validation 82

4.16 Coin Counting Computation Time Validation (Jocaml) 85

4.17 Jocaml AMS Coin Counting Communication Time Validation . . 85

4.18 Coin Counting Computation Time Validation (Voyager) 88

4.19 Voyager AMS Coin Counting Communication Time Validation . . 89

4.20 Jocaml and Voyager Matrix Multiplication Execution time Comparison . . . 97
4.21 Jocaml and Voyager Ray Tracing Execution time Comparison . . 97
4.22 Jocaml and Voyager Coin Counting Execution time Comparison . 98

5.23 Load Server Information Collection Time . . . 109
5.24 Test AMPs Coordination Time in load Server Structure . . . 110
5.25 Verified Optimal Balance . . . 111
5.26 Prediction CPU Speed for Each AMPs (25 AMPs on 15 Locations) 117
5.27 Prediction CPU Speed for Each AMPs (24 AMPs on 15 Locations) 117
5.28 Prediction CPU Speed for Each AMPs (23 AMPs on 15 Locations) 117

C.29 Prediction CPU Speed each AMPs Get(20 AMPs on 15 Locations) 215
C.30 Prediction CPU Speed each AMPs Get(19 AMPs on 15 Locations) 215
C.31 Prediction CPU Speed each AMPs Get(18 AMPs on 15 Locations) 215

D.32 `camap` and `automap` Matrix Mutiplication Movement Comparison 217
D.33 `camap` and `automap` Ray Tracing Movement Comparison . . . 218
D.34 `camap` and `automap` Double Matrix Multiplication Movement Comparison . . . 218
D.35 `camap` and `automap` Invertible Matrix Movement Comparison . . . 219
D.36 `camap` and `automap` Double Ray Tracing Movement Comparison . 220
D.37 `camap` and `automap` Five Matrix Multiplications Movement Comparison . . . 221
D.38 `camap` and `automap` Invertible Matrix Movement Comparison with Changing Loads . . . 222
D.39 `camap` and `automap` Five Matrix Multiplications Movement Comparison with Changing Loads . . . 223
D.40 Automatic and Byhand Cost Double Matrix Multiplications Movement Comparison . . . 224
D.41 Automatic and Byhand Cost Invertible Matrix Movement Comparison . . . 225

D.42 Automatic and Byhand Costs Double Ray Tracing Movement Comparison 226

List of Figures

2.1 Mobility Example in Voyager 15
2.2 Mobility Example in JavaGo 16
2.3 Mobile Channels in mHaskell 22
2.4 Mobility Example in mHaskell (Client Side) 23
2.5 Mobility Example in mHaskell (Server Side) 23

3.6 Agents, Autonomous Systems, and AMPs 51
3.7 Generic Cost Model for Autonomous Mobile Programs 54
3.8 Jocaml Three `for` loop Matrix Multiplication 56
3.9 Cost Model for AMP Matrix Multiplication 57
3.10 Static and AMP Matrix Multiplication Execution Time 62
3.11 AMP Matrix Multiplication Movement Validation 63

4.12 Skeleton Taxonomy 66
4.13 Cost Model for Collection Iteration 69
4.14 Jocaml `getGran`: Calculating Coordination Granularity 70
4.15 Jocaml `check_move`: Deciding to Move in Jocaml 71
4.16 Jocaml `getInfo` 71
4.17 Jocaml `automap` 72
4.18 Jocaml `automap` Matrix Multiplication 72
4.19 Jocaml `automap` Ray Tracing 73
4.20 Static and AMS Matrix Multiplication Execution Time Comparison (Jocaml) 77
4.21 Static and AMS Ray Tracing Execution Comparison Times (Jocaml) 77

4.22 Java Voyager `automap` . . . 78
4.23 Java Voyager `automap` Mobile Matrix Multiplication . . . 79
4.24 Java Voyager `automap` Ray Tracing . . . 79
4.25 Static and AMS Matrix Multiplication Execution Times Comparison (Voyager) . . . 82
4.26 Static and AMS Ray Tracing Execution Times Comparison (Voyager) 82
4.27 Jocaml `autofold` Definition . . . 83
4.28 Jocaml `autofold` Coin Counting . . . 84
4.29 Static and AMS Coin Counting Execution Time (Jocaml) . . . 86
4.30 Java Voyager `autofold` . . . 87
4.31 Voyager `autofold` Coin Counting . . . 88
4.32 Static and AMS Coin Counting Execution Times (Voyager) . . . 90
4.33 Mobility Structure of `automap` . . . 91
4.34 Mobility Structure of `AutoIterator` . . . 91
4.35 JavaGo `autoNext` Method in `AutoIterator` Class . . . 92
4.36 JavaGo `AutoIterator` Matrix Multiplications . . . 93
4.37 Static and `AutoIterator` Matrix Multiplications Execution Times 93
4.38 AMS Matrix Multiplication Movement Validation . . . 94
4.39 AMS Ray Tracing Movement Validation . . . 94
4.40 AMS Coin Counting Movement Validation . . . 94
4.41 Mobility Behaviour of Jocaml autonomous mobility skeletons . . . 96
4.42 Mobility Behaviour of Java Voyager autonomous mobility skeletons 96
4.43 Matrix Multiplication Communication Time Comparison . . . 99
4.44 Ray tracing Multiplication Communication Time Comparison . . 99

5.45 System with Load Server Architecture (LS) . . . 107
5.46 System without Load Server Architecture (NLS) . . . 107
5.47 Load Server and Non-Load Server Information Collecting Time . . 109
5.48 Load Server and Non-Load Server Coordination Time Comparison 110
5.49 Distribution of 7 AMPs on 3 Locations . . . 112
5.50 Distribution of 5 AMPs on 3 Locations . . . 113

5.51 Distribution of 9 AMPs on 3 Locations 113
5.52 Distribution of 7 AMPs on 4 Locations 113
5.53 Distribution of 10 AMPs on 4 Locations 113
5.54 Distribution of 13 AMPs on 4 Locations 113
5.55 Distribution of 9 AMPs on 5 Location 113
5.56 Distribution of 6 AMPs on 3 Locations 114
5.57 Distribution of 5 AMPs on 2 Locations 114
5.58 Rebalancing: 7 AMPs Adding 3 More AMPs on 4 Locations . . . 114
5.59 Rebalancing: 5 AMPs Adding 4 More AMPs on 3 Locations . . . 114
5.60 Rebalancing: 10 AMPs Removing 5 AMPs on 3 Locations 115
5.61 Distribution of 25 AMPs on Heterogeneous Network (15 Locations) 116
5.62 Actual CPU Speed for Each AMP (25 AMPs on 15 Locations) . . 118
5.63 Actual CPU Speed for Each AMP (24 AMPs on 15 Locations) . . 118
5.64 Actual CPU Speed for Each AMP (23 AMPs on 15 Locations) . . 118
5.65 Distribution of 20 AMPs on Heterogeneous Network (10 Locations) 119

6.66 Syntax of $\mathcal{J}'$ 124
6.67 Semantic Functions 125
6.68 Index Semantics for $\mathcal{J}'$ 126
6.69 An Example of Index in $\mathcal{J}'$ 127
6.70 Cost Semantics for $\mathcal{J}'$ 128
6.71 An Example of Costing in $\mathcal{J}'$ 129
6.72 Definition of Syntactic Expression Equality in $\mathcal{J}'$ 131
6.73 Definition of Contains in $\mathcal{J}'$ 132
6.74 An Example of Contains in $\mathcal{J}'$ 133
6.75 Costafter Semantics for $\mathcal{J}'$ 134
6.76 An Example of Costafter in $\mathcal{J}'$ 136
6.77 Cost Model for CAMS 137
6.78 Implementation of `camap` in Jocaml 138
6.79 CAMS and AMS Matrix Multiplication Execution Time Comparison 140
6.80 CAMS and AMS Ray Tracing Execution Time Comparison 140

6.81 CAMS and AMS Double Matrix Multiplication Execution Time Comparison . . . 141
6.82 CAMS and AMS Invertible Matrix Execution Time Comparison . 143
6.83 CAMS and AMS Double Ray Tracing Execution Time Comparison 143
6.84 CAMS and AMS Five Matrix Multiplications Execution Time Comparison . . . 144
6.85 CAMS and AMS Invertible Matrix Execution Time Comparison with Changing Loads . . . 146
6.86 CAMS and AMS Five Matrix Multiplications Execution Time Comparison with Changing Loads . . . 146
6.87 Structure of Automatic Continuation Cost Analyser . . . 149
6.88 Implementation of Index Function . . . 149
6.89 Implementation of Cost Function . . . 150
6.90 Implementation of Costafter Function in Jocaml . . . 150
6.91 The Source Code of Matrix Multiplication for Automatic Cost Analyser . . . 153
6.92 The Target Code of Automatic Cost Analyser . . . 154
6.93 Automatic and Byhand Cost Double Matrix Multiplications Execution Time Comparison . . . 155
6.94 Automatic and Byhand Cost Invertible Matrix Execution Time Comparison . . . 155
6.95 Automatic and Byhand Costs Double Ray Tracing Execution Time Comparison . . . 155

7.96 System with Super Load Server on Large Scale Network . . . 161

A.97 Syntax of $\mathcal{J}$ . . . 168
A.98 Index Semantics of $\mathcal{J}$ . . . 169
A.99 Pattern Cost Semantics of $\mathcal{J}$ . . . 171
A.100 Cost Semantics of $\mathcal{J}$ . . . 173
A.101 Definition of Expression Equality in $\mathcal{J}$ . . . 176
A.102 Definition of Contains in $\mathcal{J}$ . . . 178

A.103 Definition of Contains in $\mathcal{J}$ Cont. 179
A.104 Costafter Semantics of $\mathcal{J}$ 180
A.105 Costafter Semantics of $\mathcal{J}$ Cont. 181

C.106 Actual CPU Speed for Each AMP (20 AMPs on 10 Locations) . . 216
C.107 Actual CPU Speed for Each AMP (19 AMPs on 10 Locations) . . 216
C.108 Actual CPU Speed for Each AMP (18 AMPs on 10 Locations) . . 216

Chapter 1

Introduction

1.1 Context

Developments in networks have made it possible to exploit the computational power and resources available in global networks [42, 22]. Load management systems have been built for resources sharing using mobile computation, agents, and autonomic techniques [42].

One way of using the resources on both local and global networks is through mobile computations especially using *mobile languages* e.g. Jocaml [66] and Java Voyager [127] where executing computations can move in the network in order to better use the resources available. Mobile computation with mobile languages gives programmers control over the placement of code or active computations across a network for sharing computational resources [71]. Basically a mobile program can transport its state and code to another location in the network, where it resumes execution [75].

The agents community has focused on autonomous problem solving e.g. building self-managing, or autonomous, systems which can act flexibly in uncertain and dynamic environments. Mobile languages potentially allow agents to move more flexibly in a large scale network. That is, a mobile agent can migrate across a network to locate the resources it requires.

In distributed resource sharing systems, the task of placing work on location is termed load management, and is a branch of global scheduling techniques [24]. In load management systems, programs are transported from heavily loaded locations to lightly loaded locations to use the computational power efficiently. Using mobile computation, more flexible and efficient applications can be developed where a program can move between locations to better utilise computational resources.

One of the most important issues in load management systems, is to identify effective techniques for the distribution of the work over the locations, to achieve performance goals such as balancing the load of each location, or minimising execution time. One technique is to use *cost models* to estimate the execution time of a program [101], and the accuracy of the cost model is crucial.

This thesis investigates load management in collections of autonomous mobile computations. Programs using this kind of load management can in principle control when and where to move by consulting cost models, e.g. if a program determines that the current location is busy it will find a faster processor and migrate there.

1.2 Thesis Contributions

The research contributions together with the publications representing them are as follows.

- **Autonomous Mobile Programs** [35]: To manage load on large and dynamic networks we propose *autonomous mobile programs* (AMPs) that periodically use a cost model to decide where to execute in the network. Unusually this form of autonomous mobility affects only where the program executes and not what it does. Experiments comparing the execution time of static and AMP programs suggest that AMPs are able to dynamically relocate themselves to minimise execution time in the presence of varying external loads on shared locations.

- **The Design, Implementation and Evaluation of Autonomous Mobility Skeletons** [34]: In an autonomous mobile program, the programmer must explicitly control when the program moves. *Autonomous mobility skeletons* (AMSs) encapsulate self-aware mobile coordination for common patterns of computation over collections. AMSs are akin to algorithmic skeletons in being polymorphic higher order functions, but where algorithmic skeletons abstract over parallel coordination, AMSs abstract over autonomous mobile coordination. We have developed `automap`, `autofold` AMSs in the mobile languages Jocaml and Java Voyager, and `AutoIterator`, an unusual skeleton that abstracts over the `Iterator` interface commonly used with Java collections. While the other skeletons can be implemented using weak mobility, `AutoIterator` requires strong mobility and hence is implemented in JavaGo.

- **Cost Models for Autonomous Mobile Programs** [33, 35]: To build autonomous mobile programs, a generic cost model is developed. The cost model considers the relative CPU speeds of local and remote locations in the network, the total work the program carries out, the time elapsed at the current location, etc. to predict how long the program will take if it is running on the current location and how long will it take if it moves to a remote location. Problem specific cost models for matrix multiplication using three `for` loops (Chapter 3) and for AMSs e.g. `automap` and `autofold` in Jocaml and Java Voyager (Chapter 4) have been instantiated and validated.

- **Automatic Continuation Cost Analysis**: A limitation of the AMS is that they assume that a single higher-order function is the dominating computation for the program. In general, to deploy autonomous mobility effectively, it is necessary to know the cost of the remainder of the program, not just of a single iteration. Thus, a cost calculus to estimate the costs for the remainder of a computation at arbitrary points has been built. We have extended our AMS cost model to be parameterised on the cost of the remainder of the program. Costed autonomous mobility skeletons (CAMSs)

have been built based on the extended cost model. The CAMSs not only encapsulate the common patten of autonomous mobility but take additional cost parameters representing the costs of the remainder of the program. We have constructed an automatic cost analyser which implements the cost calculus for a Jocaml subset, produces cost equations parameterised on program variables in context, and may be used to find both cost in higher-order functions and the cost for the remainder of the program. Potential performance improvements are assessed by comparing CAMS to AMS programs and show that CAMS programs often execute faster than AMS programs, because they have more accurate cost information i.e. including the cost of the remainder of the program. Hence a CAMS program may move to a faster location when the corresponding AMS program does not.

1.3 Thesis Outline

The structure of this thesis is as follows.

Chapter 2 introduces related work, and covers concepts of *mobility*, *mobile languages*, *agents*, *autonomous systems*, *load management*, and *cost models*.

Chapter 3 explores *autonomous mobile programs*(AMPs) which are aware of their computational resource needs and sensitive to the environment in which they execute. To build autonomous mobile programs, a generic cost model is developed. Problem specific cost models are then developed and validated. Experiments show that AMPs react appropriately to changes in their environment.

Chapter 4 introduces *autonomous mobility skeletons* (AMSs) which encapsulate self-aware mobile coordination for common patterns of computation namely `automap`, `autofold`, and `AutoIterator`. A instantiated cost model for these AMSs has been built and validated. The experiments in this chapter suggest that AMPs with autonomous mobility skeletons are able to dynamically relocate themselves to minimise execution time. This chapter also shows that other skeletons e.g. `PIPE`, `FARM`, and `RaMP` can be presented as autonomous mobility skeletons using `automap` or `autofold`.

Chapter 5 presents the behaviour of collections of AMPs in both homogeneous and heterogeneous LANs. This chapter introduces a load server architecture to reduce the time to collect network load information, and then presents collections of AMPs which performs decentralised dynamic load balancing. Experiments show that collections of AMPs quickly obtain and maintain optimal or near-optimal balance. Furthermore, the system maintains balance as AMPs are added or removed.

Chapter 6 introduces a cost calculus to provide a cost for the reminder of a program at arbitrary points for a language $\mathcal{J}'$, a subset of Jocaml. We extend the AMS cost model to be parameterised on the cost of the remainder of the program and introduce costed autonomous mobility skeletons (CAMSs) e.g. `camap` and `cafold`, which use additional cost parameters to calculate the cost of completing the program execution. We have implemented an automatic cost analyser for $\mathcal{J}'$ based on the calculus which inspects programs and replaces higher-order functions with CAMSs. AMPs with sequences of higher-order functions are tested to evaluate the performance of CAMSs against AMSs.

Chapter 7 summaries the thesis and describes future work.

1.4 Publications

Unless otherwise stated, the work presented throughout this thesis including the following research publications is carried out by the author with contributions from her supervisors Prof. G. J. Michaelson and Dr. P. W. Trinder.

- Xiao Yan Deng, Greg Michaelson and Phil Trinder. Towards High Level Autonomous Mobility, in *Draft proceedings of Trends in Functional Programming*, pages 97-112, Munich, Germany, November 2004.

- Xiao Yan Deng, Phil Trinder and Greg Michaelson. Autonomous Mobile Programs. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2006 Main Conference Proceedings) (IAT'06)*, pages 177–186, December 2006. Hong Kong, IEEE Computer Society.

- Xiao Yan Deng, Greg Michaelson and Phil Trinder. Autonomous Mobility Skeletons, in *Journal of Parallel Computing*, Volume 32, Issues 7-8, Pages 463-478, September 2006.

Chapter 2

Background

Developments in distributed system technology have made it possible to build global computing platforms e.g. load management systems for sharing computational power and other resources using mobile computation, agents, and autonomic techniques [42]. In load management systems, cost models could be used to predict the behaviours of the system and decide the following behaviours of the system. Different cost models lead to different efficiency of the system. So suitable cost models are important issues in the self-managing systems.

This chapter introduces concepts of mobility, agents, autonomous systems, load management systems, and cost models.

2.1 Mobility & Mobile Languages

2.1.1 Mobility Taxonomy

The idea of mobility is not really new, and in recent years many different kinds of mobility have been explored in computer systems. These include hardware mobility and software mobility; process migration and mobile languages; weak mobility and strong mobility.

Generally speaking, mobility can be classified as *hardware mobility* or *software mobility*. Hardware mobility, which means the mobility of physical devices such as laptops, palm computers, and PDAs, deals with mobile computing. In contrast

software mobility, which means computation can move form location to location, deals with mobile computations [17].

Software mobility can be classified as *process migration*, or *mobile languages*. The main difference is which entity decides what is going to migrate. Mobile languages give the programmer the ability to control when and where to move, whereas in process migration the system decides when the process should move, so the migration is transparent to the programmer. Much research was done in the area of distributed operating system determined migration e.g. MOSIX [12], Sprite [36] and V Kernal [118].

Strong mobility and *weak mobility* is a classification for mobility given in[48]. Weak mobility e.g. `moveTo` in Java Voyager [100] is the ability to move only code from one location to another, while strong mobility is the ability of moving code together with the execution state of the program. Strong mobility is also called *transparent migration*, which is a migrated program resumes its execution at a destination site with exactly the same execution state as before migration began[51], e.g. `go` operation in Jocaml [44].

2.1.2 Calculi with Mobility

Process algebras are valuable mathematical tools for reasoning about the behaviour of concurrent and communication system. In the last ten years, researchers have produced semantics that allow communication channels or even processes to be communicated. Some process calculi feature the ability to dynamically create and exchange channel names which is referred to as *mobility*. In[30], Dal Zilio classified calculi which support mobility into two kinds. The first involves mobility of *Names* (Channels[86]), which captures the notion of the capability for a process to exchange names as values. The second involves mobility of *processes*, which means the migration of code or agents instead of the migration of references.

Other features of calculi include the ability to model:

- *Location*: For mobile computation, location is a very important notion,

where processes can migrate, so it is very important for calculi which present mobility of processes to model locations.

- *Security*: A major motivation for mobile computation is to share resources among mobile computational entities. In order to bring about the advantages of mobile computation, safety and security must be taken into consideration. Safety aims at the prevention of unintended behaviour of programs and is a precondition for security. Security is concerned with secrecy and integrity of the information and the prevention of malicious attacks. Type systems can identify what is harmful behaviour for programs and restrict the execution of programs which are potentially harmful. Well-typed processes cannot leak secret information to the environment.

- *Failure Handling*: The crash of a physical site causes the permanent failure of all its locations. More generally, any location can halt. The failure of a location should be detected at other running locations [46].

Early formalisations of concurrency and communication are CSP [61] and CCS [85]. They provide static connectivity between processes and offer an abstract model of computation where the basic resources are communication channels and the basic computation is carried out by matching input or output actions on these channels [71]. The π-Calculus [87] refines CCS by allowing fresh channel names to be dynamically created and exchanged in communication. It is a process calculus of communicating systems. It was designed to provide an abstract model of *concurrent computation* but not *distributed computation*. The π-Calculus has a clear treatment of concurrency and communication, but it does not model the notion of location or site, another very important feature of distributed system. It has been seen as the basic model of distributed system. The π-Calculus provides a simple mobility called *Name (Link) Mobility*, where links move in a virtual space of linked process[86]. The π-Calculus does not express a type system or failure handling.

Distributed π-Calculus ($D\pi$) [102] is a distributed extension of the π-Calculus with the notions of remote execution, migration, and site failure. As Riley and

Hennessy described in [102], novel features of $D\pi$ include:

- Communication channels are *explicitly located:* the use of a channel requires knowledge of both the channel and its location.

- Names are endowed with *permissions:* the holder of a name may only use that name in the manner allowed by these permissions.

The $D\pi$ calculus is a robust and useful semantic theory for a process language in which computation is distributed over different locations, in which processes may migrate from one site to another, and in which sites may fail. Dπ provides a well-defined type system to guarantee security.

The join-Calculus[45] tries to bridge the gap between concurrency theory and distributed programming. It is a small calculus, which retains the style of process calculi, and it provides built-in locality[43]. The Distributed join-Calculus[46] extends the join-Calculus with *locations* and primitives for mobility. The novelty of distributed join-Calculus[45] is the introduction of locations, which resides on a physical site and contain a group of processes and definitions. The Distributed join-Calculus provides a simple model of failure. The failure of a location can be detected at any other running location. Also the distributed join-Calculus provides a type system but it is not for security.

Ambient calculus[40], another calculus which derives its process primitives from the π-Calculus, introduces the notion of a bounded environment(ambients). An ambient,which is similar to *location* in the distributed join-Calculus, is a *bounded* place where computation happens. An ambient can be nested in other ambients and can be moved as a whole. In general, an ambient exhibits a tree structure, which is like locations in the distributed join-calculus. Ambient calculus does not provide any mechanisms for security and failure handling but some extensions[77, 117, 53] have met those aims.

The Nomadic π-Calculi are extensions of the asynchronous π-Calculus with the notions of sites and agents, allowing distributed and mobile computation to be precisely described[132]. The calculi are designed to express agent mobility, so there is no way to describe remote communication between entities. The main

entities of Nomadic π-Calculi are *sites*, *agents* and *channels*. Sites should be thought of as physical or virtual machines; each site has a unique name. Agents are units of executing code; an agent has a unique name and a body consisting of some processes; at any moment it is located at a particular site. Channels support communication within agents, and also provide targets for inter-agent communication. If an agent a must communicate with another agent b, it must migrate to the same location where agent b is located. Nomadic π-Calculi do not explicitly address network failure and reconfiguration, or security.

The Table 2.1 summarises the main properties of the calculi. The next section

Item	Location	Mobility	Security	Failure handling
π-Calculus	No	Names	No	No
$D\pi$	Yes	Processes	Yes	Yes
Join/DJoin	Yes	Processes	No	Yes
Ambient	Yes	Processes	Yes(ext)	Yes(ext)
Nomadic π	Yes	Processes	No	No

Table 2.1: Summary of Calculi

will introduce mobile languages, some of which are based on these calculi.

2.1.3 Properties of Mobile Languages

With the growth of global computer networks, programming languages are expected to have mechanisms to support effective use of network resources. Many languages which are suited for distributed and mobile program have been developed e.g. Telescript [128], Java Aglets [74], MobileML [60] and Jocaml [44]. A mobile language may have the following properties [17, 48]:

- *Mobility is under programmer control:* The language should provide mechanisms and abstractions that enable the programmer to express mobility.

- *Strong or weak mobility:* Strong mobility is the ability to allow migration of both the code and execution state. In weak mobility only the code can migrate. Mobile languages should support at least one of these two kinds of mobility.

- *Implicit or explicit mobility:* [17] *Implicit mobility* is to automatically move an executing computation/process/thread from one location in the network to another. Implicit mobility is typically exploited in small systems e.g. a single cluster or LAN [88]. *Explicit* mobile languages give the programmer control over the placement of active computations, and executes on open systems. These mobile languages operate in large-scale settings where networks are composed of heterogeneous locations [17]. This thesis focus on *explicit* mobile languages.

- *Programming is location aware:* In a mobile language, the mobility of code will usually happen when the program needs to access resources that are not in the same location as that in which the program started running. Thus, the programmer must be able to explicitly say where the computation must be moved to. This notion of location does not need to be restricted to the name of machines in a network; it can be related to the names of resources that the programmer wants to access [23]. Applications should be location-aware and may take actions based on such knowledge.

There are some other desirable features of mobile languages:

- *Formal models:* In mobile systems, external code may be executed at a user's machine and user's private code may be executed elsewhere in the network. Therefore security and safety are important. Solid formal models like the calculi discussed above are the key to formal reasoning about such properties of programs [60].

- *Architecture Independent:* Mobile languages designed to work on global distributed systems, must be able to communicate code between machines of different architectures and operating systems. The usual approach for communicating computation on heterogeneous networks is by compiling programs into architecture-independent byte-code [132].

- Other desirable features are the same as those which we have discussed for calculi such as *security* and *failure handling*.

Mobile language can be imperative such as JavaGo [109] and Java Voyager [100], or functional such as Mobile Haskell [38], Nomadic Pict [132], Facile [49, 120], MobileML [60], and Jocaml [44]. These languages will be introduced in the following sections.

2.1.4 Java Voyager

Voyager [100] supports weak mobility, and a range of mobility functions that enable communication between Voyager, SOAP, CORBA, Remote Method Invocation (RMI) and Distributed Component Object Model (DCOM) objects. Voyager also includes an activation framework for mobile agent technology. Its dynamic proxy generation removes the need for manual stub generation. Remote class-loading simplifies deployment and management of application classes.

Voyager[100] provides a set of basic and advanced services and features for distributed and mobile application development. The `moveTo` primitives indicates program migration in Voyager, and causes the following sequence of events to occur.

1. Any messages that the object is currently processing are allowed to complete and any new messages that arrive at the object are suspended. Mobility can only detect method calls that are made through Voyager Proxy objects, so no attempt may be made to move an object that might be executing methods that were invoked directly.

2. The object and all of its non-transient parts are copied to the new location using Java serialization, ignoring pass-by-reference tags. An exception is thrown when any part of the object is not serializable or when a network error occurs. To avoid copying a particular part as an object, a proxy to the part is stored instead.

3. The new addresses of the object and all of its non-transient parts are cached at the old location.

4. The old object is destroyed.

5. Suspended messages sent to the old object are resumed.

6. When a message sent via a proxy arrives at the old address of a moved object, a special exception containing the object's new address is thrown back to the stale proxy. The proxy traps this exception, rebinds to the new address, and then resends the message to the updated address. If the program at the old location crashes before a stale proxy is updated, the stale proxy is unable to successfully rebind and a message sent via the proxy generates an exception.

7. `moveTo` returns after the object is successfully moved or when a mobility exception occurs. If an exception occurs, the old object is restored to its original condition, suspended messages are resumed, the exception occurs, the old object is restored to its original condition, suspended messages are resumed, and the exception is rethrown wrapped in a MobilityException.

An example of mobility in Voyager is shown in Figure 2.1. This program is started in location "surya.macs.hw.ac.uk" at port "7000", and is sent to another location "jove" at port "8000", where it prints out "hello world".

2.1.5 JavaGo

JavaGo[108] is an extension of Java which support strong mobility. It is a toolkit that enables a Java program to migrate across computers, preserving execution state. JavaGo allows program execution including the call stack to be suspended at arbitrary program points, to be transmitted to another remote computers, and to be resumed there. In other words, JavaGo enables transparent migration for Java programs. The programmer can control the code to be transmitted. One of the advantages of JavaGo is that migration can be performed with any Java virtual machine .

The migration concept in JavaGo is simple, because it is implemented completely on JavaRMI and thus does not require complicated infrastructures. It also guarantees that the execution state is completely preserved on migration [109].

```
public class Mobility{
  public static void main( String[] args ){
    try{
      Voyager.startup( "7000" );
      // create a local drone
      IDrone drone = (IDrone) Factory.create( "Drone" );
      // get/add mobility facet
      IMobility mobility = Mobility.of( drone );
      // move the drone to and from the remote program
      mobility.moveTo( "//jove.macs.hw.ac.uk:8000" );
      drone.print( "hello world" );
    }
    catch( Exception exception ){
      System.err.println( exception );
    }
    Voyager.shutdown();
  }
}
```

Figure 2.1: Mobility Example in Voyager

In JavaGo the programmer can describe flexible migration using three language primitives (`go`, `undock` and `migratory`).

Starting Migration by Using `go`

Migration takes place by executing a `go` statement. The mobile "Hello" example in JavaGo is as follows.

```
go ("//jove:2001/JavaGoExecutor");
System.out.println ("Hello!");
```

The argument of a go statement is the name of a migration server registered in the JavaRMI registry. The migration server is an object that manages migration. When the `go` statement is executed, the execution state and object are transmitted to the destination host, where the object continues executing just after the go migration.

JavaGo allows the transmission of recursive function invocations. For example, in Figure 2.2 the migratory method `fib` computes a Fibonacci number by

```
boolean Moved = false;
public migratory int fib (int n) {
     if ( n == 0 ) {
          if ( !Moved ) {
              go ("//jove:2001/JavaGoExecutor");
              Moved = true;
          }
          return 1;
      }
      else if ( n == 1 )
          return 1 ;
      else
          return fib (n-1) + fib (n-2);
 }
```

Figure 2.2: Mobility Example in JavaGo

recursively invoking itself. During the computation, `fib` migrates only once, that is, at the point where the execution stack is the deepest.

Controlling Transparency by Using `undock`

An `undock` statement serves as a *marker* that specifies the range of the area to be migrated in the execution stack. It makes it possible to control migration transparently. When the `go` statement in the `undock` statement is executed, the code after the `go` resumes executing at the destination host, while the statements after the `undock` statement are currently executed on the departure host.

```
undock {
   go ("//jove:2001/JavaGoExecutor");
   system.out.println ("Hello!"); \\ on the destination host
}
system.out.println ("bye!"); \\on the departure host
```

Declaring Method by Using `migratory`

When migration takes place in the context of a method, the method itself must be declared as such using the `migratory` primitive. That is, the `go` statement and

the invocation of a migratory method must be written in a migratory method. If they are put in a non-migratory method, they must be enclosed within an undock statement.

2.1.6 Other Imperative Languages

Telescript[128], developed by General Magic, is a pioneer mobile language. It is an object-oriented language conceived for the development of large distributed applications. Security has been one of the most important factors in this language, together with a focus on strong mobility. In Telescript a programmer can describe migration by writing "**go** *destination*".

Obliq[21], developed at DEC, is an untyped, object-based, lexically scoped, interpreted language. Obliq allows for remote execution of procedures by means of *execution engines*. A thread in Obliq can request the execution of a procedure on a remote execution engine. Obliq supports weak mobility. Migration in Obliq is accomplished in two steps: first the object is cloned in the remote engine and then the calls to the remote object are redirected to the original object using **redirect**. The **clone** operation creates a new object with the same field names and values as the argument object. Obliq is a language designed to work in local area networks and not for wide area networks, such as the Internet[22].

Java Aglets [74] and Sumatra [7] are Java based languages. The Java Aglets API extends Java with support for weak mobility[48]. Aglets are threads in a Java interpreter. The API provides the notion of context which provides a set of basic services, e.g., retrieval of the list of aglets currently contained in that context or creation of new aglets within the context. Java Aglets provides two migration primitives: *dispatch* which make the aglet to go to the destination and *retract* which forces an aglet to come to the context where *retract* is executed.

Sumatra, developed at the University of Maryland, is a Java extension. In this language programs are able to adapt to resource changes by exploiting mobility. Sumatra provides support for strong mobility of Java threads.

These six languages are imperative. Functional languages which support mobility are described in the next few sections.

2.1.7 MobileML

MobileML[60] is a programming language based on SML[58] which has a well-founded theoretical basis. The main features of MobileML include transparent migration and dynamic linking with distributed resources by means of *contexts*. The notion of contexts can describe various forms of interactions between mobile code and environments at the destination nodes. In MobileML, if a program is moved to somewhere else, say destination l, the **go** expression is as:

```
go l;
```

A context is a program expression with holes in it. The basic operation for a context is to fill its hole with an expression. For example, consider the following context:

```
let x = 3 in 1 + Hole
```

If its hole is filled with (x*2), the expression "let x = 3 in 1 + (x*2)" can be obtained. In this expression, x in (x*2) is bound by the *let*-construct. In MobileML, mobile code travels on a network, interacting with (being provided in the holes of) contexts local to nodes. A destination is designated by the name of a context and its hole. When a variable in a piece of mobile code is dynamically bound with some value by a context, we say the value is *assimilated* through the variable by the mobile code. Such a variable is called an *assimilation variable* denoted by x'. Assimilation is the term for dynamic binding. For example, evaluating an expression

```
let f = λx.x + 1 in (go l; (f a'))
```

causes the code of `(f a')` to migrate to the context designated by `l`. If the context `l` is

```
let a = 3 in Hole
```

then,

```
let a = 3 in ((λx.x + 1)a)
```

will be evaluated, and the value 3 will be assimilated through `a` into `(f a')`.

An example of the mobile Hello program in MobileML is as follows. In this example, a server is started at location "//jove.macs.hw". The client program looks for the server, moves to it, and then print out "Hello!" at the server side. On the server side:

```
context ser with HELLO = () end;
register ser as "Ser" with "//jove.macs.hw";
```

On the client side:

```
fun print_hello () =
  let val locs = ["//jove.macs.hw/Ser:HELLO"]
  in
   ( go locs; print_string "Hello!")
  end;
```

2.1.8 Nomadic Pict

Nomadic Pict is a strongly-typed programming language based on the nomadic Pi-calculus. The language allows the construction of distributed programs structured in terms of mobile agents – units of executing computation that can migrate between machines. Communication in Nomadic Pict is location independent, so agents can communicate after migrating [132].

The language has a two-level architecture. The low level consists of well-understood, location-dependent primitives for agent creation, the migration of agents between sites, and the communication of location-dependent asynchronous messages between agents[131]. The high-level language adds location-independent ($location - transparency$) communication, so migrating processes can request identical kernel services wherever they reside and distributed objects can be invoked without knowing their physical location[132].

The main entities of the language are *sites*, *agents* and *channels*. Those have the same meaning as in Nomadic π-Calculi, see section 2.1.2.

The language inherits a rich type system from Pict[93], including higher-order polymorphism, simple recursive types and sub-typing. It adds new types `site`, `agent` of site and agent name [131].

Below is an example of a mobile agent which say “Hello” to a server in Nomadic Pict. The application is implemented with three classes of agent: the CLs(clients), which migrate from site to site; `Server` agent, which are static and are used to call clients; and a single name server agent `names`, which maintains a lookup table from the textual keys of clients to their internal agent names. They interact using location-independent communication on channel names.

```
registCL  : ^ [ String Agent]
serverCL  : ^ [ String Agent Site]
moveOn    : ^ Site
mid       : ^ String
```

On the client side, the `CL` has four parallel components: a registration message, a message sent to another `CL`, a replicated input that receives data from other `CLs` and prints it, and a replicated input that receives migration commands and executes them.

```
agent CL1 =
  ( registCL@NameServer!["myCL" CL1]
  | mid@CL2!"Outgoing data stream"
  | mid?*d = print ! (+$ "Incoming:" d)
  | moveOn ?* s =
     (migrate to s (print !"Hello!")))
```

The name server maintains a map from strings to agent names and receives new mapping on `registCL`. The map is stored as an output on the internal channel `names`.

```
new names : ^ (Map String Agent)
```

```
( names ! (Map.make ==)
| registCL?*[descr CL] = names?m = (names ! (map.add m descr CL))
| serverCL?*[descr Se S] = names? m =
           (switch (map.lookup m descr) of
              {Found>CL:Agent} -> moveOn@CL!s
            end | names!m))
```

On the server side, the Server at site `s` gets strings from the local console, sending them as requests to the name server.

```
agent Server =
  val CLname = (sys.read_line [])
  ( severlCL@NameSever![CLname Server s] )
```

The whole example is based on the PA application in [131].

2.1.9 Mobile Haskell

Mobile Haskell (mHaskell) [37] is an extension of Haskell [1, 119], a purely functional lazy language, and was designed to facilitate the construction of distributed mobile software. Mobile Haskell extends Concurrent Haskell [92] with higher order communication channels called *Mobile Channels* (MChannels), that allow the communication of arbitrary Haskell values including functions, IO actions and channels. The main features are:

- mHaskell supports the construction of open systems, enabling programs to connect and communicate with other programs and to discover new resources in the network.
- mHaskell is designed to run on heterogeneous networks.
- mHaskell takes a hybrid approach, combining byte-code and machine code. This gives the advantage of having much faster code than using only byte-code.

```
data MChannel a      -- abstract
type HostName = String
type ChanName = String

newMChannel        :: IO (MChannel a)
writeMChannel      :: MChannel a -> a -> IO ()
readMChannel       :: MChannel a -> IO a
registerMChannel   :: MChannel a -> ChanName -> IO ()
unregisterMChannel :: MChannel a -> IO()
lookupMChannel     :: HostName -> ChanName -> IO (Maybe (MChannel a))
```

Figure 2.3: Mobile Channels in mHaskell

The communication primitives in Mobile Haskell are MChannel primitives, shown in Figure 2.3.

- The `newMChannel` function is used to create a mobile channel and the functions `writeMChannel` and `readMChannel` are used to write/read data from/to a channel. MChannels are synchronous: when a value is written to a channel the current thread blocks until the value is received in the remote host. In the same way when a `readMChannel` is performed in an empty MChannel it will block until a value is received on that MChannel.

- The functions `registerMChannel` and `unregisterMChannel` register/unregister channels in a name server. Once registered, a channel can be found by other programs using `lookupMChannel` which retrieves a mobile channel from the name server.

- The `Maybe` type in Haskell has two values: `Nothing` and `Just a`, so if the lookup finds a MChannel registered with `ChanName`, it returns `Just mchannel`, or it returns `Nothing` otherwise.

- A name server is always running on every machine of the system and a channel is always registered in the local name server with the `registerMChannel` function. MChannels are single-reader channels, so only the program that created the MChannel can read values from it. Values are evaluated to normal form before being communicated.

Figures 2.4 and 2.5 show a simple communication example of Mobile Haskell. In this example the client sent a message, which print “Hello”, to a server. The first program starts by creating a new MChannel for the result. Figure 2.4 shows the client side code, and Figure 2.5 shows the server side code.

```
main = do
   mch <- newMChannel
   registerMChannel mch "mainmch"
   sendMobile mobile mch "ushas.macs.hw.uk"
   where
     mobile = do
       print ("Hello ")

   sendMobile:: IO() -> MChannel Int -> HostName -> IO Int
   sendMobile comp mch host = do
      mc <- lookupMChannel host "servermch"
      case mc of
         Just nmc -> writeMChannel nmc comp
      result <- readMchannel mch
      return result
```

Figure 2.4: Mobility Example in mHaskell (Client Side)

```
main = do
   smch <- newMChannel
   registerMChannel smch "servermch"
   readMchannel smch
```

Figure 2.5: Mobility Example in mHaskell (Server Side)

2.1.10 Jocaml

Jocaml is an extension of Objective Caml 1.07 [76], a typed programming language in the ML family with a mix of functional, imperative, and object-oriented features. It extends OCaml with support for concurrency and synchronisation,the distributed execution of programs, and the dynamic relocation of active program fragments during execution.

OCaml has several features that are strongly relevant for mobile computation [44]:

- Programs are statically typed. This is important in distributed systems, where there are many opportunities to assemble inconsistent pieces of software, and where debugging runtime type errors is problematic.
- As a programming environment, OCaml provides both native-code and byte-code compilers, with separate compilation and flexible linking.
- The OCaml runtime has good support for system programming, such as the ability to marshal and unmarshal any data types, even between heterogeneous platforms.

The programming model of Jocaml is based on the join calculus[25], and uses ML's function bindings and pattern-matching on messages to express local synchronisations. Local synchronisation means that messages always travel to a set destination, and can interact only after they reach that destination[44]. The expressions and examples in this section are from Fourent's Jocaml user's manual[66].

Jocaml Programming

Jocaml programs are made of *processes* and *expressions*. Roughly, processes are executed asynchronously and produce no result, whereas expressions are evaluated synchronously and produce values. Processes communicate by sending messages on *channels* (port names). *Messages* carried by channels are made of zero or more values, and channels are values themselves. In contrast with other process calculi (such as π-calculus), channels and the processes that listen on them are defined in a single language construct. This allows consideration of channels as functions when they have the same usage.

Channels are first-class values in Jocaml, with a communication type, which can be used to form expressions and send messages. There are two important categories of channels: asynchronous and synchronous. Synchronous channels return

values, whereas asynchronous channels do not. The definition of an asynchronous channel `echo` is

```
let  def  echo! x = print_int x
```

In this definition, the presence of `!` in the channel name indicates that this channel is asynchronous and it is not possible to know when the actual printing takes place. The definition of a synchronous channel `print` is

```
let  def  print x = print_int x;reply
```

Since there is no `!` at the end of the defined name, `print` is synchronous, thus it must return a value. It uses a `reply` construct whose semantics is to send back some values (here zero) as result.

Processes are one of the important syntactic classes of Jocaml. The most basic process sends a message on an asynchronous channel. Since only declarations and expressions are allowed at the top-level, processes are turned into expressions by "spawning" them, so the keyword `spawn` is introduced followed by a process in "{}", e.g.

```
spawn {echo 1}
```

In this example `echo 1` is a processes, and `spawn {echo 1}` is an expression.

Expressions are another important syntactic class of Jocaml. In contrast with processes, expressions evaluate to some results. Expression can occur at the top-level. Apart from OCaml expressions[76], the most basic expression sends some value on a synchronous channel, which behaves like a function.

```
let x = 1 ;;     (* an OCaml expression *)
print x ;;       (* sends value x on synchronous channel print *)
spawn { print 1; print 2; echo 3} ;; (* spawn expression *)
```

Distributed and Mobile Programming

Jocaml programs can be distributed amongst numerous machines, possibly running different systems; new machines may join or quit the computation. At any

time, every process or expression is running on a given machine. However, they may migrate from one machine to another, under the control of the language. In Jocaml, the execution of a process (or an expression) does not usually depend on its localisation i.e. the scope for defined names and values is independent of their localisation. So whenever a port name appears in a process, it can be used to form messages without knowing whether this port name is locally or remotely defined. So far, locality is transparent. When a function is sent to a remote machine, its code and the values for its local variables are also sent there, and any invocation will be executed locally on the remote machine (Strong mobility). When a synchronous port name is sent to a remote machine, only the name is sent. Invocations on this name will forward the invocation to the machine where the name is defined, much as in a remote procedure call (Weak mobility). Jocaml offers a `go` primitive to indicate program migrations.

In Jocaml *locations* are units of locality. A location contains a collection of definitions and running processes "at the same place". Every location is given a name, and these location names are first-class values. They can be registered to the name server.

Locations and functions should register in $Name - sever$, so that the local or remote programs can use them. The interface of the name server consists of two functions to register and look up arbitrary values in a "global table" indexed by plain strings. A shell command **jocns** can launch the name server.

The following is an agent-based mobile program in Jocaml. On the server side, a server `here` is created, and registered in the name server.

```
let loc here do {} ;;  (* create a new empty location "here" *)
Ns.register "here" here vartype; (* register in name server *)
Join.server ()   (* running as server *)
```

On the client side, `mobile` looks for the sever `here`, moves to it, and prints out "Hello!" at the server.

```
let loc mobile
  do {
```

```
    let here = Ns.lookup "here" vartype in /* look up the server */
    go here;  /* migrate to server "here" */
    print_string ("Hello!");
  }
```

When the migration happens there are only three messages exchanged between the machines: one for looking up the server, one for the answer, and one for the migration.

There are several other functional mobile languages, for example Facile [49], which extends Standard ML [83] with primitives for distribution, concurrency, and communication.

2.1.11 Mobile Language Summary

Table 2.2 summarises the properties of the mobile languages described above.

Languages	Mobility	Internet-Scale	Location Aware	Primitive(s)	Calculi	failure handling	Security
JavaGo	Strong	Yes	Yes	go	-	Yes	Yes
MobileML	Strong	Yes	Yes	go	λ	Yes	Yes
Nomadic Pict	Strong	Yes	Yes	migrate	Nomadic π	No	Yes
mHaskell	Weak Strong	Yes	Yes	channels moveto	-	No	Yes
Jocaml	Strong	Yes	Yes	go	Join	Yes	No
Voyager	Weak	Yes	Yes	moveTo	-	Yes	Yes

Table 2.2: Summary of Languages

Java Voyager support weak mobility for Java programs. The programmer can use the `moveTo` function to send the program code to a remote location and the program will start at that location automatically.

JavaGO supports strong mobility (transparent migration) for Java programs. The programmer can control the area to be transmitted. JavaGo guarantees that the execution state is completely preserved on migration[109].

MobileML is based on SML[58], and has a well-founded theoretical basis. The main features of MobileML include transparent migration (strong mobility).

Nomadic Pict is based on the nomadic Pi-calculus. The language allows one to build distributed programs structured in terms of mobile agents which support

strong mobility [131].

*m*Haskell supports both weak and strong mobility. For weak mobility, it has the MChannel communication primitives. For strong mobility, it provides the **moveTo** primitive.

Jocaml supports concurrency and synchronisation, the distributed execution of programs (weak mobility), and the dynamic relocation of active program fragments during execution (strong mobility). The programming model of Jocaml is based on the join calculus[25]. Programs may migrate from one machine to another, under the control of the `go` primitives.

Java Voyager, JavaGo, and Jocaml are selected as the implementation languages in this thesis, for reasons covered in Sections 3.4.1, 4.2.6, and 4.4.1.

2.2 Agents & Autonomous Systems

2.2.1 Agents

An agent is "an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives" [122]. In more detail[67], agents are: (i) clearly identifiable problem solving entities with well-defined boundaries and interfaces; (ii) situated (embedded) in a particular environment– they receive inputs related to the state of their environment through sensors and they act on the environment through effectors; (iii) designed to fulfil a specific role– they have particular objectives to achieve and have particular problem solving capabilities (services) that they can bring to bear to this end; (iv) autonomous– they have control both over their internal state and over their own behaviour; and (v) capable of exhibiting flexible problem solving behaviour in pursuit of their design objectives– they need to be both reactive (able to respond in a timely fashion to changes that occur in their environment) and proactive (able to opportunistically adopt goals and take the initiative).

From this definition four characteristics of agents can be identified, which are

situatedness, autonomy, adaptivity, and sociability[116].

- Situatedness means that the agent receives some form of sensory input from its environment, and it performs some action that changes its environment in some way. The physical world and the Internet are examples of environments in which an agent can be situated.

- Autonomy means that the agent can act without direct intervention by humans or other agents and that it has control over its own actions and internal state.

- Adaptivity means that an agent is capable of reacting flexibly to changes in its environment, taking goal-directed initiative when appropriate, and learning from its own experience, its environment, and interactions with others.

- Sociability means that an agent is capable of interacting in a peer-to-peer manner with other agents or environments.

The four properties uniquely characterise an agent. Because of the property of autonomy, agents are also called *autonomous agents*, or *intelligent agents* in some research e.g. [104, 134, 5, 133].

Another property which should be emphasised is mobility. Mobility is the ability of an agent to transport itself from one machine to another and retain its current state. Agents with mobility are called *mobile agents*. Mobility is an orthogonal property of agents, that is, not all agents are mobile. Agents that do not or cannot move are *stationary agents*. A stationary agent executes only on the system on which it begins execution. If it needs information not on that system or needs to interact with an agent on another system, it typically uses a communication mechanism, such as remote procedure calling. In contrast, a mobile agent is not bound to the system on which it begins execution. It is free to travel among the hosts in the network. Created in one execution environment, it can transport its state and code with it to another execution environment in the network, where it resumes execution. The term *state* typically means the

attribute values of the agent that help it determine what to do when it resumes execution at its destination. Code in an object-oriented context means the class code necessary for an agent to execute[75].

A mobile agent has the ability to transport itself from one system in a network to another in the same network. This ability allows it to move to a system containing an object with which it wants to interact and then to take advantage of being in the same host or network as the object. The interest in mobile agents is not motivated by the technology but rather by the benefits agents provide for creating distributed systems[75].

When adopting an agent-oriented view of the world, it soon becomes apparent that most problems require or involve multiple agents: to represent the decentralised nature of the problem, multiple loci of control, multiple perspectives, or competing interests [41]. The agent paradigm offers a new promise for building complex software because of the abstraction and flexibility it provides. These systems are conceived as organizations of coordinating agents for example multi-agent systems. “A multi-agent system (MAS) is a system composed of a population of autonomous agents, which cooperate with each other to reach common objectives, while simultaneously each agent pursues individual objectives” [134]. In order for a MAS to solve common problems coherently, the agents must communicate amongst themselves to coordinate their activities. Coordination and communication are central to MAS, for without them any benefits of interaction vanish and the group of agents quickly degenerates into a collection of individuals with chaotic behaviour. MAS is applied as a technology for solving problems in an increasingly wide range of complex applications and is inspired by models from biology e.g. Swarm [62]. Some of the MASs are autonomous systems e.g ant algorithms [82].

