Support for Flexible and Transparent Distributed Computing

Hao Liu

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University of London.

Department of Computer Science

University College London

2010

I, Hao Liu, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Modern distributed computing developed from the traditional supercomputing community rooted firmly in the culture of batch management. Therefore, the field has been dominated by queuing-based resource managers and work flow based job submission environments where static resource demands needed be determined and reserved prior to launching executions. This has made it difficult to support resource environments (e.g. Grid, Cloud) where the available resources as well as the resource requirements of applications may be both dynamic and unpredictable. This thesis introduces a flexible execution model where the compute capacity can be adapted to fit the needs of applications as they change during execution. Resource provision in this model is based on a fine-grained, self-service approach instead of the traditional one-time, system-level model. The thesis introduces a middleware based Application Agent (AA) that provides a platform for the applications to dynamically interact and negotiate resources with the underlying resource infrastructure.

We also consider the issue of transparency, i.e., hiding the provision and management of the distributed environment. This is the key to attracting public to use the technology. The AA not only replaces user-controlled process of preparing and executing an application with a transparent software-controlled process, it also hides the complexity of selecting right resources to ensure execution QoS. This service is provided by an On-line Feedback-based Automatic Resource Configuration (OAC) mechanism cooperating with the flexible execution model. The AA constantly monitors utility-based feedbacks from the application during execution and thus is able to learn its behaviour and resource characteristics. This allows it to automatically compose the most efficient execution environment on the fly and satisfy any execution requirements defined by users. Two policies are introduced to supervise the information learning and resource tuning in the OAC. The Utility Classification policy classifies hosts according to their historical performance contributions to the application. According to this classification, the AA chooses high utility hosts and withdraws low utility hosts to configure an optimum environment. The Desired Processing Power Estimation (DPPE) policy dynamically configures the execution environment according to the estimated desired total processing power needed to satisfy users' execution requirements.

Through the introducing of flexibility and transparency, a user is able to run a dynamic/normal distributed application anywhere with optimised execution performance, without managing distributed resources. Based on the standalone model, the thesis further introduces a federated resource negotiation framework as a step forward towards an autonomous multi-user distributed computing world.

Acknowledgements

Pursuing a PhD project is like climbing a high peak, step by step, accompanied with happiness, bitterness, hardships, frustration, encouragement and trust and with so many people's kind help. At this point, I would like to express my deepest gratitude to all those who gave me the possibility to complete this thesis.

This work would never have been possible without the guidance of my supervisor Søren-Aksel Sørensen to whom I am deeply indebted. I would like to thank him for providing research resources, technical discussions, stimulating suggestions and encouragement in all the time of research for and writing of this thesis. His wide knowledge in various fields and logical thinking have been of great value for me. I am also deeply grateful to my supervisor David S. Rosenblum, for his constructive comments, and for his important administrative support throughout this work.

It has been a great honour working with people who sit in the corner of the 7th floor. The discussions with them have always enhanced my research and clarified my questions. I would like to acknowledge Amril Nazir for his significant suggestions for research methodology as well as the help during software implementing and paper producing. I would like to thank Clovis Chapman for the discussions of grid scheduling and Kalman filter as well as the help in thesis writing. I would also like to thank Mohamed Ahmed particularly for identifying the potential for the use of reinforcement learning.

I would like to thank all the members of the computer cluster group, and particularly Tristan Clark, for providing computational resources and technical support during the experiments in the thesis. I would also like to thank all the other departmental support staff for their various help during the time of my PhD study.

I have enjoyed working as a teaching assistant with many lecturers, particularly Philip Treleaven, discussions with whom had a great enlightenment to me for both my research and life. The work with them had also allowed me to discuss my research with undergraduate and master students, who gave me innovative suggestions. Thanks go to all of them.

Finally, I would like to express my gratitude to many anonymous people, who provided great open source software, answered my technical questions, reviewed my work through the Internet.

I dedicate this thesis to my mum, my dad, and my wife, without whose constant encouragement, unconditional love and support I would not have been able to undertake this endeavour.