2.2.2 Autonomous Systems

Autonomous systems are also called autonomic computing systems, and a definition has been given by IBM [70]: “autonomic computing system can manage

themselves given high-level objectives from administrators" [70]. The essence of autonomic computing systems is self-management, the intent of which is to free system administrators from the details of system operation and maintenance. Autonomic systems will maintain and adjust their operation in the face of changing components, workloads, demands, and external conditions and in the face of hardware or software failures. Four aspects of self-managing are: [70, 90]

- *Self-configuration*: Automated configuration of components and systems follows high-level policies. An autonomous computing system must be able to install and set up software automatically. Self-configuration will use adaptive algorithms to determine the optimum configurations.

- *Self-optimization*: Components and systems continually seek opportunities to improve their own performance and efficiency. An autonomous system will never settle for the status quo. It will be constantly monitoring predefined system goals or performance levels to ensure that all systems are running at optimum levels. Self-optimization will be the key to allocating resources determining when an increase in processing cycles is needed, how much in needed, where they are needed, and for how long.

- *Self-healing*: The system automatically detects, diagnoses, and repairs localized software and hardware problems. Autonomous computing systems will have the ability to discover and repair potential problems to ensure that the systems run smoothly.

- *Self-protection*: The system automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent system-wide failures. Autonomous systems must identify, detect, and protect valuable assets from threats. They must maintain integrity and accuracy and be responsible for overall system security.

Different autonomic systems may have some or all these four aspects. For example, the Autonomic Job Scheduling Policy(ASP)[6] for Grid Computing is capable of dynamically scheduling resources while at the same time being self-healing

and self-protecting to various types of failures. ASP is a scheduling infrastructure which adapt to changes in application or resource failures and is capable of proactively detecting and rectifying potential faults as applications are executing[6]. ASP is demand-driven where nodes in the system look for work when their load is below a given threshold.

An example of self-configuration has been presented in [79], where a component-based programming framework has been presented to support the development of autonomic applications in Grid environments. The framework builds on the separation of composition (organization, interaction and coordination) aspects from computation behaviours. It underlies the component-based paradigm, and extends it to enable the computational behaviours of components as well as their organizations, interactions and coordinations to be managed at runtime using high-level rules.

This thesis focuses on *Self-optimization*. Autonomic systems will continually seek ways to improve their operation and to make themselves more efficient in performance or cost. For example load management systems, which will be described in Section 2.3, are self-optimization systems. The goal of load management systems are to assign to each node a number of tasks to improve the performance of the application[91].

Autonomous mobile programs (AMPs), which will be introduced in Chapter 3, are also self-optimization system. AMPs are aware of their processing resource needs and sensitive to the environment in which they execute, and are able to dynamically relocate themselves to minimise processing time in the presence of varying external loads on shared processing elements.

2.2.3 Discussion

The concepts of agents and autonomous systems come together many times. The property of autonomy of agents makes it easy to develop autonomous systems. Many autonomous systems are agent-based i.e. *the key abstraction used is that of an agent*[133]. Some agents in these autonomous systems are mobile, and can

migrate from one location to another. A figure of the relation between agents, autonomous systems and our work will be given in Section 3.2.

Many autonomous systems are using mobile agent technology, for example ethological models e.g. ant algorithms which are distributed computations. In ant algorithms[82], agents (ants) are looking for "food". An ant algorithm is tries to find the fastest path to get to the food, and uses the feedback of each mobile agent to decide which path is better. As most autonomous mobile agent systems, ant algorithms adapt the computation automatically.

2.3 Load Management Systems

As Casavant and Kuhl argue in[24], advances in hardware and software technologies have led to increased interest in the use of large-scale parallel and distributed systems for large-scale applications. One of the biggest issues in such systems is effective techniques for the distribution of processes on multiple locations. The problem is how to distribute, or schedule, the processes amongst processing elements to achieve some performance goals such as minimising execution time.

Load Management is one branch of a family of global scheduling techniques [24], where an attempt is made to truly balance the load of all locations in multicomputer, so load management is also called load balancing [24]. The goal of load management is to assign to each location a number of tasks to improve the performance of the application[91]. This thesis is concerned with load balancing in a large-scale network. In the this thesis load management refers to load balancing. There are a number of ways to classify load management techniques. The primary categories are *static* and *dynamic* [24].

2.3.1 Static/Dynamic Load Management

Static algorithms make scheduling decisions based on predictions of run-time behaviours of programs at compile-time, so static techniques do not depend on the state of the processors. Dynamic techniques rely on present and past states of the processors, and distribute load among the hosts based on these states. As a

result, dynamic techniques perform relatively better but have additional burdens of communication and processing[89, 110].

Static Load Management

Typically, the goal of static load management is to *minimise* the overall execution time of a program. Static load managements attempts to [110]:

- predict the program execution behaviour at compile time (that is, estimate the process or task, execution times, and communication delays);
- allocate processes to processors.

The major advantage of static load management is that all the overhead of the scheduling process is incurred at compile time, resulting in a more efficient execution time environment compared to dynamic scheduling methods. But due to possible conditions or network contention delays, predictions of run-time behaviours made at compile-time might deviate significantly from the real values at run-time. Therefore static load management might make inappropriate decisions and cause unpredictable performances degradation. Also, the time for computation may depend on inputs not available at compile time.

Dynamic Load Management

Dynamic load management redistributes the tasks from heavily loaded processors to lightly loaded ones based on information collected at run-time. Dynamic load management is particularly useful in a system consisting of a network of workstations in which the primary performance goal is to *maximise* utilisation of processing power instead of minimising execution time of the applications[10].

The advantage of dynamic load management over static load management is that the system need not be aware of the run-time behaviour of the applications before execution. The flexibility inherent in dynamic load management allows for adaptation to the unforeseen application requirements at run-time [97]. But due to the communication cost of load information collection and distribution,

processing cost of schedule decision making and the communication cost for task transfer, dynamic load management definitely incurs a non-zero run-time overhead. But good dynamic load management algorithms always make these costs minimized and would not invoke further load management process if the total profit obtainable from load management does not significantly exceed the total cost that may incur.

Activities in Dynamic Load Management

The activities which may happen during the execution of programs with dynamic load management are as follows[78]:

a. measure the workload of individual processors;

b. exchange load information between processors;

c. check certain conditions for load imbalance and decide whether to perform load balancing operations or not;

d. make a load management decision on how many tasks migrate from which processors to which processors;

e. choose appropriate tasks and transfer them to the corresponding destination processors.

The actual activities happening for different algorithms on differently designed systems may be different. There are many issues involved in dynamic load management such as how to measure the load of a processor, how much load information should be collected and where the processes should reside. These issues are usually grouped into several policies. A typical dynamic load management algorithm is defined by four inherent policies[39]:

- *information policy:* specifies the amount of load information made available to job placement decision-markers.
- *transfer policy:* determines the conditions under which a job should be transferred.

- *selection policy:* decides which program(s) should be selected to move.
- *placement policy:* identifies the processing element to which a job should be transferred.

These load management operations may be centralised in a single processor or distributed amongst all the processing elements that participate in the load management process. Many combined policies may also exist[10], but for all those policies some special process (load balancer) must make the decision when and where to move rather than the applications deciding. So the programmer cannot control the programs' migrations.

2.3.2 Centralised & Decentralised Load Management

Dynamic load management can be classified into centralised and decentralised management algorithms. The difference lies in *where load information is stored*. In centralised networks all the data is kept in a single processor called the central scheduler; in decentralised networks each processor maintains the load information locally and sends out updated information to the other processors whenever the data changes[137].

Centralised: A central node collects state information and constructs an estimate of the system state. The central node may be a globally shared file that is accessed and updated by all nodes. This organisation has an advantage that it incurs low overhead during estimation. The disadvantages are poor responsiveness of a central resource in a large-scale system resulting in poor scalability and the failure-proneness [97].

Decentralised: In a decentralised organisation each node of a distributed system is responsible for collecting state information and obtaining an estimate of the system state. This type of organisation has higher availability in the presence of failures, but it can potentially incur large overheads to maintain accurate state information and therefore is not easily scalable to a large-scale distributed computing system[97].

Hybrid: A hybrid organisation combines both centralised and decentralised organisations, inherits their properties, and attempts to extract advantages of both. A hybrid organisation can be implemented in the case that nodes are divided into clusters and state information is exchanged within and between clusters[136].

2.3.3 Push & Pull Policies

Generally the selection and placement policies in a dynamic load management system are combined together to decide "how is work distributed and balanced between processors (where a job should be transferred)" [80]. There are two kinds of selection and placement policy [80].

- In a *Passive load distribution policy (Pull policy)*, idle locations have to explicitly ask for work.
- In an *active load distribution policy (Push policy)*, new activity are sent to remote locations.

Pull policy , which is sometimes called *work stealing*, tries to minimise the overhead during periods in which all processors are busy anyway. However, this may yield an uneven load distribution. In contrast, push policy sends, by default, a new thread to a remote processor for execution. Although this gives a more even distribution it may yield a deterioration in the data locality of the system. In both cases, however, it is desirable to have load information about other processors available. Obtaining such information may require significant communication and therefore all machines have to find a compromise between the competing goals of an even load distribution and a minimal amount of communication.

2.3.4 Preemptive & Non-Preemptive Load Management

Load management systems distribute the system workload among the locations through transfer policies, which can be performed either *non-preemptive* or *preemptive* [73, 107]. Non-preemptive transfer entails selecting a suitable location

as the execution site for a process and initiating the process at that location. In preemptive transfer, if another location should become a better execution site, the process is transferred to that location, where it continues executing. So the non-preemptive transfer only locates processes to a suitable location once before the process is executing, however the preemptive transfer can relocate processes during the run-time when a better location is available.

Preemptive transfer is more costly than non-preemptive but more flexible, as it can migrate running processes from one location host to another. Research to date indicates that preemptive transfer yields significant performance benefits through the process migration facilities [73].

2.3.5 Dynamic Load Management Systems

Numerous dynamic load management systems e.g. [8], [94], and [52] have been proposed. This section introduces two of the most popular systems.

Load Sharing Facility (LSF)

LSF[94] is a general purpose distributed queueing system that unites a cluster of computers into a single virtual system to make better use of the resources on the network. Locations from various vendors can be integrated into a system, which can find the best location to run serial and parallel programs.

LSF can automatically select locations in a heterogeneous environment based on the current load conditions and the resources requirements of the applications. With LSF, remotely run jobs behave just like jobs run on the local location.

LSF consists of LSF Base and LSF Batch. LSF Batch is a distributed batch system built on top of LSF Base to provide batch job scheduling services to users. It accepts user jobs and holds them in queues until suitable locations are available. Location selection is a function of up-to-date load information stored in the load information manager(LIM) [52].

The LSF Base must have LSF Batch to provide job scheduling and resource allocation necessary for grids. LSF Base provides load sharing and distributed

processing service such as location selection, resource information, and transparent remote execution.

The LIM daemon monitors the locations' load and send this information to the other LIM daemons on the cluster. One of the LIMs will assume the role of master and coordinate the behaviour of the cluster. If the machine it lives on dies then one of the other LIMs will take over the coordinating role. This will continue to happen until only one location is left on the cluster thus providing a fault tolerance mechanism. The RES daemon allows jobs to be started on the machine and also provides a mechanism for running interactive jobs[11]. So LSF is using pull policy to allocate jobs.

Sun Grid Engine (SGE)

Sun Grid Engine[52] is distributed management software that optimises utilisation of resources in heterogeneous networked environments. It distributes computational workload to those available systems. So Sun Grid Engine is using a push policy to allocate jobs.

The functionality of a SGE system is performed by four daemons[52].

- The *master daemon*(information policy) maintains tables about locations, jobs, system load, and any other informations.

- The *scheduler daemon*(transfer and placement policy) maintains an up-to-date view of a grid's status. It determines which jobs are dispatched to which queues and then forwards its decisions to the master daemon, which initiates the required actions.

- The *execution daemon*(information policy) is responsible for the queues associated with the location on which it runs, and it periodically forwards the status of jobs and the load on its location to the master daemon.

- The *communication daemon* communicates over a TCP (Transmission Control Protocol) port [3]. It is used for all communication among SGE components.

2.4 Cost Models

A performance model is commonly referred as a *cost model* [81]. Usually, cost models are used to estimate the cost of a program in terms of desired metrics such as time[101], and space (memory)[63, 64]. There are two levels of cost models in general: *computation cost models*, which estimate the sequential computation time for programs; and *coordination cost models*, which predict the coordination and communication behaviours of parallel, distributed and mobile programs. Usually, coordination cost models take costs from the computation cost models into account to make better coordination decisions.

2.4.1 Computation Cost Models

According to [27, 26] there are two basic approaches by which one can estimate the execution time of a program with a given data, running on a particular computer.

- Dynamic analysis or profiling entitles measuring the execution time of the program on some data set on some machine. This approach uses an internal clock of the machine and benchmarks the execution of the program. Experimental approaches are used in measuring execution time of the program. So in this thesis the cost models for experimental approaches are called *dynamic cost models*. In this approach the benchmark is machine-specific. When the same program is run on another machine, a new set of benchmarks is needed.

- Static analysis determines the time to execute the given program by using mathematical reasoning on the source code. So this approach consists of carefully examining the program source and the data to establish a machine-independent analytical formula representing the program's performance. The analytical formula which are built in this approach are called *static cost models*.

In [16, 15] a hybrid technology to obtain execution times has been introduced using a probabilistic approach to combine both measurement and analytical approaches into a model for estimating the execution time of code. A probabilistic approach builds on experimental measurements by measuring costs for repeated executions over a suite of test cases. Under the assumption that the test suite provides representative data, it is then possible to construct statistical profiles that can be used to determine execution time to some stated probability [19].

This section introduces some static cost models which are based on performance evaluation using computer-assisted analytical techniques. In Chapter 6 we will propose a cost calculus which is related to the static cost models here. The analytical approach can be subdivided into two categories: *macroanalyses* and *microanalyses*. In performing a macroanalysis, a dominant operation of the algorithm is chosen to express the times. The O notation [72] is often used for this purpose. Aho et al. in [9] describe many macroanalyses of algorithms. In a microanalysis, program execution times are expressed as functions of the times to perform elementary operations that exist in most computers, e.g., addition, assignment, subscripting, and so on.

Many cost models have been built using microanalysis. For example, in [26, 27] Cohen et al. built time-formula to express execution times as functions of variables representing the time needed to perform common, elementary operations, e.g., addition, assignment, subscripting, loop overhead.

The basic way to build a static cost model in a microanalysis system is by using the abstract syntax tree. In the models which were built in [26, 27], the cost of an expression is the sum of the cost of the subexpressions, which are all derived from the nodes in the abstract syntax tree. Other cost models have been built by adding different properties to the abstract syntax tree. By adding size information into the cost model to get latent cost, in [101] Reistad and Gifford used static cost for data-dependent expressions, which describes the execution time of a procedure in terms of its inputs. In particular, a procedure's static dependent cost may depend on the size of input data structures and the cost of input to first-class procedures. This system produces symbolic cost expressions

that contain free variables describing the size and cost of the procedure's inputs. In [80], Loidl has built a sized time cost model for language $\mathcal{L}$ also using size information. In this model, the size information is attached to its type.

Different static cost models have been built for different systems. Early work on these problems includes that of Cohen and Zuckerman, who consider cost analysis of Algol-60 programs [28]; Wegbreit, whose pioneering work on cost analysis of Lisp programs addressed the treatment of recursion [125]; and Ramshaw [98] and Wegbreit [126], who discuss the formal verification of cost specifications. Many of the cost analysis use semantics-based methods e.g Rosendahl [103] uses abstract interpretation for cost analysis, and Wadler [124] uses projection analysis. In [101] Reistad and Gifford built static cost for data-dependent expressions.

Computation time cost models can be used for different application predications, e.g. worst case execution times[95], parallel algorithms, and so on. For example in [19], a cost model is used to predict worst case execution times for Hume [56, 57] programs. In [15], worst case execution time has been analysed for Java Byte Code.

2.4.2 Parallel Coordination Models

Section 2.4.1 has discussed cost models for sequential computation. The coordination cost models take costs from the computation cost models into account to make better coordination decision for parallel, distributed, and mobile programs.

This section uses parallel programming models as examples to explain the coordination cost models for parallel algorithms, as these cost models are more developed than these for distributed and mobile programming.

Parallel programming involves mapping a program to a multiprocessor machine. It is a complex activity involving many decisions about task allocation, scheduling, and communication. There are three approaches which can produce parallel algorithms[99, 112].

- *parallelising compilers for sequential languages*: In this approach, the compilers create parallel code from existing sequential programs, so the programmer has no or little involvement in the parallelisation process. The big problems of this approach are that not all the available parallelism can be detected because the compilers must adopt a conservative approach in generating parallel code, in order to ensure its correctness, and the code obtained by parallelising the sequential program may not be the most efficient.

- *explicit parallel programming languages*: These languages include constructs that allows the programmers to explicitly create processes that can be executed in parallel, and to manage the interactions between them. Languages such as Ada[68] and Concurrent Pascal[59] have parallel constructs integrated into them. Portable message passing libraries such as MPI[96] and PVM[13] make it possible to write portable parallel programs using sequential languages like C.

- *implicit parallelism*, in particular in functional languages: Programs in these languages are inherently parallel and there is no need for explicit parallel constructs. Higher order functions in functional languages provide a powerful control abstraction mechanism. Therefore, Cole [29] has proposed the use of algorithmic skeletons as a technique to parallelise functional languages and to program parallel machines. The idea is to capture common patterns of parallel computation in higher order functions. The major advantage of algorithmic skeletons is the portability of parallel programs written using this approach[54]. For building algorithmic skeletons, cost models are important for the efficiency of the parallelism. So the next section introduces cost models for different skeletons.

2.4.3 Cost Models for Algorithmic Skeletons

Algorithmic skeletons encapsulate the expression of parallelism, communication, synchronisation and embedding, and have an associated cost complexity [115].

The skeleton's cost complexity is calculated according to a particular cost model. A skeleton can be costed so that the communication costs for the appropriate embedding and the general cost formula under the particular cost model are provided as a cost skeleton. Then the computation costs can be supplied as parameters [115]. This provides a separation of the computation and communication costs for skeletons. Most of the cost models for algorithmic skeletons have these two parts.

Cost Models for `map` and `fold`

In [99] Rangaswami develops a parallel programming model called **HOPP** for skeleton oriented programming. A **HOPP** program could comprise one or more *phase(s)* where parallelism is exploited only within each *phase*. The parallelism in each phase depends on the number of recognised functions in the phase itself. The cost models for the recognised functions were used to statically analyse a given **HOPP** program to determine the most cost effective parallel implementation for a given architecture. In general, the cost of a program comprising of n *phases* is given by:

$$Cost \;=\; \sum_{n}^{i=1} C_{pi} \;+\; \sum_{i=0}^{n-1} C_{i,i+1} \tag{2.1}$$

where C_{pi} is the cost of phase i and $C_{i,i+1}$ is the communication cost in case there is a need for rearranging the output of phase i to suit the implementation of phase $i+1$. According to this general model, Rangaswami has developed cost models for different algorithm skeletons e.g `map, fold, scan, filter` etc. on different topologies such as linear array, 2-D Torus, and so on. The cost of `map` and `fold` on linear array are given as an example.

The cost expression for the parallel implementation of `map` on any p-processor topology is

$$C_{map} \;=\; \frac{n}{p} C_f \tag{2.2}$$

There are two parallel implementations of `fold`, `s_fold` and `g_fold`. `s_fold` is static `fold`, where the data size remains the same. `g_fold` is growing `fold`, where the size of the data changes as the `fold` is applied[106]. The costs for

the two versions are different, because in the case of `g_fold`, the size of the data communicated increases at each step and must be accounted for in the computation of the time for communication[99].

The cost expression for the parallel implementation of `s_fold` on the linear array is:

$$C_{s_fold} \;=\; C_f(\frac{n}{p} - 1 + \log p) \;+\; T^1_{com}(p-1) \tag{2.3}$$

and the cost expression for `g_fold` is:

$$C_{g_fold} \;=\; C_f(\frac{n}{p} - 1 + \log p) \;+\; \sum_{i=0}^{\log p - 1} 2^i T_{com}^{2^i \frac{n}{p} m} \tag{2.4}$$

In Equation 2.2, 2.3, and 2.4,

- n is the list size.
- p is the number of processors.
- C_f is the cost of the processing function
- T^m_{com} is the cost of communicating m elements of the list to a neighbour. It is evaluated by $T^m_{com} = K_0 + \frac{1}{k_1} ms$, where s is the size in bytes of each element of the list, K_0 and K_1 are the startup cost to initiate the communication and the cost to transfer data.

There are other cost models for parallel `map` and `fold`. Skillicorn and Cai have developed a cost calculus for the Bird Meertens Formalism (BMF)[111]. In this calculus the cost of implementing the `map` function in parallel has been built[113].

H. W. To has looked into optimising the combinations of algorithmic skeletons, where combining skeletons have been proposed to capture the common pattern of control flow. A set of primitive skeletons has been chosen e.g. `map`, and `fold` based on the observation that many highly parallel applications exploit parallelism from a large data structure, e.g. parallel abstract data types (PADTs) restricted (*rlists*) and *arrays*. And performance models for `map` and `fold` have been built for these in [121].

Cost Models for Other Algorithmic Skeletons

There are also cost models for other parallel implementations of skeletons e.g. divide and conquer (`DC`), `FARM` , and `PIPE`.

1. `DC` skeleton: Darlington et al.[31] have implemented the `DC` skeleton. The performance/cost model for DC assumes the processors are organised into a balanced binary tree and all processors will eventually be used as leaves. The execution time for DC can be predicted using the formula:

$$t_{sol_x} = \sum_{i=0}^{\log(p)-1} (t_{div_{x/2^i}} + t_{setup_{x/2^i}} + t_{comb_{x/2^i}} + t_{comm_{x/2^i}}) + t_{seq_{x/2^{\log p}}} \quad (2.5)$$

   where t_{sol_x} is the time to solve a problem of size x, t_{div_x} is the time to divide a problem of size x, t_{comb_x} is the time to combine the two results, t_{setup_x} and t_{comm_x} is the time to solve a problem of size x sequentially.

   Cole has described the time complexity of the H-tree, a well know layout of a complete binary tree on a grid of processors, for an implementation of the binary fixed divided and conquer (FDDC) skeleton in [29]. Campbell has developed a model for parallel computation called CLUMPS in [115]. The modelling of `DC` under CLUMPS has also been presented in [29].

2. `FARM` skeleton: Darlington et al. have implemented `FARM` in a similar way to their `DC` skeleton. The `FARM` consists of two major parts: one is a master processor and the other is a worker processor(s). The model for a process farm is:

$$t_{farm} = t_s + R(t_e + 2t_c) \quad (2.6)$$

   where T_{setup} is the setup time taken to allocate processors to a skeleton. R is the number of steps required to process the work pool. (t_s is the skeleton startup overhead. t_e) is the required time to solve a work packet. t_c is the communication time required to receive a result or send the data. Different cost models for `FARM` have been described in [20] and [114].

3. `PIPE` skeleton: Darlington et al[31] have developed a performance model for

pipeline (`PIPE`) skeleton which is:

$$t_{PIPE} = t_s + (t_e v + t_c)(p + n - 1) \qquad (2.7)$$

Equation 2.7 resembles a general model for `PIPE` which predicts the total execution time of a particular instance, where t_s is the startup time, t_e is the execution time for a stage for a single element, t_c is the communication time between stages, p is the number of stages and n is the number of elements in the list.

2.4.4 Cost Model Summary

Cost estimation can be performed dynamically or statically. In the general case it can not be performed entirely as a compile time analysis because costs might depend on input data, and compile time analysis do not take the real time system information into account. On the other hand, dynamic analysis risks increasing a very large run time overhead[123] and is only accurate for a specific machine and a specific set of input data.

The cost models in this thesis have two parts: compile time and run time phases. Generic and problem specific coordination cost models for autonomous mobile programs and autonomous mobility skeletons which are called at run time. Static computation cost models are also built for a subset of Jocaml. The result of this cost model is used as parameters in the coordination cost models.

2.5 Summary

How to share the resource in distributed systems is one of the most important issues in computer science. This chapter has reviewed the concepts which relate to the distributed resource sharing.

Mobile computation especially mobile languages give programmers control over the placement of code or active computations across the network[71]. Using mobile languages, programmer can write more flexible and efficient applications in distributed systems. Section 2.1 has surveyed mobile languages. Jocaml, Java

Voyager, and JavaGo are going to be used in Chapter 3, and 4 to built autonomous mobile programs and autonomous mobility skeletons.

Autonomous mobile programs are both agents and autonomous systems, and the properties of agents and autonomous systems have also been introduced in Section 2.2 with some examples. The agents community has focused on autonomous problem solving, which can act flexibly in uncertain and dynamic environments. Mobile languages provides efficient tools to make the agent move more flexibly in large scale networks, which make it possible to build self-managing systems (autonomous systems) for resource sharing using agent technology.

Another big issues in distributed systems is finding effective techniques for the distribution of the processes on multiple locations. The problem is how to distribute, or schedule, the processes amongst processing elements to achieve some performance goals such as balancing the load of each location, or minimising execution time. Load management systems have been built for meeting these requirement. Collections of autonomous mobile programs perform dynamic decentralises load management, which will be demonstrated in Chapter 5, and different load management policies have been introduced in Section 2.3.

In resource sharing systems e.g. load management systems, cost models could be used to predict the behaviours of the system and decide the following behaviours of the system. Suitable cost models are an import issue in self-managing systems. Different cost models have been reviewed in Section 2.4, as they play an important role for decision making in different systems. Coordination cost models for autonomous mobile programs and autonomous mobility skeletons will be built in Chapter 3, and 4 to predict the execution time of the programs at run time. A static computation cost model will be built in Chapter 6 to calculate the cost and remaining cost of the program, which can be used in the coordination cost model.

Chapter 3

Autonomous Mobile Programs

To manage load on large and dynamic networks we propose *autonomous mobile programs* (AMPs) that periodically use a cost model to decide where to execute in the network. Unusually this form of autonomous mobility affects only where the program executes and not what it does. We present a generic AMP cost model, together with a validated instantiation performance results for matrix multiplication AMPs. The motivation for AMPs is to minimise processing time by seeking the most favourable resources, without any requirement to visit specific processors. Thus different concrete realisations of a program may carry out the same computation in a shortest time period with given resources, but the patterns of coordination may be very different. We will explore this further below.

3.1 Introduction

Agent technology is a high-level, implementation independent approach to developing software as collections of distinct but interacting entities which cooperate to achieve some common goal. With the continuing decline in price and increase in speed of both processors and networks, it has become feasible to apply agent technology to problems involving cooperation in distributed environments, in particular, where agents may change location, typically to manipulate resources in varying locations.

Most distributed environments are shared by multiple users. In particular,

distributed agent-based systems must also contend with external competition for resources, not least for the processing elements they share. However, agent mobility in such distributed systems tends to be driven by concerns relating to the collective goal of the agent system, independent of the actual environment in which the system is running. Thus, such systems tend not to be aware of, or respond to, environmental changes which impact on the effectiveness with which they may contribute to the collective goal.

For example, if the external load on a shared processing element increases, and the agent's activity does not require its presence on that specific processing element, then it may advantageously move to a more lightly loaded processing element without otherwise affecting its behaviour. In the absence of self- and environmental awareness, however, such systems may suffer widely varying processing and response times as the local environment changes, and accurate prediction of their behaviours becomes problematic.

We have been exploring what we term *autonomous mobile programs* (AMPs) which are aware of their processing resource needs and sensitive to the environment in which they execute. Our experiments suggest that AMPs are able to dynamically relocate themselves to minimise processing time in the presence of varying external loads on shared processing elements. This work is novel in that:

- mobility is truly autonomous as the AMPs themselves use local and external load information to determine when and where to move rather than relying on a central scheduler;
- AMPs combine analytic cost models with empirical observation of their own behaviours to determine their current progress;
- The cost of movement may be kept to a very small proportion of the overall execution time.

The structure of this chapter is as follows. Section 3.2 introduces related work. Section 3.3 presents the generic cost model for autonomous mobile programs. Section 3.4 implements the cost model for matrix multiplication and validates it.

Section 3.5 presents experimental results for single AMP reaction to the change of environment. Finally, Section 3.6 summarises.

3.2 Related Work

AMPs have strong connections with both *agents* and *autonomous systems*, see Figure 3.6.

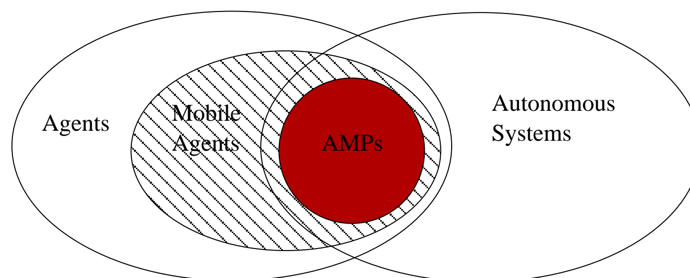


Figure 3.6: Agents, Autonomous Systems, and AMPs

AMPs differ from previous mobile agents. They have cost models and are autonomous, making decision themselves when and where to move.

AMPs are different from autonomous systems. Some autonomous systems are demand-driven e.g. Autonomic Job Scheduling Policy (ASP)[6] where nodes in the system look for work when their load is below a given threshold, but AMPs ask to move. There are schedulers in the ASP system to decide whether to move or not. In AMPs there is no scheduler at all.

Some autonomous systems are agent-based i.e. *the key abstraction used is that of an agent*[133]. Most autonomous mobile agents system adapt the computation, but AMPs adapt their coordination. According to the condition given by the programmer, AMPs make the decision when and where to move by checking the environment where they are executing. AMPs are not simply a kind of autonomous agent but a mobile agent with cost models.

AMPs are also similar to ethological models like Ant Algorithms. Both systems are searching for resources, in Ant algorithms [82] for “food”. AMPs and ants algorithms are different: firstly, an Ant algorithm is going to find the fastest path to get to the “food”, but AMPs are going to find resources and decide

which one is better. Secondly, ant algorithms use the feedback of each mobile agent(ant) to decide which path is better, but in AMP systems, the AMP does not give feedback. Instead the AMP detects the network's current information and make a decision itself.

3.3 Cost Models

Cost models are used to decide the execution times of AMPs in advance. The general cost of the total execution time of a program, T_{total}, has three components, T_{comp} is the computation time for finishing the task, T_{comm} is the communication time for migrating to another location, and T_{coord} is the coordination time for collecting or exchanging information with other programs or systems.

The computation time is necessary for every program; the other two are not necessary for all programs but they are important properties for some kinds of programs, for example parallel or mobile programs. Hence the cost model for programs is:

$$T_{total} = T_{comp}\ [+\ T_{comm}]\ [+\ T_{coord}]$$

T_{total} : The total time the programs takes

T_{comp} : The computation time

T_{comm} : The communication time

T_{coord} : The coordination time

3.3.1 Traditional Systems Costs

In traditional load management techniques, programs do not need to know how to measure the workload of locations, and how to exchange load information between locations, or to decide when and where to migrate. Thus the total time for executing a program is:

$$T_{total} = T_{comp} + T_{comm}$$

3.3.2 AMPs Costs

In AMP system the AMPs collect load information from the system or other locations, and each AMP is involved in deciding where and when to move, and also sends itself to remote locations i.e. the AMPs know all the activities which happen during the execution of the program. Thus the total time for execution is:

$$T_{total} = T_{comp} + T_{comm} + T_{coord}$$

For AMPs a cost model is used to inform the decision whether to move to a new location. Cost models are typically parametrised on: *system architecture* including *location speed* and *interconnect speed*; *cost of processing data*; *cost of communicating data*; *data size*; and *number of locations*[84]. Figure 3.7 shows the generic cost model for AMPs.

- Equation (3.9) gives the condition under which the program will move, i.e. if the time to complete in the current location is more than the time to complete in the remote location.

- Equation (3.10) states that if there are m communications in a program's lifetime and it will take T_{comm} time for each communication then the total time for communication is T_{comm} by m.

- Equation (3.11) states that if there are p processors the status of the processors is checked n times in a program's lifetime and it will take T_{coord} time for checking one processor once then the total time for coordination is T_{coord} by p by n.

- Equation (3.12) gives the condition under which the program will do the coordination work. This seeks to guarantee that the autonomous mobile program will never be worse than $100 + O$ percent of the static version. This guarantee is only valid providing that the loads on the current and target location do not change dramatically immediately after the move. For example it is easy to construct a pernicious example where each time an AMP moves to a location the load on that location becomes very high.

$$T_{total} = T_{Comp} + T_{Comm} + T_{Coord} \tag{3.8}$$

$$T_h > T_{comm} + T_n \tag{3.9}$$

$$T_{Comm} = mT_{comm} \tag{3.10}$$

$$T_{Coord} = npT_{coord} \tag{3.11}$$

$$T_{Coord} < OT_{static} \tag{3.12}$$

$$n < \frac{OT_{static}}{pT_{coord}} \tag{3.13}$$

$$T_e = W_d/S_h \tag{3.14}$$

$$T_h = W_l/S_h \tag{3.15}$$

$$T_n = W_l/S_n \tag{3.16}$$

$$W_a = \sum W_d \tag{3.17}$$

$$W_d = W_{a(thistime)} - W_{a(lasttime)} \tag{3.18}$$

$$W_l = W_{all} - W_a \tag{3.19}$$

O	:	Overhead e.g. 5%
T_{total}	:	total time
T_{static}	:	time for static program running on the current location
T_{Comm}	:	total time for communication
T_{comm}	:	time for a single communication
T_{Coord}	:	total time for coordination
T_{coord}	:	time for coordination with a single processor(location)
T_{Comp}	:	time for computation
T_e	:	time has elapsed at current location
T_h	:	time will take here
T_n	:	time will take in the next location
W_{all}	:	all work
W_a	:	the total work which has been done
W_d	:	the work has been done at current location
W_l	:	the work left
S_h	:	the current CPU speed
S_n	:	the next location CPU speed
m	:	number of communication
n	:	number of coordination
p	:	number of processor

Figure 3.7: Generic Cost Model for Autonomous Mobile Programs

- Substituting Equation (3.11) in (3.12) gives Equation (3.13), where n specifies how many times the AMP will consider moving.
- Equations (3.14), (3.15) and (3.16) relate time, work and CPU speed. The time equals the work measured by CPU speed.
- In Equation (3.17), W_d is the work that has been done at one location, so the total work is the sum of all the W_d. In other words, (3.18) shows that the work done at the current location equals all the work that has been done ($W_{a(thistime)}$) minus the total work done before the program moved to the current location ($W_{d(lasttime)}$).
- Equation (3.19) gives the remaining work, that is the total work minus all the work that has been done.

3.4 AMP Implementation

3.4.1 Mobile Programming Languages Choice

Jocaml[44] has been selected as the initial implementation language for AMPs for the following reasons.

- Jocaml is an extension of Objective Caml 1.07[76]. It has all OCaml's functional, imperative, and object-oriented features, and can build not only functional but also imperative programs.
- Jocaml supports concurrency and synchronisation, the distributed execution of programs, and the dynamic relocation of active program fragments during execution.
- Jocaml Language has strong computation mobility, with explicit primitives for migration. It works on large scale networks and is programming location-aware [17].
- Jocaml is a strict language[55], and hence has simple cost models, compared to mHaskell[38] a lazy language[119].

- The programming model of Jocaml is based on the join calculus[25] which uses ML's function bindings and pattern-matching on messages to express local synchronisations[44]. Jocaml implements join-calculus' mobility which provides transparent migration.

- It is freely available from the web site[4].

Jocaml also proved to be very expressive. Many applications have been developed using it[4].

Java Voyager[127, 100] and JavaGo[108] are used for object oriented implementation in Chapters 4 and 5.

3.4.2 Matrix Multiplication Cost Model

An AMP matrix multiplication has been developed, which can migrate from one location to another according to local and remote load information. Figure 3.8 shows the static program, which is based on the simple three `for` loops of matrix multiplication.

```
for i = 0 to n-1 do          (*first level*)
   for j = 0 to n-1 do       (*second level*)
     for k = 0 to n-1 do     (*third level*)
       m3.(i).(j) <- m3.(i).(j)+ m1.(i).(k) * m2.(k).(j);
     done
   done;
done ;;
```

Figure 3.8: Jocaml Three `for` loop Matrix Multiplication

In the first level loop, code is inserted to check whether completing at the current location will take longer than completing at the fastest location(Equation (3.9)) and if so to move.

The cubic cost model for naive matrix multiplication is well known and is used to instantiate the generic AMP cost model from section 3.3.2 to give the problem specific cost model in Figure 3.9. In Equations (3.24), and (3.25) *Sec* is

$$W_{all} = n^3 \tag{3.20}$$

$$W_a = f(i,j,k) = (i-1)n^2 + (j-1)n + k \tag{3.21}$$

$$W_l = W_{all} - W_a = n^3 - f(i,j,k) = n^3 - (i-1)n^2 + (j-1)n + k \tag{3.22}$$

$$W_d = W_{a(thistime)} - W_{a(lasttime)} = f(i,j,k)_{thistime} - f(i,j,k)_{lasttime} \tag{3.23}$$

$$T_e = \frac{W_d}{S_h} = \frac{[f(i,j,k)_{thistime} - f(i,j,k)_{lasttime}]\,Sec}{S_h} \tag{3.24}$$

$$T_h = \frac{W_l}{S_h} = \frac{[n^3 - f(i,j,k)]\,Sec}{S_h} \tag{3.25}$$

$$T_h = \frac{[n^3 - f(i,j,k)]\,T_e}{f(i,j,k)_{thistime} - f(i,j,k)_{lasttime}} \tag{3.26}$$

$$T_n = \frac{W_l}{S_n} = \frac{[n^3 - f(i,j,k)]\,Sec}{S_n} = \frac{S_h T_h}{S_n} \tag{3.27}$$

Figure 3.9: Cost Model for AMP Matrix Multiplication

a constant which converts abstract time unit into concrete time (seconds).

- Equation (3.20) shows that the total work to multiply square matrices of dimension n is n^3.
- Equation (3.21) shows that the work that has been done is a function of i, j, k.
- Substituting Equations (3.20) and (3.21) in (3.19) gives Equation (3.22). The remaining time for finishing the program is a function of i, j, k.
- The work that has been done at the current location is the total work done so far minus the total work done previously, giving Equation (3.23).
- Substituting Equation (3.23) in (3.14) the time that has elapsed at current location can be calculated, giving Equation (3.24).
- Substituting Equation (3.22) in (3.15) the time it will take at the current location can be calculated. See Equation (3.25).
- Substituting Equation (3.24) in (3.25) we get Equation (3.26). Hence the time it will take at the current location is a function of i, j, k and T_e.

- Substituting Equation (3.25) in (3.15) the time that will be taken in the next location can be predicted as Equation (3.27).

When this cost model is used in an AMP, the program can determine how much time has elapsed (T_e), and the CPU speed can be found. So it can predict how much time the program will take if it stays in the current location (T_h), and how much time it will take if it moves to a remote location (T_n). According to this information the program can make a decision about whether to move or not.

3.4.3 Validating Matrix Multiplication Cost Model

This section introduces experiments to validate the cost model for matrix multiplication. In these experiments the computation time, communication time, and coordination time are validated. The test environment is based on our local area network with locations e.g. $ncc1710$(534MHZ), $jove$(933MHZ), $lxtrinder$(1894MHZ) etc.

Computation Time Validation

The first experiments are to show if the cost models of elapsed time and remaining time are accurate, and are based on the static version of matrix multiplication on $ncc1710$ with CPU speed 534MHZ. In the experiments, the time at every row has been recorded, so the elapsed time can be calculated. Using Equation (3.26) the remaining time can be predicted. The total computation time for the program can also be predicted. The actual execution time can be calculated at the end of the program. Comparing the predicted time and the actual time, if they are similar then the cost models are accurate.

Table 3.3 shows that the predicted time is very close to the actual time, so the cost model is suitable for this program.

Communication Time Validation

The communication time is the time to send the program to the remote location. In the simple cost models we suppose the time for communication is a function

100 * 100 matrix			
Row	Elapsed Time	Remaining Time	Total Time
1	0.009315	0.922186	0.931501
2	0.018520	0.907486	0.926006
3	0.027551	0.890814	0.918365
35	0.312410	0.580190	0.892600
36	0.321268	0.571143	0.892411
37	0.330097	0.562057	0.892154
38	0.338940	0.553008	0.891948
39	0.347855	0.544081	0.891936
97	0.863379	0.026702	0.890081
98	0.872246	0.017801	0.890047
99	0.881123	0.008900	0.890023
100	0.890144	0.000000	0.890144
actual time=0.890227			

Table 3.3: Jocaml AMP Matrix Multiplication Computation Time Validation

of the size of the matrix (n*n). Hence the time for communication is calculated as:

$$T_{comm} = T_{comm1} - T_{comm2} * n^2 \tag{3.28}$$

T_{comm1} is the time for building a connection from the local location to a remote location, which is called the lookup time. T_{comm2} is the time for sending one unit of data to the remote location. Experiments have been done to test the communication time and Table 3.4 presents the results. The table shows that the time for sending the program to the remote location changes according to the size of matrix. If the size of matrix is smaller than 100*100, the time for sending is almost the same. If the matrix is bigger than 100*100, the time for sending is variable; the bigger the size the more time it takes. If the sending time ($size > 100 * 100$) is divided by n^2 (n is the size of matrix) a constant is found, so the communication time is:

T_{comm}=0.082 + (if n > 100 then 1.87*10^{-6}*n^2 else 0) seconds

In a large-scale network, the communication time is not only related to the size of data to be send, but also the network latency. In Section 7.2.1, a *super*

Size	3*3	20*20	50*50	100*100	200*200	500*500	800*800	1000*1000	Mean
1	0.13	0.08	0.07	0.09	0.17	0.53	1.27	1.75	
2	0.10	0.05	0.08	0.10	0.21	0.54	1.19	1.87	
3	0.02	0.09	0.08	0.06	0.13	0.58	1.39	2.01	
4	0.08	0.10	0.10	0.09	0.18	0.52	1.56	1.83	
5	0.02	0.07	0.08	0.12	0.18	0.66	1.28	2.20	
6	0.14	0.10	0.09	0.09	0.19	0.56	1.45	1.78	
7	0.08	0.07	0.08	0.08	0.19	0.62	1.36	2.31	
8	0.08	0.11	0.12	0.11	0.21	0.51	1.18	1.96	
9	0.11	0.09	0.08	0.08	0.22	0.49	1.19	1.76	
10	0.08	0.07	0.09	0.07	0.20	0.51	1.43	1.87	
Mean	0.07	0.08	0.08	0.08	0.17	0.50	1.21	1.76	
Tcomm1	0.082								
Tcomm2=(Mean-Tcomm1)/(n*n)					2.35E-06	1.69E-06	1.77E-06	1.68E-06	1.87E-06
T_{comm}=0.08 + (if n > 100 then 1.87*10^{-6}*n^2 else 0)									

Table 3.4: Jocaml AMP Matrix Multiplication Communication Time Calculation

Number of Host 6			Number of Host 20			Number of Host 40		
	Total Time	Single Time		Total Time	Single Time		Total Time	Single Time
1	2.51	0.42	1	8.84	0.44	1	17.85	0.45
2	2.63	0.44	2	8.73	0.44	2	17.60	0.44
3	2.53	0.42	3	8.97	0.45	3	17.59	0.44
Mean	2.56	0.43	Mean	8.85	0.44	Mean	17.68	0.44
T_{coord}=0.44								

Table 3.5: Jocaml Coordination Time Calculation

load server architecture for AMPs on large scale network is proposed, where the latency of the network is considered.

Coordination Time Validation

The coordination time is the time to discover locations, determine their loads, and make a decision about migration. In the simple cost models (Section 3.3) the total coordination time is $T_{Coord} = n * p * T_{coord}$, where T_{coord} is a single coordination time for one location, p is the number of processors, and n is the time to check the status of the locations. In AMPs it is assumed that T_{coord} is a constant, and that a program should not spend much time on this work, so the smaller the coordination time the better the efficiency. Table 3.5 shows that the coordination time for checking a single location single time is: $T_{coord} = 0.44$ seconds. So the total coordination time is: $T_{Coord} = 0.44 * p * n$ seconds.

3.4.4 Matrix Multiplication AMP Speed Up Measurement

This experiment compares the execution time of the static and AMP matrix multiplication. The test environment is based on three locations: $ncc1710$(534MHZ), $jove$(933MHZ), $lxtrinder$(1894MHZ). The loads on these three computers are almost zero. Both the static and AMP matrix multiplication are started on $ncc1710$.

Figure 3.10 shows that the larger the size of the matrix the faster the AMP version is compared with the static version. If the matrix is smaller than a certain

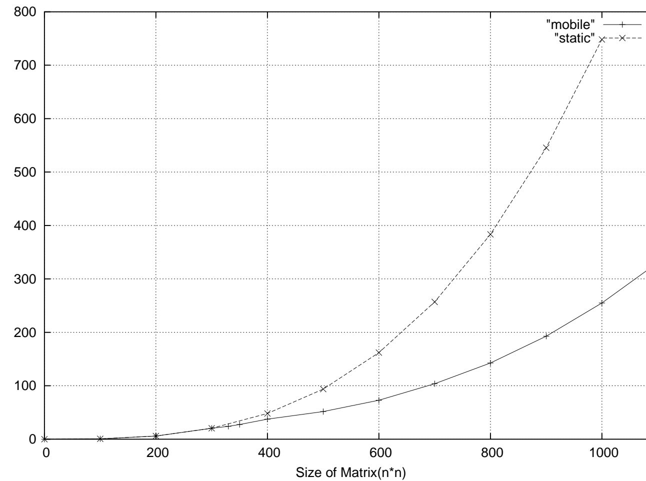


Figure 3.10: Static and AMP Matrix Multiplication Execution Time

size (here 330*330), the AMP version stays on the current location, because it will take more than $O\%$ (overhead) of the time for completion at the current location if the program does coordination and moves. So at this size, the program does not check information and move at all, and the AMP takes almost the same time as the static program. If the size of matrix is bigger than 330*330 then the AMP moves to the fastest location *lxtrinder*, and then stays there, so the AMP takes much less time than the static program.

3.5 AMP Movement Measurement

This section discusses experimental results for single AMP reaction to a change of environment. Figure 3.11 shows the movement of the matrix multiplication AMP during successive execution time periods with CPU speeds normalised by the local loads. The test environment is based on five locations with CPU speeds (Loc1 534MHZ, Loc2 933MHZ, Loc3 1894MHZ, Loc4 2000MHZ, Loc5 1100MHZ). The AMP has been started in time period 0 on $Loc1$. In time period 1 it moves to the fastest processor $Loc4$ (Move (1)). This move shows that if there is a faster location then the AMP moves to it. When $Loc4$ becomes more heavily loaded the program moves to $Loc3$, the fastest processor in period 2 (Move (2)). This move shows that AMPs can respond to changes in current location. In time

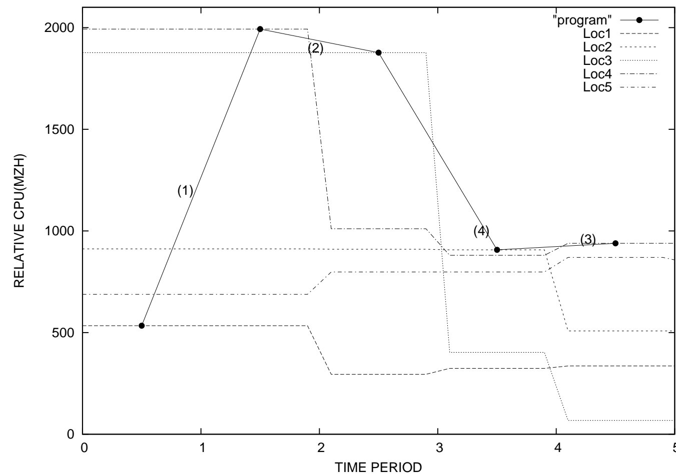


Figure 3.11: AMP Matrix Multiplication Movement Validation

period 3, even though the speed differential of *Loc*3 and *Loc*4 is small, the AMP moves to the faster one (Move (4)). In time period 4, when *Loc*4 becomes less heavily loaded, the AMP moves to it (Move (3)). This move shows that AMPs can respond to changes in other locations. So this test shows that the AMP may move repeatedly to adapt to changing loads and always find the fastest location at one moment. Similar results have been achieved for AMPs using *autonomous mobility skeletons (AMSs)* in Section 4.5.

3.6 Summary

This chapter has presented the general design of an AMP and implemented it with matrix multiplication. According to the design each AMP has a cost model giving accurate information about the time to complete and communicate for the program. Moreover it is possible to parameterise the AMP cost model with a maximum overhead, e.g. 5%, and guarantee under a reasonable assumption the autonomous mobility overheads will not exceed this. We have built the generic cost model for AMPs and instantiated it for naive matrix multiplication and evaluated it. With the problem specific cost model, the AMP matrix multiplication can make the decision when and where to move during the execution time. Experiments have been done to check that the AMP matrix multiplication programs

are faster than the static programs and the movements of the AMP are the same as we expected. Experiments in this chapter also show that a single AMP can move from location to location when the loads are changed in each location.

The advantages of an AMP architecture are as follows. It potentially scales to very large networks, with AMPs making decentralised decisions about where to execute. Indeed on very large networks only nearby locations need be considered as potential targets. The AMP architecture manages dynamic networks very easily with each AMP selecting where to execute from the current set of locations.The AMP architecture can obtain a better balance than a classical distributed load balancer as, unlike the latter, each AMP has a cost model giving accurate information about the time to complete and communicate for the program.

However the AMP architecture may introduce higher coordination costs as every AMP must obtain load information about locations, and the programmer must explicitly control when the program moves. *Autonomous Mobility skeletons* will be developed to encapsulate the mobility control for common patterns of computation in Chapter 4. A load server structure will be introduce to reduce the coordination costs, and the performances of collection of AMPs will be evaluated in Chapter 5.

Chapter 4

Autonomous Mobility Skeletons

Autonomous mobile programs (AMPs) were introduced in Chapter 3. However, a disadvantage of directly programmed AMPs is that the cost model, mobility decision function, and network interrogation are all explicit in the program. This chapter presents *autonomous mobility skeletons* (AMSs), which encapsulate all the self-aware mobile coordination for common patterns of computation over collections. AMSs are akin to algorithmic skeletons in being polymorphic higher order functions, but where algorithmic skeletons abstract over parallel coordination, AMSs abstract over autonomous mobile coordination. We build the AMS cost model for collection iteration and present the `automap`, `autofold` and `AutoIterator` AMSs, together with performance measurements of Jocaml, Java Voyager, and JavaGo implementations on modest LANs. `AutoIterator` is an unusual skeleton, abstracting over the `Iterator` interface commonly used with Java collections.

4.1 Introduction

Abstract skeletons are higher order constructs that abstract over common patterns of coordination and must be parameterised with specific computations. Concrete skeletons are executable, and the user must link computation-specific code into the appropriate skeleton. Figure 4.12 shows the relationship amongst

different species of skeletons. The notion of *algorithmic skeletons* was characterised by Cole[29] to capture common patterns of parallel coordination in a closed, or static, set of locations. *Mobility skeletons* [18] are high-level abstractions capturing common patterns of mobile coordination in an open network i.e. a dynamic set of locations. *Autonomous mobility skeletons* (AMSs) encapsulate autonomous coordination for common computations over collections, like map, fold or iteration. With AMSs, the mobile coordination is explicitly specified by the programmer, and the program does not include additional code for autonomous decisions about where to execute, as AMSs are self-aware. Using AMSs, AMPs can make the decision about when and where to move.

Figure 4.12 distinguishes between the *abstract* conception of skeletons and their *concrete realisations*. As we shall see, AMSs may have different realisations in languages with different mobile constructs. Specifically the realisation in a language with weak mobility will differ from that in a language with strong mobility.

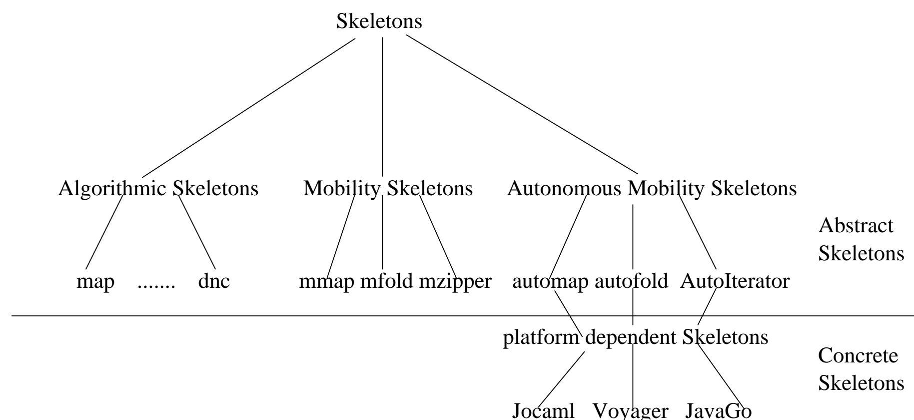


Figure 4.12: Skeleton Taxonomy

Map and Fold (Reduce) are programming models and associated implementations for processing and generating large data sets. Map and Fold have been used across a wide range of domains, including[32]:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,

- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

The `Iterator` interface is commonly used with Java collections. The following sections of this chapter introduce the design, implementation and evaluation of three AMSs (`automap`, `autofold`, and `AutoIterator`) in Jocaml, Java Voyager, and JavaGo.

We argue in [34] that the autonomous mobility skeletons in this chapter have limitations because the skeletons dynamically parameterise the cost model with measurements of performance on the preceding collection segment. If the program is reasonably regular, i.e. computing each segment of the collection represents a similar amount of work, then the cost model will be valid, and hence the movement decisions reasonable. However, as the computations become increasingly irregular, the cost model will be less valid, and hence the movement decisions may not optimise performance. We are proposing how to solve this problem in Chapter 7.

The autonomous mobility skeletons do not incorporate the costs of computations following the processing of the current collection. This restricts autonomous mobility skeletons to programs that expose useful loci of mobility at the top-levels that dominate the computation. Chapter 6 considers how to calculate the cost of the entire program instead of a single collection.

The structure of this chapter is as follows. Section 4.2 implements `automap` in Jocaml and in Voyager, and validates the AMPs with `automap`. Section 4.3 implements `autofold` in Jocaml and in Voyager, and validates the AMPs with `autofold`. Section 4.4 introduces `AutoIterator` in JavaGo. Section 4.5 presents single AMS program behaviour on dynamic load changing networks. Section 4.6

compares the AMS performance in Jocaml and in Voyager. Section 4.7 describes other skeletons. Finally, Section 4.8 summaries.

4.2 Autonomous Mobility Map (`automap`)

The higher-order function `map` applies a given function to a sequence of elements and returns a sequence of results i.e. `map f [a1; ...;an]` applies function `f` to each list element `a1, ...,an`, building the list `[f a1; ...; f an]`. The `automap` AMS performs the same computation as the map higher-order function, but may cause the program to migrate to a faster location.

4.2.1 Collection Iteration Cost Model

The problem specific cost model for AMSs which apply computation over collections e.g. `automap` and `autofold`, are all the same. Using `automap` as an example, this section introduces the cost model for collection iteration. The cost model for `autofold` is the same.

The generic AMP cost model in Section 3.3.2 is used to inform the `automap` decision about moving to a new location [35]. The problem specific cost model determines how much time has elapsed (T_e), and the *relative speed* (CPU speed * (100-load)%) in order to predict the time to complete in the current location (T_h). The network is interrogated to discover the relative speeds of available locations and the time to complete at the fastest remote location (T_n) is calculated. The AMP moves if the predicted time to complete at the current location exceeds the time to move to the best available location (T_{comm}) plus complete there (T_n), i.e. $T_h > T_{comm} + T_n$. We have instantiated the generic AMP cost model for AMSs and validated the cost model. The cost models for AMSs is similar to that for matrix multiplication in Section 3.4.2 and is given in Figure 4.13.

- Equation (4.29) shows that the total work is the length of list `l` in `automap f l`.

- Equation (4.30) gives the work that has been done is `r`.

$$W_{all} = length\ of\ list\ l = a \tag{4.29}$$
$$W_a = r \tag{4.30}$$
$$W_l = a - r \tag{4.31}$$
$$W_d = 1 \tag{4.32}$$
$$T_e = \frac{Sec}{S_h} \tag{4.33}$$
$$T_h = \frac{(a-r)Sec}{S_h} \tag{4.34}$$
$$T_h = (a-r)T_e \tag{4.35}$$
$$T_n = \frac{S_h T_h}{S_n} \tag{4.36}$$

Figure 4.13: Cost Model for Collection Iteration

- Substituting Equations (4.29) and (4.30) in Equation 3.19 (See Section 3.3.2) gives (4.31), which shows that the remaining work is the total work minus the work that has been done.
- Equation (4.32) shows that the unit work that has been done at the current location is 1, which is one element in the list.
- Substituting Equation (4.32) in (3.14) and converting to concrete time in seconds, the time that has elapsed at the current location is calculated giving Equation (4.33). The concrete time can be obtained by timing the program.
- Substituting Equation (4.31) in (3.15) and converting to concrete time, the time that the AMP will take at the current location is calculated giving Equation (4.34).
- Substituting Equation (4.33) in (4.34) and converting to concrete time gives Equation (4.35) for the time, which the AMP will take at the current location as a function of T_e.
- Substituting Equation (4.34) in (3.15) and converting to concrete time, the time that the AMP will take in the next location can be predicted as

Equation (4.36).

With this cost model, AMSs can determine how much time has elapsed (T_e), and the CPU speed can be found. So they can predict how much time the program will take if it stays in the local location (T_h), and how much time it will take if it moves to a remote location (T_n). According to this information AMSs can make a decision about whether to move or not. This cost model will be validated in Section 4.2.4.

4.2.2 Auxiliary Functions for AMS in Jocaml

Potentially AMSs could investigate moving after processing every element of the list, but this induces enormous coordination overheads. Such overheads are limited by specifying that the total coordination overhead of the program (T_{Coord}) must be less than some small percentage (O, say 5%) of the execution time of the static i.e. immobile program (T_{static}):

$T_{Coord} < OT_{static}$ (equation (3.12) in Section 3.3.2)

This overhead will be guaranteed under a reasonable assumption of load uniformity, see Section 3.3.2.

```
let getGran work f h =
  let (fh,fhtime) = timedapply f h
  in let t_static = fhtime * work
     let t_coord = tcoord numofhost
  in let times = ov * t_static / t_coord
  in let gran = if times > 0
                then work/times
                else work
  in (gran,fh,fhtime)
```

Figure 4.14: Jocaml `getGran`: Calculating Coordination Granularity

This section introduces the auxiliary functions which will be used to implement AMSs in Jocaml: the `automap` (Figure 4.17) and `autofold` (Figure 4.27). AMSs investigate moving after processing `gran` elements. Under the assumption

that the `automap` or `autofold` is the dominating computation for the program, `gran` is calculated from the time to compute a single element of the map or fold result (`h`), the length of the list (`work`), and the overhead percentage O (`ov`) by function `getGran::int->(a->b)->a->(int*b*float)` in Figure 4.14.

```
let check_move work workleft fhtime=
  let t_comm = tc work
  let t_h = fhtime * (float (workleft))
  in map (check_relspeed cur) hostlist
     let host_next = check_next cur hostlist
  in let t_n = cur.relspeed / host_next.relspeed * t_h
  in
     if (t_h > (t_n + t_comm))
     then (
       go host_next;
       cur := host_next
     )
     else cur := cur
```

Figure 4.15: Jocaml `check_move`: Deciding to Move in Jocaml

The movement check is encoded in `check_move::int->int->float->unit` function shown in Figure 4.15. Note that the sixth to last line encodes Equation (3.9). In `check_move`, globule variable `cur` is recording current location information, e.g. CPU speed and load, `tc` calculates the communication time, and `check_next` finds out the fastest location in the network, where the AMP will move.

```
let getInfo work workleft gran fhtime f h=
  check_move work workleft fhtime;
  getGran work f h
```

Figure 4.16: Jocaml `getInfo`

Function `getInfo::int->int->int->float->(a->b)->a->(int*b*float)` evaluates the benefits of a move and recalculates `gran` on the new location shown in Figure 4.16.

4.2.3 Jocaml `automap` Design and Implementation

```
let automap f l =
  let work = List.length l
  in let (fh,fhtime,gran) = getGran work f (hd l)
  in fh::automap' work (work-1) gran fhtime f t

let rec automap' work workleft gran fhtime f l =
  let xs = List.map f (take (gran-1) l)
  let (h::t) = drop (gran-1) l
  in let (gran', fhtime',fh') =
             getInfo work workleft gran fhtime f h
  in xs@(fh'::automap' work (workleft-gran) gran' fhtime' f t)
```

Figure 4.17: Jocaml `automap`

The definition of `automap` is given in Figure 4.17. It first calls `getGran` to calculate an initial granularity, before calling `automap'`. `automap'` applies the standard `map` to `gran-1` elements before calling `getInfo` to evaluate the benefits of a move and to recalculate a `gran`.

4.2.4 Jocaml `automap` Cost Model Validation

```
let rec dotprod mat1 mat2 =
  match (mat1,mat2) with
    ((h1::t1),(h2::t2)) -> h1*h2+dotprod t1 t2
  | (_,_) -> 0;;
let inner row col = (dotprod row) col;;
let rowmult row cols = map (dotprod row) cols;;
let outer cols x = rowmult x cols;;
let rowsmult rows cols = automap (outer cols) rows;;   (* automap *)
let mmultMat m1 m2 = rowsmult m1 (transpose m2);;
```

Figure 4.18: Jocaml `automap` Matrix Multiplication

`automap` is validated using two AMPs: matrix multiplication and ray tracing. Figure 4.18 shows how a Jocaml matrix multiplication may be reformulated using `automap`. At first sight, this looks like a conventional program using `map`, but

```
let findImpacts rays objects =
  automap (firstImpact objects) rays;;           (* automap *)
let top detail viewX viewY viewZ scene =
  let  rays = generateRays detail viewX viewY viewZ
  in let imps = findImpacts rays scene
  in show_imps detail detail imps;;
```

Figure 4.19: Jocaml `automap` Ray Tracing

`automap` includes calls to generic and problem specific cost functions to determine whether or not the program should move. Figure 4.19 shows part of the ray tracing program with `automap`.

According to the cost model in Section 4.2.1, the computation times, communication times, and coordination times for these two AMPs have been validated. Note that the test environment in this chapter is the same as in Section 3.4.3, which is based on our local area network with locations e.g. *ncc*1710(534MHZ), *jove*(933MHZ), *lxtrinder*(1894MHZ) etc.

Jocaml `automap` Computation Time Validation

Size	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
300*300	19.61	20.10	2.45
400*400	47.10	47.90	1.68
500*500	93.27	93.91	0.68
600*600	164.55	166.11	0.94
700*700	259.42	266.23	2.56
800*800	401.84	401.31	0.13
900*900	570.71	573.57	0.50
1000*1000	796.34	796.41	0.01

Table 4.6: Matrix Multiplication Computation Time Validation (Jocaml)

To show that the cost model of elapsed time and remaining time are accurate the `automap` matrix multiplication in Jocaml has been evaluated based on the static version. The predicted and actual computation times for a range of matrix

Size	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
50*50	117.46	116.94	1.30
60*60	191.42	185.03	3.98
70*70	271.22	271.96	1.14
80*80	391.66	388.07	0.93
90*90	530.04	530.92	0.37
100*100	725.10	707.09	3.84

Table 4.7: Ray Tracing Computation Time Validation (Jocaml)

sizes have been tested using the same techniques as in Section 3.4.3. At every element of the list, Equation (4.35) is used to predict the remaining time and the total time for the program. At the end of the program the actual execution time can be obtained. We summarise the results in Table 4.6, which show that predicted time is very close to the real time, so accurate predictions of execution time can be achieved i.e. cost models are suitable for this program. In the tables, MAPE is the *mean absolute percentage error* [129], $MAPE = \frac{Mean\ Predict - Mean\ Actual}{Mean\ Actual}$. Table 4.7 shows a similar result for the ray tracing program. Note that ray tracing is generally irregular but regular instances are evaluated in this thesis.

Jocaml `automap` Communication Time Validation

The communication time for matrix multiplication as formulated in Section 3.4.3 is:

$$T_{comm}\text{=}0.082 + (\text{if n} > 100 \text{ then } 1.87 * 10^{-6} * n^2 \text{ else } 0) \text{ seconds}$$

Table 4.8 shows validation of the AMS matrix multiplication communication time. The predicted time is close to the actual time. The worst prediction is 10.9% from the actual time, and the best is 4.7%. From experiments, the communication time for ray tracing is:

$$T_{comm}\text{=}0.020 + (\text{if n} > 20 \text{ then } 2.00 * 10^{-5} * n^2 \text{ else } 0) \text{ seconds}$$

Table 4.9 shows comparable validation of ray tracing and similar results to matrix multiplication have been obtained.

Data Size	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
3*3	0.082	0.075	8.8
20*20	0.082	0.076	7.1
50*50	0.082	0.078	4.7
100*100	0.082	0.081	1.2
200*200	0.155	0.172	10.9
300*300	0.248	0.222	10.6
400*400	0.379	0.345	9.1
500*500	0.548	0.515	5.9

Table 4.8: Jocaml AMS Matrix Multiplication Communication Time Validation

Data Size	Mean Actual	Mean Predict	Mean Absolute Percentage Error MAPE(%)
3*3	0.0173	0.020	13.6
20*20	0.0239	0.020	19.6
50*50	0.0632	0.070	10.0
100*100	0.2196	0.220	0.2
200*200	0.8926	0.819	8.9
300*300	1.9471	1.818	7.1
400*400	3.1513	3.217	2.0

Table 4.9: Jocaml AMS Ray Tracing Communication Time Validation

The communication time, T_{comm}, will be used in Equation (3.9) to make the moving decision in the cost model from Section 3.3.1.

Jocaml Coordination Time Validation

From the experiments on the LAN specified in Section 3.4.3, the coordination time for checking a single location once has been estimated as: $T_{coord} = 0.44 seconds$. So the total coordination time is: $T_{Coord} = 0.44 * p * n$ seconds, where p is the number of processors and n is the times to check the status of the processors, see Equation 3.11 in Section 3.3.1. T_{Coord} will be used in Equation 3.12 to decide when the AMPs check.

Table 4.10 shows validation of the coordination time model for matrix multiplication. The predicted time is very close to the actual time: the worst prediction is 2.06% of the actual time and the best is 0.09%. The coordination time

Locations	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
2	0.88	0.86	2.06
3	1.32	1.30	1.23
4	1.76	1.75	0.64
5	2.20	2.16	1.82
15	6.60	6.61	0.09
25	11.00	11.02	0.21

Table 4.10: Jocaml Coordination Time Validation

is independent to the computation, so all the AMPs in Jocaml have the same coordination time.

Improving the Prediction of Communication Time

Table 4.8 shows the prediction of the communication time for matrix multiplication and ray tracing. The worst prediction of is 10.9%, which is not as good as the prediction of the computation time (2.56% in Table 4.6) and coordination time (2.06% in Table 4.10). Similar results have also been noticed in the experiments for ray tracing and coin counting [69] (Section 4.3.2) programs in both Jocaml and Voyager in this and following sections.

The reason is that the current prediction of communication time is parameterised on the size of data to be sent but not the latency between locations, which may also affect the communication time. So in future work in Section 7.2, we suggest that the communication time should be parameterised on both the data size and network latency, which may provide better prediction of communication time.

4.2.5 Jocaml `automap` AMPs Speed Up Measurement

Figure 4.20 shows the execution times of the `automap` matrix multiplication programs against the static programs. The test environment is based on three locations with CPU speeds 534MHZ, 933MHZ and 1894MHZ. The loads on these three locations are almost zero. Both the static and the AMS programs are

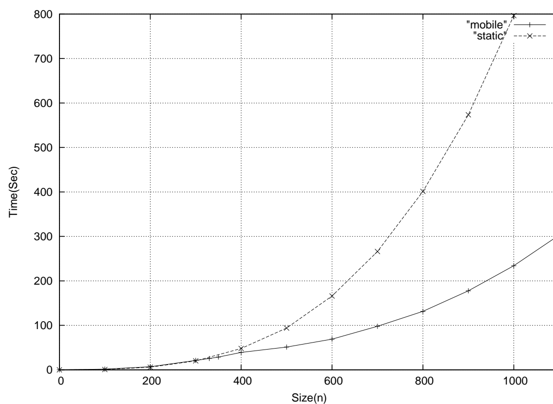


Figure 4.20: Static and AMS Matrix Multiplication Execution Time Comparison (Jocaml)

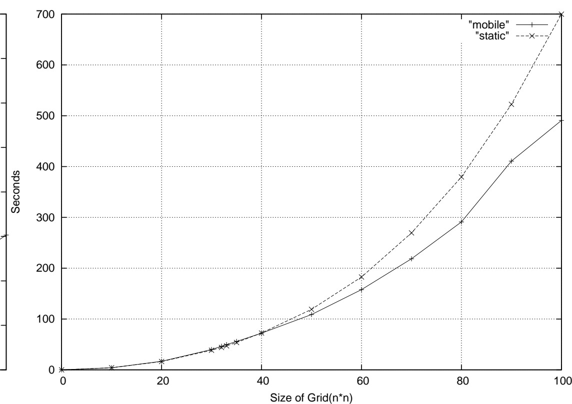


Figure 4.21: Static and AMS Ray Tracing Execution Comparison Times (Jocaml)

started on the first location. The result is similar to the `for` loop matrix multiplication programs in Section 3.4.4. Similar results have been achieved for ray tracing programs as shown in Figure 4.21.

4.2.6 Java Voyager `automap` Design and Implementation

It is appealing to implement Java AMSs, as Java is a very widely used language and there are numerous mobile Java variants. Voyager[127] is a popular Java with weak mobility, providing a set of basic and advanced services and features for distributed application development. Voyager ORB includes distributed naming service and mobile agent technology [127]. Two AMSs have been developed in Voyager. This section introduces `automap`. Section 4.3.4 will introduce `autofold`.

The Voyager `automap` performs the same computation as, and similar coordination to, the Jocaml `automap`. Figure 4.22 gives the definition of `automap` in Voyager, where the Java `check_move` and `getGran` auxiliary functions have the same functionality as those in Jocaml in Section 4.2.2. The Voyager auxiliary functions are shown in Appendix B.2. As Java 1.4 has no parametric polymorphism, the Voyager `automap` operates on a list of `Object` and returns a list of `Object`.

An AMS matrix multiplication is readily written in Java Voyager using `automap`, shown in Figure 4.23. The new class `Auto` has an object `auton`, which includes

```
public Object[] automap (Superclass obj, Object[] l){
     ... ...

  for(int i=0;i<work;i++){ // map
    if( (i-checkPos) == 0 ){
      timestart = System.currentTimeMillis();
      resultl[i] = proxy.mapf (l[i]);
      timeend = System.currentTimeMillis();
      fhtime = timeend-timestart;
      gran = getGran (work,fhtime);
      checkPos = checkPos + gran;
      check_move (work,(work-i-1),fhtime,mobility);
    }
    else {
      resultl[i] = proxy.mapf (l[i]);
    }
  }
  return resultl;
}
```

Figure 4.22: Java Voyager `automap`

`automap`. Class `RowMult` has a function `mapf`, which is the function the `map` will apply to the collection. When `auton.automap (rowM, mat1)` is called, `automap` will apply `rowM.mapf` on array `mat1`, and at the same time `automap` makes the decision of when and where to move. An AMS ray tracer is also written in Java Voyager as shown in Figure 4.24.

4.2.7 Java Voyager `automap` Cost Model Validation

The execution times, communication time, and coordination time have been validated for matrix multiplication and ray tracing in Voyager, as for the programs in Section 4.2.4 in Jocaml.

Java Voyager `automap` Computation Time Validation

Table 4.11 shows the predicted and actual execution times of a range sizes of matrix multiplication in Voyager. We conclude that the predicted time is very close to the actual time i.e. cost models are suitable for this program. Table 4.12

```
public static void main (String[] args){
  int[][] mat1 = makeMatrix(size);
  int[][] mat2 = makeMatrix(size);
  int[][] matT = transpose(mat2);

  RowMult rowM = new RowMult(matT);
  Auto auton = new Auto();
  int[][] res = auton.automap (rowM, mat1);  /*  automap */
}
```

Figure 4.23: Java Voyager `automap` Mobile Matrix Multiplication

```
  public static void top(int detail,double viewX,double viewY,
                       double viewZ,Poly[] scene,String port)
  {
    Ray[] rays;
    Impact[] imps;
    RayFunctions rf = new RayFunctions();
    rays=rf.generateRays(detail,viewX,viewY,viewZ);

    FirstImpact fi = new FirstImpact(scene);
    Auto auton = new Auto(port);

    imps = auton.automap(fi,rays);   /* automap */
    return;
  }
```

Figure 4.24: Java Voyager `automap` Ray Tracing

shows a similar result for ray tracing.

Java Voyager Communication Time Validation

From experiments , the communication time for matrix multiplication in Voyager is:

$$T_{comm}\text{=0.029 + (if n > 50 then } 5.07*10^{-6}*n^2 \text{ else 0) seconds}$$

and the communication time for ray tracing is:

$$T_{comm}\text{=0.035 + (if n > 20 then } 3.97*10^{-5}*n^2 \text{ else 0) seconds}$$

Size	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
300*300	1.29	1.27	1.55
400*400	3.11	3.00	3.51
500*500	5.63	6.03	6.72
600*600	9.75	10.20	4.41
700*700	15.75	16.00	1.56
800*800	23.00	23.80	3.36
900*900	32.40	33.40	2.99
1000*1000	45.25	46.20	2.06

Table 4.11: Matrix Multiplication Computation Time Validation (Voyager)

Size	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
50*50	31.88	32.37	2.13
60*60	45.00	45.50	3.96
70*70	58.80	61.05	3.68
80*80	78.40	79.36	3.63
90*90	97.20	100.12	2.91
100*100	122.50	123.98	3.62

Table 4.12: Ray Tracing Computation Time Validation (Voyager)

Table 4.13 shows validation of our Voyager `automap` matrix multiplication communication time. The predicted time is close to the actual time: the worst prediction is 12.5% from the actual time and the best is 2.0%. Table 4.14 validates the communication time for ray tracing. The communication time, T_{comm}, is used in Equation 3.9 to make the moving decision in the cost model in Section 3.3.1.

Java Voyager Coordination Time Validation

From the experiments, the coordination time in Voyager for checking a single location once has been estimated as: $T_{coord} = 0.25 seconds$. So the total coordination time is: $T_{Coord} = 0.25 * p * n$ *seconds*, where p is the number of processors and n is the times to check the status of the processors, see Equation 3.11 in Section 3.3.1. T_{Coord} will be used in Equation 3.12 to decide when the AMP shall take checking.

Data Size	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
3*3	0.029	0.028	5.0
20*20	0.029	0.028	5.0
50*50	0.042	0.047	12.5
100*100	0.081	0.079	2.3
200*200	0.236	0.259	9.9
300*300	0.495	0.510	3.0
400*400	0.857	0.840	2.0
500*500	1.323	1.276	3.5

Table 4.13: Voyager AMS Matrix Multiplication Communication Time Validation

Data Size	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
3*3	0.035	0.0328	7.2
20*20	0.035	0.0387	9.4
50*50	0.135	0.1539	14.3
100*100	0.433	0.4325	0.1
200*200	1.624	1.7047	4.9
300*300	3.611	3.6811	1.9
400*400	6.392	6.6080	3.4

Table 4.14: Voyager AMS Ray Tracing Communication Time Validation

Table 4.15 shows validation of the coordination time model for matrix multiplication in Voyager. The predicted time is very close to the actual time: the worst prediction is 11.6% from the actual time and the best is 1.0%. The coordination time is independent of the computation, so all the AMPs in Voyager have the same coordination time.

4.2.8 Java Voyager `automap` AMPs Speed Up Measurements

Figure 4.25 compares the execution times of static and AMS versions of Voyager matrix multiplications, using the apparatus from section 4.2.5. Figure 4.26 shows a similar result for ray tracing.

Locations	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
3	0.76	0.75	1.0
4	1.01	0.89	11.6
5	1.26	1.15	9.0
15	3.78	3.92	3.9
25	6.30	6.55	4.0

Table 4.15: Voyager Coordination Time Validation

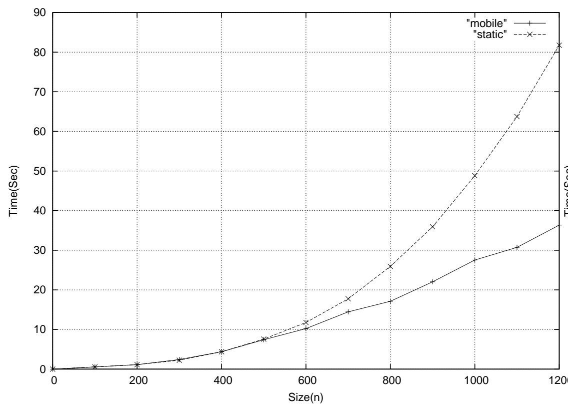


Figure 4.25: Static and AMS Matrix Multiplication Execution Times Comparison (Voyager)

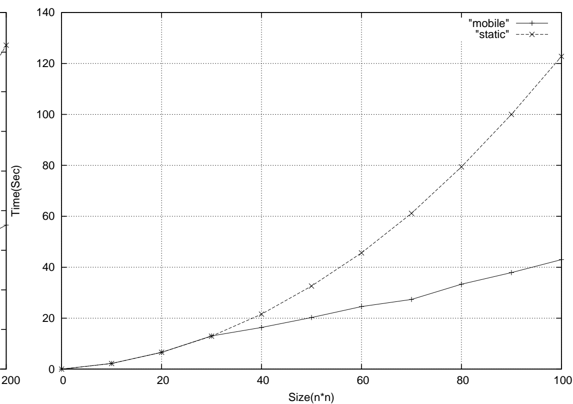


Figure 4.26: Static and AMS Ray Tracing Execution Times Comparison (Voyager)

4.3 Autonomous Mobility Fold (`autofold`)

The higher-order function `fold` processes a data structure in some order and builds a return value. Typically, `fold` deals with two things: a combining function, and a data structure, e.g. a list of elements. The fold proceeds to combine elements of the data structure using the function in some systematic way. The standard fold, `fold f a [b1; ...; bn]`, computes `f (... (f (f a b1) b2) ...) bn`. `autofold f a [b1;...;bn]` computes the same value but may migrate to a faster location.

The cost model for `autofold` is the same as that for `automap` which is given in Figure 4.13 in Section 4.2.1. This cost model will be used in the implementations of `autofold` in Jocaml and Voyager in Section 4.3.1 and 4.3.4

4.3.1 Jocaml autofold Design and Implementation

The definition of `autofold` in Jocaml is given in Figure 4.27. It uses the same aux-

```
let autofold f accu l =
  let work = length l
  in let (fh,fhtime,gran) = getGran work (f accu) h
  in autofoldl' work (work-1) gran fhtime f fh t

let rec autofold' work workleft gran fhtime f accu l =
  let xs = fold f accu (take (gran-1) l)
  let (h::t) = drop (gran-1) l
  in let (gran', fhtime',fh') =
              getInfo work workleft gran fhtime (f xs) h
  in autofoldl' work (workleft-gran) gran' fhtime' f fh' t
```

Figure 4.27: Jocaml `autofold` Definition

iliary functions `getGran`, `check_move`, and `getInfo` in Section 4.2.2 as `automap` in Jocaml. `autofold` first calls `getGran` to calculate an initial granularity before calling `autofold'`. `autofold'` applies standard `fold` to `gran-1` elements before calling `getInfo` to evaluate the benefits of a move and to recalculate a `gran`.

4.3.2 Jocaml autofold Cost Model Validation

`autofold` has been used to construct a coin counting problem that uses a genetic algorithm [69] to find a minimal and maximal set of coins that sum to a target figure [135]. The coin counting problem is to find suitable numbers of each kind of coin, if there are `n` kinds of coin presenting different values, e.g. 100, 50, 20, 10, 5, 2 and 1, and a target sum of money is given. The sum of coins' values must be the same as the given money and the total number of coins must be between minimum and maximum numbers. Therefore, the problem could be expressed as `p min max`, where `p` represents target money, `min` represents minimum number of coins and `max` represents maximum number of coins [135]. Figure 4.28 shows the coin counting program using `autofold`.

```
let repeat n f p =
  let intf p n = f p in
  let l = makelist n in
  autofold intf p l                    (* autofold *)

let next (gsPop,gsCache) =
  let gParentA = selectRawTournament gsPop false 2 rawtournrate true
  and gParentB = selectRawTournament gsPop false 2 rawtournrate true in
  let gChild = crossNpt gParentA gParentB (glength-1) crossoverrate
  in (gsPop, push gChild gsCache)

let loop p min max gsPop =
  let (_,Gstack(pgwH,nE)) = ntimes nPopsize next (gsPop,empty()) in
  let pgwH' = List.map (Decimal.mutate maxdec mutationrate) pgwH in
  let gsPop' = Gstack((List.map (coins p min max) pgwH') , nE) in
  let gsPop'' = rankRaw gsPop' false in
  gsPop''

let rec test p min max =
  let (Gstack(pgwH,nE)) = makeinit nPopsize [] nPopsize in
  let gsPop = Gstack(List.map (coins p min max) pgwH,nE) in
  let gsPop' = rankRaw gsPop false in
  let gsPop'' = repeat 100 (loop p min max) gsPop' in
 gsPop''
```

Figure 4.28: Jocaml `autofold` Coin Counting

Jocaml `autofold` Computation Time Validation

The coin counting program has been evaluated, to show that the cost model of elapsed time and remaining time are accurate for `autofold`, as has been done in Section 4.2.4 for matrix multiplication. Table 4.16 shows that the predicted time is very close to the real time. So an accurate predictions of processing time can be achieved, i.e. cost models are also effective for this program.

Jocaml `autofold` Communication Time Validation

The communication time for the coin counting program is different from matrix multiplication. In the matrix multiplication, when the program moves to the remote location, it brings the matrix with it, so the communication time is a

Size	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
10	4.48	4.63	3.22
20	9.25	9.16	1.08
30	13.38	13.65	1.94
40	17.84	18.14	1.64
50	22.81	22.69	0.52
60	26.89	27.12	0.84
70	31.54	31.68	0.42
80	35.95	36.17	0.60
90	40.39	40.79	0.99
100	45.25	45.62	0.80

Table 4.16: Coin Counting Computation Time Validation (Jocaml)

function of the matrix size. In the coin counting problem, there is no heavy data like a matrix and the program only brings the code, so the communication time should be a constant. Using the same techniques as in Section 3.4.3, the communication time for coin counting has been calculated as $T_{comm} = 0.012$ *seconds*. T_{comm} will be used in Equation 3.9 to make the moving decision in the cost model in Section 3.3.1. Table 4.17 shows validation of the coin counting communication

Data Size	Mean Actual	Mean Predict	Mean Absolute Percentage Error MAPE(%)
20	0.0119	0.012	0.6
50	0.0117	0.012	2.7
100	0.0135	0.012	12.2
200	0.0114	0.012	5.1
300	0.0123	0.012	2.2
400	0.0122	0.012	1.3
500	0.0129	0.012	7.2
600	0.0136	0.012	13.3
700	0.0120	0.012	0.2
800	0.0121	0.012	1.2
900	0.0137	0.012	14.5
1000	0.0121	0.012	0.9

Table 4.17: Jocaml AMS Coin Counting Communication Time Validation

time model. The predicted time is close to the actual time: the worst prediction is 14.5% from the actual time and the best is 0.2%.

Coordination Time Validation

The coordination time for coin counting program in Jocaml is the same as the coordination time for the matrix multiplication in Jocaml which has been validated in Section 4.2.4.

4.3.3 Jocaml `autofold` AMPs Speed Up Measurements

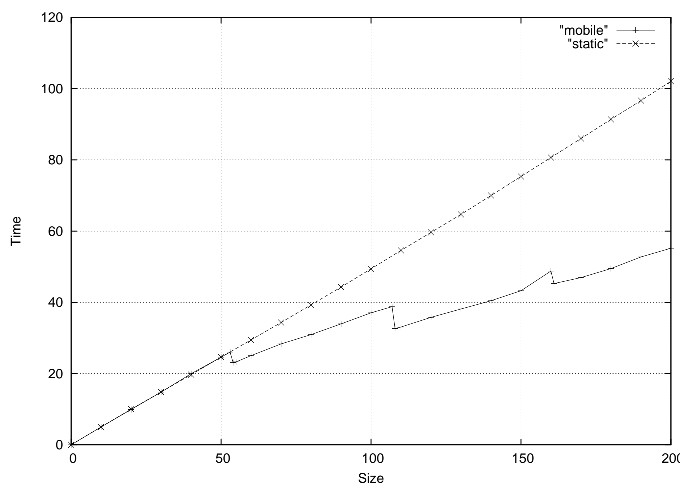


Figure 4.29: Static and AMS Coin Counting Execution Time (Jocaml)

Figure 4.29 shows the execution times of static and `autofold` coin counting programs. As before, once the program has a sufficiently large execution time, it benefits from moving to a faster location. In this figure, there are three clear irregularities in the mobile version plot. That is because as the size of the program increases, `gran` (see Figure 4.14) may decrease. So at some points, even if the size of the program is increased, it may move earlier to the faster location than the smaller program, so the bigger program finishes faster than the smaller program. For example, the program with size 50 matrix does not move, but the one with size 60 matrix moves. Similarly, the `gran` of size 100 is 51, but the `gran` of size 110 is 37, so the size 110 program moves to a faster location earlier than size 100 program. So there is an irregularity at point 110 in the plot. These irregularities also arise in Figure 4.20, and Figure 4.21, but they are too small to be noticed.

4.3.4 Java Voyager `autofold` Design and Implementation

```
public Object[] autofold (Superclass obj,Object b,Object[] l){
          .......

  for(int i=0;i<work;i++){ // fold
    if( (i-checkPos) == 0 ){
      timestart = System.currentTimeMillis();
      result = proxy.foldf (result, l[i]);
      timeend = System.currentTimeMillis();
      fhtime = timeend-timestart;
      gran = getGran (work,fhtime);
      checkPos = checkPos + gran;
      check_move (work,(work-i-1),fhtime,mobility);
    }
    else{
      result = proxy.foldf (result, l[i]);
    }
  }
  return resultl;
}
```

Figure 4.30: Java Voyager `autofold`

`autofold` is also readily constructed in Java Voyager. Figure 4.30 shows the definition of `autofold`. The `check_move` and `getGran` auxiliary functions in voyager have the same functionality as in Jocaml which has been presented in Section 4.2.2. The auxiliary functions in Voyager are presented in Appendix B.2.1.

4.3.5 Java Voyager `autofold` Cost Model Validation

Figure 4.31 shows the coin counting program in Voyager. Using the same techniques as in Section 4.2.7, the execution times, communication time, and coordination time for coin counting have been evaluated.

Java Voyager `autofold` Computation Time Validation

Table 4.18 shows the predicted and actual execution time of coin counting programs with a range sizes of maximum counts. We conclude that the predicted

```
public static void main (String[] args) {
   int gs = Integer.parseInt(args[0]);
   int size = Integer.parseInt(args[1]);
   int[] l = mkArray(gs);

   Coins c = new Coins(size);
   Auto auton = new Auto();
   int sum = auton.autofold(c,0,l); /* autofold */
}
```

Figure 4.31: Voyager `autofold` Coin Counting

Size	Mean Predict	Mean Actual	Mean Absolute Percentage Error MAPE(%)
1000	2.53	2.46	3.01
2000	4.43	4.20	5.45
3000	6.09	5.93	2.68
4000	7.81	7.58	3.08
5000	9.51	9.33	1.92
6000	10.40	10.97	5.20
7000	12.72	12.62	0.76
8000	13.80	14.25	3.18
9000	15.41	16.02	3.85
10000	16.74	17.66	5.24

Table 4.18: Coin Counting Computation Time Validation (Voyager)

time is very close to the actual time, i.e. the cost model in Section 4.2.1 is also effective for this program.

Java Voyager `autofold` Communication Time Validation

Section 4.3.1 shows the communication time for the coin counting program is a constant. From experiments, the communication time for coin counting in Voyager is: $T_{comm} = 0.074$ *seconds*. T_{comm} will be used in Equation 3.9 to make the moving decision in the cost model in Section 3.3.1. Table 4.19 shows validation of the coin counting communication time. The predicted time is close to the actual time: the worst prediction is 11.1% from the actual time and the best is 1.6%.

Data Size	Mean Actual	Mean Predict	Mean Absolute Percentage Error MAPE(%)
20	0.0673	0.074	9.0
50	0.0678	0.074	8.4
100	0.0799	0.074	8.0
200	0.0669	0.074	9.7
300	0.0658	0.074	11.1
400	0.0704	0.074	4.9
500	0.0795	0.074	7.4
600	0.0752	0.074	1.6
700	0.0756	0.074	2.2
800	0.0796	0.074	7.5
900	0.0784	0.074	6.0
1000	0.0787	0.074	6.4

Table 4.19: Voyager AMS Coin Counting Communication Time Validation

Java Voyager Coordination Time Validation

The coordination time for coin counting program is the same as the coordination time for the matrix multiplication which has been validated in Section 4.2.7.

4.3.6 Java Voyager `autofold` AMPs Speed Up Measurements

Figure 4.32 compares the execution times of static and AMS versions of a Java Voyager coin counting program. These results are again similar to those for the Jocaml results.

4.4 An Autonomous Mobile Iterator

4.4.1 `AutoIterator` & JavaGo

`AutoIterator` is a class that implements the Java `Iterator` interface, which specifies a generic mechanism to enumerate the elements of a collection. The methods in the `Iterator` interface are `hasNext, next` and `remove`. The method `hasNext` is a `boolean` valued method that returns `true` if one or more elements

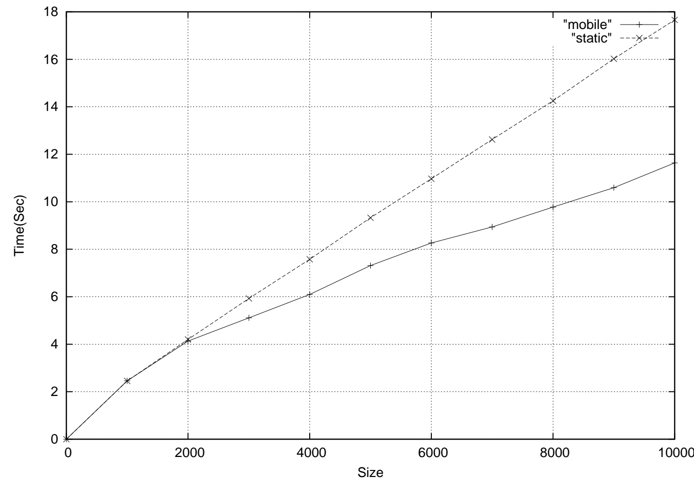


Figure 4.32: Static and AMS Coin Counting Execution Times (Voyager)

remain to be examined. The method `next` returns an unexamined element of the object. The method `remove` deletes the last element that was returned by `next` [105]. The `AutoIterator` class implements all three methods, and extends `Iterator` with `autonext`, which has the same functionality as `next` but can make autonomous mobility decisions.

Figure 4.33 and 4.34 show the mobility structure of programs using `automap` and `AutoIterator`. `automap` and `autofold` can use weak mobility as the computation to be moved is encapsulated in the function mapped or folded. In `AutoIterator`, however, there is no encapsulated computation to be weakly moved and strong mobility is required to move the entire collection. Hence Voyager, with only weak mobility, cannot be used. JavaGo [108] supports strong mobility.

In the program using `automap`, we can just send `Object A` with the data in the object to another location. This is weak mobility. So for building `automap` weak mobility is required. `automap` can also be built using strong mobility such as Jocaml. In the program using `AutoIterator`, `autoNext()` implements the autonomous mobility in the entire program. `autoNext()` does nothing but get the next data from the list, and the computation is the rest of the program after `autoNext()` function. So `AutoIterator` must send the whole computation to another location. So strong mobility is necessary. The mobility structure of

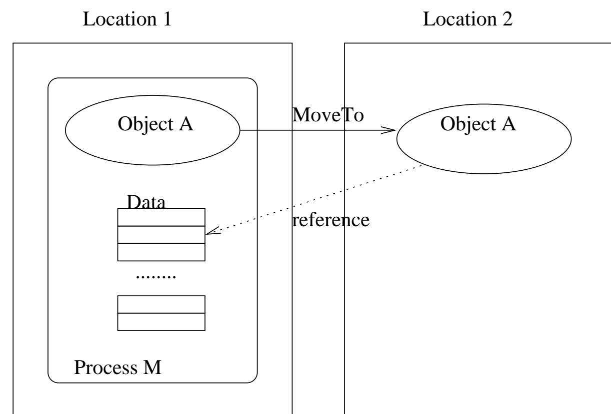


Figure 4.33: Mobility Structure of `automap`

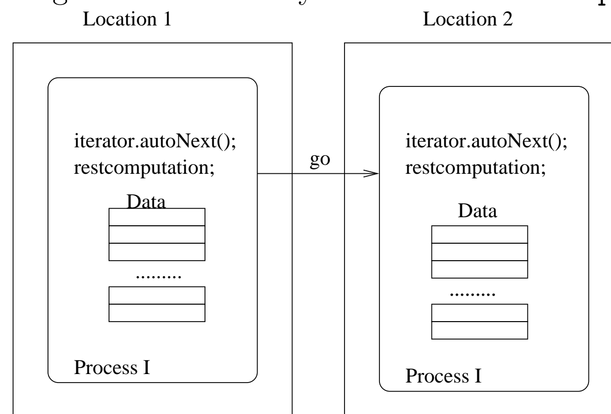


Figure 4.34: Mobility Structure of `AutoIterator`

`autofold` is the same as `automap`.

4.4.2 `AutoIterator` Implementation

Figure 4.35 shows an `autoNext` implementation again using the analogous `check_move` and `getGran` functions. The entire `AutoIterator` class is given in Appendix B.3. `AutoIterator` is very similar to `automap` and `autofold`. It counts the time of computation on the first element of the list, and calculates `gran`. Then it makes the decision autonomously of whether to move or not and where to move after every `gran` elements.

```
public migratory Object autoNext() {
    if (nextIndex < work){
      if(nextIndex == 0){
        timestart = System.currentTimeMillis();
        timeend = timestart;
      }
      else
        if((nextIndex-checkPos) == 0 ){
          timestart = timeend;
          timeend = System.currentTimeMillis();
          fhtime = timeend-timestart;
          check_move (size,(work-nextIndex-1),fhtime);
          gran = getGran (work,fhtime);
          checkPos = checkPos + gran;
        }
      return list.get(nextIndex++);
    }
    else
      throw new NoSuchElementException("No next element");
}
```

Figure 4.35: JavaGo `autoNext` Method in `AutoIterator` Class

4.4.3 `AutoIterator` AMPs Speed Up Measurements

Figure 4.36 shows how `AutoIterator` can be used to implement matrix multiplications. Each element of the list is a `MatrixMul` object and includes two matrices and a function `Multiplication`, which multiplies the two matrices. `AutoIterator` enumerates each object using `autoNext` and performs the multiplication. The entire program is in Appendix B.3

Figure 4.37 shows the execution times of static and `AutoIterator` versions of a JavaGo matrix multiplications program. Once again, the AMS version is faster.

4.5 AMP with AMS Movement Measurement

This section discusses experimental results for a single AMP with an AMS reacting on the change of environment. Experiments have been conducted to test if

```
undock {
  AutoIterator ai = new AutoIterator(al);  /* AutoIterator */
  while (ai.hasNext()){
    MatrixMul iu = (MatrixMul)ai.autoNext();
    int[][] mat = iu.Multiplication();
  }
}
```

Figure 4.36: JavaGo `AutoIterator` Matrix Multiplications

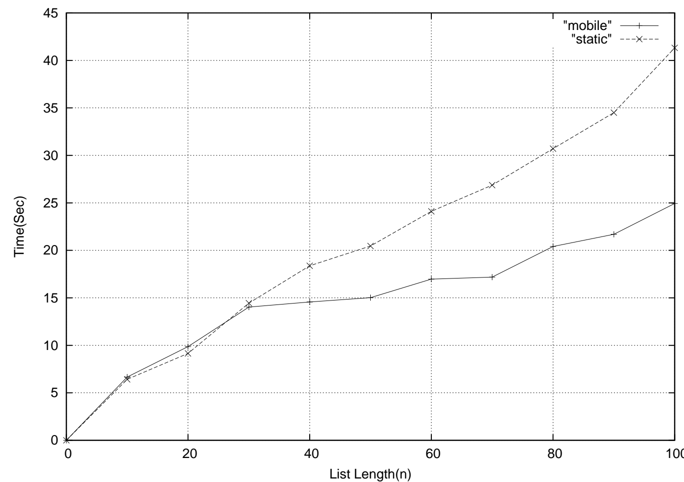


Figure 4.37: Static and `AutoIterator` Matrix Multiplications Execution Times

the program moves as we expect using the the same techniques as in Section 3.5.

Figure 4.38 shows the movement of the matrix multiplication with AMS during successive execution time periods with CPU speeds normalised by the local loads. The test environment is based on five locations with CPU speeds (*Loc*1 534MHZ, *Loc*2 933MHZ, *Loc*3 1894MHZ, *Loc*4 2000MHZ, *Loc*5 1100MHZ). The AMP has been started in time period 0 on *Loc*1. In time period 1 it moved to the fastest processor *Loc*3. When *Loc*3 became more heavily loaded the program moved to *Loc*5, the fastest processor in period 2. In time period 3, *Loc*4 became less loaded, and was the fastest location at that moment, so the program moved to it. In time period 7 *Loc*5 was a little faster than *Loc*2. So the program moved to *Loc*5 rather than staying on *Loc*2. Many AMPs have reproducible behaviour. Figures 4.39 and 4.40 show similar results for the AMS ray tracing and the AMS coin counting.

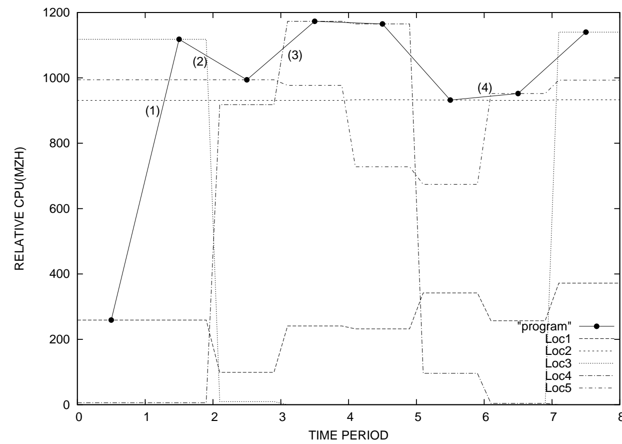


Figure 4.38: AMS Matrix Multiplication Movement Validation

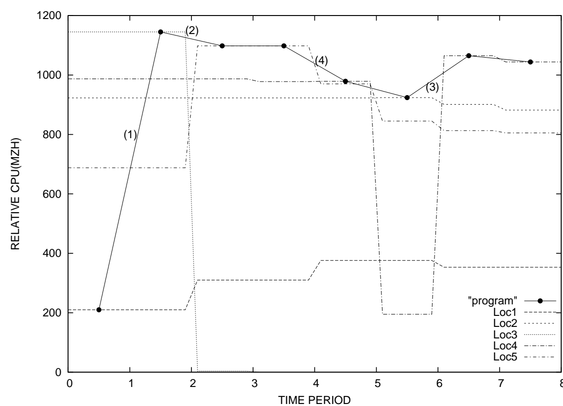


Figure 4.39: AMS Ray Tracing Movement Validation

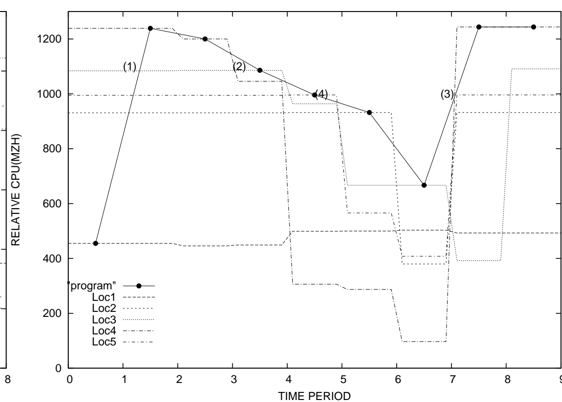


Figure 4.40: AMS Coin Counting Movement Validation

The following conclusions can be drawn from the figures:

- The program may move repeatedly to adapt to changing loads and always find the fastest location in one step.
- Move (1) shows that if there is a faster location then the AMP moves to it.
- Move (2) shows that AMPs can respond to changes in current location.
- Move (3) shows that AMPs can respond to changes in other locations.
- Move (4) shows that even if the speed differential is small, the AMP moves.