List of Abbreviations

AA	Application Agent
AppLes	Application-Level Scheduling
ART	Automatic Resource Tuner
AU	Average Utility
BSP	Bulk Synchronous Parallel
CFD	Computational Fluid Dynamics
DAG	Directed Acyclic Graph
DPPE	Desired Processing Power Estimation
DRM	Distributed Resource Manager
DRMAA	Distributed Resource Management Application API
DT	Domain Tree
ESD	Execution Satisfaction Degree
FLOPS	Floating point Operations Per Second
GARA	Globus Architecture for Reservation and Allocation
GRAM	Globus Resource Allocation Manager
GSI	Globus Security Infrastructure
HEP	High Energy Physics
JSDL	Job Specification Description Language
LComm	Local Communication
MDS	Monitoring and Discovery Service
MPI	Message Passing Interface
MPP	Massively Parallel Processing
MU	Marginal Utility
NAT	Network Address Translation
NWS	Network Weather Service
OAC	Online Feedback-based Automatic Resource Configuration
PBS	Portable Batch Queuing System
PMWComm	Process Management and Wide-area Communication
PSA	Parameter Sweep Application
PVM	Parallel Virtual Machine

QoS	Quality of Service
RAS	Resource Acquisition Server
RC	Resource Configuration
RD	Resource Discoverer
RTT	Round-Trip Time
SGE	Sun Grid Engine
SLA	Service Level Agreement
SP	Selection Probability
SPEC	Standard Performance Evaluation Corporation
SPMD	Single Process, Multiple Data
UD/UT/UC	unit data/unit time/unit cost
UPID	Unique Process ID
VM	Virtual Machine

Contents

1 Introduction **15**

1.1 Problem Statement 15

1.2 Research Scope 17

1.2.1 Distributed Memory System 17

1.2.2 Computational Distributed Application 17

1.2.3 User-Level Middleware 18

1.2.4 Dynamic Applications 18

1.3 Research Contribution 18

1.4 Thesis Outline 20

2 Background **22**

2.1 Distributed Resource Infrastructure 22

2.1.1 Supercomputer 22

2.1.2 Cluster Computing 23

2.1.3 Grid Computing 24

2.1.4 Cloud Computing 26

2.2 Distributed Applications 27

2.2.1 Embarrassingly Parallel Computation 27

2.2.2 Parallel Computation 27

2.2.3 Workflow 28

2.2.4 Dynamic Applications 29

2.3 Distributed Application Execution 30

2.4 Chapter Summary 32

3 Motivation and Requirements **33**

3.1 Flexible Execution 34

3.1.1 On-line Resource Allocation 34

3.1.2 Dynamic Load Balancing 35

3.2 Providing Resource Management Transparency 37

3.2.1 Application Frameworks 37

3.2.2 Automatic Resource Selection 38

3.3 Chapter Summary 41

4 Model Design 42

4.1 Application Agent (AA) Layer 43

4.2 Application Layer 44

4.2.1 Distributed Object Graphs 45

4.2.2 Load Balancing 46

4.2.3 Master-Worker (Task-based) 47

4.2.4 Developing Adaptive Applications 49

4.3 Transparent Resource Management 49

4.4 Chapter Summary 50

5 Application Agent 51

5.1 Overview 52

5.2 Application Programming Interface 53

5.3 Architecture 54

5.3.1 Unique Process ID (UPID) 54

5.3.2 LComm 56

5.3.3 PMWComm 56

5.3.4 Resource Discoverer (RD) 60

5.3.5 Information Record and Synchronization 61

5.4 Implementation 62

5.4.1 PVM Implementation 64

5.5 Evaluation 65

5.5.1 Basic Performance Test 65

5.5.2 Feasibility Test 67

5.5.3 Flexible Execution VS Static Execution 69

5.6 Chapter Summary 71

6 Automatic Resource Configuration and the Utility Classification Policy 73

6.1 Execution Satisfaction Degree 74

6.2 Online Feedback-based Automatic Resource Configuration 76

6.2.1 Implementation 77

6.3 The Utility Classification Policy 80

6.3.1 The Policy 81

6.3.2 Policy Evaluation 84

6.4 Experiment I - Two Dimension Heat Equation 85

6.4.1 Simulation Setup 85

6.4.2 Simulation Validation 86

6.4.3 Experimental Setup 87

6.4.4 Experiment I.1 - Homogenous Environment . . . 88
6.4.5 Experiment I.2 - Heterogeneous Environment . . . 90
6.4.6 Experiment I.3 - Heterogeneous Environment with Cross-domain Delay . . . 92
6.4.7 Experiment I.4 - Dynamic Resource Environment . . . 94
6.5 Experiment II - Economy-based Task Farming . . . 95
6.5.1 Experimental Setup . . . 95
6.5.2 Experiment II.1 . . . 97
6.5.3 Experiment II.2 - Small Grain Size . . . 101
6.6 The Utility Classification Policy Summary . . . 103