4.6 Jocaml and Java Voyager Comparison

As `automap` and `autofold` are developed both in Jocaml and in Java Voyager, this chapter compares these two languages in their mobility behaviour, the computation time, communication time, and coordination time of the AMSs.

4.6.1 Mobility Behaviour Comparison

The Mobility Behaviour of Jocaml AMSs

In Figure 4.41 `automap` is used as an example to explain the coordination behaviour of the Jocaml AMSs. `autofold` has the same behaviour as `automap`. As Jocaml supports strong mobility, the program moves along with its execution state. In the figure, a Jocaml program with `automap` is started, which applies `f` to list `l` in location 1 (1). `automap` will automatically decide whether and where the program moves automatically. So the whole program moves to location 2 with its data and context (2). In location 2, the `automap` consumes the input list (3), and produces a result list (4).

The Mobility Behaviour of Voyager AMSs

As Voyager supports only weak mobility, when the program moves it communicates only the code, and not the execution state. Figure 4.42 shows the coordination behaviour of the Voyager `automap`. Here, a Voyager program with `automap` is started, which applies `f` in `Object A` to list `l` in location 1. The program sends the code of `Object A` to location 2 (1). The system built a reference from location 2 to the data in location 1 (2). In location 2, `f` fetches data from location 1, produces a result and returns it to location 1 (3). After the program has finished, the code of `Object A` stays in location 2 and waits for another migration but the data in location 1 will never move (4).

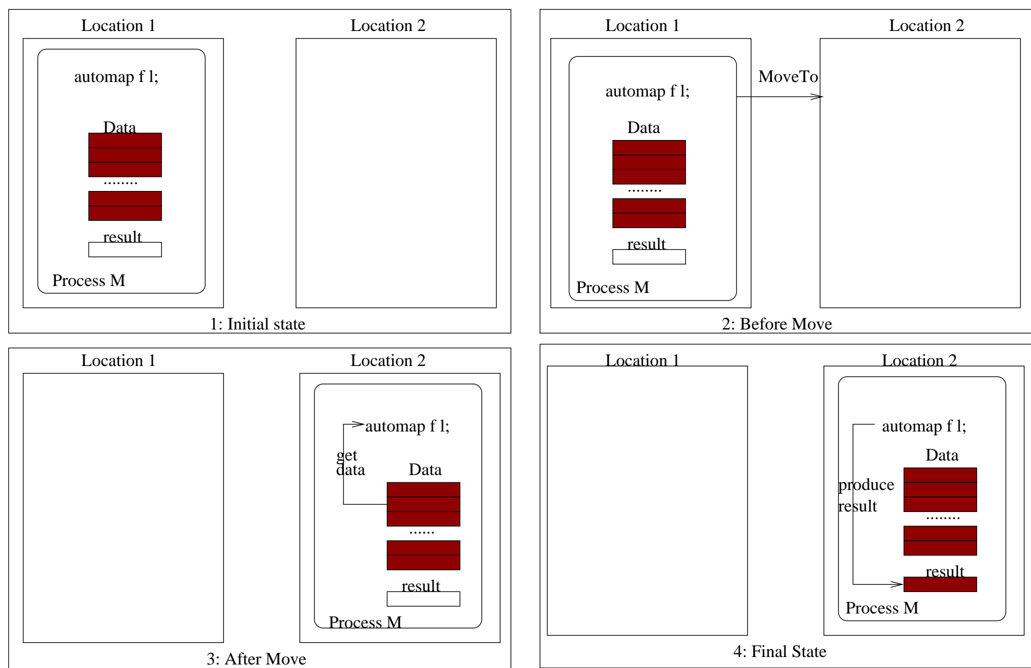


Figure 4.41: Mobility Behaviour of Jocaml autonomous mobility skeletons

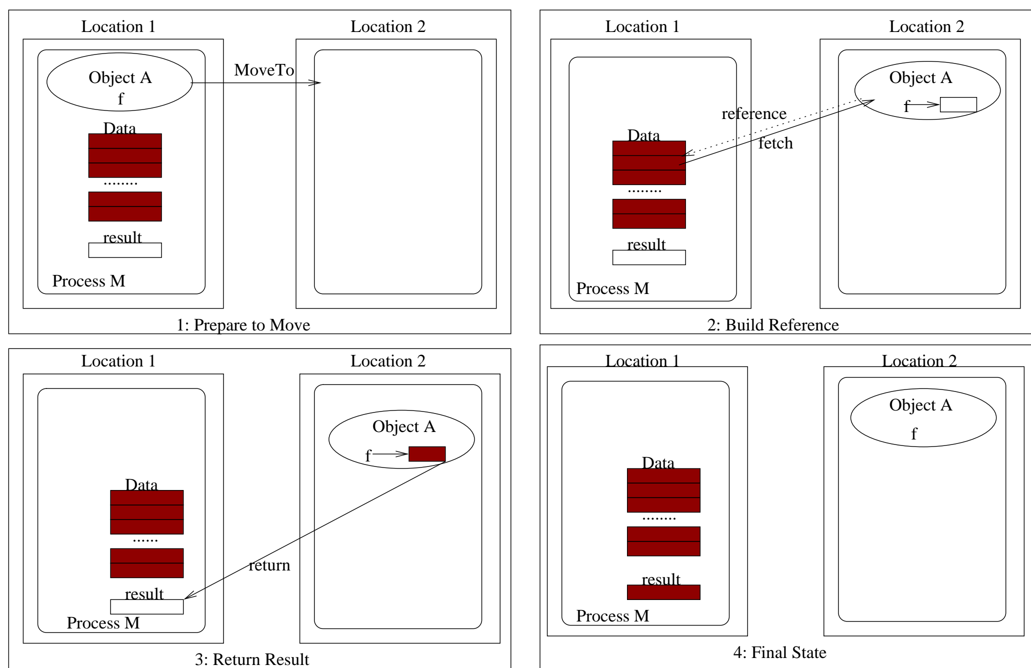


Figure 4.42: Mobility Behaviour of Java Voyager autonomous mobility skeletons

4.6.2 Computation Time Comparison

Substantial differences in the execution times obtained with Jocaml and with Java Voyager can be seen in Figures 4.20 and 4.25. Table 4.20 summarises and compares the execution times of static versions of the Jocaml and Java Voyager matrix multiplication programs. The time complexity of matrix multiplication is $O(n^3)$, so in the third and sixth columns of the table, "time/size^3" is used as a measurement of the time taken for a single matrix element multiplication. From this table, Jocaml matrix multiplications take on average 16.4 times longer than Java Voyager.

Jocaml			Voyager		
size	time	time/size^3	size	time	time/size^3
300	20.10	7.4e^{-7}	300	1.27	4.7e^{-8}
400	47.90	7.5e^{-7}	400	3.00	4.6e^{-8}
500	93.91	7.5e^{-7}	500	6.03	4.8e^{-8}
600	166.11	7.7e^{-7}	600	10.20	4.7e^{-8}
700	266.23	7.7e^{-7}	700	16.00	4.7e^{-8}
800	401.31	7.8e^{-7}	800	23.80	4.6e^{-8}
900	573.57	7.9e^{-7}	900	33.40	4.6e^{-8}
1000	796.41	7.9e^{-7}	1000	46.20	4.6e^{-8}
Average		7.7e^{-7}	Average		4.7e^{-8}
Jocaml/Voyager = 16.4					

Table 4.20: Jocaml and Voyager Matrix Multiplication Execution time Comparison

Jocaml			Voyager		
size	time	time/size^2	size	time	time/size^2
50	116.94	0.05	50	32.37	0.01
60	191.42	0.05	60	45.50	0.01
70	271.22	0.06	70	61.05	0.01
80	391.66	0.06	80	79.36	0.01
90	530.04	0.07	90	100.12	0.01
100	725.10	0.07	100	123.98	0.01
Average		0.06	Average		0.01
Jocaml/Voyager = 6					

Table 4.21: Jocaml and Voyager Ray Tracing Execution time Comparison

Jocaml			Voyager		
size	time	time/size	size	time	time/size
30	13.65	0.45	3000	5.93	0.0019
40	18.14	0.45	4000	7.58	0.0019
50	22.69	0.45	5000	9.33	0.0019
60	27.12	0.45	6000	10.97	0.0018
70	31.68	0.45	7000	12.62	0.0018
80	36.17	0.45	8000	14.25	0.0018
90	40.79	0.45	9000	16.02	0.0018
100	45.62	0.46	10000	17.66	0.0018
Average		0.45	Average		0.0018
Jocaml/Voyager = 250					

Table 4.22: Jocaml and Voyager Coin Counting Execution time Comparison

Similar results have been obtained for ray tracing and coin counting, where Voyager programs are faster than the correspond programs in Jocaml. Table 4.21 shows that the Jocaml ray tracing program is on average six times slower than Java Voyager. For the coin counting program on average Jocaml is 250 times slower than Java Voyager and Table 4.22 summaries the results.

4.6.3 Communication Time Comparison

The communication time for matrix multiplication in Jocaml and Voyager are shown in Tables 4.8 and 4.13. Figure 4.43 summaries the two tables. The figure shows that the communication time in Jocaml and Voyager increase as the size of the matrix increase. If the size of matrix is smaller than 100*100 the Jocaml communication time is bigger than for Voyager, but if the size of the matrix is larger than 100*100 the Jocaml communication time is smaller than for Voyager. Figure 4.44 shows the communication time for ray tracing in Jocaml and Voyager.

The communication time for coin counting is constant both in Jocaml and in Voyager, there is no heavy data like matrix, and the program only brings the code of the program. The communication time for coin counting programs is 0.074 seconds in Voyager and 0.012 seconds in Jocaml. So the communication time in Jocaml is smaller than that in Voyager.

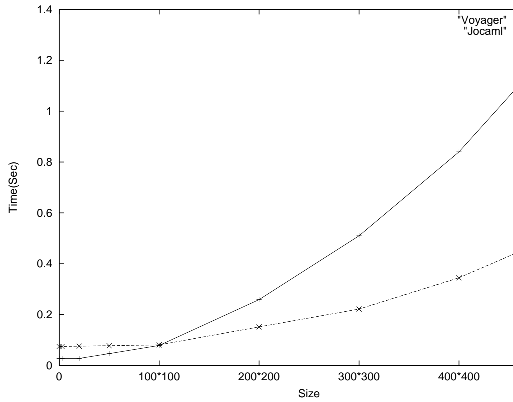


Figure 4.43: Matrix Multiplication Communication Time Comparison

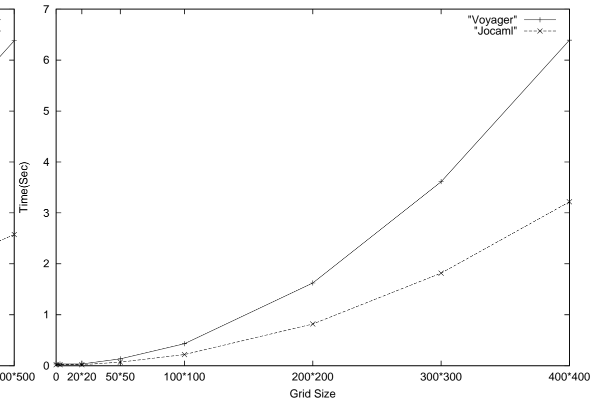


Figure 4.44: Ray tracing Multiplication Communication Time Comparison

4.6.4 Coordination Time Comparison

According to equation 3.11 in Section 3.3.1, the total coordination time is:

$$T_{Coord} = n * p * T_{coord}$$

where T_{coord} is the coordination time with a single processor. In one programming environment the T_{coord} should be the same for different programs, but it should be different in different environments e.g. Voyager and Jocaml. Sections 4.2.4 and 4.2.7 show T_{coord} is 0.44 seconds in Jocaml and 0.25 seconds in Voyager. So the coordination time in Voyager is smaller than in Jocaml. Hence the coordination overhead in Voyager is smaller than in Jocaml. We expect that the smaller the overhead the better the coordination information the programs will obtain. As a result a *Load Server Architecture* is introduced in Section 5.2 to reduce the coordination overhead in Voyager.

4.7 Other Skeletons

4.7.1 `PIPE` Skeleton

The **PIPE** skeleton composes a list of functions together so that elements can be streamed through them[31]. The type of `PIPE` is:

$$\texttt{PIPE} :: [\alpha \rightarrow \alpha] \rightarrow (\alpha \rightarrow \alpha)$$

and the definition is:

`PIPE` = foldr (.)

4.7.2 `FARM` Skeleton

The **FARM** skeleton applies a function to each of a list of jobs. The function also takes an environment, which represents data which is common to all of the jobs[31]. The type of `FARM` is:

`FARM` :: $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow ([\beta] \rightarrow [\gamma])$

and the definition is:

`FARM` = map . (f env)

4.7.3 `DC` Skeleton

Many algorithms work by splitting a large task into several sub-tasks, solving the sub-tasks independently, and combining the results. This approach is known as divide-and-conquer and it is captured by the `DC` skeleton.

In `DC`, `t` presents trivial tasks, `s` presents tasks which have been solved. Larger tasks are divided by function `d` into sub-tasks, and the sub-tasks are passed to be solved recursively. The sub-results are then combined by function `c` to produce the main result[31]. The type of `DC` is:

`DC` :: $(\alpha \rightarrow Bool) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow [\alpha]) \rightarrow ([\beta] \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$

One of the implementation of `DC` is:

```
DC t s d c x | t x = s x
             | not ( t x ) = (c . map ( DC t s d c ) . d ) x
```

4.7.4 `RaMP` Skeleton

Another common class of algorithms describes systems where each object in the system can potentially interact with any other object. Each individual interaction is calculated and the results are combined to produce a result for each object. This is described by the `RaMP` skeleton ('Reduce-and-Map-over-Paris'). This skeleton is typically used for initial specification and implemented by transformation to an alternative form, for example by farming out the calculations for each object or by pipelining over the functions `f` and `g`[31]. The type of `FARM` is:

$$\texttt{RaMP} :: (\alpha \to \alpha \to \beta) \to (\beta \to \beta \to \beta) \to [\alpha] \to [\beta]$$

The definition of `RaMP` is:

$$\texttt{RaMP} \text{ f g xs} = \text{map h xs}$$

$$\text{where h x} = \text{foldr g (map (f x) xs)}$$

4.7.5 Discussion

The definition of `PIPE`, `FARM`, `RaMP` show that these three skeletons can be implemented using `map` or `fold`. So it would be easy to build autonomous versions of these three skeletons by using `automap` or `autofold`. The `DC` skeleton can be implemented using `map`, but `DC` performs irregular problems naturally. The autonomous mobility `DC` can not be implemented just using `automap` or `autofold`. It is difficult to predict the execution time for `DC`. So it is difficult to build cost models for `DC`.

4.8 Summary

AMSs have been proposed to encapsulate common patterns of self-aware mobile coordination aiming to minimise execution time in networks with dynamically changing loads. By analogy with other skeleton species, they hide low level mobile coordination details from users and provide higher level loci for designing load-aware mobile systems.

Abstract AMSs with concrete realisations for the common higher-order functions e.g. `map` and `fold` have been demonstrated. The realisations are provided both in the functional language context shared with other skeleton species, using Jocaml, and in an object-oriented context using mobile Javas. A novel `AutoIterator` skeleton has also been demonstrated for the widely used object-oriented `Iterator` interface.

AMS cost models are dynamic and substantially implicit. During the traversal of a collection, the skeleton implementation periodically measures the time to compute a single collection element, and uses the value to parameterise an implicit cost for the remainder of the traversal. The validations for the computation time, communication time, and coordination time of AMSs suggest that the problem specific cost model is suitable for the AMSs.

The experiments in this chapter also suggest that, for our set of test programs, AMSs can offer considerable savings in execution times, which scale well as overall execution times increase i.e. the AMS programs are faster than the static programs when there are faster locations in the network. Experiments in this chapter also show that single AMP can move from location to locations when the loads are changed in each location.

The mobility behaviour, the computation time, the communication time, and the coordination time of the AMSs in Jocaml and in Java Voyager are compared, as `automap` and `autofold` are developed in both languages. The mobility behaviour of Jocaml and Java Voyager are different, as Jocaml supports strong mobility, but Java Voyager supports weak mobility. The computation time in Jocaml is slower than that in Java Voyager, but the communication time in Jocaml is generally faster than in Java Voyager. The coordination time in Voyager is faster, which is an advantage, as we expect that the smaller the overhead the better the coordination information the programs will get. Chapter 5 will introduce a *Load Servers Architecture* to reduce the coordination overhead in Voyager and investigate load management using collections of AMPs on homogeneous and heterogeneous networks.

Chapter 5

Autonomous Load Management

Emergent phenomena occur when a collection of individuals interact without central control to produce results which are not explicitly programmed [2]. In AMPs, an emergent collective behaviour is that a collection of AMPs performs decentralised [137] dynamic load balancing. This chapter investigates *autonomous load management* using collections of AMPs on homogeneous and heterogeneous networks. The idea behind autonomous load management is that in dynamic networks an individual AMP determines when and where to move according to the status of the locations such as the CPU speed and load.

As the AMP architecture introduces a coordination overhead for collecting system information, and we expect that the smaller the overhead the better the coordination information the programs will obtain, this chapter introduces a *Load Server Architecture* to reduce the time of exchanging information between AMPs in the collections. The performance of collections of AMPs on homogeneous and heterogeneous networks are also presented, which shows that collections of AMPs quickly obtain and maintain a balance.

5.1 Introduction

5.1.1 Autonomous Load Management

The idea behind *autonomous load management* is that in dynamic networks individual AMPs determine when and where to move according to the status of their locations such as the CPU speed and load. Autonomous load management is dynamic load management. However it is different from traditional static and dynamic load management in the following ways.

- The application itself rather than a special process (load balancer) decides when and where to move.
- There is no central process to collect load information or make the decision to move.
- It may operate on a dynamic network i.e. both locations and AMPs can join or leave the network.

The aim of autonomous load management is different to traditional dynamic load management. Both dynamic and autonomous load management redistribute the processes from heavily loaded locations to lightly loaded ones based on the information collected at run-time. The goal of dynamic load management is *maximising utilisation of the processing power*[10]. Autonomous load management aims to minimise execution time of AMPs, which is similar to static load management. So, we can say this autonomous load management uses the dynamic method to solve a problem which dynamic load management cannot solve.

5.1.2 Autonomous Load Management Activities

The activities, which may happen during autonomous load management, are different from dynamic load management systems and are as follows.

- measure the currently workload;
- exchange load information between the currently available locations;

- check certain conditions e.g. relative CPU speed of locations for load imbalance, and decide whether to perform load management operation or not;
- make a load management decision about the location to which the current program should move;

The first three activities are the same as for dynamic load management [78] (Section 2.3), but the last activities are different. In dynamic load management the load balancer makes the decision of which task to move, but in autonomous load management the program decides itself whether to move or not and to which location.

5.1.3 Autonomous Load Management Policies

As autonomous load management is dynamic load management, we can also use the same dynamic load management policies e.g. push policy and the decentralized policy (Section 2.3) to control the activities during the execution of a program. Let us classify autonomous load management using the *information*, *transfer*, and *placement* load management policies from Section 2.3.1.

- *information policy:* specifies the amount of load information made available to job placement decision-makers. The information policy can be centralised or decentralised[97]. Because of the inherent feature of our AMPs that every program is the decision-maker we choose a decentralised information policy, in which every location keeps a copy of the system state. In this kind of policy the program can get information from local node, which take less time for coordination than from remote node.
- *transfer policy:* determines the conditions under which an AMP should be transferred. In the transfer policy AMPs perform a preemptive policy set by the cost models. According to the condition ($T_h > T_{comm} + T_n$) in the cost models in Section 3.3, AMPs can determine whether to move or not.
- *placement policy:* identifies the location to which a program should be transferred. As for the information policy, autonomous load management uses

a decentralised placement policy. The placement policy can either be a pull or push policy (Section 2.3.3). A push policy has been chosen in our autonomous mobile program, because the program sends itself to another location, rather than the idle location asking for jobs. The push policy makes the program simpler than the pull policy.

To achieve effective autonomous load management performance, the following three issues are important:

- to develop *cost models* for autonomous mobile computations that identify the costs of completing the computation at different location and of moving to a new location.
- to construct an effective implementation that minimizes the costs of coordination.
- to evaluate the load balancing performance on homogeneous and heterogeneous networks.

The generic and problem specific cost models for AMPs have been developed in Chapters 3 and 4. This chapter focuses on the second and third issues. The structure of this chapter is as follows. Section 5.2 introduces the load server architecture to reduce the coordination time in AMPs. Section 5.3 presents behaviour of collections of AMPs on homogeneous and heterogeneous networks. Finally, Section 5.4 summarises. Note in this chapter load management refers to load balancing in Chapter 2.

5.2 Load Server Architecture

In initial experiments, each AMP obtained load information from all other locations, but to reduce the coordination overhead we quickly introduced the *load server architecture* in Voyager. In the load server architecture, each location has a load server that maintains information about location loads. Specifically, the load server records CPU speed, the number of AMPs and the load of each location.

The advantages of the load server architecture are that it reduces the coordination time for AMPs, the time to discover load information, and also reduces network traffic.

5.2.1 Load Server and Non-Load Server System Structure

Figure 5.45 shows the system with load severs (LS), and Figure 5.46 shows the system without load servers (NLS). Without a load server an AMP must talk

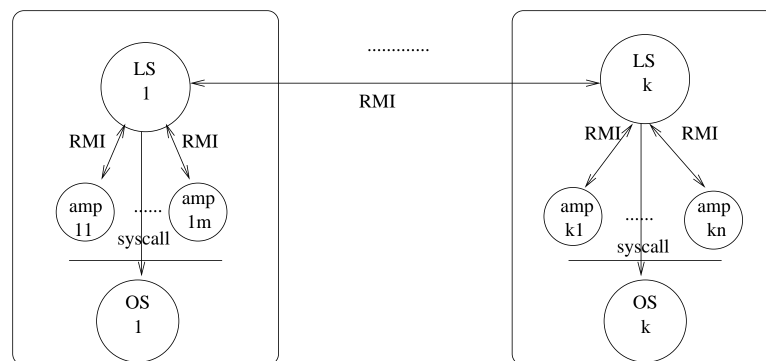


Figure 5.45: System with Load Server Architecture (LS)

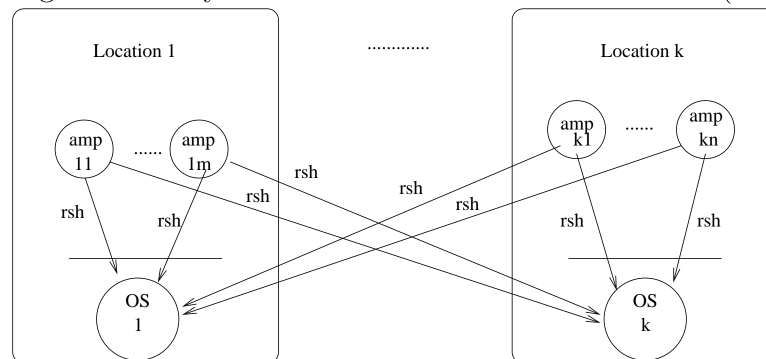


Figure 5.46: System without Load Server Architecture (NLS)

to each location by remote shell command “rsh”, which is very expensive. Using load servers, each load server just needs to detect the local information by talking to the operating system (OS) and acquire the information of other locations by Java RMI, which spends much less time than talking to a remote OS by “rsh”. AMPs are not in charge of collecting system information any more, and they can collect information from local load server, which saves much time compared to obtaining information by talking to each location. This architecture also reduces

the network traffic. For example, if there are two locations, Loc1 and Loc2, and each location has two AMPs: amp11 and amp12 on Loc1; amp21 and amp22 on Loc2. In the non-load server architecture, there are four communications between two locations (amp11 and amp12 talk to the OS on Loc2, and amp21 and amp22 talk to the OS on Loc1). In the load server architecture there is only two remote communications, which are the load servers on Loc1 and Loc2 talk to each other by Java RMI. So the load server architecture reduces the network traffic.

5.2.2 Information Renewal Time

It is important that AMPs obtain recent information such as relative CPU speed (*(CPU speed) *(100-loads)%*) of locations in the network. So the shorter the time to collect new information the fresher information AMPs will obtain. In the non-load server architecture the time for collecting new information is the coordination time, as each AMP obtains system information individually via remote system calls. According to the equation, $T_{Coord} = 0.25 * n * p \ seconds$ (see Section 4.2.7), where 0.25 is the time for a single check of one location, p is the number of processors, and n is the time to check the status of the processors, the time for checking information once of p locations is $T_{Coord} = 0.25 * p \ seconds$.

In the load server architecture, AMPs are separated for the information checking; load severs do this job instead. The time for load servers to collect the information of locations has two parts: One is the time for collecting local information via a system call to talk to the local OS, and the other is collecting remote information. For this the local load server does not need to talk with remote OSs, as it can exchange the information to remote load servers by Java RMI, which is much faster than remote system call.

Table 5.23 tests the time for a load server collecting information from different numbers of locations. The test is based on the LAN specified in Section 3.4.3. The table shows that if there is only one location in the network, the time for checking information is the time of the system call to get the local information. From the experiments the average system call time is 0.094 seconds. If there are

locations	1 (local)	2	3	4	5	10	15	20	25
Min	0.081	0.090	0.099	0.102	0.106	0.131	0.158	0.187	0.215
Max	0.125	0.124	0.175	0.189	0.172	0.179	0.192	0.241	0.277
Mean	**0.094**	0.010	0.121	0.127	0.137	0.161	0.167	0.203	0.248
Total RMI	0.000	0.005	0.027	0.033	0.043	0.067	0.073	0.109	0.154
Single RMI	–	0.005	0.014	0.011	0.011	0.007	0.005	0.006	0.006
Mean RMI	**0.008**								
T_{coord}	0.094+0.008*(p-1)								

Table 5.23: Load Server Information Collection Time

p ($p>1$) locations, the checking information time includes the time for checking local location information (a single system call time) and the time for checking the information of remote locations, which is the time of exchanging information time with other load servers by Java RMI. From experiments the single RMI time is 0.008 seconds. So in the p locations load server architecture, the time for checking information once is *0.094+0.008*(p-1)* seconds.

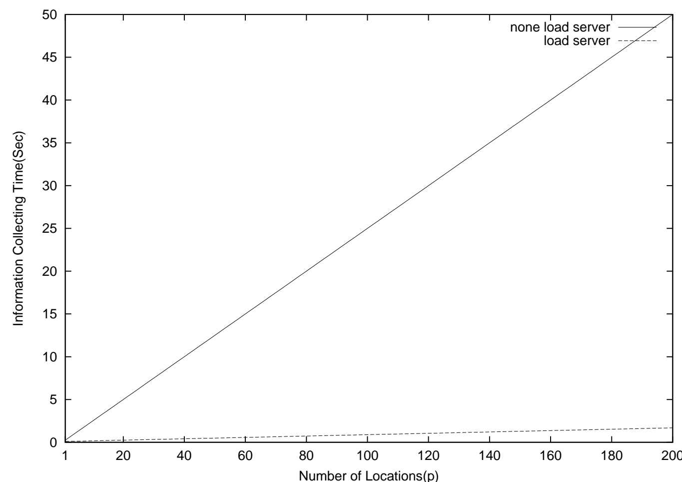


Figure 5.47: Load Server and Non-Load Server Information Collecting Time

Figure 5.47 compares the information collecting time in the load server and non-load server architectures, and shows the following.

- The information collecting time in the load server architecture is faster than in the non-load server architecture: *0.094+0.008 * (p-1)* seconds in load server architecture and *0.25 * p* seconds in the non-load server architecture.

- As the number of local server increases, LS is even faster than the NLS.

5.2.3 AMP Coordination Time

Locations	1	2	3	4	5	10	15	20	25
Min	0.008	0.009	0.007	0.008	0.009	0.009	0.009	0.009	0.008
Max	0.012	0.018	0.012	0.015	0.014	0.016	0.015	0.019	0.013
Mean	0.010	0.014	0.009	0.013	0.011	0.010	0.012	0.011	0.010
T_{coord}	0.011								

Table 5.24: Test AMPs Coordination Time in load Server Structure

Table 5.24 shows the AMP coordination time in the load server architecture. AMPs are separated from information collecting. So the coordination time is not the time for detecting information for the whole network as in non-load server architecture, which has been passed to load servers. The coordination time is the time for AMP to talk to the local load server. So T_{Coord} should be a constant rather than a variable like $T_{Coord} = 0.25 * n * p\ seconds$, in non-load server architecture (see Section 4.2.7).

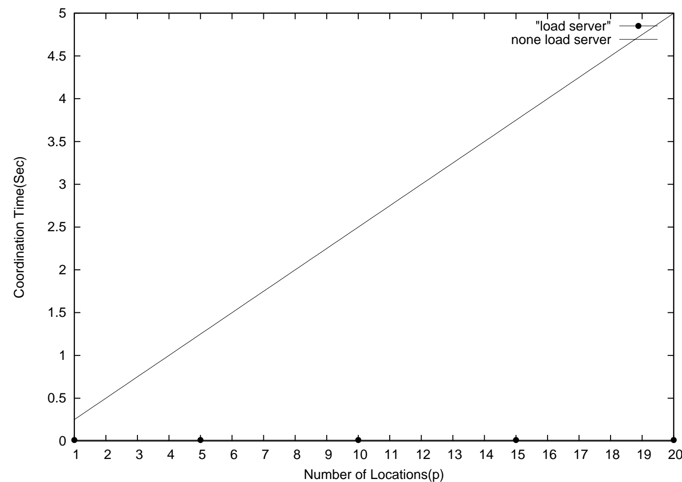


Figure 5.48: Load Server and Non-Load Server Coordination Time Comparison

Figure 5.48 compares the coordination in the load server and non-load server architectures. As we shall see, the coordination time in the load server architecture is a very small constant, here 0.011 seconds, but it increases as the number

of location increases in the non-load server architecture. So using load servers can reduce the AMP coordination time.

5.3 Collections of AMPs Measurements

Experiments for collections of AMPs on both homogeneous and heterogeneous networks are conducted.

5.3.1 Collections of AMPs on Homogeneous Networks

For homogeneous systems, we measured AMP behaviour on a dedicated network, where all locations have the same CPU speed (3193MHz) and communications latency i.e locations ($linux$01-$linux$28) in our local network.

Optimal Balances

We initially hypothesised that every location would have an equal number of AMPs, but experiments have shown that the *initiating location*, where the AMPs are started is more heavily loaded than the others, because of the communication with the remote process shipping from it, and hence has fewer AMPs. We define *optimal balance* as each location having the same number of AMPs, except the initiating location which may have fewer. For small numbers of AMPs the initiating location has just one AMP and other locations have $\frac{a-1}{p-1}$ AMPs, where a is the total number of AMPs, and p is the total number of locations.

AMPs	5 AMPs	7 AMPs	9 AMPs	10 AMPs	13 AMPs
3 Locs	1/2/2	1/3/3	1/4/4	-	-
4 Locs	-	1/2/2/2	-	1/3/3/3	1/4/4/4
5 Locs	-	-	22221	-	-

Table 5.25: Verified Optimal Balance

Our experiments show that collections of AMPs achieve optimal balance as predicted. In Table 5.25, the first row is the number of AMPs started, the first

column is the number of locations used, and the remaining columns summarise the distribution of AMPs on the locations with the initiating location listed first. For example, if we run seven AMPs on three locations and we start all seven AMPs on *Loc*1, then we expect that there will be three AMPs on both *Loc*2 and *Loc*3, but just one AMPs on *Loc*1.

For illustration, the movement of 7 AMPs between 3 locations is shown in Figure 5.49. The AMPs are started on *Loc*1 in time period 0. Four AMPs move to *Loc*2 and two move to *Loc*3 in time period 1, which is not an optimal balance, so one AMP in *Loc*2 moves to *Loc*3 in time period 2. After this move the system achieves an optimal balance and the AMPs do not move again. We have

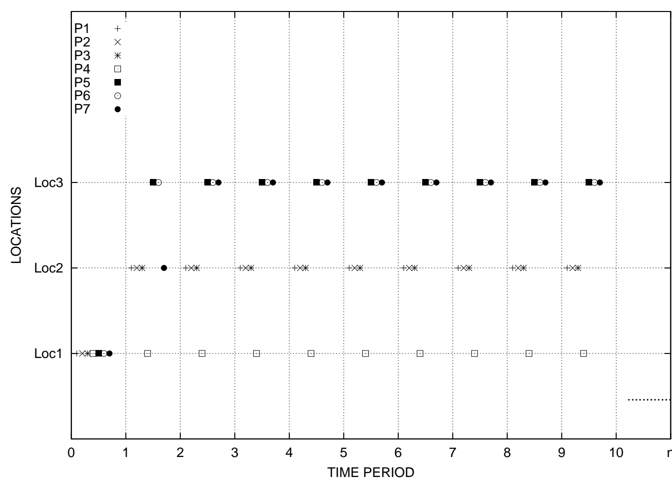


Figure 5.49: Distribution of 7 AMPs on 3 Locations

also made experiments with five AMPs on three location, nine AMPs on three locations, seven AMPs on four locations, ten AMPs on four locations, thirteen AMPs on four locations, nine AMPs on five locations and achieve similar results. Figures 5.50, 5.51, 5.52, 5.53, 5.54 and 5.55 show our test results.

Near Optimal Balances

Near-optimal balance is when each location except the initiating location may not have the same number of AMPs, but the number of AMPs on each location differs by no more than one. In Figure 5.56, six AMPs are started on *Loc*1. Three of them move to *Loc*3, two of them move to *Loc*2, one stays on *Loc*1, and the AMPs

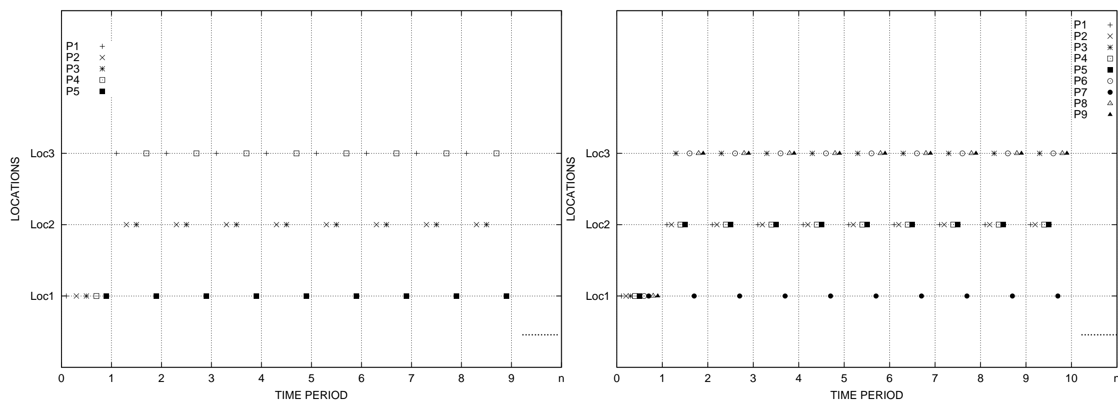


Figure 5.50: Distribution of 5 AMPs on 3 Locations

Figure 5.51: Distribution of 9 AMPs on 3 Locations

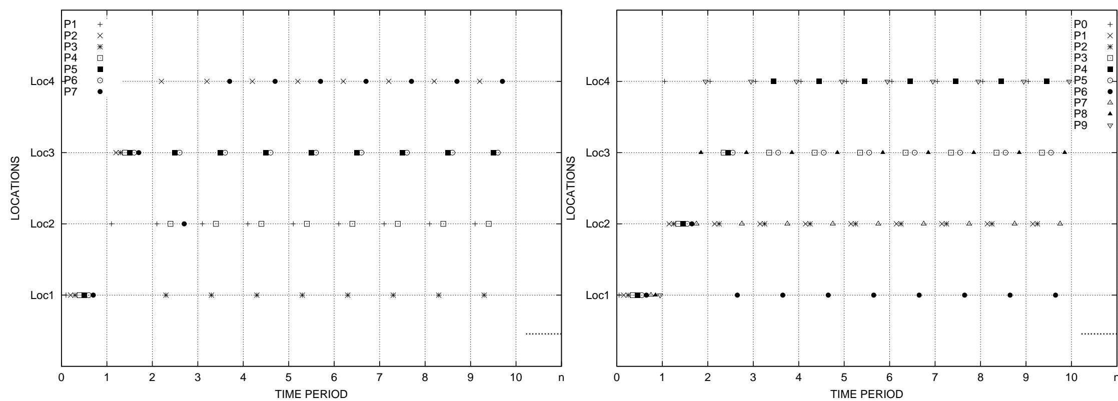


Figure 5.52: Distribution of 7 AMPs on 4 Locations

Figure 5.53: Distribution of 10 AMPs on 4 Locations

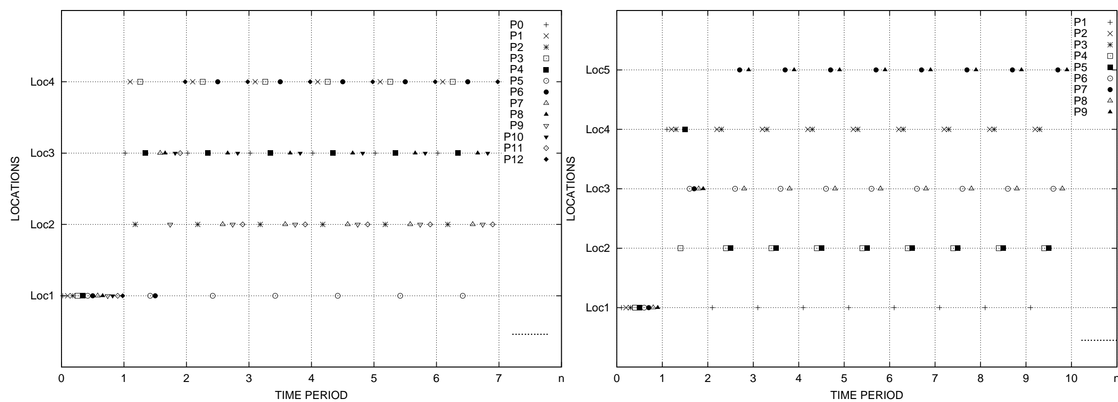


Figure 5.54: Distribution of 13 AMPs on 4 Locations

Figure 5.55: Distribution of 9 AMPs on 5 Location

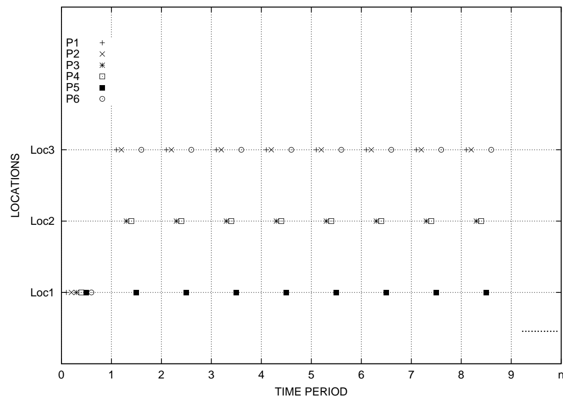


Figure 5.56: Distribution of 6 AMPs on 3 Locations

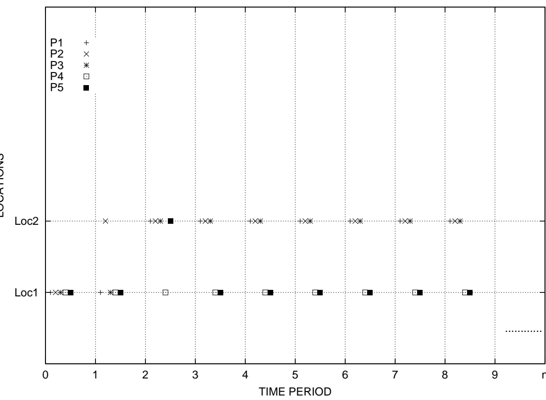


Figure 5.57: Distribution of 5 AMPs on 2 Locations

do not move again after the best distribution. We have also made experiments with five AMPs on two locations and get similar results, shown in Figure 5.57.

Adding More AMPs

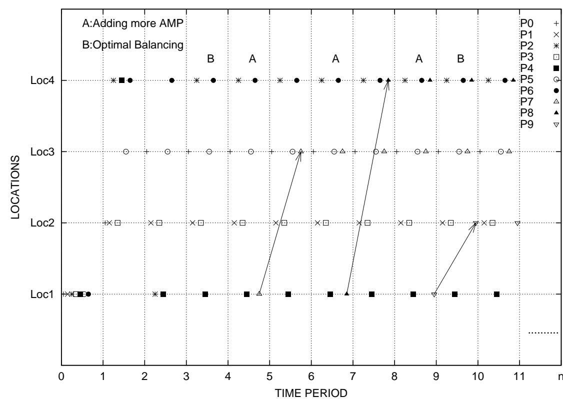


Figure 5.58: Rebalancing: 7 AMPs Adding 3 More AMPs on 4 Locations

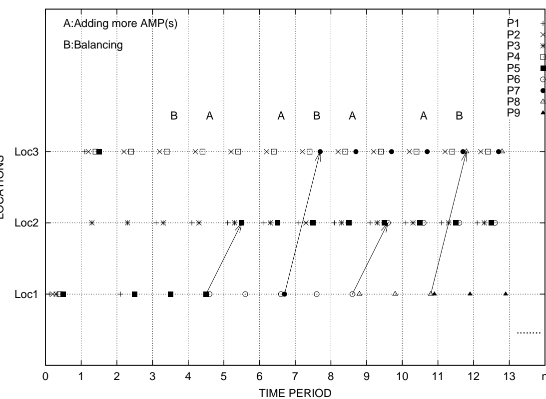


Figure 5.59: Rebalancing: 5 AMPs Adding 4 More AMPs on 3 Locations

Figure 5.58 shows that if we add AMPs to a balanced AMPs system, the AMPs can rebalance themselves. Seven AMPs are initially started on *Loc*1. Once the system is balanced, one AMP is started in each time period of 4, 6 and 8, and the system achieves balance in time period 9. Figure 5.59 shows that we start five AMPs on *Loc*1, after the system is balanced, we start four more AMPs, then the system achieves a new balance in time period 12.

Removing some AMPs

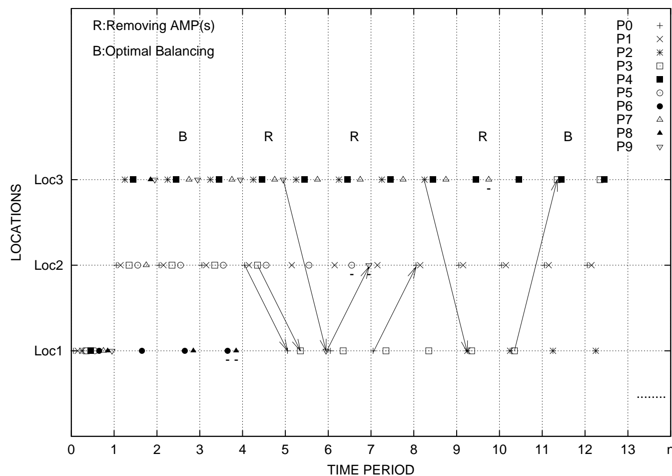


Figure 5.60: Rebalancing: 10 AMPs Removing 5 AMPs on 3 Locations

The rebalancing as AMPs are removed follows a similar patten to adding AMPs. In Figure 5.60, firstly we start five big AMPs and five small AMPs on *Loc*1, and they become balanced. Then the five small AMPs finish one by one, so the balance is broken, but after a few moves the system achieves a new balance again in time period 11.

5.3.2 Collections of AMP on Heterogeneous Network

For testing the behaviour of multiple AMPs on a heterogeneous network we have made two experiments. One is the behaviour of 25 AMPs on a heterogeneous network of 15 locations, with CPU speeds 3193MHz (Loc1-Loc5), 2168MHZ (Loc6-Loc10), and 1793MHz (Loc11-Loc15). The other is the behaviour of 20 AMPs on 10 locations with CPU speeds 3139MHz (Loc1-Loc5), 2167MHZ (Loc6), 1793MHz (Loc7-Loc10).

For illustration, the movement of 25 AMPs between the 15 locations is shown in Figure 5.61, where "B" is the *balanced state*. In the balanced state if any AMP move to any other location it cannot get as much *average relative CPU speed* (*(CPU speed) *(100-loads)% / (Number of AMP)*) as at the current location. In this state, AMPs will stay in the current locations and not move any more until

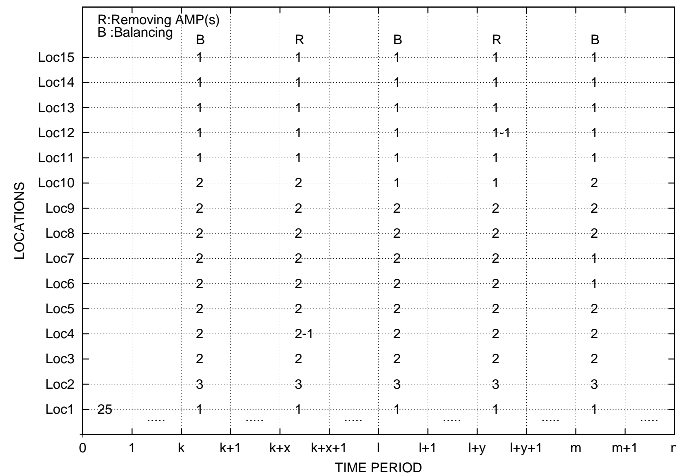


Figure 5.61: Distribution of 25 AMPs on Heterogeneous Network (15 Locations)

the balance is broken. We start 25 AMPs on Loc1 in time period “0”. After some movements of each AMP, the system achieves a balance in time period “k” and the AMPs maintain the balance and do not move any more until time period “k+x”, when one of the AMPs is finished and the balance is broken. So the 24 AMPs move again and reach a new balance in time period “l”, and once again 23 AMPs reach balanced state in time period “m”.

Table 5.26 shows the states of the 15 locations, when 25 AMPs reach balance. In the balanced state, every AMP has the maximum average relative CPU speed in the network, and the *initiating location* (*Loc*1) has fewer AMPs, as where the AMPs are started is more heavily loaded than the others, because of the communication with the remote process shipping from it. When the 25 AMPs achieve the balanced state, 50% CPU of Loc1 has been used for communication rather than for AMP execution, and there is 1 AMP on Loc1 (3193MHz), so the AMP on this location can get 1597MHz CPU speed (CPU * (100%-load%)/number of AMP). There are 3 AMPs on Loc2 (3193MHz), so each AMP on this location can get 1064MHz CPU, and there is 1 on Loc11 (1793MHz) with no other loads so the AMP gets 1793MHz CPU. If one of the AMPs on Loc2 is going to move to Loc11, there will be 2 AMPs on Loc7, and each AMP will get less than 897MHz (1793MHz/2) CPU, which is less than the current CPU(1064MHz). The other AMPs are in the same state.

Total AMPs	Location	CPU speed s	Number of AMPs at each location n	Other Loads (CPU%) l	Relative Speed Each AMP had r=s*(100-l)%/n
25 AMPs	Loc1	3193MHz	1	50	1597MHz
	Loc2	3193MHz	3	0	1064MHz
	Loc3-Loc5	3193MHz	2	0	1597MHz
	Loc6-Loc10	3193MHz	2	0	1597MHz
	Loc11-Loc15	1793MHz	1	0	1793MHz

Table 5.26: Prediction CPU Speed for Each AMPs (25 AMPs on 15 Locations)

Total AMPs	Location	CPU speed s	Number of AMPs at each location n	Other Loads (CPU%) l	Relative Speed Each AMP had r=s*(100-l)%/n
24 AMPs	Loc1	3193MHz	1	49	1628MHz
	Loc2	3193MHz	3	0	1064MHz
	Loc3-Loc5	3193MHz	2	0	1597MHz
	Loc6-Loc9	3193MHz	2	0	1597MHz
	Loc10	2167MHz	1	0	2167MHz
	Loc11-Loc15	1793MHz	1	0	1793MHz

Table 5.27: Prediction CPU Speed for Each AMPs (24 AMPs on 15 Locations)

Total AMPs	Location	CPU speed s	Number of AMPs at each location n	Other Loads (CPU%) l	Relative Speed Each AMP had r=s*(100-l)%/n
23 AMPs	Loc1	3193MHz	1	48	1660MHz
	Loc2	3193MHz	3	0	1064MHz
	Loc3-Loc5	3193MHz	2	0	1597MHz
	Loc6-Loc7	2167MHz	1	0	2167MHz
	Loc8-Loc10	2167MHz	2	0	1084MHz
	Loc11-Loc15	1793MHz	1	0	1793MHz

Table 5.28: Prediction CPU Speed for Each AMPs (23 AMPs on 15 Locations)

The last column in Tables 5.26 shows the maximum relative CPU speed each AMP obtains, but the AMPs may not use all these CPU speeds. Figures 5.62 shows the *actual relative CPU speed* (*(CPU speed) * loads% / (Number of AMP)*) each AMP uses at balance state. The figure shows that when 25 AMPs on 15 locations achieve balanced states every AMP has effective resource between 150MHZ and 400MHZ with only three exceptions. So every AMP has similar *average relative CPU speed* at the balance state. Similar results were found when some AMPs are finished and the system achieves new balance states. Tables 5.27 and 5.28 show the states of the locations at the new balance states of 24 and 23 AMPs, and Figures 5.63 and 5.64 show the *actual relative CPU speed* for 24 and 23 AMPs.