7 Automatic Resource Configuration and the DPPE Policy 105
7.1 The DPPE Policy . . . 105
7.1.1 Applying Kalman Filter . . . 106
7.1.2 The Policy . . . 108
7.1.3 Experiment I - Mandelbrot . . . 108
7.1.4 Experiment II - Bulk Synchronous Parallel . . . 111
7.1.5 Policy Summary . . . 113
7.2 OAC Related Work . . . 114
7.2.1 Control Theory and Reinforcement Learning . . . 114
7.2.2 Feedback-based Scheduling . . . 115
7.2.3 Performance Measurement and Tuning . . . 115
7.3 Chapter 6 & Chapter 7 Summary . . . 116

8 Federated Resource Negotiation 118
8.1 Framework . . . 119
8.1.1 Negotiation Policies . . . 120
8.2 Using the Federated Negotiation to Improve Multiple Application Satisfactions . . . 121
8.2.1 Experiment . . . 121
8.3 Chapter Summary . . . 123

9 Conclusion 124
9.1 Contributions . . . 124
9.1.1 The Two-layer Model and the Application Agent . . . 125
9.1.2 Online Automatic Resource Configuration and Two Tuning Policies . . . 126
9.1.3 Federated Resource Negotiation . . . 126
9.2 Discussion of Adaptive Application Related Work . . . 126
9.3 Future Directions . . . 128
9.3.1 Development of the Application Agent . . . 128
9.3.2 OAC Investigation . . . 132
9.3.3 Autonomous Distributed Computing . . . 132

Appendices **132**

A Condor and SGE Systems **133**

A.1 Condor . . . 133

A.1.1 Run Applications in Condor . . . 133

A.2 Sun Grid Engine . . . 135

A.2.1 Run Applications in SGE . . . 135

B Application Agent (AA) User Manual **137**

B.1 Feature . . . 137

B.2 Installation . . . 137

B.3 Run . . . 138

B.4 AA Console . . . 139

B.5 Main API . . . 140

C Simulating the 2D Heat Equation Experiment **143**

C.1 Simulation . . . 144

D Lunar Surface Temperature Mapping **145**

D.1 A Little Experiment Background . . . 145

D.2 Mathematical Algorithms . . . 145

D.2.1 Surface Temperature . . . 146

D.2.2 Heat Conduction . . . 146

D.2.3 Energy Flux Fraction . . . 147

D.3 Application Development . . . 148

D.3.1 Load Balancing . . . 148

D.3.2 ESD Report . . . 149

D.4 Application Execution . . . 149

D.4.1 From the User's Point of View . . . 149

D.4.2 Under the Hood . . . 149

Bibliography **152**

List of Figures

2.1 Distributed application types. 27
2.2 A one-dimensional finite difference parallel application 28
2.3 A dynamic application example 30
2.4 Resource reservation with backfilling scheduling. 31
2.5 The long tail problem 31

3.1 Unstructured communication pattern 37
3.2 Steps in AppLes methodology 39

4.1 Traditional distribution computing model VS Proposed distributed computing model . . 43
4.2 An overview of the two-layer model. 44
4.3 Distributed automata graph 45
4.4 The CFD-OG model. 45
4.5 Master-worker paradigm 48
4.6 An overview of the online feedback-based automatic resource configuration. 50

5.1 AA Daemons 53
5.2 Unique Process Identifier 55
5.3 UPID examples 56
5.4 Local communication performed by the LComm implemented by PVM or MPI. 56
5.5 Cross-domain communication 58
5.6 The process addition is achieved by PMWComms 58
5.7 Place multiple PMWComms to connect multiple domains 59
5.8 A Domain Tree. 59
5.9 How the AA works with DRMs. 61
5.10 Information is synchronized hierarchically. 62
5.11 The implementation of the AA. 63
5.12 The process of obtaining a resource. 64
5.13 AA basic performance test 66
5.14 The AA deploys an application in the Condor and HPC cluster from a laptop. 69
5.15 The dynamic change of the AA virtual machine 70
5.16 Dynamic process addition for the Mandelbrot application 70

6.1 The plots of ESD functions. 76
6.2 How the AA performs the OAC. 77
6.3 The Automatic Resource Tuner (ART) daemon 78
6.4 The automatic resource configuration: application code structure. 79
6.5 A WWG testbed composed of 11 types of resources. [BM02] 81
6.6 2D heat equation computation. 86
6.7 The grid cells are distributed into 3 hosts 87
6.8 Simulation validation. 87
6.9 The filtered ESD is much smoother. 88
6.10 Results of experiment I.1 - homogeneous environment, QoS = 0.8 iterations/UT 88
6.11 Results of experiment I.1 - homogeneous environment, QoS = 1.0 iterations/UT 90
6.12 Results of experiment I.2 - heterogeneous environment, QoS = 0.8 iterations/UT 91
6.13 Results of experiment I.3, central locus = cluster E 92
6.14 Results of experiment I.3, central locus = cluster D 94
6.15 Results of experiment I.4 - dynamic environment, QoS = 0.8 iterations/UT 94
6.16 Simulated parameter sweep computation 96
6.17 Define the ESD based on the deadline QoS requirement 98
6.18 The historical AU of type $5 \sim 9$ in the first execution of test 2. 99
6.19 First execution in Experiment II.1 101
6.20 Second execution in Experiment II.1 103
6.21 First execution in Experiment II.2 104