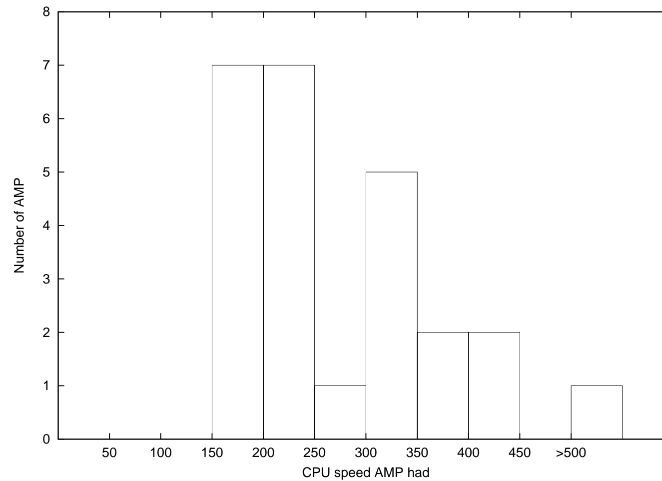


Figure 5.62: Actual CPU Speed for Each AMP (25 AMPs on 15 Locations)

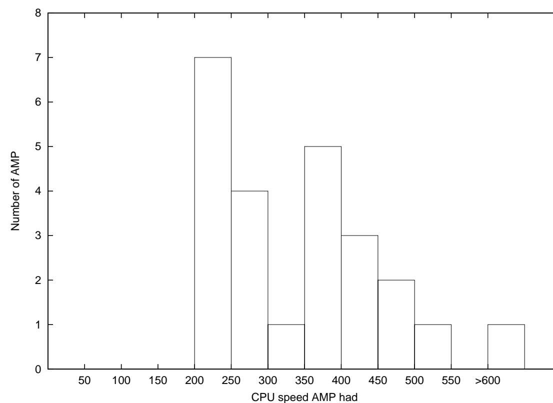


Figure 5.63: Actual CPU Speed for Each AMP (24 AMPs on 15 Locations)

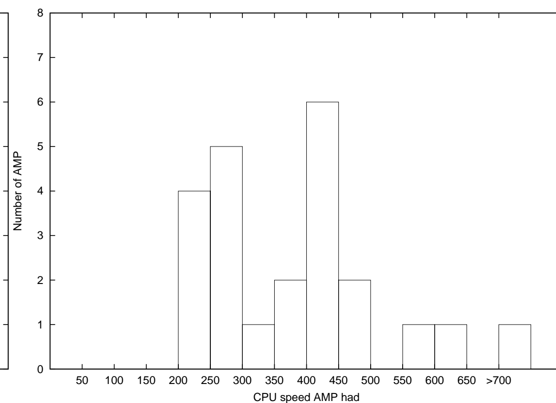


Figure 5.64: Actual CPU Speed for Each AMP (23 AMPs on 15 Locations)

The behaviours of 20 AMP programs on 10 locations with CPU speeds 3139MHz (Loc1-Loc5), 2167MHZ (Loc6), 1793MHz (Loc7-Loc10) are also measured, and get the similar results to the first experiment. Figure 5.65 shows the balanced state for 20 AMPs and the rebalanced states for 19 and 18 AMPs on 10 locations.

Tables C.29, C.30, and C.31 in Appendix C show the location information at the balance state when there are 20, 19, or 18 AMPs on the 10 locations. Figures C.106, C.107, and C.108 show the actual relative CPU speed available to 20, 19 and 18 AMPs.

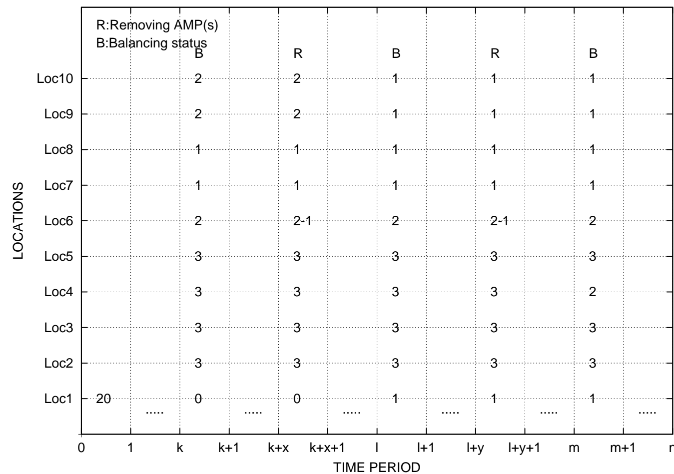


Figure 5.65: Distribution of 20 AMPs on Heterogeneous Network (10 Locations)

5.4 Summary

The experiments in this chapter compare the coordination times of AMPs with the load server architecture to that with the non-load server architecture. The results suggest that by introducing the load server architecture, the coordination time for AMPs are reduced, so the AMPs can obtain fresher information than in the non-load server architecture. The load server also reduces network traffic.

Experiments on a homogeneous and heterogeneous network show that collections of AMPs quickly obtain and maintain optimal or near-optimal balance. The results also show that if the optimal or near-optimal balance is broken by adding in or removing AMPs, collections of independent AMPs rebalance quickly with a small number of moves.

In a homogeneous system, if the ratio of AMPs to locations is ideal, an optimal balance is relatively quickly obtained with every location, except the initiating location, hosting the same number of AMPs. Similarly, in a homogeneous system, even if the ratio of AMPs to locations is not ideal, an near-optimal distribution can be obtained. Furthermore, the system maintains balance as AMPs are added or removed. In a heterogeneous system, AMPs can achieve balance with similar average relative CPU speeds.

The last three chapters introduced the concepts of AMPs which make decentralised decisions about where to execute, and AMSs which encapsulate common

patterns of self-aware mobile coordination. However problem specific cost models need to be developed by hand and the cost model assumes that only one higher-order function is the dominating computation for the program. To solve this problem a cost calculus will be introduced in Chapter 6, which can produce the cost of the entire program.

Chapter 6

Automatic Continuation Cost Analysis

Autonomous mobility skeletons and associated cost models are proposed and constructed in Chapter 4. These make the assumption that a single higher-order function is the dominating computation for the program, because the model in Chapter 4 only considers the cost of the higher-order function and not the remaining cost of the the program. For example, if there are multiple higher-order functions in the program, e.g. in program `(automap f1 l1); (automap f2 l2)`, `automap f2 l2` is the remainder of `automap f1 l1` in the program. When `automap f1 l1` make the decision to move or not, it might not move if it only considers the cost of itself, but it might move if it also considers the cost of `automap f2 l1`. So the main question is how to calculate the costs of the reminder of the higher-order functions in the program.

This chapter explores a calculus to manipulate, and ultimately automatically statically extract the cost of the remainder (*costafter*) of a program, at arbitrary points. The remainder is the *continuation* [50] in a program, which is a representation of the execution state of a program e.g. the call stack or values of variables at a certain point[130]. The advantage of the cost calculus is that it is not necessary to provide a closed form solution as environmental information for a computation is always available implicitly at run-time.

An automatic continuation cost analyser, which implements the cost calculus, is built to produce cost equations parametrised on program variables in context, and may be used to find both costs in higher order functions and the costafter of the higher order functions i.e. the cost of the continuation.

We extend our AMS cost model to be parametrised on the costafter of the skeletons, and improved AMSs i.e. costed autonomous mobility skeletons (CAMSs) have been built based on the extended cost model. CAMSs e.g. `camap` and `cafold`, not only encapsulate the common pattern of autonomous mobility like AMSs in Chapter 4, but take two additional cost parameters: the cost of the remainder and the cost in the higher-order function. The possible performance improvements are assessed by comparing CAMS to AMS programs. The results show that, CAMS programs perform more effectively than AMS programs, because they have more accurate cost information. Hence a CAMS program may move to a faster location when the corresponding AMS program does not.

6.1 Introduction

To calculate the costafter of an arbitrary point in a program, the cost of every expression must be calculated. To illustrate the concept this section gives a small language, $e ::= n \mid e{+}e$, where n is integer. Using expression `2+3` as an example, the costafter of `2` can be calculated as $\frac{\mathrm{E} \vdash_c 3 \,\$\, c_3 \qquad \mathrm{E} \vdash_c + \,\$\, c_+}{\mathrm{E} \vdash_a 2 \trianglelefteq (2{+}3) \,\pounds\, c_3{+}c_+}$, where c_3 is the cost of `3`, c_+ is the cost of "`+`" and the judgement is that the costafter of `2` is $c_3{+}c_+$. So to calculate the costafter of `2` the cost of `3` and the cost of "`+`" need to be calculated. The semantic functions $\vdash_c$ and $\vdash_a$ produce the cost and costafter of expression respectively in the environment E, which will be introduced in Section 6.3.

The problem with this calculation is if there are two similar expressions or one expression in two or more different place in the program, it is difficult to identify the expression whose costafter needs to be calculated. For example, in expression `10+10`, there are two `10`s. The question is to calculate the costafter of `10` in `10+10`, but it is difficult to decide which `10` is required, and the costafters of these two `10`s are different. To solve this problem, every expression in a program

is given a unique number, which is called its *index*. The process of giving these indices is called indexing. After indexing, the expression `10+10` becomes $< 3, < 1, 10 > + < 2, 10 >>$. The two `10`s become $< 1, 10 >$ and $< 2, 10 >$, which are different to each other. The general three stages to calculate the costafter are:

- To index the program.
- To calculate the cost of expressions in a program.
- To calculate the costafter of the point required.

To calculate the cost of an expression is standard and uses techniques similar to cost models in [80, 103, 124]. The third stage, to calculate the costafter, is novel, which is to predict a continuation cost in a program.

The structure of this chapter is as follows. Section 6.2 defines a small strict higher-order language $\mathcal{J}'$. Section 6.3 builds the cost calculus for language $\mathcal{J}'$, which includes *indexing* the program (Subsection 6.3.1), *calculating the costs* of every expressions (Subsection 6.3.2), *calculating the costafter* of the given expression (Subsection 6.3.3). Section 6.4 builds the cost model for CAMSs, and implements CAMSs. Section 6.5 compares the performance of the programs using CAMSs with the programs AMSs. Section 6.6 describes the implementation of an automatic continuation cost analyser which implements the cost calculus. This section also compares the performances of analyser produced CAMSs to the hand analysed CAMSs in Section 6.5. Section 6.7 summaries. Note that in this chapter higher-order functions are AMSs e.g. `automap`, and `autofold` or CAMSs e.g. `camap` or `cafold`, according to the context.

6.2 Syntax of Language $\mathcal{J}'$

A cost semantics has been built for language $\mathcal{J}$, which is a subset of Jocaml. $\mathcal{J}$ is a core functional language and readily able to describe non-trivial programs like matrix multiplication and ray tracing. The full syntax and cost calculus of $\mathcal{J}$ is presented in Appendix A. To explain the principles, this section introduces

a simple language $\mathcal{J}'$, a subset of $\mathcal{J}$, and Section 6.3 uses $\mathcal{J}'$ to introduce the cost calculus. Figure 6.66 shows the abstract syntax of $\mathcal{J}'$. To simplify the

e ::=		expression
	$\mid$ k	constant
	$\mid$ v	variable
	$\mid$ `fun` $v \rightarrow e$	lambda (abstract)
	$\mid$ $e\ e$	application
	$\mid$ $e\ op\ e$	operation
	$\mid$ `map` $e\ e$	`map` (higher order function)
	$\mid$ e `(*` e `*)`	user cost
	$\mid$ $< n, e >$	index i.e. expression e has index n

op ::= + | - | * | / |
> | < | >= | <= | = | ! = |
:: (cons) |
; (sequence)

Figure 6.66: Syntax of $\mathcal{J}'$

presentation it is assumed that variable (v) names in the program are unique. Note that $\mathcal{J}$ also contains `fold` higher-order function.

$\mathcal{J}'$ is a core lambda calculus with two unusual expression, the *index* expression and *user cost* expression. The *index* expression is presented because the whole program has been indexed, so every expression in a program has a unique integer as an index to avoid repetitive costafter of one expression in different locations. The semantics of index are explained in Section 6.3.1. Costing recursive functions is undecidable. Thus to deal with the cost of recursive functions, *user costs* are introduced into language $\mathcal{J}'$, which will be explained in Section 6.3.2.

6.3 Cost Calculus for $\mathcal{J}'$

The cost calculus includes three parts. The first part indexes the abstract syntax tree (AST) to produce *indexed abstract syntax tree* (IAST), i.e. gives every expression a unique integer as an index (Section 6.3.1). The second part calculates the cost of each expression (Section 6.3.2). The third part calculates the costafter of each expression (Section 6.3.3). Figure 6.67 shows the semantic functions which

$\vdash_i \,:\, n \rightarrow e \rightarrow e * n$	index
$\vdash_c \,:\, env \rightarrow e \rightarrow e * cost$	expression cost
$\equiv \,:\, e \rightarrow e \rightarrow boolean$	expression equality
$\in \,:\, e \rightarrow e \rightarrow boolean$	expression contains
$\vdash_a \,:\, env \rightarrow e \rightarrow e \rightarrow cost$	costafter

Figure 6.67: Semantic Functions

will be used in the calculus to present semantics of index, cost, costafter, etc. The meaning of these functions will be given in the sections where the functions are used. In these semantic functions, *env* is a semantic domain, which is the environment used in the calculus, and is presented as E. The type of *env* is, *env* : $(v * cost)^*$, and *cost* is an integer.

6.3.1 Index Semantics for $\mathcal{J}'$

In the cost calculus, every expression in a program has a unique integer as an index to avoid repeatedly calculating the costafter of same expression in a program. For example, in expression `((a+b)+(a+c))` expression `a` occurs twice in the expression, so it is difficult to identify the costafter of which `a` should be calculated. To do so, the original abstract syntax trees for programs are decorated to make every expressions are unique in the program.

Figure 6.68 shows the semantics of index, where $\vdash_i$ is a semantic function, which takes two parameters, the expression and an integer(i), which is the current index of the expression, and returns a tuple of index expression and another integer which is i increased by 1. The type of this function is: $\vdash_i \,:\, n \rightarrow e \rightarrow e * n$, where n is integer, e is expression.

- Equation (6.37) shows if the current index is i the expression is a constant k, after indexing the new expression is $< i, k >$ and the current index becomes $i + 1$.

- Equation (6.38) is similar to Equation (6.37), just the expression is a variable (v) instead of constant.

$$\frac{}{i \vdash_i k \Rightarrow_i (< i, k >, i+1)} \tag{6.37}$$

$$\frac{}{i \vdash_i v \Rightarrow_i (< i, x >, i+1)} \tag{6.38}$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i')}{i \vdash_i \texttt{fun}\ x \ \rightarrow \ e \Rightarrow_i (< i', \texttt{fun}\ x \ \rightarrow \ e' >, i'+1)} \tag{6.39}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i (e_1\ e_2) \Rightarrow_i (< i'', (e_1'\ e_2') >, i''+1)} \tag{6.40}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i e_1\ op\ e_2 \Rightarrow_i (< i'', e_1'\ op\ e_2' >, i''+1)} \tag{6.41}$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i')}{i \vdash_i e\ (*\ c\ *) \Rightarrow_i (< i', e'\ (*\ c\ *) >, i'+1)} \tag{6.42}$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i \texttt{map}\ e_1\ e_2 \Rightarrow_i (< i'', \texttt{map}\ e_1'\ e_2' >, i''+1)} \tag{6.43}$$

$$\frac{}{i \vdash_i < i', e > \Rightarrow_i (< i', e >, i)} \tag{6.44}$$

Figure 6.68: Index Semantics for $\mathcal{J}'$

- Equation (6.39) decorates lambda expressions. In this equation the dummy does not need to be decorated, only the body of the lambda expression does. For indexing $\texttt{fun}\ x \ \rightarrow \ e$, if the current index is i, and after indexing e the current index is i', and e becomes e', the indexed lambda expression is $< i', \texttt{fun}\ x \ \rightarrow \ e' >$ and the current index is $i'+1$.

- Equation (6.40) decorates application expressions $(e_1\ e_2)$. If the current index is i after indexing e_1, the current index becomes i' and e_1 becomes e_1', and then after indexing e_2, the current index is i'' and e_2 becomes e_2', then the index for the application expression is i'' and the current index becomes $i''+1$.

- Equation (6.41) decorates operation expressions using the same rules as in Equation (6.40).

- In Equation (6.42), user cost expressions $e\ (*\ c\ *)$ are indexed. In this expression only e, the expression part, has been indexed but not the cost part $(*\ c\ *)$.

- In Equation (6.43), `map` expressions are indexed, which is similar to Equation (6.40).

- If the expression is an index expression, it should not be indexed again. So in Equation (6.44), the return expression is the index expression, and the current index number is still i.

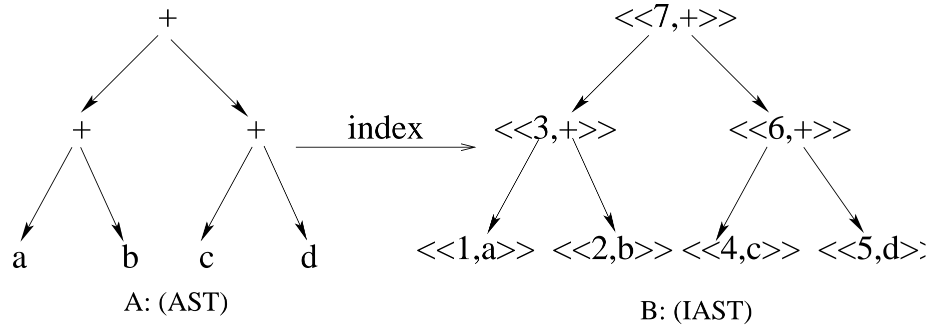


Figure 6.69: An Example of Index in $\mathcal{J}'$

Figure 6.69 shows an example of indexing an AST. In the figure, tree A is the original abstract syntax tree for expression e, `((a+b)+(c+d))`, and tree B is the indexed abstract syntax tree. In tree B, the index of node `a` is `1`, the index of node `b` is `2`, the index of node `(a+b)` is `3`, and the index of the entire expression `((a+b)+(c+d))` is `7`. The indexed expression for e is e_i: `<7,(<3,(<1,a>+<2,b>)> +<6,(<4,c>+<5,d>)>)>`. The result from the function in Section 6.6.2, which implements the index semantics, is the same.

6.3.2 Cost Semantics for $\mathcal{J}'$

Two kinds of cost model have been introduced in Section 2.4.1. *Computation cost models* estimate the sequential computation time for programs. *Coordination cost models* predict the coordination and communication behaviours of parallel, distributed and/or mobile programs. Usually, coordination cost models take costs which have been calculated from the computation cost models into account to make more efficient coordination decisions. The coordination cost models for AMPs and AMSs have been built in Sections 3.4.2 and 4.2.1. This section introduces static computation cost models.

Many cost models use semantics-based methods e.g [80, 101]. We build a static computation cost model i.e. cost semantics for language $\mathcal{J}'$ to calculate the cost of each expression in the program. The cost semantics is based on the cost semantics in [80] and simplifies this work. The cost semantics for $\mathcal{J}'$ does not consider the type or size system of the language.

Figure 6.70 shows the cost semantics of Language $\mathcal{J}'$. Semantic function $\vdash_c$ takes the environment (*env*) and an expression (e), and returns a tuple of the expression and the cost (*cost*) of the expression under the environment. The type of this function is: $\vdash_c$: $env \to e \to e * cost$.

$$\frac{}{\mathrm{E} \vdash_c k \,\$\, 0} \tag{6.45}$$

$$\frac{}{\{v, c\} + \mathrm{E} \vdash_c v \,\$\, c + 1} \tag{6.46}$$

$$\frac{\mathrm{E} \vdash_c e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_c e_1 \; op \; e_2 \,\$\, 1 + c_1 + c_2} \tag{6.47}$$

$$\frac{\mathrm{E} \vdash_c e \,\$\, c}{\mathrm{E} \vdash_c \texttt{fun}\; x \;\to\; e \,\$\, c} \tag{6.48}$$

$$\frac{\mathrm{E} \vdash_c e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_c (e_1 \; e_2) \,\$\, c_1 + c_2} \tag{6.49}$$

$$\frac{}{\mathrm{E} \vdash_c e \; (* \; c \; *) \,\$\, c} \tag{6.50}$$

$$\frac{\mathrm{E} \vdash_c e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \texttt{map}\; e_1 \; e_2 \,\$\, c_1 * (length \; e_2) + c_2} \tag{6.51}$$

$$\frac{\mathrm{E} \vdash_c e \,\$\, c}{\mathrm{E} \vdash_c < i, e > \,\$\, c} \tag{6.52}$$

Figure 6.70: Cost Semantics for $\mathcal{J}'$

- Equation (6.45) infers the cost of an constant is 0 in environment E.
- Equation (6.46) shows that the cost of the value of variable , here c, has been stored in the environment. If we try to calculate the cost of a variable, we need to look up the environment $\{v, c\} + \mathrm{E}$, and the total cost of variable is the cost to access the variable and the cost of the value of the variable, so the total cost is $c + 1$.

$$
\begin{array}{lll}
& \text{cost of} < 7, (< 3, (< 1, a > + < 2, b >) > + < 6, (< 4, c > + < 5, d >) >) > & \\
\Rightarrow & \text{cost of} (< 3, (< 1, a > + < 2, b >) > + < 6, (< 4, c > + < 5, d >) >) & (6.52) \\
\Rightarrow & 1\text{+cost of} < 3, (< 1, a > + < 2, b >) > \text{+cost of} < 6, (< 4, c > + < 5, d >) > & (6.47) \\
\Rightarrow & 1\text{+cost of} (< 1, a > + < 2, b >)\text{+cost of} < 6, (< 4, c > + < 5, d >) > & (6.52) \\
\Rightarrow & 1\text{+}(1\text{+cost of} < 1, a >\text{+cost of} < 2, b >)\text{+cost of} < 6, (< 4, c > + < 5, d >) > & (6.47) \\
\Rightarrow & 1\text{+}(1\text{+cost of } a\text{+cost of} < 2, b >)\text{+cost of} < 6, (< 4, c > + < 5, d >) > & (6.52) \\
\Rightarrow & 1\text{+}(1\text{+}(1\text{+}c_a)\text{+cost of} < 2, b >)\text{+cost of} < 6, (< 4, c > + < 5, d >) > & (6.46) \\
\Rightarrow & 1\text{+}(1\text{+}(1\text{+}c_a)\text{+cost of } b)\text{+cost of} < 6, (< 4, c > + < 5, d >) > & (6.52) \\
\Rightarrow & 1\text{+}(1\text{+}(1\text{+}c_a)\text{+}(1\text{+}c_b))\text{+cost of} < 6, (< 4, c > + < 5, d >) > & (6.46) \\
\Rightarrow & 1\text{+}(1\text{+}(1\text{+}c_a)\text{+}(1\text{+}c_b))\text{+cost of} (< 4, c > + < 5, d >) & (6.52) \\
\Rightarrow & 1\text{+}(1\text{+}(1\text{+}c_a)\text{+}(1\text{+}c_b))\text{+}(1\text{+cost of} < 4, c >\text{+cost of} < 5, d >) & (6.47) \\
\Rightarrow & 1\text{+}(1\text{+}(1\text{+}c_a)\text{+}(1\text{+}c_b))\text{+}(1\text{+cost of } c\text{+cost of} < 5, d >) & (6.52) \\
\Rightarrow & 1\text{+}(1\text{+}(1\text{+}c_a)\text{+}(1\text{+}c_b))\text{+}(1\text{+}(1\text{+}c_c)\text{+cost of} < 5, d >) & (6.46) \\
\Rightarrow & 1\text{+}(1\text{+}(1\text{+}c_a)\text{+}(1\text{+}c_b))\text{+}(1\text{+}(1\text{+}c_c)\text{+cost of } d) & (6.52) \\
\Rightarrow & 1\text{+}(1\text{+}(1\text{+}c_a)\text{+}(1\text{+}c_b))\text{+}(1\text{+}(1\text{+}c_c)\text{+}(1\text{+}c_d)) & (6.46)
\end{array}
$$

Figure 6.71: An Example of Costing in $\mathcal{J}'$

- Equation (6.47) performs the cost of operation expressions (e_1 *op* e_2). If the cost of e_1 is c_1, and the cost of e_2 is c_2 then the cost of e_1 *op* e_2 is $1 + c_1 + c_2$, where 1 is the cost for getting the operator.

- Equation (6.48) performs the cost of lambda expressions `fun` $x \rightarrow e$, which is the the cost of the body e.

- Equation (6.49) shows that the cost of application expression ($e_1 \;\; e_2$) is the cost of e_1 (c_1) plus the cost of e_2 (c_2).

- Equation (6.50) enables the user cost to provide in particular for a recursive functions. It is difficult to calculate the static cost of recursive functions. So in e $(* \; c \; *)$, the cost of e is c which is calculated by hand rather by the cost analyser automatically. The cost is passed to the cost analyser. The cost analyser will take c as the cost of e in the costing process.

- Equation (6.51) infers the cost of `map` expression under environment E. Here only the regular cases of map are considered. So under this condition function e_1 applying on every elements of list e_2 has the same cost. So the total cost of `map` expression is $c_1 * (length \; e_2) + c_2$, where c_1 is the cost of function e_1 applying on the first element of list e_2.

- Equation (6.52) shows that under environment E the cost of index expression $< i, e >$ is the cost of expression e (c).

Figure 6.71 shows how the indexed AST from Figure 6.69 is costed. The cost of expression `((a+b)+(c+d))` from the program, which implements the cost semantics of $\mathcal{J}'$, is `(1+((1+((1+0)+(1+0)))+(1+((1+0)+(1+0)))))`, where the four `(1+0)`s are the cost of `a`, `b`, `c`, and `d`, because in the program `a`, `b`, `c`, and `d` are integers and the cost of integer is 0. So the result is the same as that calculated by hand in Figure 6.71. The program is in Section 6.6.2.

6.3.3 Costafter Semantics for $\mathcal{J}'$

Section 6.3.2 explains the cost semantics of $\mathcal{J}'$, but calculating the costs of expression is not the object of this work; the object is to calculate the costafter of the expressions in program. The costafter is the cost of the remainder i.e. the continuation of the program.

There are two approaches to calculate the cost of the continuations in a program. One is translating the direct program into continuation passing style (CPS) [47] to obtain the continuation of the current expression, then calculating the cost of the continuation. Another is the approach which is presented in this section. This approach passes the cost of the remainder of the program directly rather than passing the continuation back to the current expression. There are two advantages of this approach. First, we do not need to translate the program into CPS. Second, it is easier to pass an integer (cost) back to the current expression than the execution state of the program e.g. the call stack or values of variables.

This section introduces the costafter semantics of $\mathcal{J}'$. In the costafter two definitions will be used: syntactic expression equality, and expression containment.

Syntactic Equality of Expression

Figure 6.72 shows the definition of syntactic expression equality, which compares if two expressions are structurally the same, presented as: $\equiv : e \rightarrow e \rightarrow boolean.$

$$\frac{e = k}{e \equiv k} \tag{6.53}$$

$$\frac{e = v}{e \equiv v} \tag{6.54}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(e_1\ op\ e_2) \equiv (e_3\ op\ e_4)} \tag{6.55}$$

$$\frac{x_1 \equiv x_2 \qquad e_1 \equiv e_2}{(\texttt{fun}\ x_1 \rightarrow e_1) \equiv (\texttt{fun}\ x_2 \rightarrow e_2)} \tag{6.56}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(e_1\ e_2) \equiv (e_3\ e_4)} \tag{6.57}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(\texttt{map}\ e_1\ e_2) \equiv (\texttt{map}\ e_3\ e_4)} \tag{6.58}$$

$$\frac{e_1 \equiv e_2 \qquad c_1 \equiv c_2}{(e_1\ (*\ c_1\ *)) \equiv (e_2\ (*\ c_2\ *))} \tag{6.59}$$

$$\frac{}{< i, e_1 > \equiv < i, e_2 >} \tag{6.60}$$

Figure 6.72: Definition of Syntactic Expression Equality in $\mathcal{J}'$

- Equation (6.53) defines constant expression equality. If expression e is an constant and its value is equal to k, so expression e is equal to expression k.

- Equation (6.54) performs variable equality. In $\mathcal{J}'$, it is assumed that variables (v) names in the program are unique, so if two variables expressions have the same name, the two expressions are equal.

- Equation (6.55) performs the equality of two operation expressions. Expression (e_1 op e_2) equals to expression (e_3 op e_4), if e_1 equals to e_3 and e_2 equals to e_4.

- Equation (6.56) performs the equality of two lambda expression. Expression `fun` ($x_1 \rightarrow e_1$) equals to expression `fun` ($x_2 \rightarrow e_2$), if variable x_1 equals to variable x_2 and expression e_1 equals to e_2.

- Equations (6.57) and (6.58) define application expression and `map` expression equality, which are all similar to Equation (6.56).

- Equation (6.59) shows that two user cost expression are equal if both the expression parts and cost parts are equal.
- Equation (6.60) defines the index expression equality. In the calculus, the whole program has been indexed, so every expression in a program has a unique integer as an index, so in $\mathcal{J}'$, two index expressions are equal to each other only if their indices are equal.

Expression Contains

$$\frac{e_1 \equiv e_2}{e_1 \in e_2} \tag{6.61}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{fun}\ x \ \rightarrow e_2} \tag{6.62}$$

$$\frac{e_1 \in e_2}{e_1 \in (e_2\ e_3)} \tag{6.63a}$$

$$\frac{e_1 \in e_3}{e_1 \in (e_2\ e_3)} \tag{6.63b}$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ op\ e_3} \tag{6.64a}$$

$$\frac{e_1 \in e_3}{e_1 \in e_2\ op\ e_3} \tag{6.64b}$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ (*\ c\ *)} \tag{6.65}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{map}\ e_2\ e_3} \tag{6.66a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{map}\ e_2\ e_3} \tag{6.66b}$$

$$\frac{e_1 \in e_2}{e_1 \in < i, e_2 >} \tag{6.67}$$

Figure 6.73: Definition of Contains in $\mathcal{J}'$

Figure 6.73 gives the definition of contains. Semantic function $\in$ takes two expressions, if the second expression contains the first expression it returns true or else returns false.

- Equation (6.61) identifies that if expression e_1 is equal to expression e_2 then the two expressions contain each other.

- Equation (6.62) shows that if e_1 is contained in e_2, then e_1 is contained in `fun` $x \rightarrow e2$.
- Equation (6.63a) to (6.67) gives the definition of contains for different expressions, which are all similar to Equation (6.62).

$$
\begin{array}{lrcll}
 & b & = & b & \\
 & b & \equiv & b & (6.54) \\
\Rightarrow & b & \in & b & (6.61) \\
\Rightarrow & b & \in & <2, b> & (6.67) \\
\Rightarrow & b & \in & (<1, a> + <2, b>) & (6.64b) \\
\Rightarrow & b & \in & <3, (<1, a> + <2, b>)> & (6.67) \\
\Rightarrow & b & \in & <3, (<1, a> + <2, b>)> + <6, (<4, c> + <5, d>)> & (6.64a) \\
\Rightarrow & b & \in & <7, (<3, (<1, a> + <2, b>)> + <6, (<4, c> + <5, d>)>)> & (6.67)
\end{array}
$$

Figure 6.74: An Example of Contains in $\mathcal{J}'$

An example to deduce if expression `(a+,b)+(c+d)` contains expression `b` is given in Figure 6.74, where expression `<7,(<3,(<1,a>+<2,b>)>+<6,(<4,c>+<5,d>)>)>` is indexed expression `(a+b)+(c+d)`.

Costafter Semantics for $\mathcal{J}'$

Figure 6.75 shows the semantics of costafter of an expression e in different expressions. Semantic function $\vdash_a$ takes the environment (env) and two expressions. The return value of $\vdash_a$ is a cost, which is the costafter of the first expression in the second expression. The type of this function is: $\vdash_a : env \rightarrow e \rightarrow e \rightarrow cost$.

- Equation (6.68) states that if expression e is equal to e' then the costafter of e in e' is 0.
- Equation (6.69a) and (6.69b) define the costafter of e in lambda expressions. If the costafter of e in e_1 is c, then the costafter of e in lambda expression `fun` $x \rightarrow e_1$ is c too. If e_1 does not contains e then the costafter of e in `fun` $x \rightarrow e_1$ is 0.
- Equation (6.70a), (6.70b), and (6.70c) define the costafter of e in application expressions (e_1 e_2). If e_1 contains e, then the costafter of e in (e_1 e_2) is the

$$\frac{e \;\equiv\; e'}{\mathrm{E} \vdash_a e \trianglelefteq e' \,\pounds\, 0} \tag{6.68}$$

$$\frac{\mathrm{E} \vdash_a e \trianglelefteq e_1 \,\pounds\, c}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{fun}\ x \ \rightarrow e_1 \,\pounds\, c} \tag{6.69a}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{fun}\ x \ \rightarrow e_1 \,\pounds\, 0} \tag{6.69b}$$

$$\frac{e \;\in\; e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \,\pounds\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ e_2) \,\pounds\, c_1 + c_2} \tag{6.70a}$$

$$\frac{e \;\in\; e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,\pounds\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ e_2) \,\pounds\, c_2} \tag{6.70b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ e_2) \,\pounds\, 0} \tag{6.70c}$$

$$\frac{e \;\in\; e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \,\pounds\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ op\ e_2) \,\pounds\, 1 + c_1 + c_2} \tag{6.71a}$$

$$\frac{e \;\in\; e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,\pounds\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ op\ e_2) \,\pounds\, c_2 + 1} \tag{6.71b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\ op\ e_2) \,\pounds\, 0} \tag{6.71c}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq e_1\ (*\ c\ *) \,\pounds\, 0} \tag{6.72}$$

$$\frac{e \;\in\; e_1 \qquad \mathrm{E} \vdash_c \texttt{map}\ e_1\ e_2 \,\$\, c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \,\pounds\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{map}\ e_1\ e_2 \,\pounds\, c + c_1 + c_2} \tag{6.73a}$$

$$\frac{e \;\in\; e_2 \qquad \mathrm{E} \vdash_c \texttt{map}\ e_1\ e_2 \,\$\, c \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,\pounds\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{map}\ e_1\ e_2 \,\pounds\, c + c_2} \tag{6.73b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{map}\ e_1\ e_2 \,\pounds\, 0} \tag{6.73c}$$

$$\frac{\mathrm{E} \vdash_a e \trianglelefteq e_1 \,\pounds\, c}{\mathrm{E} \vdash_a e \trianglelefteq\ < i, e_1 >\ \pounds\, c} \tag{6.74}$$

Figure 6.75: Costafter Semantics for $\mathcal{J}'$

costafter of e in e_1, here c_1, plus the cost of e_2, here c_2. If e_2 contains e then the costafter of e in $(e_1\ e_2)$ is the cost after e in e_2. If e is not contained in e_1 or e_2, then the costafter of e in $(e_1\ e_2)$ 0.

- Equation (6.71a), (6.71b), and (6.71c) define the costafter of e in operation expressions e.g. $(e_1\ op\ e_2)$. If e_1 contains e, then the costafter of e in $(e_1\ e_2)$ is the costafter of e in e_1 (c_1), plus the cost of e_2 (c_2) plus 1, which is the cost for getting the operator.

- In language $\mathcal{J}'$ the cost of recursive function can not be calculated automatically, so the costafters in recursive functions are not considered. The user cost expression gives the cost of a recursive function, so the costafter of any expression in a user cost expressions is 0, see Equation (6.72).

- Equation (6.73a), (6.73b), and (6.73c) define the costafter of e in `map` expression `map` $e_1\ e_2$. Equation (6.73a) shows that if e_1 contains e, then the costafter of e in `map` $e_1\ e_2$ is the costafter of e in e_1, plus the cost of e_2, and plus the cost of `map` expression, because after e_1 and e_2 `map` expression will be executed. A similar situation of e_2 containing e is handled in Equation (6.73b). Equation (6.73c) shows that if e is not contained in e_1 or e_2 then the costafter of e in `map` $e_1\ e_2$ is 0.

- Equation (6.74) shows that the costafter of e in index expression $< i, e_1 >$ is the same as the costafter of e in expression e_1.

Figure 6.76 gives an example to show how to calculate the costafter of a subexpression in a expression. From the deduction in Figure 6.74, expression `b` is contained in the indexed expression of `((a+b)+(c+d))`, e_i (Section 6.3.1). The costafter of expression `b` in expression e_i is calculated in Figure 6.76.

The result from the program, which implements the costafter semantics, is `((0+1)+((1+((1+0)+(1+0)))+1))`, where the first `0` is the costafter of `b` in `b` and the second and the third `0`s are the cost of `c`, and `d`, because in the program `a`, `b`, `c`, and `d` are integers and the cost of integer is 0. So the result is the same as that calculated by hand in Figure 6.76.

	costafter of b in	
	$< 7, (< 3, (< 1, a > + < 2, b >) > + < 6, (< 4, c > + < 5, d >) >) >$	
$\Rightarrow$	costafter of b in	
	$(< 3, (< 1, a > + < 2, b >) > + < 6, (< 4, c > + < 5, d >) >)$	(6.74)
$\Rightarrow$	1+costafter of b in	
	$< 3, (< 1, a > + < 2, b >) >$+cost of $< 6, (< 4, c > + < 5, d >) >$	(6.71a)
$\Rightarrow$	1+costafter of b in	
	$(< 1, a > + < 2, b >)$+cost of $< 6, (< 4, c > + < 5, d >) >$	(6.74)
$\Rightarrow$	1+(1+costafter of b in	
	$< 2, b >$)+cost of $< 6, (< 4, c > + < 5, d >) >$	(6.71b)
$\Rightarrow$	1+(1+costafter of b in b)+cost of $< 6, (< 4, c > + < 5, d >) >$	(6.74)
$\Rightarrow$	1+(1+0)+cost of $< 6, (< 4, c > + < 5, d >) >$	(6.68)
$\Rightarrow$	1+(1+0)+(1+ (1+c_c) + (1+c_d)) (From Figure 6.71)	

Figure 6.76: An Example of Costafter in $\mathcal{J}'$

6.4 Costed Autonomous Mobility Skeletons

A cost calculus to manipulate and ultimately automatically statically extract costafter at arbitrary points has been built in Section 6.3. This section extends the AMS cost model to be parameterised on the costafter of the higher-order functions, and improved AMSs have been built based on the extended cost model. The improved AMSs are called *costed autonomous mobility skeletons* (CAMSs). CAMSs e.g. `camap` and `cafold` not only encapsulate the common pattern of autonomous mobility like AMSs in Chapter 4, but take two additional cost parameters: the cost of the remainder and the cost in the higher-order function.

6.4.1 Cost Model for Costed Autonomous Mobility Skeletons

The cost model for CAMSs is similar to that for AMSs in Section 4.2.1 and is given in Figure 6.77. In this cost model:

- In Equation (6.75) the total work is the cost in the CAMS and the costafter of the CAMS, rather than just the total work in Equation (4.29) in the cost model for AMS.
- Equation (6.76) gives the work that has been done, `r`.

$$W_{all} = costf * (length\ of\ the\ list) + costafter \ = \ a \tag{6.75}$$

$$W_a = r \tag{6.76}$$

$$W_l = W_{all} - W_a = a - r \tag{6.77}$$

$$W_d = costf \tag{6.78}$$

$$T_e = \frac{W_d}{S_h} Sec = \frac{costf}{S_h} Sec \tag{6.79}$$

$$T_h = \frac{W_l}{S_h} Sec = \frac{(a-r)Sec}{S_h} \tag{6.80}$$

$$T_h = \frac{((a-r)Sec)T_e}{Sec} = \frac{(a-r)}{costf} T_e \tag{6.81}$$

$$T_n = \frac{W_l}{S_n} = \frac{(a-r)Sec}{S_n} = \frac{S_h T_h}{S_n} \tag{6.82}$$

Figure 6.77: Cost Model for CAMS

- Equation (6.77) shows that the remaining work is the total work minus the work that has been done.

- Equation (6.78) shows that the work that has been done at the current location is the cost of `f`, which is the cost for processing one element in the list.

- Substituting Equation (6.78) in (3.14), the time that has elapsed at current location can be calculated giving Equation (6.79).

- Substituting Equation (6.77) in (3.15) the time it will take at the current location can be calculated giving Equation (6.80).

- Substituting Equation (6.79) in (6.80) Equation (6.81) can be got, which shows the time it will take at the current location is a function of T_e.

- Substituting Equation (6.80) in (3.15) the time that will be taken in the next location can be predicted as Equation (6.82).

6.4.2 An Implementation of Costed Autonomous Mobility Skeletons

camap and cafold CAMSs are implemented in Jocaml. An implementation of camap is shown in Figure 6.78. The implementation cafold is very similar to camap. The check_move function is the same as in Section 4.2.2. The checkInfo function is similar to the getInfo function in Section 4.2.2, which is to evaluate the benefits of a move and to recalculate when should check again rather to calculate a gran in function getInfo.

```
let rec camap' f l costf costaftermap fhtime workleft=
   let (h::t) = l in
   if (((!t_current)-.(!t_last)) >= (!whencheck))
   then
   (check_move costf workleft fhtime;
    let (fh,fhtime') = timedapply f h in
    t_current := Unix.gettimeofday();
    checkInfo costf fhtime';
    fh::camap' f t costf costaftermap fhtime' (workleft-costf)
   )
   else
   (let fh = f h in
    t_current := Unix.gettimeofday();
    fh::camap' f t costf costaftermap fhtime (workleft-costf)
   )

let camap f l costf costaftermap =
   let (h::t) = l in
   (let localwork = costf * (length l) in
    let work = localwork + costaftermap in
    let (fh,fhtime) = timedapply f h in
    t_current := Unix.gettimeofday();
    checkInfo costf fhtime;
    fh::camap' f t costf costaftermap fhtime (work-costf)
   )
```

Figure 6.78: Implementation of camap in Jocaml

6.5 Evaluating Costed Autonomous Mobility Skeletons

This section is using `camap` and `automap` as examples to evaluate the performance of CAMSs against AMSs. Six AMPs have been built, which use different numbers of higher-order functions to demonstrate the behaviours of CAMSs and AMSs. To construct the AMPs using `camap f l costf costafter`, the `costf` and `costafter` are calculated by hand according to the cost calculus for $\mathcal{J}'$. Section 6.6 will introduce an automatic continuation cost analyser which can automatically transform higher-order functions into CAMSs.

In this chapter, CAMS programs refer to AMPs with CAMSs, and the same as AMS programs.

6.5.1 Single Higher Order Function Examples

The initial hypothesis is if there is only one higher-order functions in the AMP, the performances of the CAMS programs should be the the same as the corresponding AMS programs. That is the AMPs should exhibit the same movement behaviour at the same moment and hence have the same execution times. Two single higher-order function AMPs have been tested: matrix multiplication and ray tracing.

Matrix Multiplication Comparison

Different size matrix multiplication have been executed to see if the CAMS programs have the same execution time as the corresponding AMS programs. The test environment has three locations with CPU speeds 534MHZ(ncc1710), 933MHZ(jove) and 1894MHZ(lxtrinder). The loads on these three locations are almost zero, and both the CAMS and AMS programs are started on the first location.

Figure 6.79 shows that the CAMS programs behave the same as the corresponding AMS programs. Both the CAMS and AMS programs start to move when the matrix size increased to 330*330 (See Table D.32 in Appendix D), and

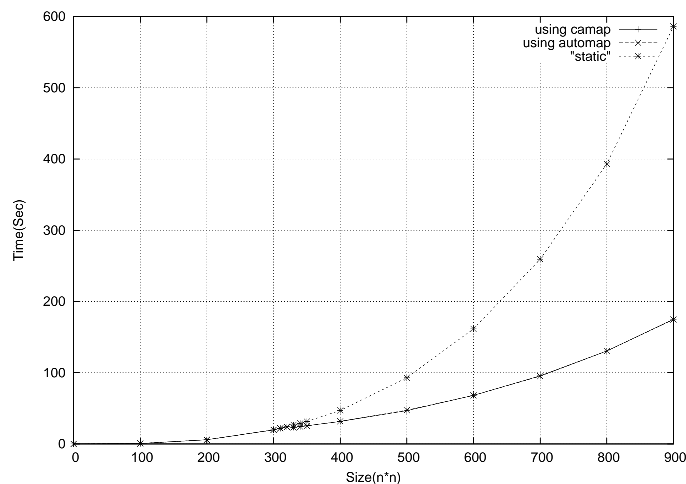


Figure 6.79: CAMS and AMS Matrix Multiplication Execution Time Comparison

hence the execution time of the CAMS and AMS programs are almost identical.

Ray Tracing Comparison

Figure 6.80 shows similar results for ray tracing AMPs. The exact execution time are shown in Table D.33 in Appendix D.

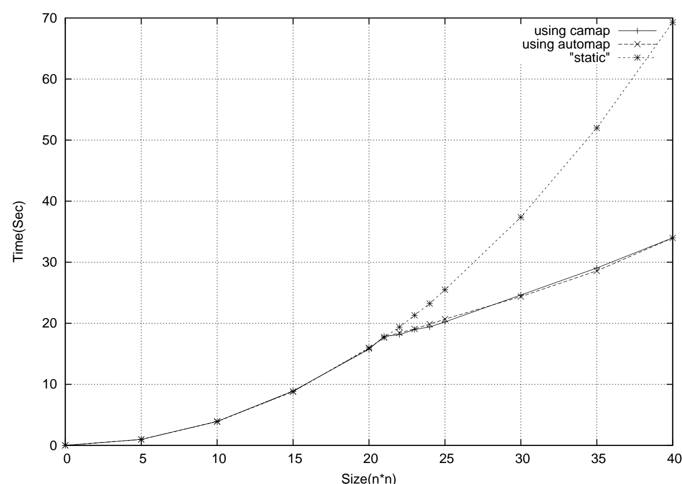


Figure 6.80: CAMS and AMS Ray Tracing Execution Time Comparison

From the results of matrix multiplication and ray tracing, it can be concluded that when there is only one higher-order functions in the AMPs, the performances of the CAMS programs are as good as the AMS programs.

6.5.2 Sequential Composed Higher Order Function Examples

This section investigates the performance of the skeletons when an AMP contains the sequential composition of several higher-order functions. Four experiments have been performed with sequential CAMSs or AMSs. The AMPs are double matrix multiplication, invertible matrix, double ray tracing, and five matrix multiplications. The test environments in this section are the same as in Section 6.5.1.

Double Matrix Multiplication Comparison

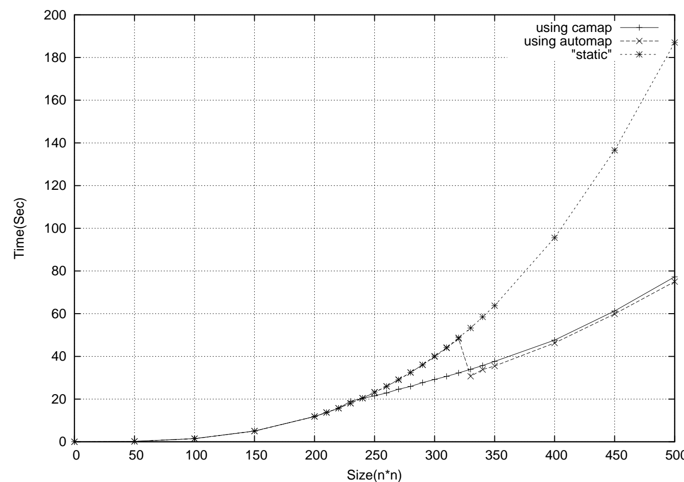


Figure 6.81: CAMS and AMS Double Matrix Multiplication Execution Time Comparison

Figure 6.81 compares the CAMS and AMS double matrix multiplication programs, where two matrix multiplication are performed sequentially. The CAMS program starts moving at matrix size of 230*230, but the AMS program starts moving still at 330*330 (see Table D.34 in Appendix D), because the CAMSs consider the total remaining costs of the entire program but AMSs only consider the remaining costs in the function.

Figure 6.81 shows that in some cases the CAMS programs are slower than the corresponding AMS programs, for the following reasons.

- When the matrix is large enough for both the CAMS and AMS programs to move to faster location with the same size of matrix e.g. 330*330, the AMS

program move at an *earlier element* (not size) than the CAMS program. For example in the AMP of size 330*330 double matrix multiplication, the first `automap` in the program only calculate the 330 elements in itself, and decides to move once, so the program moves at the 165th element. However, the first `camap` in the CAMS program, calculates 330*2=660 elements, and considers moving twice, so the program moves at the 220th element. Hence, the CAMS program moves to a faster location later than the AMS program and it is a little slower.

- Another reason is that the CAMS programs may perform more checks than the corresponding AMS programs. Still using the program with matrix size of 330*330 double matrix multiplication as the example, after the AMS program moves to a faster location the first `automap` considers the remaining 165 elements only, so it might not check information again. However the first `camap` in the CAMS program considers 110+330=440 elements, so it might check the locations information again.

In these two cases the CAMS programs may be a little slower than the corresponding AMS programs, but are both faster than the static programs.

Invertible Matrix Comparison

The invertible matrix program takes two matrix and checks if they are invertible to each other. The essence of the program is as follows:

```
let m12 = mmult m1 m2;;
let isId12 = checkEqual m12 idMat;;
let m21 = mmult m2 m1;;
let isId21 = checkEqual m21 idMat;;
```

In this program there are two matrix multiplications, so there are two higher-order functions as in double matrix multiplications. The `checkEqual` function checks if two matrices are identical, which takes little execution time, so the invertible matrix program is very similar to double matrix multiplication.

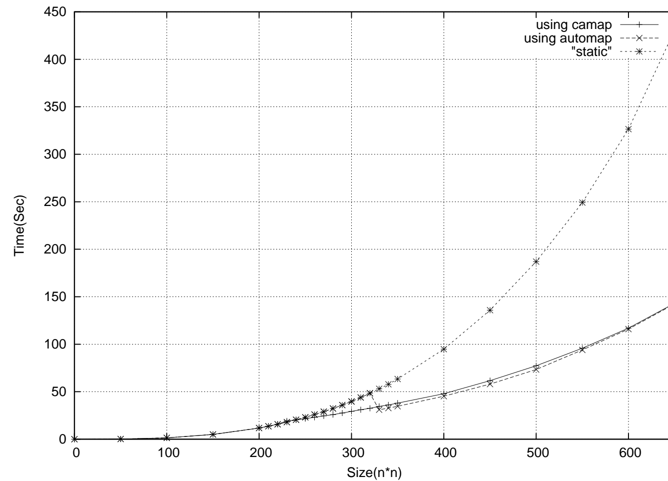


Figure 6.82: CAMS and AMS Invertible Matrix Execution Time Comparison

Figure 6.82 compares the execution time of a range of sizes of CAMS invertible matrix programs with AMS programs. The results are similar to double matrix multiplication. The exact execution times are shown in Table D.35 in Appendix D.

Double Ray Tracing Comparison

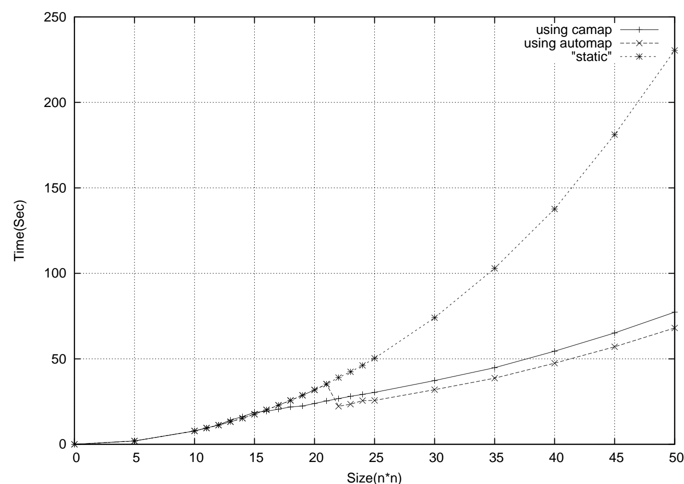


Figure 6.83: CAMS and AMS Double Ray Tracing Execution Time Comparison

Figure 6.83 compares the execution time of CAMS and AMS double ray tracing AMPs, where ray tracing is done twice. The results are similar to the results for double matrix multiplication. The exact execution times are shown in Table D.36 in Appendix D.

Five Matrix Multiplication Comparison

Figure 6.84 compares movements of CAMS and AMS five matrix multiplication AMPs, where matrix multiplication are done five times. Hence, there are five higher-order functions in the program. The results are similar to the results for double matrix multiplication. The exact execution times are shown in Table D.37 in Appendix D.

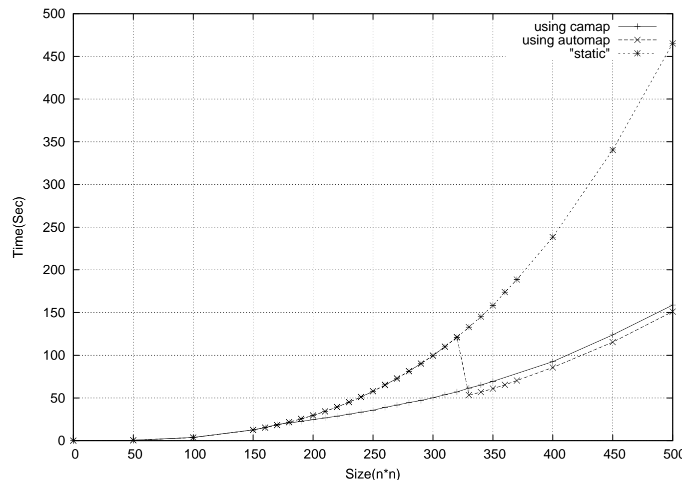


Figure 6.84: CAMS and AMS Five Matrix Multiplications Execution Time Comparison

Comparing Figure 6.84 with Figure 6.81 shows that the CAMS five matrix multiplication program starts to move at size 170*170, which is smaller than the CAMS double matrix multiplication program at size 230*230. However, the AMS five matrix multiplication program still starts to move at size of 330*330 matrix, which is the same as the AMS double matrix multiplication program. The reason is that in CAMS five matrix multiplication, the first `camap` considers the costs of 5 higher-order functions, but the first `automap` in the corresponding AMS program still only considers the cost of the current collection computation.

Discussion

From the results for AMPs with different number of higher-order functions, the following properties of CAMS and AMS can be noted:

- In some cases the CAMS programs are faster than the corresponding AMS

programs because CAMS programs move to a faster location with smaller size data than the corresponding AMS programs, and the more higher-order functions in a AMP, the earlier the CAMS program moves compared with the corresponding AMS program.

- The CAMS programs may do more checks than the corresponding AMS programs. This is a disadvantage for execution time i.e. in this case the CAMS programs may be slower than the corresponding AMS programs, but is an advantage for reacting to the change of environment, which will be considered in Section 6.5.3.

- For the same size of matrix, if both the CAMS and corresponding AMS programs move to a faster location, then the AMS programs move at an *earlier element* (not size) than the corresponding CAMS programs. In this case, the AMS programs are faster than the corresponding CAMS programs.

- Even if the AMS programs may move at an earlier element than the corresponding CAMS programs, the ratio of CAMS program execution compared to the corresponding AMS program becomes smaller when the data becomes bigger, as the number of the element at which the CAMS program starts moving compared to the AMS program becomes smaller when the data becomes bigger. For example, in an AMP l is the length of list in the higher-order functions, n is number of higher-order function, and m is the checks for one higher-order function. The *gran*, at which the AMP moves, for `automap` is $gran_{automap} = \frac{l}{m+1}$, and for `camap` is $gran_{camap} = \frac{l*n}{m*n+1}$. $gran_{camap}/gran_{automap} = \frac{l*n}{m*n+1}/\frac{l}{m+1}$ can be simplified as $1 + \frac{n-1}{m*n+1}$. So when n is a constant, the bigger the m the smaller the ratio. As bigger AMPs can do more checks, which is the bigger m, the bigger the AMP, the smaller the ratio. The assumption here is that the n higher-order functions are working on the same function and the same list.

6.5.3 Performance Comparison with Changing Load

When there is more than one higher-order function in the AMP, CAMS programs may do more checks than the corresponding AMS programs (see Section 6.5.2). This is an advantage for reacting to the change of environment. The experiments in this section test the behaviour of CAMS and AMS programs when the loads of locations in the network are changed. Tests for two AMPs have been done: invertible matrix and five matrix multiplications. The tests are based on four locations, (1)ncc1710, (2)jove, (3)lxtrinder, and (4)linux81, with CPU speed of 534MHz, 933MHz, 1894MHz, and 2800MHz. At the beginning, the first three location are idle and the fourth location is heavily loaded with relative CPU speed of 56MHz.

Figure 6.85 compares the execution time of CAMS to AMS invertible matrix AMPs. Both CAMS and AMS invertible matrix AMPs are tested one by one. The AMPs are started on ncc1710 and move to lxtrinder as expected. At the same time linux81 finishes the work and becomes idle.

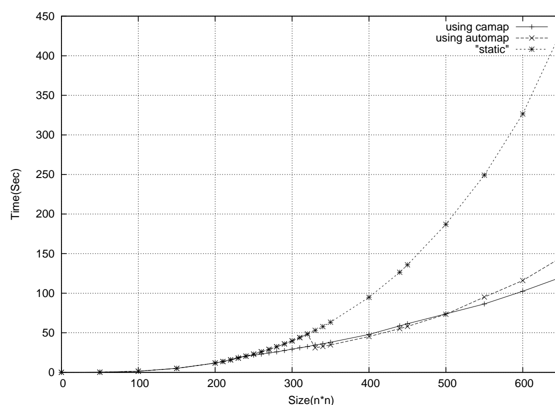


Figure 6.85: CAMS and AMS Invertible Matrix Execution Time Comparison with Changing Loads

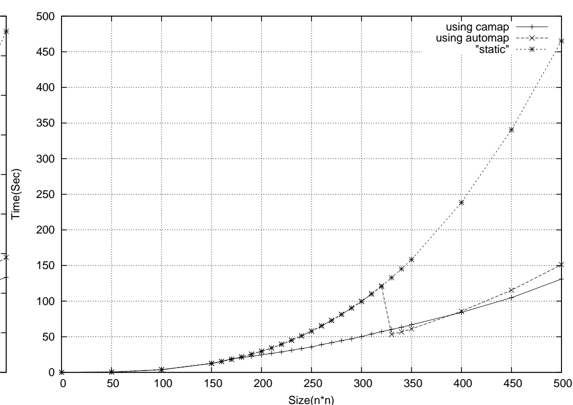


Figure 6.86: CAMS and AMS Five Matrix Multiplications Execution Time Comparison with Changing Loads

When the sizes of matrices are larger than 450*450, the CAMS programs move to the faster location linux81 again, but the corresponding AMS programs do not, as the `camap` in the CAMS programs do more checks than `automap` in the corresponding AMS programs. In this case, the CAMS programs may finish faster than the corresponding AMS programs. When the size of the matrix is larger

than 450*450 but smaller than 500*500, the CAMS programs is slower than the corresponding AMS programs, as the additional coordination time is bigger than the reduced execution time in a faster location. When the size of the matrix is larger than 500*500, the CAMS programs is faster than the corresponding AMS programs.

When the size of matrix is smaller than 450*450, the CAMS programs do not move, because the cost of moving (T_{comm}) to another location is too big compared to the reduced execution time in a faster location. In this case, the results are similar to the result in Figure 6.82.

The exact execution times are showed in Table D.38 in Appendix D. Similar results are obtained for five matrix multiplication AMPs, shown in Figure 6.86. Also see Table D.39 in Appendix D for details.

Note that the more higher-order functions in AMPs, the smaller size data with which the CAMS programs start to move again. In invertible matrix, there are two higher-order functions and the CAMP programs move again when the size of the matrix is larger than 450*450. In five matrix multiplication, there are five higher-order functions and the CAMS programs move again when the size of the matrix is larger than 330*330.

6.5.4 Conclusion

From the results in Sections 6.5.1, 6.5.2, and 6.5.3, the following conclusion can be drawn:

- If there is only one high-order function dominating the computation, CAMS programs reproduce the movement of the corresponding AMS programs (Section 6.5.1).

- If there is more than one higher-order functions in AMPs, CAMS programs move with smaller size data e.g. matrix than the corresponding AMS programs, and the more higher-order functions in AMPs, with the smaller data the CAMS programs move than the corresponding AMS programs and hence the greater the potential for gains (Section 6.5.2).

- When both the CAMS and AMS programs move, the CAMS programs may be slower than the corresponding AMS programs, but the larger the data in the AMPs the earlier the CAMS programs move relative to the corresponding AMS programs (Section 6.5.2).

- The CAMS programs react to the change of environment more sensitively than the corresponding AMS programs (Section 6.5.3).

6.6 Automatic Continuation Cost Analyser

6.6.1 Structure of the Automatic Continuation Cost Analyser

The cost calculus has been implemented as an automatic cost analyser in Jocaml. The analyser produces cost equations parameterised on program variables in context, and is used to find both cost in higher order functions and the costafter of the higher order functions. As *costafter* is the cost of the continuations, the analyser is also called an *automatic continuation cost analyser*. The analyser takes programs in a subset of Jocaml with higher-order functions as input and outputs Jocaml AMPs with CAMSs. Figure 6.87 shows the structure of the automatic continuation cost analyser. The analyser has four parts.

1. The parser takes Jocaml programs with higher-order functions e.g. `map` and `fold` and outputs the abstract syntax tree (AST).

2. The indexer is an implementation of the index semantics in Section 6.3.1. The indexer takes the AST and decorates every nodes in the AST which gives every expression a unique integer as an index, so the output of indexer is an *indexed abstract syntax tree (IAST).*

3. The coster takes the IAST and outputs the cost after each node (costafter) in IAST. The coster first calculates the cost of each expression, which implements the cost semantics in Section 6.3.2, and then calculates the

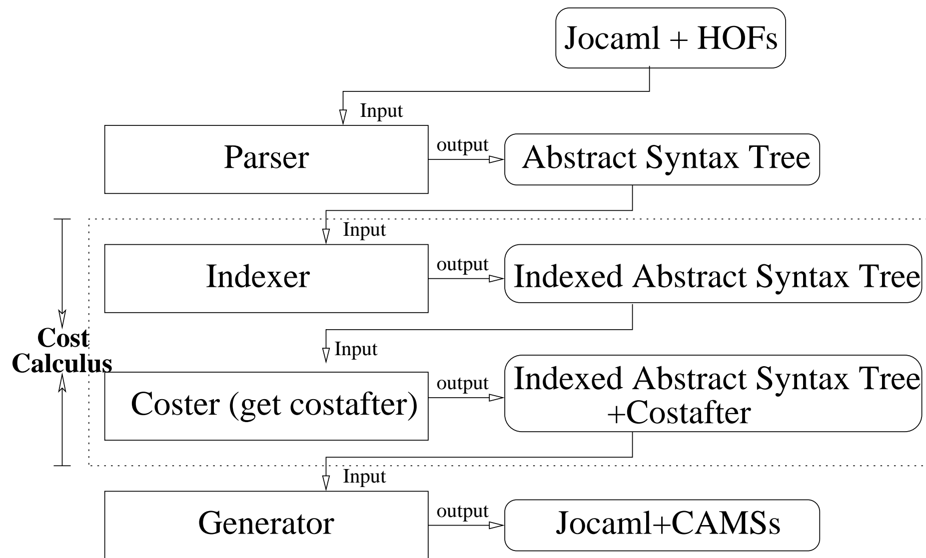


Figure 6.87: Structure of Automatic Continuation Cost Analyser

costafter of each expression, which implements the costafter semantics in Section 6.3.3.

4. The Generator generates a Jocaml program from the IAST, which has the same functionality as the Jocaml program but using CAMSs e.g. `camap` or `cafold` instead of higher-order functions `map` and `fold`.

6.6.2 An Implementation of the Cost Calculus

In the automatic continuation cost analyser, `index, cost,` and `costafter` functions have been implemented, with the semantics given in Section 6.3.1, 6.3.2, and 6.3.3.

```
let rec index i e =
     match e with
      (VAR s) -> (INDEX (i,VAR s),i+1) |
      (INT i1) -> (INDEX (i,INT i1),i+1) |
      .......
```

Figure 6.88: Implementation of Index Function

Figure 6.88 shows part of the implementation of the `index`. In the code, `index` takes the current index number `i` and the expression to be indexed `e` and returns

the indexed expression and the next index number `i+1`. The type of `index` is: `int -> expression -> (expression * int)`.

```
let rec cost env e =
     match e with
      (VAR i) -> (*cost env*)  (lookup env i) |
      (INT _) -> INT 0 |
      (OP(_,e1,e2)) -> OP(LADD,INT 1,OP(LADD,cost env e1,cost env e2)) |
      .......
```

Figure 6.89: Implementation of Cost Function

Figure 6.89 shows part of the code of the `cost` function. The `cost` function implements the cost semantics introduced in Section 6.3.2. The type of `cost` is: `env -> expression -> int`.

```
let rec costafter env e1 e2 =
  if e1=e2
  then INT 0
  else costafter' env e1 e2

and costafter' env e e' =
  match e' with
    (VAR i) -> INT 0 |
    (INT i) -> INT 0 |
    (OP(_,e1,e2)) ->
      if contains e e1
      then OP(LADD,costafter env e e1,OP(LADD,cost env e2,INT 1))
      else
        if contains e e2
        then OP(LADD,costafter env e e2,INT 1)
        else INT 0 |
      .......
```

Figure 6.90: Implementation of Costafter Function in Jocaml

Figure 6.90 is the implementation of `costafter`. The `costafter` implements the costafter semantics introduced in Section 6.3.3. The type of `costafter` is: `env -> expression -> expression -> int`.

6.6.3 Generator: Generating Autonomous Mobility Skeletons

Section 6.6.2 shows the programs to calculate the costafter of an arbitrary node in abstract syntax trees, but this thesis is interested in the nodes of higher-order functions e.g `map` and `fold`, which can be converted into CAMSs.

The functionality of the generator is to find the higher-order functions in a program and translate them to CAMSs with the costafters. For example, if the original program is `map f l`, the object program after the generator is `camap f l costf costafter`, where `costf` is the cost of `f` applied to the first element of `l`, which can be calculated using the cost semantics, and `costafter` is the cost after the `map` expression in the program, which can be calculated using costafter semantics.

This section uses expression e, `(map (fun x -> x+1) [1;2]);(map (fun y -> y-1) [3;4])`, as an example to explain how to convert higher-order functions to CAMS. Expression e has two sub-expressions e_1, `(map (fun x -> x+1) [1;2])`, and e_2, `(map (fun y -> y-1) [3;4])`, so e can be presented as $e_1; e_2$. To construct CAMSs, four issues should be considered:

(1) the cost of `(fun x -> x+1)`,

(2) the costafter of e_1 in e,

(3) the cost of `(fun y -> y-1)`,

(4) and the costafter of e_2 in e.

From the cost equations in Section 6.3.2, the cost of `(fun x -> x+1) (hd [1;2])` can be calculated, which is `(fun x -> (1+((1+0)+0))) (hd [1;2])` (Equations (6.49), (6.48), (6.46)). The simplified result is 2.

The costafter of e_1 in e is the costafter of e_1 in e_1, which is 0 (Equation (6.68)), plus the cost of e_2, plus 1 (Equation (6.71a)). The cost of e_2 is `((fun y -> (1+((1+0)+0))) (hd [3;4]))*(length [3;4])`, which is simplified as 4, so the total costafter of e_1 in e is `5`. The generator converts e_1 to `(camap (fun x`

`-> (x+1)) [1;2] (2) (5))`. Similarly, e_2 is converted to `(camap (fun y -> (y-1)) [3;4] (2) (1))`.

The result from the program is the same as that which is calculated by hand. The generator takes the original program,

```
(map (fun x -> x+1) [1;2]);
(map (fun y -> y-1) [3;4])
```

and generates an AMP with CAMSs. The AMP is:

```
(camap (fun x -> (x+1)) [1;2]
  (((fun x -> (1+((1+0)+0))) (hd [1;2])))  (* cost of (fun x -> x+1) *)
  ((0+(((fun y -> (1+((1+0)+0))) (hd [3;4]))*(length [3;4])))+1)
                                          (* costafter of first map *)
);
(camap (fun y -> (y-1)) [3;4]
  (((fun y -> (1+((1+0)+0))) (hd [3;4])))  (* cost of (fun x -> x+1) *)
  (0)                                     (* costafter of second map *)
)
```

The AMP can be simplified to:

```
(camap (fun x -> (x+1)) [1;2] (2) (5) );
(camap (fun y -> (y-1)) [3;4] (2) (1) )
```

6.6.4 Matrix Multiplication Example

Figure 6.91 gives the source code of matrix multiplication in Jocaml for the analyser. Note the user costs for `dist`, `tranpose`, `dotprod`, and `rowmult`.

The analyser takes the source code as input and outputs the target code of matrix multiplication AMP, where top level `map` has been converted to `camap`. Figure 6.92 shows the target code of the CAMSs matrix multiplication. Note this example is in $\mathcal{J}$, which is an extension of $\mathcal{J}'$. The cost calculus for $\mathcal{J}$ is been presented in Appendix A.

```
let rec dist = (fun vec1 -> fun vec2 ->
  match (vec1,vec2) with
    ([],vec) -> [] |
    ((h1::t1),(h2::t2)) -> (h1::h2)::(dist t1 t2) |
    ((h1::t1), []) -> [h1]::(dist t1 []))
  (* fun mv11-> fun mv22 ->  3*(length mv11)*(length mv22) *)
  in let rec transpose = (fun e -> fold_right dist e [])
     (* fun le -> 2*(length le)*(length le) *)
  in let rec dotprod = (fun mat1 -> fun mat2 ->
       match (mat1,mat2) with
         ((h1::t1),(h2::t2)) -> h1*h2+(dotprod t1 t2) |
         (m1,m2) -> 0 )
     (* fun l1 -> fun l2 -> (4*(length l2)) *)
  in let rec rowmult  = (fun cls -> fun row -> map (dotprod row) cls )
     (* fun rowc -> fun lsc -> 4*(length rowc)*(length rowc) *)
  in let rec rowsmult = (fun rows -> fun cols ->
       map ( rowmult cols) rows )
  in let tm2 = transpose mat2
  in rowsmult mat1 tm2
```

Figure 6.91: The Source Code of Matrix Multiplication for Automatic Cost Analyser

6.6.5 Comparing Automatic and Hand Analysis

This section compares the performances CAMSs generated by the analyser with the CAMSs which have been produced by hand in Section 6.5. Comparison of double matrix multiplication, invertible matrix, and double ray tracing AMPs have been made.

Figure 6.93 compares the execution times of different sizes of double matrix multiplication, and shows that the execution time of analyser produced CAMS programs are very similar to the hand analysed CAMS programs. See Table D.40 in Appendix D for the details of execution time.

Similar results are found for invertible matrix AMPs and double ray tracing AMPs, which are shown in Figures 6.94 and 6.95. See Tables D.41 and D.42 in Appendix D for the details of execution time.

```
let rec dist =
  ((fun vec1 -> (fun vec2 -> match (vec1,vec2) with
    ([],vec) -> [] |
    ((h1::t1),(h2::t2)) -> ((h1::h2)::((dist t1) t2)) |
    ((h1::t1),[]) -> ([h1]::((dist t1) [])))));;
let rec transpose =
  ((fun e -> (fold_right dist e [])));;
let rec dotprod =
  ((fun mat1 -> (fun mat2 -> match (mat1,mat2) with
      ((h1::t1),(h2::t2)) -> ((h1*h2)+((dotprod t1) t2)) |
      (m1,m2) -> 0)));;
let rec rowmult =
  ((fun cls -> (fun row -> (map (dotprod row) cls))));;
let rec rowsmult =
  (fun rows -> (fun cols -> (map (rowmult cols) rows)));;
let tm2 = (transpose mat2)
 in
 (camap (rowmult tm2) (mat1) (((4*(length tm2))*(length tm2))) (0))
```

Figure 6.92: The Target Code of Automatic Cost Analyser

6.7 Summary

This chapter introduces a core functional language $\mathcal{J}'$, a subset of Jocaml, in Section 6.2. To provide the costafter i.e. the cost for the reminder of a computation at arbitrary points, a cost calculus for $\mathcal{J}'$ has been built. The cost calculus calculates the costafter by three steps: to index the AST into IAST, which is to distinguish duplicate expressions; to calculate the cost of expression in a program; to calculate the costafter of each higher-order function. Section 6.3 introduces the semantics for index, cost, and costafter, the three parts of the cost calculus.

To validate the cost calculus, costed autonomous mobility skeletons (CAMSs) e.g. `camap` and `cafold` have been built by hand in Section 6.4 with the cost model for them. The CAMSs take additional cost parameters e.g. costafter.

To evaluates the performance of CAMSs against AMSs, six AMPs have been built with single or with sequential composed higher-order functions in Section 6.5. If there is only one high-order function dominating the computation, CAMS programs reproduce the movement of the corresponding AMS programs.

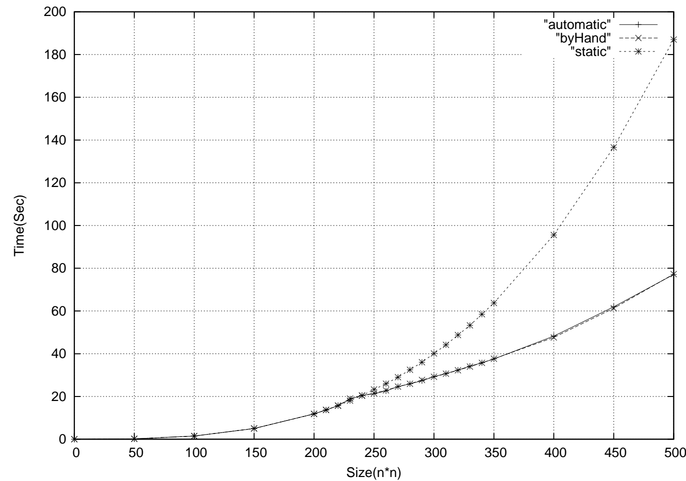


Figure 6.93: Automatic and Byhand Cost Double Matrix Multiplications Execution Time Comparison

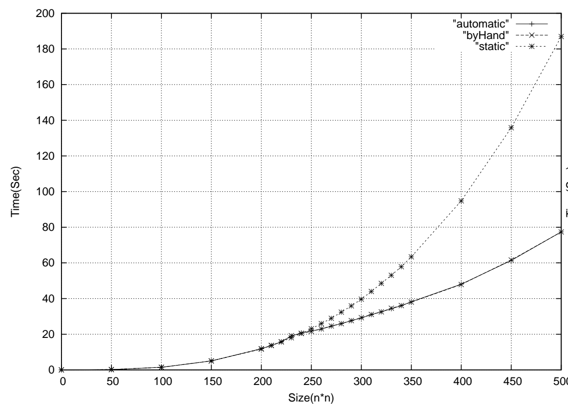


Figure 6.94: Automatic and Byhand Cost Invertible Matrix Execution Time Comparison

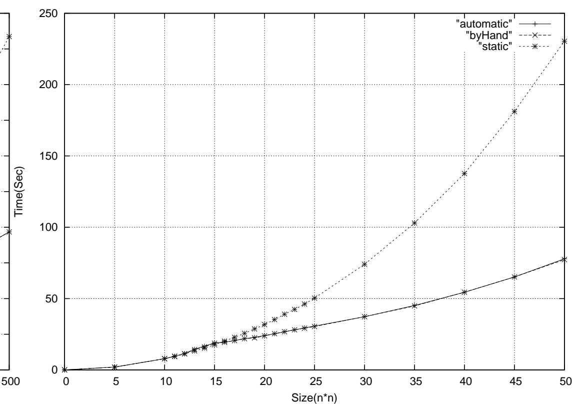


Figure 6.95: Automatic and Byhand Costs Double Ray Tracing Execution Time Comparison

If there is more than one higher-order functions in the AMPs, CAMS programs move with smaller size data than the corresponding AMS programs, and the more higher-order functions in the AMPs, with the smaller data the CAMS programs move than the corresponding AMS programs and hence the greater the potential for gains. When both the CAMS and AMS programs move, the CAMS programs may be slower than the corresponding AMS programs, but the larger the data in the AMPs the earlier the CAMS programs move relative to the corresponding AMS programs. The CAMS programs react to the change of environment more sensitively than the corresponding AMS programs.

The automatic continuation cost analyser implements the cost calculus. The

analyser converts higher-order functions into CAMSs automatically. Experiments in Section 6.6 show that the performance of CAMS programs which are constructed automatically for the analyser are very similar to the CAMS programs which have been produced by hand.

Chapter 7

Conclusion and Future Work

7.1 Summary

One of the biggest issues for distributed systems is how to share resources, computational or data. This thesis demonstrated autonomous mobility for *self-optimization* of computational resource.

Chapter 2 introduces the concepts related to distributed systems such as *mobility*, *agents* , *autonomous systems*, *Load management* systems, and *cost models.*

Chapter 3 proposes autonomous mobile programs (AMPs), which can periodically use a cost model to decide mobility effects. The advantages of an AMP model are as follows. AMPs making decentralised decisions about where to execute. Indeed on large networks only nearby locations need be considered as potential targets. The model manages dynamic networks very easily with each AMP selecting where to execute from the current set of locations. The AMP model can obtain a better balance than a classical distributed load balancer as, unlike the latter, each AMP has a cost model giving accurate information about the time to complete and to communicate the program. Moreover it is possible to parameterise the AMP cost model with a maximum overhead, e.g. 5%, and guarantee, under reasonable assumptions that autonomous mobility overheads will not exceed it.

Chapter 4 introduces *autonomous mobility skeletons*. The disadvantages of AMPs are that the programmer must explicitly control when the program moves in the AMPs, and AMPs also require an accurate model of computation. To encapsulate self-aware mobile coordination for common patterns of computation over collections, autonomous mobility skeletons (AMSs) have been developed. AMSs are akin to algorithmic skeletons in being polymorphic higher order functions, but where algorithmic skeletons abstract over parallel coordination, AMSs abstract over autonomous mobile coordination. By analogy with other skeleton species, they hide low level mobile coordination details from users and provide higher level loci for designing load-aware mobile systems.

This chapter also demonstrates abstract AMSs with concrete realisations for the common higher-order functions `map` and `fold`. The realisations are provided both in the functional language context shared with other skeleton species, using Jocaml, and in an Object-Oriented context using mobile Javas e.g. Java Voyager and JavaGo. We have also demonstrated a novel `AutoIterator` skeleton for the widely used Object-Oriented `Iterator` interface. Our experiments suggest that, for our set of test programs, AMSs can offer considerable savings in execution times, which scale well as overall execution times increase.

Cost models for AMSs are dynamic and substantially implicit. During the traversal of a collection, the skeleton implementation periodically measures the time to compute a single collection element, and uses the value to parameterise an implicit cost for the remainder of the traversal.

Chapter 5 considers collections of AMPs, and a load server architecture to reduce the cost of coordination, which is the time for AMPs to get information. The AMP architecture introduces coordination costs as every AMPs must obtain load information about locations. The coordination cost is the additional cost of AMPs against the sequential program, so the smaller the coordination cost the better the performance of AMPs. Minimising the effect of the coordination cost must be take into account when building AMPs. In the load server architecture, every location has a load server, which collects local information and get information from other load server. The AMPs get information about locations from

the local load server rather than collecting themselves.

Collections of AMPs behave like a decentralised load balancing system. The behaviours of collection of AMPs have been explored under the load server structure on both homogeneous and heterogeneous networks. Collections of AMPs quickly obtain and maintain optimal or near-optimal balance until the balance is broken.

Chapter 6 presents cost calculus that can statically predict the costs of the remainder i.e. costafter of a program at arbitrary points. The cost models for AMSs in Chapter 4 assume that a single higher-order functions is the dominating computation for the program. To relax this constraint, more sophisticated cost models are parameterised on the costafter of the higher-order functions. Costed autonomous mobility skeletons (CAMSs) have been implemented with additional cost parameters for the cost of the remainder of the program based on the new cost model. An automatic cost analyser which implements the cost calculus has been implemented for language $\mathcal{J}$, a subset of Jocaml in Jocaml. The cost analyser inspects programs in $\mathcal{J}$ and replaces higher-order functions with CAMSs.

Experiments have been made to compare CAMSs with AMSs in different programs, where a single higher-order function or multiple higher-order functions are in the programs. If there is only one high-order function dominating the computation, CAMS program reproduce the movement of the corresponding AMS programs If there is more than one higher-order functions in AMPs, the CAMS programs react to changes of environment more sensitively than the corresponding AMS programs, because they have more accurate cost information i.e. including the cost of the remainder of the program. Hence a CAMS program may move to a faster location when the corresponding AMS program does not.

7.2 Contributions and Further Work

Section 1.2 discussed four research contributions in this thesis. This section discuses further work for these contributions.

7.2.1 AMPs on Large Scale Networks

The first contribution of this thesis is *Autonomous Mobile Programs* [35].

The current experiments have been done on small local area networks. We would like to generalise the AMPs architecture to large scale network e.g. WAN, Grid, etc., where the communication time and system architecture are different from our current system. To generalise AMPs on large scale networks, three activities should be considered.

- To propose a new architecture with super load servers.
- To improve the cost model.
- To evaluate the result.

To generalise AMPs on large scale networks, a new architecture with super load servers has been proposed in Figure 7.96. In this architecture, the large scale network has been divided into clusters according to the network latency. In the same cluster the latencies between any two locations are similar. The architecture for the cluster is the same as the architecture which has been shown in Figure 5.45 in Sections 5.2. The communication time is parameterised on the size of data to be send. Every load server collects local information such as relative CPU speed and exchanges information with other servers.

The super load servers are in charge of collecting the information of the cluster and exchanging information with other clusters. The information includes the relative CPU speed of each location and the AMP number in each location in the cluster. The super load server also keep a record of network latencies between the cluster and remote clusters. The super load server passes the information about remote clusters to the load servers in the local cluster. AMPs in the cluster can collect this information from the local load servers, where AMPs are running on them.

The generic cost model for AMPs has been built in Section 3.3.2. This cost model should be extended to consider the information from remote clusters. The

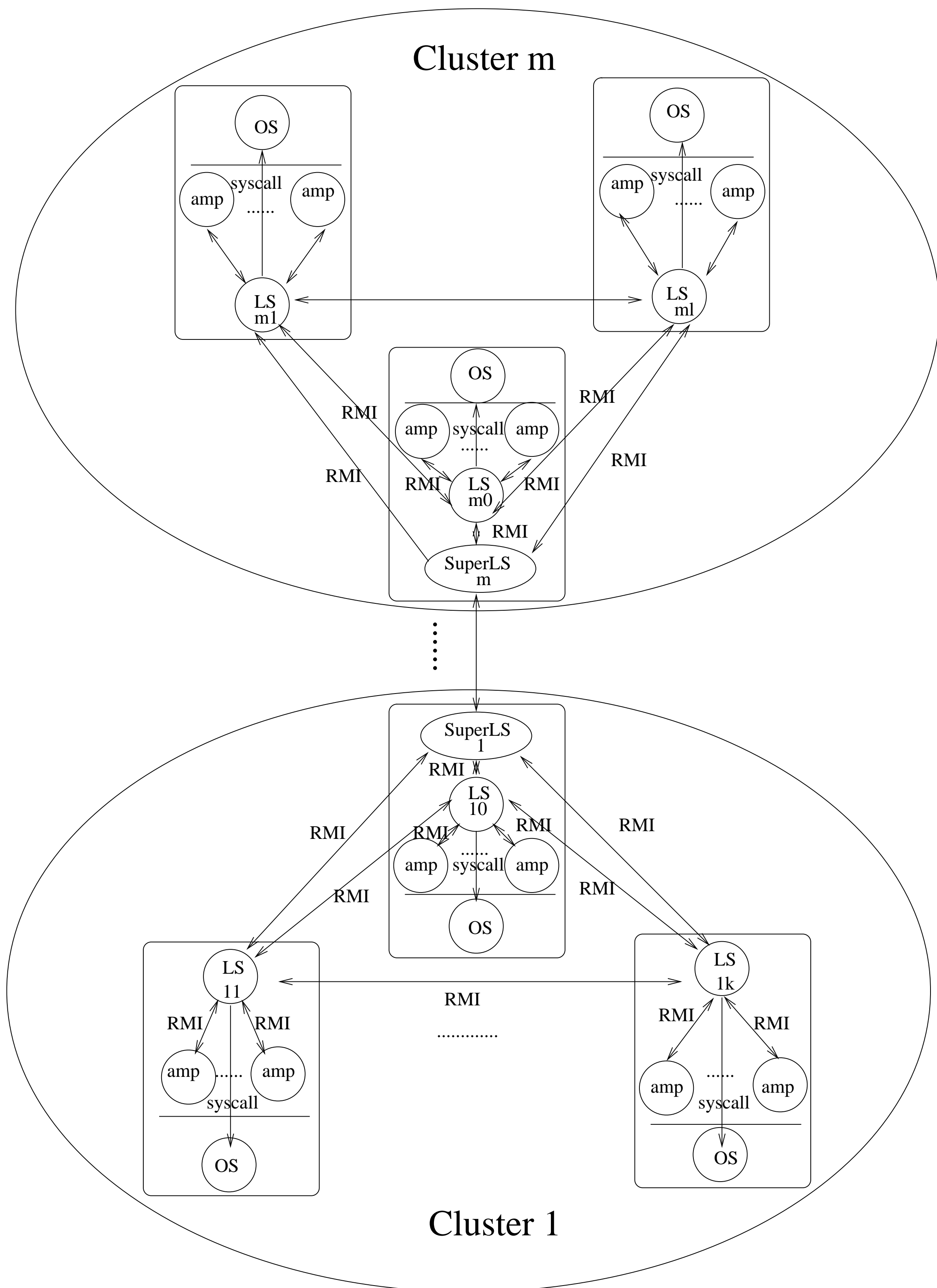


Figure 7.96: System with Super Load Server on Large Scale Network

cost model in Section 3.3.2 compares the execution time for completing the computation at the current location (T_h) with the that at the fastest location in the network (LAN) (T_n). In the new architecture, the cost model should compare the execution time at the current location with the times at the fastest locations in each cluster. T_h is the same as Equation (3.15) in Section 3.3.2, but T_n can be calculated using Equation (7.83).

$$T_n = min\ (T_{n_i} + T_{comm_i}) \quad {}_{i=1...m} \tag{7.83}$$

T_{n_i} is the time to finish at the fastest location in cluster i, which can be got using Equation (3.16) in Section 3.3.2. T_{comm_i} is communication time from current cluster to cluster i. The communication time is parameterised on both the data size and network latency between these two clusters, which can be obtained from the local super load server. The AMP will compare these kinds of time, and choose the fastest location to move to. T_n is the execution time to finish at the fastest location plus the communication time from current cluster to the next cluster. Equation (7.84) gives the condition under which the program will move, i.e. if the time to complete in the current location is more than the time to complete in the remote location.

$$T_h > T_n \tag{7.84}$$

To evaluate the AMP performance in the new architecture, three experiments should be conducted.

- To validate the AMPs speedup against the static programs. See Section 3.4.3.
- To validate single AMP movement. See Section 3.5.
- To validate the load balancing performance for collections of AMPs (Section 5.3). This validation involves much more AMPs and locations than that in Section 5.3, so simulation test may be used.

One advantage of the super load server architecture is that the super load servers in the system are like super-peer in a p2p network[14], which makes it

possible for a super load server to reach every other location in the system, thus realising the concept of a integrated schema formed from all possible information sources. This is achieved by classifying locations into clusters[14]. The other advantage is that the super load server architecture is arranged in different layers, which make it to extend the architecture easily by introducing *super-super load server* etc..

7.2.2 Autonomous Mobility for Irregular Computation

The second contribution is the *Design, Implementation and Evaluation of Autonomous mobility skeletons* [34].

Autonomous mobility skeletons have a limitation because the skeletons dynamically parameterise the cost model with measurements of performance on the preceding collection element. If the program is reasonably regular, i.e. computing each element of the collection represents a similar amount of work, then the cost model will be valid, and hence the movement decisions reasonable. However, as the computations become increasingly irregular, the cost model will be less valid, and hence the movement decisions may not optimise performance. We would like to generalise autonomous mobility skeletons to irregular problems with cost models and strategies to adapt to their behaviour.

New autonomous mobility skeletons should be built to predict the remaining computation time for irregular computation over collections. Two possible approaches can be used when building the new autonomous mobility skeletons.

- The first approach is to take the total computation time for processing n elements instead of one element, then calculate the average time for processing one elements, which is the total for processing n elements divided by n. The average time can be used to predict the computation time for processing the remaining elements.

- Another approach is to take the separate computation times for processing n elements, then compare these times and find their regularity, e.g. the difference between two nearby elements. The difference can be used to

predict the computation time for processing the remaining elements. For example, if the times for processing the first five elements are 1, 2, 3, 4, and 5 seconds, then the time for processing the sixth element can be assumed as 6 seconds.

For validating the new approaches, programs for irregular computation should be developed, and the performance using new approaches should be compared to the performance using the autonomous mobility skeletons in Chapter 4.

7.2.3 Resource Driven Mobility

The third contribution is the *Cost Model* autonomous mobile programs [35, 33].

For further development of AMPs, we would like to build automatic resource driven mobility. We propose to investigate the application of a generic cost-based ethology to autonomous mobile multi-agent system. Specifically, we will use evolved biological foraging strategies to better engineer scalable self-organising resource-location systems in large-scale dynamic networks. The cost model determines whether parametric foraging behaviours can effectively solve the problem of locating 'nearby' resources in a large dynamic environment e.g.

- load management, i.e. foraging for computational resources,
- storage management, i.e. foraging for repository space, and
- distributed information retrieval, i.e. foraging for distributed information.

The actives for building biological foraging strategies are:

1. To develop a cost-model for generic behaviours of collections of mobile agents on large-scale networks. By investigating the properties of groups of agents that interact with one another in specific ways, we will extend significantly our cost model in this thesis, which is based on individual mobile agents, e.g. consider how animals make decisions about moving between patches of their environments differing in resources, when competing with other animals [65].

2. To describe the foraging behaviours as design patterns with an initial realisation as a library of classes in a mobile Java, e.g. Voyager.

3. To evaluate the foraging behaviours in comparison with conventional techniques e.g. load management, storage management, and distributed information retrieval.

7.2.4 Automatic Continuation Cost Analysis for Java

Last but not least the *Automatic Continuation Cost Analysis* is another contribution in this thesis.

We have developed an experimental automatic cost analyser, from a structural operational semantic execution time model, for a substantial Jocaml subset. Java is more mainstream and widely used than Jocaml, and is imperative/Object-Oriented. We would like to build an automatic cost analyser for Java to convert static Java programs into Java AMPs automatically. Three steps should be taken to build the analyser.

- to built a cost calculus for Java,
- to implement the cost calculus in Java, and
- to evaluate.

The cost calculus for Java predicts the costafter of Java program at arbitrary points. For building a cost calculus for Java, there is significant challenge in making models, and hence analyses, of patterns and of Object-Oriented constructs, in particular in the presence of arbitrary inheritance. For making models, the costs of Objects should be taken into account, and the calculation of the costafter might not been exactly determined at compile time, because the cost of some Objects might be known at run time. To solve these problems, we are exploring rewriting the cost calculus in continuation passing style [47], to provide costs for the reminder of a computation at arbitrary points during its execution. Thus the cost of Objects can be calculated at run time rather than at compile time. A

limitation of the proposed approach is that the analysis can provide the cost of the Objects if the source code is available, but not otherwise.

The implementation and the evaluation of the cost analyser will using the similar techniques as in Sections 6.5 and 6.6.

Appendix A

Cost Calculus for $\mathcal{J}$

In Chapter 6 a cost semantics has been built for a very small language $\mathcal{J}'$ which is a subset of Jocaml. Language $\mathcal{J}$ extends $\mathcal{J}'$ with *null, boolean, list, tuple, pattern, compose, if, match*, and *fold*, and facilitates the construction of non-trival programs like matrix multiplication, and ray tracing in Section 6.5. In this appendix the cost sematics of $\mathcal{J}$ is given.

A.1 Syntax of $\mathcal{J}$

Figure A.97 shows the abstract syntax of $\mathcal{J}$. To simplify the presentation it is assumed that variables (*id*) names in the program are unique. In *let*, fun, and *match* expressions, p is pattern, which is also expression.

A.2 Cost Calculus for $\mathcal{J}$

A.2.1 Index Semantics

Figure A.98 shows the semantics of index for $\mathcal{J}$, where $\vdash_i$ is a function, which takes two parameters, an expression and an integer(i), and returns a tuple of index expression and another integer which is i increased by 1. The type of this function is: $\vdash_i : n \rightarrow e \rightarrow e * n$, where n is integer, e is expression.

- Equation (A.85) shows that if the current index is i the expression is a

e ::=		expression
	`null`	`null` expression
	c	constant
	id	variable
	`(`e...e`)`	tuple
	`[ ]`	empty list
	`[`e...e`]`	list
	`fun` $p \rightarrow e$	lambda (abstract)
	e e	application
	e op e	operation
	`let` p `=` e `in` e	`let`
	`if` e `then` e `else` e	condition
	`match` e `with` $p \rightarrow e$ ‖...‖ $p \rightarrow e$	`match`
	`map` e e	`map` (higher order function)
	`fold` e e e	`fold` (higher order function)
	e `(*` e `*)`	user cost
	$<n,e>$	index i.e. expression e has index n, where n is an integer

p (pattern) ::= id | `(`p...p`)` | `[`p...p`]` | p`::`p

op ::= + | - | * | / |
> | < | >= | <= | = | ! = |
:: (cons) |
; (sequence)

Figure A.97: Syntax of $\mathcal{J}$

constant c, after indexing the new expression is $< i, c >$ and the current index becomes $i + 1$.

- Equation (A.86) is similar to Equation (A.85), but the expression is a variable (id) instead of integer.

- Equation (A.87) decorates lambda expressions. In this equation the pattern does not be decorated, only the body of the lambda expression does. For indexing `fun` $p \rightarrow e$, if the current index is i, and after indexing e the current index is i', and e becomes e', the index lambda expression is $< i',$ `fun` $p \rightarrow e' >$ and the current index is $i' + 1$.

- Equation (A.88) decorates application expressions (e_1 e_2). If the current index is i, after indexing e_1 the current index becomes i' and e_1 becomes e_1',

$$\frac{}{i \vdash_i c \Rightarrow_i (< i, c >, i+1)} \quad (A.85)$$

$$\frac{}{i \vdash_i id \Rightarrow_i (< i, id >, i+1)} \quad (A.86)$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i')}{i \vdash_i \texttt{fun}\ p \ \rightarrow \ e \Rightarrow_i (< i', \texttt{fun}\ p \ \rightarrow \ e' >, i'+1)} \quad (A.87)$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i (e_1\ e_2) \Rightarrow_i (< i'', (e_1'\ e_2') >, i''+1)} \quad (A.88)$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i \texttt{let}\ p \ = \ e_1\ \texttt{in}\ e_2 \Rightarrow_i (< i'', \texttt{let}\ p \ = \ e_1'\ \texttt{in}\ e_2' >, i''+1)} \quad (A.89)$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i e_1\ op\ e_2 \Rightarrow_i (< i'', e_1'\ op\ e_2' >, i''+1)} \quad (A.90)$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_n \Rightarrow_i (e_n', i'')}{i \vdash_i [e_1\ ...\ e_n] \Rightarrow_i (< i'', [e_1'\ ...\ e_n'] >, i''+1)} \quad (A.91)$$

$$\frac{}{i \vdash_i [\ \ ] \Rightarrow_i (< i, [\ \ ] >, i+1)} \quad (A.92)$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i')}{i \vdash_i e\ (*\ c\ *) \Rightarrow_i (< i', e'\ (*\ c\ *) >, i'+1)} \quad (A.93)$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'')}{i \vdash_i \texttt{map}\ e_1\ e_2 \Rightarrow_i (< i'', \texttt{map}\ e_1'\ e_2' >, i''+1)} \quad (A.94)$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_n \Rightarrow_i (e_n', i'')}{i \vdash_i (e_1\ ...\ e_n) \Rightarrow_i (< i'', (e_1'\ ...\ e_n') >, i''+1)} \quad (A.95)$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'') \qquad i'' \vdash_i e_3 \Rightarrow_i (e_3', i''')}{i \vdash_i \texttt{fold}\ e_1\ e_2\ e_3 \Rightarrow_i (< i''', \texttt{fold}\ e_1'\ e_2'\ e_3' >, i'''+1)} \quad (A.96)$$

$$\frac{i \vdash_i e_1 \Rightarrow_i (e_1', i') \qquad i' \vdash_i e_2 \Rightarrow_i (e_2', i'') \qquad i'' \vdash_i e_3 \Rightarrow_i (e_3', i''')}{i \vdash_i \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow_i (< i''', \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3' >, i'''+1)} \quad (A.97)$$

$$\frac{i \vdash_i e \Rightarrow_i (e', i') \qquad i' \vdash_i e_1 \Rightarrow_i (e_1', i'') \qquad i'' \vdash_i e_n \Rightarrow_i (e_n', i''')}{i \vdash_i \texttt{match}\ e\ \texttt{with}\ p_1 \rightarrow e_1 \| ... \| p_n \rightarrow e_n \Rightarrow_i (< i''', \texttt{match}\ e'\ \texttt{with}\ p_1 \rightarrow e_1' \| ... \| p_n \rightarrow e_n' >, i'''+1)} \quad (A.98)$$

$$\frac{}{i \vdash_i < i', e > \Rightarrow_i (< i', e >, i)} \quad (A.99)$$

Figure A.98: Index Semantics of $\mathcal{J}$

and then after indexing e_2, the current index is i'' and e_2 becomes e_2', then the index for the application expression is i'' and the current index becomes $i'' + 1$.

- Equation (A.89) decorates `let` expressions using the same rules as in Equation (A.88), Equation (A.90) decorates operation expressions, and Equation (A.91) decorates lists.
- Equation (A.92) indexes the empty list, which is similar to Equation (A.86).
- In Equation (A.93), user cost expressions e $(* \ c \ *)$ are indexed, where only e has been indexed.
- In Equation (A.94), `map` expressions are indexed, which is similar to Equation (A.88).
- Equation (A.95) indexes tuples, which is similar to Equation (A.91).
- Equation (A.96), and Equation (A.97) index `fold` and `if` expressions, which are similar to Equation (A.88).
- Equation (A.98) indexes `match` expressions, which is similar to Equation (A.89) and Equation (A.87), where patterns are not indexed.
- If the expression is an index expression, it should not be indexed again. So in Equation (A.99), the return expression is the index expression, and the current index number is still i.

A.2.2 Cost Semantics

In `let`, `fun`, and `match` expressions, p is pattern, which is also expression, and the syntax of pattern is shown in Figure A.97.

To calculate the cost of `let`, `fun`, and `match` expressions, the cost of pattern should be got first, so this section first presents the pattern cost semantics, and then presents the cost semantics.

Pattern Cost Semantics of $\mathcal{J}$

Figure A.99 shows the semantics of pattern cost, where $\vdash_p$ takes the environment (*env*) and an pattern expression, and returns a new environment which is the old environment extended with the pattern and its cost. The type of this function is: $\vdash_p\ :\ env \rightarrow p \rightarrow env\ *\ cost$, where (*env*) is a semantic domain, which stores the names of variable and the cost of the variable.

$$\frac{}{\mathrm{E} \vdash_p\ id\ \Rightarrow_p\ (\{id, 1\} + \mathrm{E}, 1)} \tag{A.100}$$

$$\frac{\mathrm{E} \vdash_p\ p_1\ \Rightarrow_p\ (\mathrm{E}_1, c_1) \qquad ... \qquad \mathrm{E}_{n-1} \vdash_p\ p_n\ \Rightarrow_p\ (\mathrm{E}_n, c_n)}{\mathrm{E}_{n-1} \vdash_p\ (p_1...p_n)\ \Rightarrow_p\ (\mathrm{E}_n, c_1 + ... + c_n)} \tag{A.101}$$

$$\frac{}{\mathrm{E} \vdash_p\ [\ \ ]\ \Rightarrow_p\ (\mathrm{E}, 0)} \tag{A.102}$$

$$\frac{\mathrm{E} \vdash_p\ p_1\ \Rightarrow_p\ (\mathrm{E}_1, c_1) \qquad ... \qquad \mathrm{E}_{n-1} \vdash_p\ p_n\ \Rightarrow_p\ (\mathrm{E}_n, c_n)}{\mathrm{E}_{n-1} \vdash_p\ [p_1...p_n]\ \Rightarrow_p\ (\mathrm{E}_n, n + c_1 + ... + c_n)} \tag{A.103}$$

$$\frac{\mathrm{E} \vdash_p\ p_1\ \Rightarrow_p\ (\mathrm{E}_1, c_1) \qquad \mathrm{E}_1 \vdash_p\ p_2\ \Rightarrow_p\ (\mathrm{E}_2, c_2)}{\mathrm{E}_{n-1} \vdash_p\ (p_1 :: p_2)\ \Rightarrow_p\ (\mathrm{E}_2, 1 + c_1 + c_2)} \tag{A.104}$$

Figure A.99: Pattern Cost Semantics of $\mathcal{J}$

- Equation (A.100) shows that the pattern cost of *id* is 1. The semantic function $\vdash_p$ returns this cost, and at the same time updates the environment E with $id, 1$.

- Equation (A.101) shows that the pattern cost of tuples. If the cost of the first element of tuple $(p_1...p_n)$ (p_1) is c_1, and the environment E becomes E_1, and the cost of the last element of the tuple (p_n) is c_n, and the environment is updated to E_n, then the pattern cost of the tuple is $c_1 + ... + c_n$, and the environment becomes E_n.

- Equation (A.102) shows that the pattern cost of the empty list is 0.

- Equation (A.103) shows that the pattern cost of lists, which is similar to Equation (A.101).

- Equation (A.104) shows if two patterns p_1 and p_2 are combined together, the total cost is the pattern cost of p_1 (c_1) plus the pattern cost of p_2 (c_1) and plus 1, which is the cost for combining. The environment is updated as well.

Cost Semantics of $\mathcal{J}$

Figure A.100 shows the cost semantics of $\mathcal{J}$, where $\vdash_c$ takes the environment (*env*) and an expression (e), and returns a tuple of the expression and the cost (*cost*) of the expression under the environment. The type of this function is: $\vdash_c$: $env \rightarrow e \rightarrow e * cost$

- Equation (A.105) infers that the cost of an constant is 0 in environment E.

- Equation (A.106) shows that the cost of the value of variable, here c, has been stored in the environment, if the cost of a variable is needed, we need to look up the environment $\{id, c\} + \mathrm{E}$.

- Equation (A.107) performs the cost of operation expressions (e_1 *op* e_2). So if the cost of e_1 is c_1, and the cost of e_2 is c_2 then the cost of e_1 *op* e_2 is $1 + c_1 + c_2$.

- Equation (A.108) performs the cost of lambda expressions `fun` $p \rightarrow e$, which have two part, the cost of pattern p (c_p), and the cost of the body e. The cost of pattern p can be get using the pattern cost semantics. Under the environment E if the cost of e is c, then the cost of `fun` $p \rightarrow e$ is $c_p + c$.

- Equation (A.109) shows that under environment E the cost of `let` expression `let` $p = e_1$ `in` e_2 has three parts, the cost of e_1 (c_1), the cost of e_2 (c_2), and the cost of p (c_p).

- Equation (A.110) shows that the cost of application expression ($e_1 \;\; e_2$) is the cost of e_1 (c_1) and the cost of e_2 (c_2).

- In any environment E the cost of empty list [ ] is 0, as shown in Equation (A.111).

$$\overline{\mathrm{E} \vdash_c \ c \,\$\, 0} \tag{A.105}$$

$$\overline{\{id, c\} + \mathrm{E} \vdash_c \ id \,\$\, c} \tag{A.106}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \ e_1 \ op \ e_2 \,\$\, 1 + c_1 + c_2} \tag{A.107}$$

$$\frac{\mathrm{E} \vdash_p \ p \Rightarrow_p (\mathrm{E}_1, c_p) \qquad \mathrm{E}_1 \vdash_c \ e \,\$\, c}{\mathrm{E} \vdash_c \ \mathtt{fun}\ p \ \rightarrow \ e \,\$\, c_p + c} \tag{A.108}$$

$$\frac{\mathrm{E} \vdash_p \ p \Rightarrow_p (\mathrm{E}_1, c_p) \qquad \mathrm{E}_1 \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \ \mathtt{let}\ p \ = \ e_1 \ \mathtt{in}\ e_2 \,\$\, c_p + c_1 + c_2} \tag{A.109}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \ (e_1 \ e_2) \,\$\, c_1 + c_2} \tag{A.110}$$

$$\overline{\mathrm{E} \vdash_c \ [\ \ ] \,\$\, 0} \tag{A.111}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad ... \qquad \mathrm{E} \vdash_c \ e_n \,\$\, c_n}{\mathrm{E} \vdash_c \ [e_1 ... e_n] \,\$\, c_1 + ... + c_n} \tag{A.112}$$

$$\overline{\mathrm{E} \vdash_c \ e \ (*\ c\ *) \,\$\, c} \tag{A.113}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2}{\mathrm{E} \vdash_c \ \mathtt{map}\ e_1 \ e_2 \,\$\, c_1 * (length \ e_2) + c_2} \tag{A.114}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad ... \qquad \mathrm{E} \vdash_c \ e_n \,\$\, c_n}{\mathrm{E} \vdash_c \ (e_1 ... e_n) \,\$\, c_1 + ... + c_n} \tag{A.115}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2 \qquad \mathrm{E} \vdash_c \ e_3 \,\$\, c_3}{\mathrm{E} \vdash_c \ \mathtt{fold}\ e_1 \ e_2 \ e_3 \,\$\, c_1 * (length \ e_2) + c_2 + c_3} \tag{A.116}$$

$$\frac{\mathrm{E} \vdash_c \ e_1 \,\$\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2 \qquad \mathrm{E} \vdash_c \ e_3 \,\$\, c_3}{\mathrm{E} \vdash_c \ \mathtt{if}\ e_1 \ \mathtt{then}\ e_2 \ \mathtt{else}\ e_3 \,\$\, c_1 + max(c_2 + c_3)} \tag{A.117}$$

$$\frac{\mathrm{E} \vdash_c \ e \,\$\, c \ \ \mathrm{E} \vdash_p \ p_1 \Rightarrow_p (\mathrm{E}_1, c_1) \ \ \mathrm{E}_1 \vdash_c \ e_1 \,\$\, c'_1 \ \ ... \ \ \mathrm{E}_n \vdash_p \ p_n \Rightarrow_p (\mathrm{E}_n, c_n) \ \ \mathrm{E}_n \vdash_c \ e_n \,\$\, c'_n}{\mathrm{E} \vdash_c \ \mathtt{match}\ e \ \mathtt{with}\ p_1 \rightarrow e_1 \| ... \| p_n \rightarrow e_n \,\$\, c + c_1 + ... + c_n + max(c'_1 ... c'_n)} \tag{A.118}$$

$$\frac{\mathrm{E} \vdash_c \ e \,\$\, c}{\mathrm{E} \vdash_c < i, e > \ \$\, c} \tag{A.119}$$

Figure A.100: Cost Semantics of $\mathcal{J}$

- Equation (A.112) shows that under environment E the cost of list $[e_1...e_n]$ is the sum of the costs of every elements in the list.

- Equation (A.113) enables user specified costs. In particular, it is difficult to get the static cost of recursive functions. So in e $(* \ c \ *)$, the cost of e is c which is calculated by hand rather by the cost analyser automatically. The cost is passed to the cost analyser.

- Equation (A.114) infers the cost of a `map` expression under environment E. Here only the regular cases of `map` are considered. Under this condition function e_1 applied to every elements of list e_2 has the same cost. So the total cost of a `map` expression is $c_1 * (length\ e_2) + c_2$, where c_1 is the cost of function e_1 applied to the first element of list e_2.

- Equation (A.115) shows the cost of tuples, which is similar to the cost of lists in Equation (A.112).

- Equation (A.116) shows the cost of `fold` expressions, which is similar to the cost of `map` expressions in Equation (A.114).

- Equation (A.117) finds the cost of `if` expression `if` e_1 `then` e_2 `else` e_3. If the cost of e_1 is c_1, the cost of e_2 is c_2, and the cost of e_3 is c_3, then the cost of the if expression is c_1 plus the maximum of c_2 and c_3.

- Equation (A.118) finds the cost of a `match` expression `match` e `with` $p_1 \ \rightarrow \ e_1 \ \| \ ... \ \| \ p_n \ \rightarrow \ e_n$. Firstly the cost of patterns from p_1 to p_n are calculated as c_1 to c_n, and at the same time the environment has been updated according to the pattern semantics in Figure A.99. Then the cost of each expression from e_1 to e_n should be got as c'_1 to c'_n. The cost of the `match` expression is the sum of c_1 to c_n plus the maximum of c'_1 to c'_n.

- Equation (A.119) shows that under environment E the cost of an index expression $< i, e >$ is the cost of expression e (c).

A.2.3 Costafter Semantics

This section introduces the costafter semantics of $\mathcal{J}$. In costafter, definitions of expression equality and expression contains will be used.

Expression Equality

Figure A.101 shows the definition of expression equality, where $\equiv$ applies expression equality, which checks if two expressions are the same, presented as: $\equiv : e \rightarrow e \rightarrow boolean$.

- Equation (A.120) defines constant expression equality. If expression e is an constant and its value is equal to c, so expression e is equal to expression c.
- Equation (A.121) defines variable equality. In $\mathcal{J}$, it is assumed that variables (id) names in the program are unique, so if two variables expressions have the same name, the two expression are equal.
- Equation (A.122) defines the equality of two operation expressions. Expression (e_1 *op* e_2) equals expression (e_3 **op** e_4), if e_1 equals e_3 and e_2 equals e_4.
- Equation (A.123) defines the equality of two lambda expression. Expression `fun` ($p_1 \rightarrow e_1$) equals expression `fun` ($p_2 \rightarrow e_2$), if pattern p_1 equals pattern p_2 and expression e_1 equals e_2. The patterns are also expressions, so the expression equality can also be used for patterns.
- Equation (A.124), Equation (A.125), and Equation (A.126) define application expressions, let expressions and `map` expressions equality, which are all similar to Equation (A.123).
- Equation (A.127) shows that two list expressions are equal, if the elements in one list are equal to the elements in the other list.
- Equation (A.128) shows that empty lists are always equal.

$$\frac{e = c}{e \equiv c} \tag{A.120}$$

$$\frac{e = id}{e \equiv id} \tag{A.121}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(e_1\ op\ e_2) \equiv (e_3\ op\ e_4)} \tag{A.122}$$

$$\frac{p_1 \equiv p_2 \qquad e_1 \equiv e_2}{(\texttt{fun}\ p_1 \rightarrow e_1) \equiv (\texttt{fun}\ p_2 \rightarrow e_2)} \tag{A.123}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(e_1\ e_2) \equiv (e_3\ e_4)} \tag{A.124}$$

$$\frac{p_1 \equiv p_2 \qquad e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(\texttt{let}\ p_1\ =\ e_1\ \texttt{in}\ e_2) \equiv (\texttt{let}\ p_2\ =\ e_3\ \texttt{in}\ e_4)} \tag{A.125}$$

$$\frac{e_1 \equiv e_3 \qquad e_2 \equiv e_4}{(\texttt{map}\ e_1\ e_2) \equiv (\texttt{map}\ e_3\ e_4)} \tag{A.126}$$

$$\frac{e_i \equiv e'_i \qquad i = 1...n}{[e_1...e_n] \equiv [e'_1...e'_n]} \tag{A.127}$$

$$\frac{}{[\ \ ] \equiv [\ \ ]} \tag{A.128}$$

$$\frac{e_i \equiv e'_i \qquad i = 1...n}{(e_1...e_n) \equiv (e'_1...e'_n)} \tag{A.129}$$

$$\frac{e_1 \equiv e_4 \qquad e_2 \equiv e_5 \qquad e_3 \equiv e_6}{(\texttt{fold}\ e_1\ e_2\ e_3) \equiv (\texttt{fold}\ e_4\ e_5\ e_6)} \tag{A.130}$$

$$\frac{e_1 \equiv e_4 \qquad e_2 \equiv e_5 \qquad e_3 \equiv e_6}{(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \equiv (\texttt{if}\ e_4\ \texttt{then}\ e_5\ \texttt{else}\ e_6)} \tag{A.131}$$

$$\frac{e \equiv e' \qquad p_i \equiv p'_i \qquad e_i \equiv e'_i \qquad i = 1..n}{(\texttt{match}\ e\ \texttt{with}\ p_1 \rightarrow e_1 \| ... \| p_n \rightarrow e_n) \equiv (\texttt{match}\ e'\ \texttt{with}\ p'_1 \rightarrow e'_1 \| ... \| p'_n \rightarrow e'_n)} \tag{A.132}$$

$$\frac{e_1 \equiv e_2 \qquad c_1 \equiv c_2}{(e_1\ (*\ c_1\ *)) \equiv (e_2\ (*\ c_2\ *))} \tag{A.133}$$

$$\frac{}{< i, e_1 > \equiv < i, e_2 >} \tag{A.134}$$

Figure A.101: Definition of Expression Equality in $\mathcal{J}$

- Equation (A.129) defines the expression equality for tuples, which is similar to lists equality in Equation (A.127).
- Equation (A.130) defines the expression equality for `fold` expressions, which is similar to Equation (A.123).
- Equation (A.131) defines the expression equality for if expressions,
- and Equation (A.132) defines the expression equality for `match` expressions, which are all similar to Equation (A.123).
- In Equation (A.133), two user cost expression are equal, if both the expression parts and cost parts of the expressions are equal.
- In the calculus, the whole program has been indexed, so every expression in a program has a unique integer as an index, thus two index expressions are equal to each other only if their indices are equal.

Expression Contains

Figure A.102 and A.103 give the definition of contains, where $\in$ takes two expression. If the second expression contains the first expression it returns true, or it returns false if not.

- Equation (A.135) identifies that if expression e_1 equal to expression e_2 then the two expressions contain each other.
- Equation (A.136) shows that if e_1 is contained in e_2, then e_1 is contained in `fun` $p \rightarrow e2$.
- Equation (A.137a) to (A.147c) give the definition of contains for different expressions, which are all similar to Equation (A.136).

Semantics of Costafter

Figure A.104 and A.105 show the semantics of costafter of one expression e.g e in different expressions, where $\vdash_a$ takes the environment (env) and two expressions.

$$\frac{e_1 \equiv e_2}{e_1 \in e_2} \tag{A.135}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{fun}\ p \rightarrow e_2} \tag{A.136}$$

$$\frac{e_1 \in e_2}{e_1 \in (e_2\ e_3)} \tag{A.137a}$$

$$\frac{e_1 \in e_3}{e_1 \in (e_2\ e_3)} \tag{A.137b}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{let}\ p = e_2\ \texttt{in}\ e_3} \tag{A.138a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{let}\ p = e_2\ \texttt{in}\ e_3} \tag{A.138b}$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ op\ e_3} \tag{A.139a}$$

$$\frac{e_1 \in e_3}{e_1 \in e_2\ op\ e_3} \tag{A.139b}$$

$$\frac{e_1 \in e_i \qquad i = 2..n}{e_1 \in [e_2\ ...\ e_n]} \tag{A.140}$$

$$\frac{e_1 \in e_2}{e_1 \in e_2\ (*\ c\ *)} \tag{A.141}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{map}\ e_2\ e_3} \tag{A.142a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{map}\ e_2\ e_3} \tag{A.142b}$$

$$\frac{e_1 \in e_2}{e_1 \in < i, e_2 >} \tag{A.143}$$

$$\frac{e_1 \in e_i \qquad i = 2..n}{e_1 \in (e_2\ ...\ e_n)} \tag{A.144}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{fold}\ e_2\ e_3\ e_4} \tag{A.145a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{fold}\ e_2\ e_3\ e_4} \tag{A.145b}$$

$$\frac{e_1 \in e_4}{e_1 \in \texttt{fold}\ e_2\ e_3\ e_4} \tag{A.145c}$$

Figure A.102: Definition of Contains in $\mathcal{J}$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{if } e_2 \texttt{ then } e_3 \texttt{ else } e_4} \tag{A.146a}$$

$$\frac{e_1 \in e_3}{e_1 \in \texttt{if } e_2 \texttt{ then } e_3 \texttt{ else } e_4} \tag{A.146b}$$

$$\frac{e_1 \in e_4}{e_1 \in \texttt{if } e_2 \texttt{ then } e_3 \texttt{ else } e_4} \tag{A.146c}$$

$$\frac{e_1 \in e_2}{e_1 \in \texttt{match } e_2 \texttt{ with } p_3 \rightarrow e_3 \| ... \| p_n \rightarrow e_n} \tag{A.147a}$$

$$\frac{e_1 \in p_i \qquad i = 3...n}{e_1 \in \texttt{match } e_2 \texttt{ with } p_3 \rightarrow e_3 \| ... \| p_n \rightarrow e_n} \tag{A.147b}$$

$$\frac{e_1 \in e_i \qquad i = 3...n}{e_1 \in \texttt{match } e_2 \texttt{ with } p_3 \rightarrow e_3 \| ... \| p_n \rightarrow e_n} \tag{A.147c}$$

Figure A.103: Definition of Contains in $\mathcal{J}$ Cont.

The return value of $\vdash_a$ is a cost, which is the costafter of the first expression in the second expression. The type of this function is: $\vdash_a \, : \, env \rightarrow e \rightarrow e \rightarrow cost$

(1) Equation (A.148) states that `if` expression e equal to e' then the costafter of e in e' is 0.

(2) Equations (A.149a) and (A.149b) define the costafter of e in lambda expressions. If the costafter of e in e_1 is c, then the costafter of e in lambada expression `fun` $p \rightarrow e_1$ is c too. If e_1 does not contains e then the costafter of e in `fun` $p \rightarrow e_1$ is c is 0.

(3) Equations (A.150a), (A.150b), and (A.150c) define the costafter of e in application expressions e.g. $(e_1\ e_2)$. If e_1 contains e, then the costafter of e in $(e_1\ e_2)$ is the costafter of e in e_1, here is c_1, plus the cost of e_2, here is c_2. If e_2 contains e then the costafter of e in $(e_1\ e_2)$ is the cost after e in e_2. If e does not contains in e_1 or e_2, then the costafter of e in $(e_1\ e_2)$ 0.

(4) Similar to the equations in costafter rule (3), Equation (A.151a), (A.151b), (A.151c), and (A.151d) define the costafter of e in `let` expression $let\ p = e_1\ in\ e_2$.

(5) Equations (A.152a), (A.152b), and (A.152c) define the costafter of e in operation expressions e.g. $(e_1\ op\ e_2)$. If e_1 contains e, then the costafter of

$$\frac{e \;\equiv\; e'}{\mathrm{E} \vdash_a e \trianglelefteq e' \,£\, 0} \tag{A.148}$$

$$\frac{\mathrm{E} \vdash_a e \trianglelefteq e_1 \,£\, c}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{fun}\; p \;\rightarrow\; e_1 \,£\, c} \tag{A.149a}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{fun}\; p \;\rightarrow\; e_1 \,£\, 0} \tag{A.149b}$$

$$\frac{e \in e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \,£\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; e_2) \,£\, c_1 + c_2} \tag{A.150a}$$

$$\frac{e \in e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,£\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; e_2) \,£\, c_2} \tag{A.150b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; e_2) \,£\, 0} \tag{A.150c}$$

$$\frac{\mathit{notFun}\; e_1 \qquad e \in e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \,£\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{let}\; p \;= e_1 \;\texttt{in}\; e_2 \,£\, c_1 + c_2} \tag{A.151a}$$

$$\frac{\mathit{notFun}\; e_1 \qquad e \in e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,£\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{let}\; p \;= e_1 \;\texttt{in}\; e_2 \,£\, c_2} \tag{A.151b}$$

$$\frac{e \in e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,£\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{let}\; p \;= e_1 \;\texttt{in}\; e_2 \,£\, c_2} \tag{A.151c}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq \texttt{let}\; p \;= e_1 \;\texttt{in}\; e_2 \,£\, 0} \tag{A.151d}$$

$$\frac{e \in e_1 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_1 \,£\, c_1 \qquad \mathrm{E} \vdash_c e_2 \,\$\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; \mathit{op}\; e_2) \,£\, 1 + c_1 + c_2} \tag{A.152a}$$

$$\frac{e \in e_2 \qquad \mathrm{E} \vdash_a e \trianglelefteq e_2 \,£\, c_2}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; \mathit{op}\; e_2) \,£\, c_2 + 1} \tag{A.152b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq (e_1\; \mathit{op}\; e_2) \,£\, 0} \tag{A.152c}$$

$$\frac{e \in e_i \qquad \mathrm{E} \vdash_a e \trianglelefteq e_i \,£\, c_i \qquad \mathrm{E} \vdash_c e_{i+1} \,\$\, c_{i+1} \qquad \ldots \qquad \mathrm{E} \vdash_c e_n \,\$\, c_n}{\mathrm{E} \vdash_a e \trianglelefteq [e_1 \ldots e_n] \,£\, c_i + \;\ldots\; + c_n} \tag{A.153a}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq [e_1 \ldots e_2] \,£\, 0} \tag{A.153b}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq [\;\;] \,£\, 0} \tag{A.154}$$

$$\frac{}{\mathrm{E} \vdash_a e \trianglelefteq e_1 \; (* \; c \; *) \,£\, 0} \tag{A.155}$$

Figure A.104: Costafter Semantics of $\mathcal{J}$

$$\frac{e \ \in \ e_1 \qquad \mathrm{E} \vdash_c \texttt{map}\ e_1\ e_2 \,\$\, c \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_1 \,\pounds\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{map}\ e_1\ e_2 \,\pounds\, c + c_1 + c_2} \tag{A.156a}$$

$$\frac{e \ \in \ e_2 \qquad \mathrm{E} \vdash_c \texttt{map}\ e_1\ e_2 \,\$\, c \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_2 \,\pounds\, c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{map}\ e_1\ e_2 \,\pounds\, c + c_2} \tag{A.156b}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{map}\ e_1\ e_2 \,\pounds\, 0} \tag{A.156c}$$

$$\frac{\mathrm{E} \vdash_a \ e \trianglelefteq e_1 \,\pounds\, c}{\mathrm{E} \vdash_a \ e \trianglelefteq \ < i, e_1 > \ \pounds\, c} \tag{A.157}$$

$$\frac{e \ \in \ e_i \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_i \,\pounds\, c_i \qquad \mathrm{E} \vdash_c \ e_{i+1} \,\$\, c_{i+1} \qquad ... \qquad \mathrm{E} \vdash_c \ e_n \,\$\, c_n}{\mathrm{E} \vdash_a \ e \trianglelefteq (e_1 \ ... \ e_n) \,\pounds\, c_i + \ ... \ + c_n} \tag{A.158a}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq (e_1 \ ... \ e_n) \,\pounds\, 0} \tag{A.158b}$$

$$\frac{e \ \in \ e_1 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_1 \,\pounds\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2 \qquad \mathrm{E} \vdash_c \ e_3 \,\$\, c_3}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,\pounds\, c_1 + max(c_2, c_3)} \tag{A.159a}$$

$$\frac{e \ \in \ e_2 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_2 \,\pounds\, c_2}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,\pounds\, c_2} \tag{A.159b}$$

$$\frac{e \ \in \ e_3 \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_3 \,\pounds\, c_3}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,\pounds\, c_3} \tag{A.159c}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \,\pounds\, 0} \tag{A.159d}$$

$$\frac{e \ \in \ e_1 \qquad \mathrm{E} \vdash_c \texttt{fold}\ e_1\ e_2\ e_3 \,\$\, c \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_1 \,\pounds\, c_1 \qquad \mathrm{E} \vdash_c \ e_2 \,\$\, c_2 \qquad \mathrm{E} \vdash_c \ e_3 \,\$\, c_3}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{fold}\ e_1\ e_2\ e_3 \,\pounds\, c + c_1 + c_2 + c_3} \tag{A.160a}$$

$$\frac{e \ \in \ e_2 \qquad \mathrm{E} \vdash_c \texttt{fold}\ e_1\ e_2\ e_3 \,\$\, c \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_2 \,\pounds\, c_2 \qquad \mathrm{E} \vdash_c \ e_3 \,\$\, c_3}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{fold}\ e_1\ e_2\ e_3 \,\pounds\, c + c_2 + c_3} \tag{A.160b}$$

$$\frac{e \ \in \ e_3 \qquad \mathrm{E} \vdash_c \texttt{fold}\ e_1\ e_2\ e_3 \,\$\, c \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_3 \,\pounds\, c_3}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{fold}\ e_1\ e_2\ e_3 \,\pounds\, c + c_3} \tag{A.160c}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{fold}\ e_1\ e_2\ e_3 \,\pounds\, 0} \tag{A.160d}$$

$$\frac{e \ \in \ e' \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e' \,\pounds\, c' \qquad \mathrm{E} \vdash_p \ p_i \Rightarrow_p (\mathrm{E}_i, c_i) \qquad \mathrm{E} \vdash_c \ e_i \,\$\, c'_i \qquad i = 1..n}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{match}\ e'\ \texttt{with}\ p_1 \rightarrow e_1 \ \|...\| \ p_n \rightarrow e_n \,\pounds\, c' + c_1 + ... + c_n + max(c'_1 ... c'_n)} \tag{A.161a}$$

$$\frac{e \ \in \ e_i \qquad \mathrm{E} \vdash_a \ e \trianglelefteq e_i \,\pounds\, c_i}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{match}\ e'\ \texttt{with}\ p_1 \rightarrow e_1 \ \|...\| \ p_n \rightarrow e_n \,\pounds\, c_i} \tag{A.161b}$$

$$\frac{}{\mathrm{E} \vdash_a \ e \trianglelefteq \texttt{match}\ e'\ \texttt{with}\ p_1 \rightarrow e_1 \ \|...\| \ p_n \rightarrow e_n \,\pounds\, 0} \tag{A.161c}$$

Figure A.105: Costafter Semantics of $\mathcal{J}$ Cont.

e in (e_1 e_2) is the costafter of e in e_1 (c_1), plus the cost of e_2 (c_2) plus 1, which is the cost for getting the operator, so is the 1 in Equation (A.152b).

(6) Equation (A.153a) defines that the costafter of e in list $[e_1...e_n]$, if e contains in e_i (i=1..n), then the costafter of e in the list is the costafter of e in e_i, plus the costs of all elements in the list after e_i ($e_{i+1}...e_n$). If e does not contains in any element in the list, the costafter of e in the list is 0. See Equation (A.153b).

(7) According to Equation (A.154), the costafter of any expression e in empty lists is 0.

(8) The user cost expression give the cost of recursive function, so the costafter of any expression in a user cost expressions is 0, see Equation (A.155).

(9) Equations (A.156a), (A.156b), and (A.156c) define the costafter of e in `map` expression `map` e_1 e_2. Equation (A.156a) shows that if e_1 contains e, then the costafter of e in `map` e_1 e_2 is the costafter of e in e_1, plus the cost of e_2, and plus the cost of `map` expression, because after we get e_1 and e_2 the `map` expression will be executed. The similar situation of e_2 containing e is defined in Equation (A.156b). Equation (A.156c) shows if e is not contained in e_1 or e_2 then the costafter of e in `map` e_1 e_2 is 0.

(10) Equation (A.157) shows that the costafter of e in index expression $< i, e_1 >$ is the same as the costafter of e in expression e_1.

(11) Equation (A.158a) and Equation (A.158b) define the costafters in tuples, which are similar to the equations for lists in Equation (A.153a) and (A.153b).

(12) Equation (A.159a) to Equation (A.159d) define the costafter of e in `if` expressions. If e is in e_1 and the the costafter of e in e_1 is c_1, then the costafter of e in the `if` expression is c plus the maximum cost of e_2 (c_2) and e_3 (c_3). See Equation (A.159a). If e is in e_2 and the the costafter of e in e_2 is c_2, then the costafter of e in the `if` expression is c_2. See Equation (A.159b).

Similarly, if e is in e_3 and the the costafter of e in e_3 is c_3, then the costafter of e in the `if` expression is c_3. See Equation (A.159c).

(13) Equation (A.160a) to Equation (A.160d) give the costafter e in `fold` expressions, which are similar to the costafter in *map* expression in Equation (A.156a), (A.156b), and (A.156c).

(14) Equation (A.161a) to Equation (A.161c) define the costafters of e in `match` expressions, which are similar to the costafter in `if` expression in Equation (A.159a), (A.159b), (A.159c), and (A.159d).

Appendix B

Autonomous Mobility Skeletons Code

In this appendix, Section B.1 gives the Jocaml code of autonomous mobility skeletons, Section B.2 gives the Voyager code of autonomous mobility skeletons, and Section B.3 gives the JavaGo code of `AutoIterator`.

B.1 Jocaml Autonomous Mobility Skeletons

```
(* *************************************** *)
(*   AN IMPLEMENTATION OF AMSs IN JOCAML   *)
(* *************************************** *)


(* Globle Data and Structure *)
type server = {
                mutable servername : string; mutable running : bool;
                mutable speed:float; mutable cpunum  : float;
                mutable load:float;mutable relspeed : float
};;  (* New type to store the information of locations*)


let hostlist = [
{servername="ncc1710";running=false;speed=0.;cpunum=1.;load=1.;relspeed=0.};
```

```
{servername="ncc1711";running=false;speed=0.;cpunum=1.;load=1.;relspeed=0.};
{servername="lxtrinder";running=false;speed=0.;cpunum=1.;load=1.;relspeed=0.};
];;

let current = ref {servername=Unix.gethostname();
            running=true ;speed=0.0;cpunum=1.;load=1.;relspeed=0.5
};;

let  numofhost = List.length hostlist;;

(* ************************** *)
(*    Auxiliary functions      *)
(* ************************** *)
let check_running hn =
   let running = Unix.system
                  ("rsh "^hn.servername^" ps -A | grep mobilesev")
   in
   match running with
     (WEXITED s) -> if s = 1
                    then ( printf "WEXITED=\%d" s)
                    else (
                          hn.running <-true;
                         )
   | (WSIGNALED s) -> printf "WSIGNALED  %d" s
   | (WSTOPPED s) ->  printf "WSTOPPED %d" s;
;;

(* check the CPU speed of server hn *)
let check_speed hn =
  let buff = String.create 40 in
  let _ = Unix.system ("rsh "^hn.servername^
```

```
          " cat /proc/cpuinfo |grep MHz|wc -l >> "^"cpuspeed.tmp")
  in ();
  let _ = Unix.system ("rsh "^hn.servername^
         " cat /proc/cpuinfo |grep MHz >> "^"cpuspeed.tmp")
  in ();
  let fd_read = Unix.openfile ("cpuspeed.tmp") [O_RDWR] 777
  in
  let _ = Unix.read fd_read buff 0 40
  in  Unix.ftruncate fd_read 0;
      Unix.close fd_read;
  let indcpuNum = String.index buff 'c' in
  let cpuNum = String.sub buff (indcpuNum-2) 2 in
  let indspeed = String.index buff ':' in
  let speedString = String.sub buff (indspeed+2) 7 in
  let speed = float_of_string speedString in
  hn.speed <-speed;
  let cpunum = float_of_string cpuNum in
  hn.cpunum <- cpunum;
;;

(* check the relative CPU speed of server hn *)
let check_relspeed current hn =
  let getselfcpu () =
    let buff = String.create 20
    in
    let pid = string_of_int (Unix.getpid())
    in
    let _ = Unix.system ("ps -p "^pid^" -opcpu >> "^"relspeed.tmp") in ();
    let fd = Unix.openfile ("relspeed.tmp") [O_RDWR] 777 in
    let _ = Unix.read fd buff 0 20 in ();
    Unix.ftruncate fd 0;
```

```
    Unix.close fd;
    let idString = String.sub buff 5 4 in
    let id =  float_of_string idString in
    id
  in
  let buff = String.create 20 in
  let _ = Unix.system ("rsh "^hn.servername^
          " /u1/pg/xyd3/showcpu >> "^"relspeed.tmp") in ();
  let fd = Unix.openfile ("relspeed.tmp") [O_RDWR] 777
  in
  let _ = Unix.read fd buff 0 20
  in ();
  Unix.ftruncate fd 0;
  Unix.close fd;
  let loadString = String.sub buff 0 5
  in
  let load =
    let load = float_of_string loadString in
    if load > (hn.cpunum*. 100.) then (100.*.hn.cpunum)
                             else load
  in
  if hn.servername = current.servername
    then hn.load <- load -. (getselfcpu())
    else hn.load <- load;
  hn.relspeed <- (hn.cpunum -.(hn.load/.100.0)) *. hn.speed/.hn.cpunum;
;;

let tcomm n =
       let t_c_v = if n<100
                  then 0.0
                  else 1.87e-6 *. float(n*n)
```

```
        and t_c_c =  0.082 in
        t_c_c +. t_c_v
;;

let  tcoord p = 0.44 *. (float p);;

let move nserver =
  let host = Ns.lookup (nserver.servername) vartype in
  go host;;

let timedapply f h =
  let time1 = Unix.gettimeofday() in
  let fh = f h in
  let time2 = Unix.gettimeofday() in
  (fh,time2-.time1)
;;

let rec check_next current hostl =
  match hostl with
  [] -> current |
  hosth::hostt ->
    if (current.relspeed > hosth.relspeed)
    then check_next current hostt
    else check_next hosth hostt
;;

let check_move work workleft fhtime=
  let t_comm = tc work
  and t_h = fhtime *. (float (workleft))
  in
  map (check_relspeed cur) hostlist;
```

```
  let host_next = check_next cur hostlist in
  let t_n =
    if  (cur.servername <> host_next.servername)
    then (cur.relspeed) /. (host_next.relspeed) *.t_h +. t_c
    else t_h
  in
    if (t_h > t_n) && (cur.servername <> host_next.servername)
    then (
      move host_next;
      current := host_next
    )
    else cur
;;

let getGran work f h =
  let (fh,fhtime) = timedapply f h
  in
  let t_static = fhtime *. (float (work))
  and t_coord = tcoord (numofhost)
  in
  let times = Pervasives.truncate((5./.100.*.t_static)/.t_coord)
  in
  let gran =
    if times > 0
    then work/(times+1)+1
    else work+1
    in (fh,fhtime,gran)
;;

let getInfo work workleft gran fhtime f h=
  let cur' = check_move work workleft fhtime
```

```
  and (fh',fhtime',gran') = getGran work f h
  in
  (cur', gran', fhtime',fh')
;;

let init () =
  map check_running hostlist;
  check_speed current ;
  map check_speed hostlist ;
  check_relspeed current current ;
  map (check_relspeed current) hostlist ;
;;

let rec take n l =
   match n,l with
     _,[] -> [] |
     0,_ -> [] |
     m,h::t -> h::take (m-1) t
;;

let rec drop n l =
  match n,l with
    _,[] -> [] |
    0,l  -> l |
    m,h::t -> drop (m-1) t
;;

(* *************************************** *)
(*               automap                   *)
(* *************************************** *)
let rec automap' work workleft gran fhtime f l =
```

```
  match l with
  [] -> [] |
  hd::tl ->
   let xs = List.map f (take (gran-1) l)
   and (l') = drop (gran-1) l  in
   match l' with
   []->[]|
   h::t ->
    let (gran', fhtime',fh') = getInfo work workleft gran fhtime f h
    in xs@(fh'::automap' work (workleft-gran) gran' fhtime' f t)
;;

let automap f l =
  match l with
    [] -> [] |
    h::t ->
      let work = List.length l
      in let (fh,fhtime,gran) = getGran work f h
      in fh::automap' work (work-1) gran fhtime f t;;

(* ************************************** *)
(*                autofold                *)
(* ************************************** *)
let rec autofold' work workleft gran fhtime f accu l =
  match l with
  [] -> [] |
  _ ->
   let xs = foldl f accu (take (gran-1) l)
   and (h::t) = drop (gran-1) l
   in let (gran', fhtime',fh') = getInfo work workleft gran fhtime (f xs) h
   in autofold' work (workleft-gran) gran' fhtime' f fh' t;;
```

```
let autofold cur f accu l =
  match l with
    [] -> [] |
    h::t ->
    let work = length l
    in let (fh,fhtime,gran) = getGran work (f accu) h
    in fh::autofold' work (work-1) gran fhtime f accu t;;
```

B.2 Voyager Autonomous Mobility Skeletons

B.2.1 `Auto` Class Implementation

```
/* ******************************************** */
/*   AN IMPLEMENTATION OF AMSs IN JAVA VOYAGER  */
/* ******************************************** */

import java.io.*;
import java.util.*;
import java.lang.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.awt.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.mobility.*;
import com.objectspace.lib.util.*;

public class Auto{

   /* Globle Data and Structure */
   protected String localName = getLocalName();
   protected Server current = new Server(localName) ;
```

```
protected Server next = new Server() ;
protected ILoadServer ls = null;
protected float genCpu=0;
protected Server[] serverlist = null;

protected String getLocalName(){
   String hostName = null;
   try {
     String hostNameLong =
       java.net.InetAddress.getLocalHost().getHostName();
     int pos = hostNameLong.indexOf('.');
     if(pos>=0){
       hostName = hostNameLong.substring(0,pos);
     }
     else{
       hostName = hostNameLong;
     }
   }
   catch ( Exception exception )
   {
   System.err.println( exception );
   }
   return hostName;
}

private String getCmdRet (String cmd){
  String cmdRet = "";
  try{
   Process p =Runtime.getRuntime().exec(cmd);
   InputStream pin = p.getInputStream();
   InputStreamReader cin = new InputStreamReader(pin);
```

```
      BufferedReader in = new BufferedReader(cin);
      String line = "";
      while ((line=in.readLine())!=null){
        cmdRet=cmdRet + line;
      }
      in.close();
    }
    catch(Exception ex){
      System.out.println("getCmdRet " + ex);
    }
    return cmdRet;
  }

/* ************************ *
 *   Auxiliary functions     *
 * ************************ */
private native int GetPid();
static {
    System.load("/u1/pg/xyd3/Java/loadServerRay/mobile/GetPidImpl.so");
}

private float getselfcpu (){
  float load = 0;
  int pid = GetPid();
  try{
    String str = getCmdRet("ps -p "+pid+" -opcpu");
    int pos = str.indexOf('U');
    String loadStr = str.substring(pos+1,pos+5);
    load = Float.parseFloat(loadStr);
  }
  catch( Exception ex){
```

```
    System.err.println( ex );
  }
  return load;
}

protected double tcomm (int n){
  double t_comm = 0;
  if (n<50)
    t_comm =  0.029;
  else
    t_comm = 0.029+5.07 * 0.000001 * ((double)(n*n));
  return t_comm;
}

protected double tcoord (int p){
  return 0.25 * (double) p;
}

protected void checkRelSpeed(){
  try{
    float load = 0;
    this.serverlist = ls.getServers();
    int ampnum=0;
    for (int i=0;i<(this.serverlist.length);i++){
      ampnum = this.serverlist[i].ampnum;
      if(serverlist[i].servername.equals(current.servername)){
        this.current = this.serverlist[i];
        load = serverlist[i].load+serverlist[i].comingload
                   -(this.genCpu/serverlist[i].speed);
        if (load < 0){load = 0;}
      }
```

```
      else{
        load = serverlist[i].load + serverlist[i].comingload;
        ampnum=ampnum+1;
      }
      serverlist[i].relspeed =
         ((serverlist[i].cpunum-(load/100))/ serverlist[i].cpunum)
          *serverlist[i].speed / ampnum;
    }
  }
  catch( Exception exception ){
    System.err.println( exception );
  }
  return ;
}

protected void checkNext () {
  checkRelSpeed();
  this.next = this.current;
  for (int i=0;i<serverlist.length;i++){
    if( ((this.next.relspeed)<(serverlist[i].relspeed))
         & serverlist[i].running){
      this.next = serverlist[i];
    }
  }
  return;
}

protected synchronized void check_move (int datasize,
                                        int workleft,
                                        double fhtime,
                                        IMobility mobility){
```

```
  try{
    double t_c = tcomm (datasize);
    double t_h = fhtime * (double) workleft;
    double t_n = t_h;
    checkNext();
    if (!(current.servername.equals(next.servername))){
      t_n = (current.relspeed) / (next.relspeed) * t_h + t_c;
      if (t_h > t_n){
        // before move set previous location value
        ls.setSleepTime(true,3000);
        ls.setAmpNum(-1); //previous host Amp -1
        //lookup next host
        this.ls = (ILoadServer)Naming.lookup
                    ("//"+next.servername+":9000/LS" );
        ls.setAmpNum(1);
        mobility.moveTo("//"+next.servername+":8000"); //move next host
        this.current.servername = new String(this.next.servername);
        System.out.println("The program is on "+current.servername);
       }
     }
  }
  catch( Exception exception ){
    System.err.println( exception );
  }
  return;
}

protected int getGran (int work, double  fhtime){
  double staticTime = fhtime * (double)work;
  double coordTime = tcoord (1);
  int times = (int)( (5.0/100.0*staticTime) / coordTime );
```

```
  int gran = work;

  if (times > 0)
    gran = work/(times+1)+1;
  return gran;
}

public Auto () {};
public Auto (String port){
  try{
    Voyager.startup(port);
    String sName = "//"+localName+":9000/LS";
    this.ls = (ILoadServer) Naming.lookup( sName );
    this.current = ls.getCurrent();
    ls.setAmpNum(1);
  }
  catch(Exception ex){
    System.out.println("Start Local Server: " + ex);
  }
}

/* ********************************* *
 *            automap                *
 * ********************************* */
public Impact[] automap (Superclass obj, Ray[] rays){
  Impact[] result=new Impact[rays.length];
  try{
    long timestart = 0;
    long timeend = 0;
    double fhtime = 0;
    int work = rays.length;
```

```
    int gran = work;
    int checkPos = 1;

    ISuperclass proxy = (ISuperclass) Proxy.of(obj);
    IMobility mobility = Mobility.of(proxy); //bulid mobility

    for(int i=0;i<work;i++){
      if ((i-checkPos) == 0){
        timestart = System.currentTimeMillis();
        result[i]=(Impact)proxy.mapf(rays[i]);
        timeend = System.currentTimeMillis();
        fhtime = (double)(timeend-timestart)/1000.0;
        gran = getGran (work,fhtime);
        checkPos = checkPos + gran;
        check_move (obj.datasize,(work-i-1),fhtime,mobility);

        //set how ofter should the server check information
        ls.setSleepTime(false,(int)(fhtime*1000)*gran);
      }
      else{
        result[i]=(Impact)proxy.mapf(rays[i]);
      }
    }
    ls.setAmpNum(-1); //counting the AMP number
  }
  catch( Exception exception ){
    System.err.println( exception );
  }
     Voyager.shutdown();
     return result;
}
```

```
/* ********************************* *
 *            autofold               *
 * ********************************* */
public int autofold (Superclass obj,int x,int[] l){
  int result = x;
  try{
    long timestart = 0;
    long timeend = 0;
    double fhtime = 0;
    int work = l.length;
    int gran = work;
    int checkPos = 1;
    double totaltime = 0;

    ISuperclass proxy = (ISuperclass) Proxy.of(obj);
    IMobility mobility = Mobility.of(proxy); //bulid mobility

    for(int i=0;i<work;i++){ // fold
      if((i-checkPos) == 0 ) {
        timestart = System.currentTimeMillis();
        result = proxy.foldf (result, l[i]);
        timeend = System.currentTimeMillis();

        fhtime = (double)(timeend-timestart)/1000.0;
        gran = getGran (work,fhtime);
        checkPos = checkPos + gran;
        check_move (obj.datasize,(work-i-1),fhtime,mobility);
       }
       else {
         result = proxy.foldf (result, l[i]);
       }
```

```
      }
      ls.setAmpNum(-1);
    }
    catch( Exception exception ){
      System.err.println( exception );
    }
    Voyager.shutdown();
    return result;
  }
}
```

B.2.2 Load Server Implementation

```
import java.io.*;
import java.util.*;
import java.lang.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class LoadServer extends UnicastRemoteObject
                     implements ILoadServer, Serializable{

   public String localName = getLocalName();
   public Server[] serverlist = {
                                  (new Server("linux02")),
                                  (new Server("linux06")),
                                  (new Server("linux09"))};
   public int numOfServer = serverlist.length;
   static public int sleeptime = 3000;
```

```
public int ampNum = 0;
public Server next = new Server(localName);
public Server local = new Server(localName) ;
private  CheckInfo ci = null;

protected String getLocalName() {
   String hostName = null;
   String hostNameLong = null;
   try {
     hostNameLong = java.net.InetAddress.getLocalHost().getHostName();
     int pos = hostNameLong.indexOf('.');
     if(pos>=0){
       hostName = hostNameLong.substring(0,pos);
     }
     else{
       hostName = hostNameLong;
     }
   }
   catch ( Exception exception )
   {
   System.err.println( exception );
   }
   return hostName;
}

private String getCmdRet (String cmd){
  String cmdRet = "";
  try{
    Process p =Runtime.getRuntime().exec(cmd);
    InputStream pin = p.getInputStream();
    InputStreamReader cin = new InputStreamReader(pin);
```

```
      BufferedReader in = new BufferedReader(cin);
      String line = "";
      while ((line=in.readLine())!=null){
        cmdRet=cmdRet + line;
      }
      in.close();
    }
    catch(Exception ex){
      System.out.println("getCmdRet " + ex);
    }
    return cmdRet;
  }

/* ********************************** *
 *   get local information functions  *
 * ********************************** */
  protected float checkCpuNum (Server hn){
    float num = 1;
    String numStr =
        getCmdRet ("rsh "+hn.servername+
          " cat /proc/cpuinfo |grep MHz|wc -l");
    num = Float.parseFloat(numStr);
    return num;
  }

  protected float checkSpeed (Server hn){
    float speed = 0;
    String str =
        getCmdRet ("rsh "+hn.servername+" cat /proc/cpuinfo |grep MHz" );
    int pos = str.indexOf(':');
    String speedStr = str.substring(pos+1,pos+9);
```

```
  speed = Float.parseFloat(speedStr);
  return speed;
}

protected boolean checkRunning (Server hn){
  boolean running = false;
  String runningStr = getCmdRet ("ps -C voyager");
  int pos = runningStr.indexOf("voyager");
  if (pos != -1)  {
    running = true;
  }
  return running;
}

protected float checkLoad (Server hn){
  float load = 0;
  String usedCpuStr =
     getCmdRet ("/u1/pg/xyd3/showcpu");
  float usedCpu = Float.parseFloat(usedCpuStr);
  if (usedCpu > (hn.cpunum * 100))
    load = 100 * hn.cpunum;
  else
    load =usedCpu;
  return load;
}

protected void checkStaticInfo (){
  local.speed=checkSpeed(this.local);
  local.cpunum=checkCpuNum(this.local);
  return;
}
```

```
  protected void check(){
    try{
      for(int i=0;i<numOfServer;i++){
        Server s = this.serverlist[i];
        if (!s.servername.equals(localName)){
          ILoadServer ls = (ILoadServer)Naming.lookup
                       ("//"+s.servername+":9000/LS" );
          serverlist[i] = ls.getInfo();
        }
        else{
          serverlist[i].running=checkRunning(serverlist[i]);
          serverlist[i].load = checkLoad(serverlist[i]);
        }
      }
    }
    catch(Exception ex){
       System.out.println("Start Local Server: " + ex);
    }
    return;
  }

/*****************************
 *constructor
 *****************************/
  protected void setLocal(){
    for (int i=0;i<numOfServer;i++){
      if (serverlist[i].servername.equals(localName)){
        this.local = serverlist[i];
        serverlist[i].isLocal = true;
        serverlist[i].running=checkRunning(serverlist[i]);
```

```
        serverlist[i].load = checkLoad(serverlist[i]);
      }
    }
    return;
  }

  protected void init (){
     setLocal();
     checkStaticInfo();
     ci = new CheckInfo();
     ci.start();
  }

  public LoadServer() throws RemoteException{
    try{
      init();
    }
    catch(Exception ex){
      System.out.println("Init Server Info: " + ex);
    }
 }

/* *************************** *
 *   CheckInformation Thread    *
 * *************************** */
  class CheckInfo extends Thread {
    public CheckInfo() {}
    public void run(){
      try{
        while (true){
          long timestart = System.currentTimeMillis();
```

```
          check();
          long timeend = System.currentTimeMillis();
          System.out.println("coord Time: " + (timeend-timestart));
          sleep(sleeptime);
        }
      }
      catch(Exception ex){
        System.out.println("Start Load Server: " + ex);
      }
      return;
    }
  }

/* ************************************* *
 *         remote use motheds            *
 * ************************************* */
  public synchronized Server[] getServers() throws RemoteException{
    return this.serverlist;
  }

  public synchronized Server getCurrent() throws RemoteException{
    return this.local;
  }
  public synchronized Server getNext()throws RemoteException{
    return this.next;
  }
  public synchronized int getAmpNum()throws RemoteException{
    return this.local.ampnum;
  }
  public synchronized void setAmpNum(int io)throws RemoteException{
    this.local.ampnum = this.local.ampnum + io;
```

```
    return;
  }
  public synchronized void setFlag(String servername, boolean flag)
                                            throws RemoteException{
    int i=0;
    while (!serverlist[i].servername.equals(servername)) i++;
    serverlist[i].enable = flag;
    return;
  }
  public synchronized void setFlag(boolean flag)throws RemoteException{
    local.enable = flag;
    return;
  }

  public synchronized void setComingLoad(float t) throws RemoteException{
    local.comingload  = local.comingload + t;
  }

  public synchronized void setComingLoad(String servername,float t)
                                                throws RemoteException{
    for(int i=0;i<numOfServer;i++){
      if (serverlist[i].servername.equals(servername)){
        serverlist[i].comingload  = serverlist[i].comingload + t;
      }
    }
    return;
  }

  public synchronized void setSleepTime(boolean change,int t)
                                               throws RemoteException{
    if ((t<this.sleeptime)||(change)){
```

```
      this.sleeptime = t;
    }
    return;
  }
  public synchronized Server getInfo()throws RemoteException{
    return this.local;
  }

/* ************************************** *
 *             main function              *
 * ************************************** */
  public static void main(String[] args)throws Exception {
        try{
            LoadServer ls = new LoadServer();
            String sName = "//"+ls.localName+":9000/LS";
            System.out.println(ls.localName);
            Naming.rebind(sName, ls);
        }
        catch(Exception ex){
             System.out.println("Start Load Server: " + ex);
       }
  }
}
```

B.3 JavaGo Autonomous Mobile Iterator

```
/* The auxiliary functions and globle structures in Class AutoIterator
are the same as in Class Auto in Java Voyager, so this class only give
the code which are different from Class Auto.  */
import java.io.*;
import java.util.*;
```

```
import java.lang.*;
import javago.*;

public class AutoIterator implements Iterator,Serializable{

/* ********************************** *
 *   Auxiliary functions are omited    *
 * ********************************** */

  private ArrayList list;
  private int nextIndex;
  private int work;

/* ************************** *
 *         constructor         *
 * ************************** */
  public AutoIterator(ArrayList theList){
    list = theList;
    nextIndex = 0;
    work = list.size();

     try{
       init(); //initialize server
      }
      catch(Exception ex){
        System.out.println("Start Local Server: " + ex);
      }
  }

  public boolean hasNext(){
    return nextIndex < work;
```

```
  }

  public Object next() {
     if (nextIndex < work)
        return list.get(nextIndex++);
      else{
        throw new NoSuchElementException("No next element");
      }
  }

  private int checkPos = 1;
  private long timestart = 0;
  private long timeend = 0;
  private double fhtime = 0;
  private int gran = work;

/* *************************** *
 *           autoNext          *
 * *************************** */
  public migratory Object autoNext() throws javago.NotifyGone, IOException{
    try{
      if (nextIndex < work){
        if(nextIndex == 0){
          timestart = System.currentTimeMillis();
          timeend = timestart;
        }
        else{
          if((nextIndex-checkPos) == 0 ){
            if(nextIndex!=1){
              check_move (this.datasize,(work-nextIndex-1),fhtime);
            }
```

```
            timestart = timeend;
            timeend = System.currentTimeMillis();
            fhtime = (double)(timeend-timestart)/1000.0;
            gran = getGran (work,fhtime);
            checkPos = checkPos + gran;
          }
        }
        return list.get(nextIndex++);
      }
      else{
        throw new NoSuchElementException("No next element");
      }
    }
    catch( Exception exception ){
        System.err.println( exception );
    }
  }

  public void remove(){
    throw new UnsupportedOperationException("remove not supported");
  }

/* ********************************** *
 *    example: matrix multiplications   *
 * ********************************** */
  public static void main(String args[]){
    try{
      undock {
        String port=null;
        int listlength = Integer.parseInt(args[0]);
        ArrayList al = new ArrayList();
```

```
        for (int i=0;i<listlength;i++){
          MatrixMul ii = new MatrixMul();
          al.add(i,ii);
        }
        long timestart = System.currentTimeMillis();
        AutoIterator ai = new AutoIterator(al);  /* AutoIterator */
        while (ai.hasNext()){
          MatrixMul iu = (MatrixMul)ai.autoNext();
          int[][] mat = iu.Multiplication();
        }
      }
    }
    catch (Exception e) {
      System.out.println ("migration failed!");
      System.out.println (e.getMessage());
    }
  }
}
```

Appendix C

Balance Status of Collections of AMPs

This appendix shows the behaviour of 20 AMPs on 10 locations with CPU speeds 3139MHz (Loc1-Loc5), 2167MHZ (Loc6), and 1793MHz (Loc7-Loc10). Tables C.29, C.30, and C.31 show the location information at balance state when there are 20, 19, or 18 AMPs on the 10 locations. Figures C.106, C.107, and C.108 in Appendix C show the actual relative CPU speed available to 20 AMPs and then becomes 19 and 18 AMPs. In Figure C.106, most AMPs have average relative CPU speed from 200MHz to 400MHz (18 out of 20 AMPs). There is one AMP on Loc1 with CPU speed 650-700MHz. Similar results were got when there are 19 and 18 AMPs, which are shown in Figure C.107 and Figure C.108. See Section 5.3.2 for details.

Total AMPs	Location	CPU speed s	Number of AMPs at each location n	Other Loads (CPU%) l	Relative Speed Each AMP had r=s*(100-l)%/n
20 AMPs	Loc1-Loc5	3193MHz	0	80	-
	Loc6	2167MHZ	2	0	1084MHz
	Loc7-Loc8	1793MHz	1	0	1793MHz
	Loc9-Loc10	1793MHz	2	0	897MHz

Table C.29: Prediction CPU Speed each AMPs Get(20 AMPs on 15 Locations)

Total AMPs	Location	CPU speed s	Number of AMPs at each location n	Other Loads (CPU%) l	Relative Speed Each AMP had r=s*(100-l)%/n
19 AMPs	Loc1	3193MHz	1	59	1309MHz
	Loc2-Loc5	3193MHz	3	0	1064MHz
	Loc6	2167MHZ	2	0	1084MHz
	Loc7-Loc10	1793MHz	1	0	1793MHz

Table C.30: Prediction CPU Speed each AMPs Get(19 AMPs on 15 Locations)

Total AMPs	Location	CPU speed s	Number of AMPs at each location n	Other Loads (CPU%) l	Relative Speed Each AMP had r=s*(100-l)%/n
18 AMPs	Loc1	3193MHz	1	53	1501MHz
	Loc2,3,5	3193MHz	3	0	1064MHz
	Loc4	3193MHz	2	0	1597MHz
	Loc6	2167MHZ	2	0	1084MHz
	Loc7-Loc10	1793MHz	1	0	1793MHz

Table C.31: Prediction CPU Speed each AMPs Get(18 AMPs on 15 Locations)

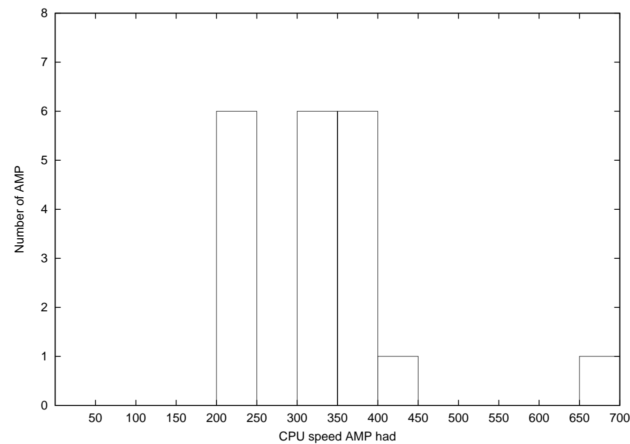


Figure C.106: Actual CPU Speed for Each AMP (20 AMPs on 10 Locations)

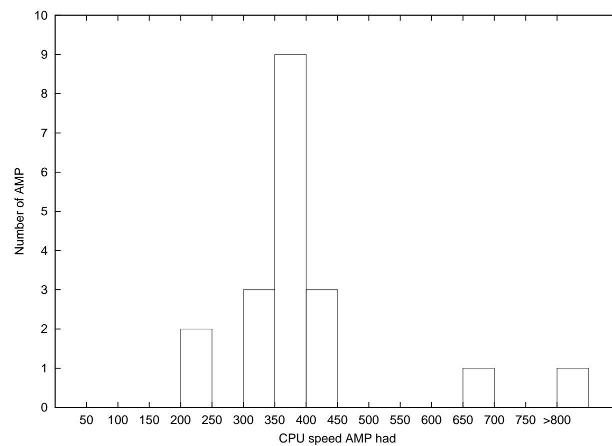


Figure C.107: Actual CPU Speed for Each AMP (19 AMPs on 10 Locations)

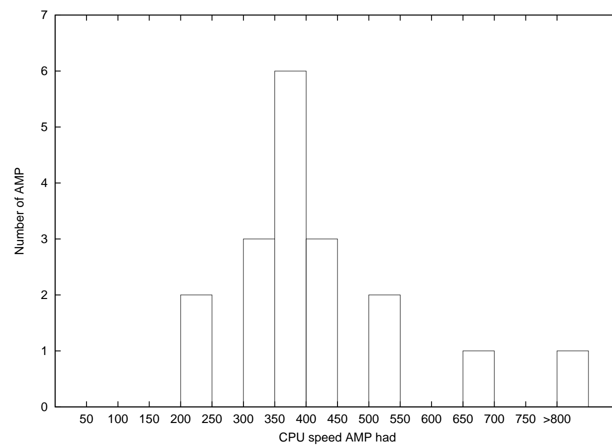


Figure C.108: Actual CPU Speed for Each AMP (18 AMPs on 10 Locations)

Appendix D

Automatic Cost Analyser Validation

Tables D.32 to D.39 compare CAMS to AMS performances. Tables D.40 to D.42 compare the execution times CAMS programs from the analyser automatically and CAMS programs produced by hand.

(1)ncc1710 (2)jove (3)lxtrinder					
Size	Static	`automap` Mobile		`camap` Mobile	
	time	time	status	time	status
100*100	0.695	0.700	Stay	0.700	Stay
200*200	5.848	5.866	Stay	5.867	Stay
300*300	19.727	19.808	Stay	19.808	Stay
310*310	21.870	21.950	Stay	21.932	Stay
320*320	24.152	24.116	Stay	24.116	Stay
330*330	26.853	23.191	(1)− > (3)	23.362	(1)− > (3)
340*340	28.880	24.206	(1)− > (3)	24.412	(1)− > (3)
350*350	31.471	25.400	(1)− > (3)	25.571	(1)− > (3)
400*400	47.060	31.768	(1)− > (3)	31.498	(1)− > (3)
500*500	92.984	47.454	(1)− > (3)	46.786	(1)− > (3)
600*600	161.655	68.218	(1)− > (3)	68.217	(1)− > (3)
700*700	259.260	95.104	(1)− > (3)	95.672	(1)− > (3)
800*800	393.167	130.496	(1)− > (3)	130.606	(1)− > (3)
900*900	586.220	174.681	(1)− > (3)	175.023	(1)− > (3)

Table D.32: `camap` and `automap` Matrix Mutiplication Movement Comparison

(1)ncc1710 (2)jove (3)lxtrinder					
Size	Static	automap Mobile		camap Mobile	
	time	time	status	time	status
5*5	0.967	0.971	Stay	0.982	Stay
10*10	3.876	3.931	Stay	3.908	Stay
15*15	8.803	8.822	Stay	8.922	Stay
20*20	15.972	15.873	Stay	15.750	Stay
21*21	17.670	17.709	Stay	17.868	Stay
22*22	19.378	18.449	(1)−>(3)	18.125	(1)−>(3)
23*23	21.305	19.065	(1)−>(3)	18.978	(1)−>(3)
24*24	23.221	19.854	(1)−>(3)	19.408	(1)−>(3)
25*25	25.493	20.679	(1)−>(3)	20.226	(1)−>(3)
30*30	37.376	24.370	(1)−>(3)	24.644	(1)−>(3)
35*35	51.978	28.591	(1)−>(3)	29.057	(1)−>(3)
40*40	69.288	33.977	(1)−>(3)	33.997	(1)−>(3)

Table D.33: camap and automap Ray Tracing Movement Comparison

(1)ncc1710 (2)jove (3)lxtrinder					
Size	Static	automap Mobile		camap Mobile	
	time	time	status	time	status
50*50	0.173	0.176	Stay	0.174	Stay
100*100	1.467	1.451	Stay	1.472	Stay
150*150	5.042	4.997	Stay	5.023	Stay
200*200	11.865	11.804	Stay	11.834	Stay
210*210	13.703	13.751	Stay	13.720	Stay
220*220	15.732	15.743	Stay	15.690	Stay
230*230	18.042	17.998	Stay	18.990	(1)−>(3)
240*240	20.426	20.379	Stay	20.358	(1)−>(3)
250*250	23.249	23.087	Stay	21.518	(1)−>(3)
260*260	25.960	25.947	Stay	22.872	(1)−>(3)
270*270	29.000	29.134	Stay	24.631	(1)−>(3)
280*280	32.457	32.442	Stay	25.899	(1)−>(3)
290*290	36.030	36.077	Stay	27.696	(1)−>(3)
300*300	40.131	39.844	Stay	29.222	(1)−>(3)
310*310	44.179	43.946	Stay	30.642	(1)−>(3)
320*320	48.726	48.314	Stay	32.254	(1)−>(3)
330*330	53.299	30.687	(1)−>(3)	33.984	(1)−>(3)
340*340	58.485	33.846	(1)−>(3)	35.733	(1)−>(3)
350*350	63.753	35.475	(1)−>(3)	37.630	(1)−>(3)
400*400	95.571	46.269	(1)−>(3)	47.621	(1)−>(3)
450*450	136.587	59.867	(1)−>(3)	61.256	(1)−>(3)
500*500	187.014	75.013	(1)−>(3)	77.283	(1)−>(3)

Table D.34: camap and automap Double Matrix Multiplication Movement Comparison

(1)ncc1710 (2)jove (3)lxtrinder					
Size	Static	automap Mobile		camap Mobile	
	time	time	status	time	status
50*50	0.175	0.171	Stay	0.172	Stay
100*100	1.437	1.461	Stay	1.422	Stay
150*150	5.005	4.993	Stay	5.011	Stay
200*200	11.876	11.774	Stay	11.691	Stay
210*210	13.692	13.671	Stay	13.712	Stay
220*220	15.731	15.689	Stay	15.717	Stay
230*230	17.997	17.994	Stay	19.162	(1)− > (3)
240*240	20.476	20.440	Stay	20.433	(1)− > (3)
250*250	23.051	23.038	Stay	21.623	(1)− > (3)
260*260	25.976	25.997	Stay	23.019	(1)− > (3)
270*270	28.909	29.128	Stay	24.607	(1)− > (3)
280*280	32.400	32.270	Stay	25.912	(1)− > (3)
290*290	35.872	35.993	Stay	27.631	(1)− > (3)
300*300	39.657	39.780	Stay	29.284	(1)− > (3)
310*310	43.995	43.906	Stay	31.069	(1)− > (3)
320*320	48.518	48.332	Stay	32.510	(1)− > (3)
330*330	52.902	31.053	(1)− > (3)	34.575	(1)− > (3)
340*340	57.844	32.791	(1)− > (3)	36.093	(1)− > (3)
350*350	63.384	34.701	(1)− > (3)	38.025	(1)− > (3)
400*400	94.777	45.322	(1)− > (3)	47.895	(1)− > (3)
450*450	135.806	58.046	(1)− > (3)	61.640	(1)− > (3)
500*500	186.985	73.376	(1)− > (3)	77.369	(1)− > (3)
550*550	249.145	94.038	(1)− > (3)	95.831	(1)− > (3)
600*600	326.493	115.928	(1)− > (3)	116.898	(1)− > (3)
650*650	430.764	142.560	(1)− > (3)	143.259	(1)− > (3)

Table D.35: `camap` and `automap` Invertible Matrix Movement Comparison

(1)ncc1710 (2)jove (3)lxtrinder					
Size	Static	`automap` Mobile		`camap` Mobile	
	time	time	status	time	status
5*5	1.929	1.969	Stay	1.964	Stay
10*10	7.863	7.751	Stay	7.820	Stay
11*11	9.445	9.586	Stay	9.532	Stay
12*12	11.233	11.182	Stay	11.280	Stay
13*13	13.266	13.129	Stay	14.196	Stay
14*14	15.289	15.240	Stay	16.173	Stay
15*15	17.575	17.710	Stay	18.710	Stay
16*16	20.135	20.167	Stay	19.320	(1)− > (3)
17*17	22.844	22.870	Stay	20.566	(1)− > (3)
18*18	25.599	25.705	Stay	21.825	(1)− > (3)
19*19	28.666	28.634	Stay	22.429	(1)− > (3)
20*20	31.773	31.815	Stay	23.888	(1)− > (3)
21*21	35.282	35.177	Stay	25.356	(1)− > (3)
22*22	39.037	22.242	(1)− > (3)	26.703	(1)− > (3)
23*23	42.371	23.490	(1)− > (3)	28.067	(1)− > (3)
24*24	46.133	25.553	(1)− > (3)	29.216	(1)− > (3)
25*25	50.324	25.678	(1)− > (3)	30.404	(1)− > (3)
30*30	74.030	31.933	(1)− > (3)	37.282	(1)− > (3)
35*35	102.907	38.708	(1)− > (3)	44.799	(1)− > (3)
40*40	137.621	47.448	(1)− > (3)	54.451	(1)− > (3)
45*45	181.171	57.037	(1)− > (3)	65.112	(1)− > (3)
50*50	230.401	68.033	(1)− > (3)	77.334	(1)− > (3)

Table D.36: `camap` and `automap` Double Ray Tracing Movement Comparison

(1)ncc1710 (2)jove (3)lxtrinder					
Size	Static	`automap` Mobile		`camap` Mobile	
	time	time	status	time	status
50*50	0.434	0.424	Stay	0.434	Stay
100*100	3.653	3.625	Stay	3.655	Stay
150*150	12.507	12.524	Stay	12.555	Stay
160*160	15.280	15.277	Stay	15.241	Stay
170*170	18.234	18.249	Stay	18.961	(1)− > (3)
180*180	21.628	21.606	Stay	20.772	(1)− > (3)
190*190	25.497	25.480	Stay	22.491	(1)− > (3)
200*200	29.548	29.565	Stay	24.598	(1)− > (3)
210*210	34.290	34.267	Stay	26.623	(1)− > (3)
220*220	39.454	39.348	Stay	28.675	(1)− > (3)
230*230	45.004	45.020	Stay	30.951	(1)− > (3)
240*240	51.214	51.019	Stay	33.346	(1)− > (3)
250*250	57.871	57.827	Stay	35.609	(1)− > (3)
260*260	65.436	64.941	Stay	38.950	(1)− > (3)
270*270	72.719	72.952	Stay	41.647	(1)− > (3)
280*280	81.246	81.422	Stay	44.618	(1)− > (3)
290*290	90.343	90.251	Stay	47.046	(1)− > (3)
300*300	99.482	99.697	Stay	50.197	(1)− > (3)
310*310	109.958	110.152	Stay	53.850	(1)− > (3)
320*320	120.877	121.269	Stay	57.101	(1)− > (3)
330*330	132.859	53.371	(1)− > (3)	61.579	(1)− > (3)
340*340	145.197	56.868	(1)− > (3)	65.193	(1)− > (3)
350*350	158.205	61.013	(1)− > (3)	69.352	(1)− > (3)
360*360	173.871	65.315	(1)− > (3)	73.370	(1)− > (3)
370*370	188.731	70.451	(1)− > (3)	77.500	(1)− > (3)
400*400	238.351	85.709	(1)− > (3)	92.567	(1)− > (3)
450*450	340.371	115.377	(1)− > (3)	123.887	(1)− > (3)
500*500	465.234	151.124	(1)− > (3)	158.788	(1)− > (3)

Table D.37: `camap` and `automap` Five Matrix Multiplications Movement Comparison

(1)ncc1710 (2)jove (3)lxtrinder (4)linux81					
Size	Static	`automap` Mobile		`camap` Mobile	
	time	time	status	time	status
50*50	0.175	0.171	Stay	0.172	Stay
100*100	1.437	1.461	Stay	1.422	Stay
150*150	5.005	4.993	Stay	5.011	Stay
200*200	11.876	11.774	Stay	11.691	Stay
210*210	13.692	13.671	Stay	13.712	Stay
220*220	15.731	15.689	Stay	15.717	Stay
230*230	17.997	17.994	Stay	19.162	$(1)->(3)$
240*240	20.476	20.440	Stay	20.433	$(1)->(3)$
250*250	23.051	23.038	Stay	21.623	$(1)->(3)$
260*260	25.976	25.997	Stay	23.019	$(1)->(3)$
270*270	28.909	29.128	Stay	24.607	$(1)->(3)$
280*280	32.400	32.270	Stay	25.912	$(1)->(3)$
290*290	35.872	35.993	Stay	27.631	$(1)->(3)$
300*300	39.657	39.780	Stay	29.284	$(1)->(3)$
310*310	43.995	43.906	Stay	31.069	$(1)->(3)$
320*320	48.518	48.332	Stay	32.510	$(1)->(3)$
330*330	52.902	31.053	$(1)->(3)$	34.575	$(1)->(3)$
340*340	57.844	32.791	$(1)->(3)$	36.093	$(1)->(3)$
350*350	63.384	34.701	$(1)->(3)$	38.025	$(1)->(3)$
400*400	94.777	45.322	$(1)->(3)$	47.895	$(1)->(3)$
440*440	126.478	55.173	$(1)->(3)$	58.757	$(1)->(3)$
450*450	135.806	58.046	$(1)->(3)$	61.505	$(1)->(3)->(4)$
500*500	186.985	73.376	$(1)->(3)$	73.850	$(1)->(3)->(4)$
550*550	249.146	95.083	$(1)->(3)$	86.464	$(1)->(3)->(4)$
600*600	326.493	116.056	$(1)->(3)$	102.449	$(1)->(3)->(4)$
650*650	430.764	145.170	$(1)->(3)$	120.165	$(1)->(3)->(4)$

Table D.38: `camap` and `automap` Invertible Matrix Movement Comparison with Changing Loads

(1)ncc1720 (2)jove (3)lxtrinder (4)linux81					
Size	Static	`automap` Mobile		`camap` Mobile	
	time	time	status	time	status
50*50	0.434	0.424	Stay	0.434	Stay
100*100	3.653	3.625	Stay	3.655	Stay
150*150	12.507	12.524	Stay	12.555	Stay
160*160	15.280	15.277	Stay	15.241	Stay
170*170	18.234	18.249	Stay	18.961	(1)−>(3)
180*180	21.628	21.606	Stay	20.772	(1)−>(3)
190*190	25.497	25.480	Stay	22.491	(1)−>(3)
200*200	29.548	29.565	Stay	24.598	(1)−>(3)
210*210	34.290	34.267	Stay	26.623	(1)−>(3)
220*220	39.454	39.348	Stay	28.675	(1)−>(3)
230*230	45.004	45.020	Stay	30.951	(1)−>(3)
240*240	51.214	51.019	Stay	33.346	(1)−>(3)
250*250	57.871	57.827	Stay	35.609	(1)−>(3)
260*260	65.436	64.941	Stay	38.950	(1)−>(3)
270*270	72.719	72.952	Stay	41.647	(1)−>(3)
280*280	81.246	81.422	Stay	44.618	(1)−>(3)
290*290	90.343	90.251	Stay	47.046	(1)−>(3)
300*300	99.482	99.697	Stay	50.197	(1)−>(3)
310*310	109.958	110.152	Stay	53.850	(1)−>(3)
320*320	120.877	121.269	Stay	57.101	(1)−>(3)
330*330	132.859	53.371	(1)−>(3)	60.356	(1)−>(3)−>(4)
340*340	145.197	56.868	(1)−>(3)	63.223	(1)−>(3)−>(4)
350*350	158.205	61.013	(1)−>(3)	66.878	(1)−>(3)−>(4)
360*360	173.871	65.315	(1)−>(3)	69.705	(1)−>(3)−>(4)
370*370	188.731	70.451	(1)−>(3)	72.974	(1)−>(3)−>(4)
400*400	238.351	85.709	(1)−>(3)	84.129	(1)−>(3)−>(4)
450*450	340.371	115.377	(1)−>(3)	104.813	(1)−>(3)−>(4)
500*500	465.234	151.124	(1)−>(3)	130.905	(1)−>(3)−>(4)

Table D.39: `camap` and `automap` Five Matrix Multiplications Movement Comparison with Changing Loads

(1)ncc1710 (2)jove (3)lxtrinder					
Size	Static	Automatic		byHand	
	time	time	status	time	status
50*50	0.173	0.171	Stay	0.174	Stay
100*100	1.467	1.473	Stay	1.472	Stay
150*150	5.042	5.006	Stay	5.023	Stay
200*200	11.865	11.869	Stay	11.834	Stay
210*210	13.703	13.677	Stay	13.720	Stay
220*220	15.732	15.740	Stay	15.690	Stay
230*230	18.042	18.836	$(1)->(3)$	18.990	$(1)->(3)$
240*240	20.426	20.510	$(1)->(3)$	20.358	$(1)->(3)$
250*250	23.249	21.253	$(1)->(3)$	21.518	$(1)->(3)$
260*260	23.249	22.616	$(1)->(3)$	22.872	$(1)->(3)$
270*270	29.000	24.605	$(1)->(3)$	24.631	$(1)->(3)$
280*280	32.457	25.912	$(1)->(3)$	25.899	$(1)->(3)$
290*290	36.030	27.371	$(1)->(3)$	27.696	$(1)->(3)$
300*300	40.131	29.286	$(1)->(3)$	29.222	$(1)->(3)$
310*310	44.179	30.724	$(1)->(3)$	30.642	$(1)->(3)$
320*320	48.726	32.341	$(1)->(3)$	32.254	$(1)->(3)$
330*330	53.299	34.165	$(1)->(3)$	33.984	$(1)->(3)$
340*340	58.485	35.769	$(1)->(3)$	35.733	$(1)->(3)$
350*350	63.753	37.569	$(1)->(3)$	37.630	$(1)->(3)$
400*400	95.571	48.255	$(1)->(3)$	47.621	$(1)->(3)$
450*450	136.587	61.946	$(1)->(3)$	61.256	$(1)->(3)$
500*500	187.014	77.085	$(1)->(3)$	77.283	$(1)->(3)$

Table D.40: Automatic and Byhand Cost Double Matrix Multiplications Movement Comparison

(1)ncc1710 (2)jove (3)lxtrinder					
Size	Static	automatic		byHand	
	time	time	status	time	status
50*50	0.175	0.171	Stay	0.172	Stay
100*100	1.437	1.455	Stay	1.422	Stay
150*150	5.005	5.037	Stay	5.011	Stay
200*200	11.876	11.839	Stay	11.691	Stay
210*210	13.692	13.711	Stay	13.712	Stay
220*220	15.731	15.736	Stay	15.717	Stay
230*230	17.997	18.973	(1)−>(3)	19.162	(1)−>(3)
240*240	20.476	20.758	(1)−>(3)	20.433	(1)−>(3)
250*250	23.051	21.774	(1)−>(3)	21.623	(1)−>(3)
260*260	25.976	22.928	(1)−>(3)	23.019	(1)−>(3)
270*270	28.909	24.543	(1)−>(3)	24.607	(1)−>(3)
280*280	32.400	25.951	(1)−>(3)	25.912	(1)−>(3)
290*290	35.872	27.565	(1)−>(3)	27.631	(1)−>(3)
300*300	39.657	29.110	(1)−>(3)	29.284	(1)−>(3)
310*310	43.995	30.994	(1)−>(3)	31.069	(1)−>(3)
320*320	48.518	32.581	(1)−>(3)	32.510	(1)−>(3)
330*330	52.902	34.290	(1)−>(3)	34.575	(1)−>(3)
340*340	57.844	36.039	(1)−>(3)	36.093	(1)−>(3)
350*350	63.384	38.170	(1)−>(3)	38.025	(1)−>(3)
400*400	94.777	48.063	(1)−>(3)	47.895	(1)−>(3)
450*450	135.806	61.364	(1)−>(3)	61.640	(1)−>(3)
500*500	186.985	77.417	(1)−>(3)	77.369	(1)−>(3)

Table D.41: Automatic and Byhand Cost Invertible Matrix Movement Comparison

(1)ncc1710 (2)jove (3)lxtrinder					
Size	Static	Automatic		byHand	
	time	time	status	time	status
5*5	1.929	1.952	Stay	1.964	Stay
10*10	7.863	7.882	Stay	7.820	Stay
11*11	9.445	9.528	Stay	9.532	Stay
12*12	11.233	11.478	Stay	11.280	Stay
13*13	13.266	14.408	Stay	14.196	Stay
14*14	15.289	16.647	Stay	16.173	Stay
15*15	17.575	18.926	Stay	18.710	Stay
16*16	20.135	19.493	(1)− > (3)	19.320	(1)− > (3)
17*17	22.844	20.543	(1)− > (3)	20.566	(1)− > (3)
18*18	25.599	21.868	(1)− > (3)	21.825	(1)− > (3)
19*19	28.666	22.787	(1)− > (3)	22.429	(1)− > (3)
20*20	31.773	24.058	(1)− > (3)	23.888	(1)− > (3)
21*21	35.282	25.348	(1)− > (3)	25.356	(1)− > (3)
22*22	39.037	26.718	(1)− > (3)	26.703	(1)− > (3)
23*23	42.371	28.065	(1)− > (3)	28.067	(1)− > (3)
24*24	46.133	29.420	(1)− > (3)	29.216	(1)− > (3)
25*25	50.324	30.705	(1)− > (3)	30.404	(1)− > (3)
30*30	74.030	37.258	(1)− > (3)	37.282	(1)− > (3)
35*35	102.907	45.125	(1)− > (3)	44.799	(1)− > (3)
40*40	137.621	54.417	(1)− > (3)	54.451	(1)− > (3)
45*45	181.171	65.127	(1)− > (3)	65.112	(1)− > (3)
50*50	230.401	77.895	(1)− > (3)	77.334	(1)− > (3)

Table D.42: Automatic and Byhand Costs Double Ray Tracing Movement Comparison

Bibliography

[1] Haskell 98 Language and Libraries The Revised Report. Technical report, December 2002. http://www.haskell.org/onlinereport/.

[2] Emergent, 2003, accessed 2007. http://www.beart.org.uk/Emergent/.

[3] TCP, Transmission Control Protocol, 2007. Accessed: March 2007, http://www.networksorcery.com/enp/protocol/tcp.htm.

[4] The Jocaml System, accessed: 2005. http://pauillac.inria.fr/jocaml/.

[5] ASAP Research Theme: Multi-Agents. University of Nothtingham, School of CS, ASAP, Accessed: 2007. http://www.asap.cs.nott.ac.uk/themes/ma.shtml.

[6] J. Abawajy. Autonomic Job Scheduling Policy for Grid Computing. In *Lecture Notes in Computer Science, LNCS 3516*, pages 213–220, Germany, May 2005. Internation Conference on Computationl Science - ICCS 2005, part 3, Springer.

[7] A. Acharya, M. Ranganathan, and J. H. Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. In *Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 111–130, London, UK, 1997. Springer-Verlag.

[8] I. Ahmad, A. Ghafoor, and K. Mehrotra. Performance prediction of distributed load balancing on multicomputer systems. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages

830–839, Albuquerque, New Mexico, United States, 1991. ACM Press, New York, USA.

[9] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[10] A. Al Zain. Integrated High Performance Computation on a Grid Network. Technical report, Department of Computer Sciences:Heriot-Watt University, January 2003.

[11] M. Antonioletti. Load Sharing Across Networked Computers. Technical report, Edinburgh Parallel Computing Centre:The University of Edinburgh, December 1997.

[12] A. Barak, S. Guday, and R. Wheeler. *The Mosix Distributed Operating System: Load Balancing for Unix*. Springer-Verlag, 1993.

[13] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Press, Jan. 1995.

[14] Z. Bellahsene and M. Roantree. Querying Distributed Data in a Super-Peer Based Architecture. In *Database and Expert Systems Applications*, volume 3180/2004 of *Lecture Notes in Computer Science*, pages 296–305. Springer Berlin / Heidelberg, October 2004.

[15] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *the 12th EuroMicro Conference on Real-Time Systems*, pages 81–88, Stockholm, Sweden, June 2000.

[16] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 279, Washington, DC, USA, 2002. IEEE Computer Society.

[17] A. R. D. Bois. *Mobile Computation in a Purely Functional Language*. PhD thesis, School of Mathematical and Computer Science, Heriot-Watt Univeristy, Edinburgh, UK, August 2005.

[18] A. R. D. Bois, P. Trinder, and H. Loidl. Towards Mobility Skeletons. *Parallel Processing Letters*, 15(3):273–288, 2005.

[19] A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann. Worst-case execution times for a purely functional language. In *18th IFL 2006, Budapest*. Springer, 2006.

[20] D. J. Busvine. *Detecting parallel structures in functional programs*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, April 1993.

[21] L. Cardelli. A language with distributed scope. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 286–297, San Francisco, California, United States, 1995. ACM Press.

[22] L. Cardelli. Abstrations for Mobile Computation. *Secure Internet Programming*, pages 51–94, 1999.

[23] A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32, Boston, Massachusetts, United States, 1997. ACM Press.

[24] T. Casavant and J. Kuhl. A Taxonomy of Scheduling in General-Pupose distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[25] G. Chen, M. Odersky, C. Zenger, and M. Zenger. A Functional View of Join. Technical Report ACRC-99-016, LAMP, University of South Australia, 1999.

[26] J. Cohen. Computer-assisted microanalysis of programs. *Commun. ACM*, 25(10):724–733, 1982.

[27] J. Cohen and A. Weitzman. Software tools for micro-analysis of programs. *Software - Practice and Experience*, 22(9):777–808, 1992.

[28] J. Cohen and C. Zuckerman. Two Languages for Estimating Program Efficiency. *Commun. ACM*, 17(6):301–308, 1974.

[29] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1989.

[30] S. Dal Zilio. Mobile Processes: a Commented Bibliography. In *MOVEP'2k – 4th Summer school on Modelling and Verification of Parallel processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 206–222, Nantes, France, June 2000. Springer Berlin / Heidelberg.

[31] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *PARLE '93: the 5th International Conference on Parallel Architectures and Languages Europe*, pages 146–160, London, UK, 1993. Springer-Verlag.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, USA, 2004.

[33] X. Y. Deng, G. Michaelson, and P. Trinder. Towards High Level Autonomous Mobility. In H.-W. Loidl, editor, *Draft Proceedings of Trends in Functional Programming*, pages 97–112, Munich, Germany, November 2004.

[34] X. Y. Deng, G. Michaelson, and P. Trinder. Autonomous Mobility Skeletons. *Journal of Parallel Computing*, Volume 32, Issues 7-8:Pages 463–478 Algorithmic Skeletons, September 2006.

[35] X. Y. Deng, P. Trinder, and G. Michaelson. Autonomous Mobile Programs. In *IAT '06: Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2006 Main Conference Proceedings) (IAT'06)*, pages 177–186, Hong Kong, December 2006. IEEE Computer Society, Washington, DC, USA.

[36] F. Douglis and J. K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[37] A. Du Bois, P. Trinder, and H. Loidl. *m*Haskell: Mobile Computation in a Purely Functional Language. *Journal of Universal Computer Science*, 11(7):1234–1254, 2005.

[38] A. Du Bois, P. Trinder, and H.-W. Loidl. Implementing Mobile Haskell. In *Trends in Functional Programming*, volume 4, pages 79–94, Edinburgh, Scotland, September 2003.

[39] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed System. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, 1986.

[40] First Int'l. Conf. on Foundations of Software Science and Computation Structures. *Mobile Ambients*. Springer-Verlag, 1998.

[41] I. Foster, N. R. Jennings, and C. Kesselman. Brain Meets Brawn: Why Grid and Agents Need Each Other. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 8–15, New York, 2004. IEEE Computer Society, Washington, DC, USA.

[42] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization . *International Journal of High Performance Computing Applications*, 15:200–222, 2001.

[43] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Palaiseau, France, November 1998.

[44] C. Fournet, F. L. Fessant, L. Maranget, and A. Schmitt. Jocaml: a Language for Concurrent Distributed and Mobile Programming. In *Proceedings of the Fourth Summer School on Advanced Functional Programming*, pages 19–24, St Anne's College, Oxford, August 2002. Springer-Verlag.

[45] C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, United States, 1996. ACM Press.

[46] C. Fournet, G. Gonthier, and J. J. Levy. A Calculus of Mobile Agents. In *Proceedings of the 7th International Conference on Concurrency Theory*, pages 406–421, London, UK, 1996. Springer-Verlag.

[47] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.

[48] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[49] A. Giacalone, P. Mishra, and S. Prasad. Facile: a symmetric integration of concurrent and functional programming. *Int. J. Parallel Program.*, 18(2):121–160, 1989.

[50] M. J. Gordon. *The Denotational Description of Programming Languages An Introduction*. Springer-Verlag, 1979.

[51] R. S. Gray. Agent TCL: A Transportable Agent System. In T. Finin and J. Mayfield, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM '95)*, Baltimore, Maryland, 1995. ACM Press New York, USA.

[52] A. Group. Sun's Grid Computing Solutions Outdistance the Competition, May 2002. http://www.sun.com/software/grid/docs/Grid_competitive.pdf.

[53] X. Guan, Y. Yang, and J. You. Making ambients more robust. In *Int'l. Conf. on Software: Theory and Practice*, pages 377–384, Beijing, China, Aug 2000.

[54] M. Hamdan. *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons*. PhD thesis, Department of Computing and Electrial Engineering, Heriot-Watt University, Edinburgh, Scotland, January 2000.

[55] K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*. Springer, 1999.

[56] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proceedings of GPCE 03 Generative Programming and Component Engineering*, LNCS, pages 37–56, Erfurt, Germany, September 2003. Springer-Verlag.

[57] K. Hammond, G. Michaelson, and P. Vasconcelos. Bounded Space Programming using Finite State Machines and Recursive Functions: the Hume Approach. *ACM Transactions on Software Engineering Methodology*, 2006. Submitted.

[58] M. R. Hansen and H. Rischel. *Introduction to Programming using SML*. Addison-Wesley, 1999.

[59] P. B. Hansen. The Programming Language Concurrent Pascal. *The Origin of Concurrent Programming: from Semaphores to Remote Procedure Calls*, pages 297–318, 2002. Springer-Verlag New York, Inc.

[60] M. Hashimoto and A. Yonezawa. MobileML: A Programming Language for Mobile Computation. In *Proceedings of the 4th International Conference on Coordination Languages and Models*, volume 1906 of *Lecture Notes In Computer Science*, pages 198–215. Springer-Verlag London, UK, 2000.

[61] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[62] J. Hoffmeyer. The Swarming Body. In *Proceedings of the Fifth Congress of the International Association for Semiotic Studies*, pages 937–940, Berkeley, 1994.

[63] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–197, New Orleans, Louisiana, USA, 2003. ACM Press New York, NY, USA.

[64] M. Hofmann and H.-W. Loidl. Automatic Prediction of Resource Bounds for Embedded Systems. Technical report, Ludwig-Maximilians Universitat, Munchen, March 2005.

[65] A. I. Houston and J. M. McNamara. Animal Behaviour. volume 36, pages 166–174. 1988.

[66] Institut National de Recherche en Informatique et en Automatique. *The JoCaml language beta release: Documentation and user's manual*, January 2001.

[67] N. R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001.

[68] R. Jha, J. K. II, and D. Cornhill. Ada Program Partitioning Language: A Notion for Distributing Ada Programs. *IEEE Transactions on Software Engineering*, 15(3):271–280, 1989. IEEE Computer Society, Los Alamitos, CA, USA.

[69] J.Hawkins and A.Abdallah. A Generic Functional Genetic Algorithm. In P.Trinder and G.Michaelson, editors, *Draft proceedings of the First Scottish Functional Programming Workshop*, pages 151–168, Heriot-Watt University, Edinburgh, 1999.

[70] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[71] Z. Kirli. *Mobile Computation with Functions*. PhD thesis, University of Edinburgh, Laboratory for Foundations of Computer Science:Division of Informatics, 2001.

[72] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison Wesley, Reading, Massachusetts, vol. 1 edition, July 1997.

[73] P. Krueger and M. Livny. A Comparison of Preemptive and Non-Preemptive Load Distributing. In *8th International Conference on Distributed Computing Systems, 1988.*, pages 123–130, San Jose, CA, USA, June 1988. IEEE Xplore.

[74] D. B. Lange and M. Oshima. Mobile agents with Java: The Aglet API. *World Wide Web*, 1(3):111–121, 1998.

[75] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Commun. ACM*, 42(3):88–89, 1999.

[76] X. Leroy and D. Doligez. *The Objective Caml system release 3.07: Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, September 2003.

[77] F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 352–364, Boston, MA, USA, 2000. ACM Press.

[78] C. Liu. Dynamic Load Balancing in Parallel and Distributed Computation: A Survey, accessed: Jan 2004. http://www.cs.nmsu.edu/˜cliu/ilp/loadbalance/survey.html.

[79] H. Liu and M. Parashar. A Component Based Programming Framework for Autonomic Applications. In *the International Conference on Autonomic*

Computing, pages 10–17, New York, NY, USA, 2004. IEEE Computer Society.

[80] H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, University of Glasgow, April 1998. Department of Computing Science.

[81] A. Merlin and G. Hains. A Generic Cost Model For Concurrent and Data-parallel Meta-computing. In *Forth Workshop on Automated Verification of Critical Systems (AVOCS'04)*, Electronic Notes in Theoretical Computer Science, London, UK, September 2004. Springer. Long preliminary version appears as LIFO RR2004-06.

[82] P. E. Merloti. Optimization Algorithms Inspired by Biological Ants and Swarm Behavior. Technical report, San Diego State University, Artificial Intelligence, CS550, San Diego, June 2004.

[83] G. Michaelson. *Elementary Standard ML.* UCL Press, 1995.

[84] G. Michaelson and P. Trinder. Distributed and Parallel Technologies. lecture notes, 2004. Heriot-Watt University.

[85] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[86] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[87] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes Pt.1. *Information and Computation*, 100(1):1–40, Sept 1992.

[88] D. Milojicic, F. Douglis, and R. Wheeler. *Mobility: processes, computers, and agents.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[89] S. A. Mondal. Survey on Load Balancing in a LAN Environment, Jun 1999. http://www.infy.com/knowledge-capital/thought-papers/loadbal2.pdf.

[90] R. Murch. *Autonomic Computing.* Published by IBM Press, 1st edition, March 2004.

[91] H. Nishikawa and P. Steenkiste. A General Architecture for Load Balancing in A Distributed-Memory Environment. In *ICDCS'1993: 13th IEEE International Conference on Distributed Computing Systems*, pages 47–54, Pittsburgh, PA, USA, May 1993. IEEE Computer Society Press.

[92] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, United States, 1996. ACM Press.

[93] B. C. Pierce and D. N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing, pages 455–494. MIT Press, 2000.

[94] Platform Computing, Inc. *LSF 6.0 User's Guide*, Accessed: November 2003. http://www.platform.com/Products/Platform.LSF.Family/.

[95] P. Puschner and G. Bernat. WCET Analysis of Reusable Portable Code. In *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 45, Delft University of Technology, Delft, The Netherlands, 2001. IEEE Computer Society, Washington, DC, USA.

[96] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*, chapter 4: Message-Passing Programming, pages 92–96. McGraw Hill Professional, 2003.

[97] K. Qureshi and M. Hatanaka. An Introduction to Load Balancing for Parallel Raytracing on HDC Systems. *Current Science, Tutorials*, 78(7):818–820, April 2000.

[98] L. H. Ramshaw. *Formalizing the Analysis of Algorithms.* PhD thesis, Stanford University Department of Computer Secience, 1979.

[99] R. Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model*. PhD thesis, Department of Computer Science, Edinburgh University, 1996.

[100] Recursion Software, Inc, 2591 North Dallas Parkway, Suite 200, Frisco, TX 75034. *Voyager User Guide*, May 2005. http://www.recursionsw.com/Voyager/Voyager_User_Guide.pdf.

[101] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and Functional Programming*, pages 65–78, Orlando, Florida, United States, 1994. ACM Press New York, NY, USA.

[102] J. Riley and M. Hennessy. A Typed Language for Distributed Mobile Processes (extended abstract). In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–390, San Diego, California, United States, 1998. ACM Press.

[103] M. Rosendahl. Automatic Complexity Analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156, Imperial College, London, United Kingdom, 1989. ACM Press New York, NY, USA.

[104] S. Russell and P. Norvig, editors. *Artificial Intelligence: A Modern Approach*. 1995.

[105] S. Sahni. *Data Structures, Algorithms, and Applications in Java*. Mc Graw Hill, University of Florida, 2000.

[106] N. Scaife. *A Dual Source Parallel Architecture for Computer Vision*. PhD thesis, Deptment of Computing and Electrical Engineering, Heriot-Watt University, May 2000.

[107] T. Schnekenburger. Load Balancing in CORBA: A Survey of Concepts, Patterns, and Techniques. *J. Supercomput.*, 15(2):141–161, 2000.

[108] T. Sekiguchi. JavaGo, May 2006. http://homepage.mac.com/t.sekiguchi/javago/index.html.

[109] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation. In *COORDINATION '99: Proceedings of the Third International Conference on Coordination Languages and Models*, pages 211–226, London, UK, 1999. Springer-Verlag.

[110] B. A. Shirazi, A. R. Hurson, and K. M. Kavi, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[111] D. B. Skillicorn. *Parallelism and the Bird-Meertens Formalism*. Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1992.

[112] D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1995.

[113] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, 1995.

[114] H. Sreekantaswamy, S. Chanson, and A. Wagner. Performance Prediction Modeling of Multicomputers. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 278–285, Yokohama, Japan, Jun 1992. IEEE Xplore.

[115] D. C. Submitted. *CLUMPS: A Candidate Model Of Efficient, General Purpose Parallel Computation*. PhD thesis, Department of Computer Science, the University of Exeter, Oct. 1994.

[116] K. P. Sycara. The Many Faces of Agents. *AI Magazine*, 19(2): Summer:11–12, 1998.

[117] D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. *International Journal of Information Security*, 2(3):126–144, 2004.

[118] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable Remote Execution Facilities for the V-system. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 2–12, Orcas Island, Washington, United States, 1985. ACM Press, New York, NY, USA.

[119] S. Thompson. *The Craft of Functional Programming*. Addison-Wesley, second edition, 1999.

[120] B. Thomsen, L. Leth, S. Prasad, T.-S. Kuo, A. Kramer, F. Knabe, and A. Giacalone. Facile Antigua Release – Programming Guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, Munich, Germany, 1993.

[121] H. W. To. *Optimising the Parallel Behaviour of Combinations of Program Components*. PhD thesis, University of London Imperial College of Science, Technology and Medicine Department of Computing, 1995.

[122] P. T. Tosic and G. A. Agha. Towards a Hierarchical Taxonomy of Autonomous Agents. In *IEEE SMC'2004: International Conference on Systems, Man and Cybernetics*, pages 3421–3426, Hague, The Netherlands, October 2004. IEEE Xplore.

[123] M. Vancea and A. Vancea. A Cost Model for the AND Parallel Execution of Logic Programs, 2002. STUDIA UNIV. BABES-BOLYAI, INFORMATICA, Volume XLVII, Number 2: 67-74.

[124] P. Wadler. Strictness Analysis Aids Time Analysis. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132, San Diego, California, United States, 1988. ACM Press, New York, USA.

[125] B. Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9):528–539, 1975.

[126] B. Wegbreit. Verifying program performance. *J. ACM*, 23(4):691–699, 1976.

[127] T. Wheeler. Voyager Architecture Best Practices. Technical report, Recursion Software, March 2005. http://www.recursionsw.com/Voyager/2005-03-31-Voyager_Architecture_Best_Practices.pdf.

[128] J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*, pages 437–472, Menlo Park, CA, 1997. AAAI/MIT Press.

[129] Wikipedia. Mean absolute percentage error. December 2006, Accessed: March 2006. http://en.wikipedia.org/wiki/Mean_Absolute_Percentage_Error.

[130] Wikipedia. Continuation. February 2007, Accessed: March 2007. http://en.wikipedia.org/wiki/Continuation.

[131] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, pages 42–52, Palm Springs, CA, USA, 2000. IEEE Educational Activities Department.

[132] P. T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation.* PhD thesis, Wolfson College, University of Cambridge, April 2000.

[133] M. Wooldridge. Agent-Based Software Engineering. *IEE Proceedings Software Engineering*, 144(1):26–37, 1997.

[134] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, June 1995.

[135] J. Xu. Comparative Evaluation of a Parallel Genetic Algorithm. Master's thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, September 2002.

[136] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software - Practice and Experience*, 23(12):1305–1336, 1993.

[137] A. Y. Zomaya and Y.-H. Teh. Observations on Using Genetic Algorithms for Dynamic Load-Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(9):899–911, 2001.