7.1 A histogram of observed Φ noise. 107
7.2 Satisfying the 300 seconds deadline. 110
7.3 Satisfying the 600 seconds deadline. 111
7.4 Resource usage comparison between automatic and manual resource selection 111
7.5 The BSP master-worker model 112
7.6 Resource configuration for the BSP application. QoS = 0.03iterations/second 114

8.1 A multi-AA distributed computing world 119
8.2 The federated resource negotiation 120
8.3 An experiment of the federated resource negotiation. 122

9.1 The two-layer model in the future. 131

C.1 Object-oriented 2D heat equation experiment simulation framework 144

D.1 Temperature mapping application execution speed 150
D.2 The number of resource configured during the temperature mapping execution. 150
D.3 Lunar temperature mapping 151

List of Tables

5.1 A domain table . . . 62
5.2 A host table . . . 63
5.3 Flexible execution VS static execution . . . 69

6.1 Simulated resource environment for experiment I.1 $\sim$ I.4. . . . 89
6.2 Simulated resource environment composed of 10 heterogeneous hosts. . . . 96
6.3 Test results in experiment II.1 . . . 98
6.4 Comparing the Utility Classification with the DBC algorithms . . . 102
6.5 Test results in experiment II.2. . . . 102

7.1 Summary of three actions: addition, releasing, and replacing. . . . 109
7.2 Comparison of two OAC polices . . . 117

8.1 An example of the AA-World table . . . 119

9.1 Software related to adaptive applications. . . . 128

List of Algorithms

1 Generalized Master-Worker algorithm. 48
2 Algorithm to find the connection route in a DT. 60
3 A local load balancing algorithm 67
4 AU Update 83
5 Selection Probability (SP) Calculation 83
6 Tuning Algorithm 84
7 A simple global load balancing algorithm 86

Chapter 1

Introduction

In recent years, the way computers are used has changed significantly. The emergence of small electronic devices with excellent display capabilities and graphical user interfaces has completed the natural evolution towards an interactive information society that was started by the desktop computer. Users now expect such personal devices to perform tasks that are well beyond their processing or storage capabilities and the idea of a portable, handheld "supercomputer" no longer seems a science fiction concept. At the same time the supercomputer concept has changed significantly. By connecting hundreds of readily available workstations it is now possible to generate virtual computers with processing or storage capabilities that rival those of the high cost, dedicated supercomputer. The advent of high-speed networks has enabled the integration of computational resources which are geographically distributed and administered at different domains. Such distributed systems, normally managed by *Distributed Resource Mangers* (DRMs) and resource co-allocators [CFK99], provide users with simple access to a distributed set of computational resources to run resource hungry applications.

The computer revolution has not only increased our expectations regarding the power of personal processing devices, it has also changed the way we operate these devices. The age where users were prepared to formulate a problem, submit it and wait long periods of time for answers is long gone. The user of today expects not only to receive continuous updates regarding the progress of an application but also to steer its progress and these expectations remove some of the traditional assumptions about high performance applications, namely that they are repetitive, predictable and that their results are not urgently needed.

1.1 Problem Statement

To run an application in a distributed system, users are currently required to submit it to the DRMs. They will in turn schedule resources, deploy application components and start executions on behalf of users. In most systems, jobs are submitted to a batch queue where they will sit for some unknown length of time until the required combination of resource becomes available. Or users can request a reservation. Advance Reservation [Fos99] reserves the required resources in the future time slot and fixes the start time for the job. In order to be effective, detailed knowledge is required to predict the execution time and perform more efficient scheduling by using other approaches such as backfilling

[LZ07]. The job execution model is a classic batch processing environment: users submit jobs and wait for results. Jobs are set up in advance so they can be run to completion without human interaction. It is perfectly suitable for supporting the embarrassingly parallel workload or batch jobs where tasks are independent of each other. Some examples include large scale face recognition that involves comparing thousands of input faces with similarly large number of faces, computer simulations comparing many independent scenarios, genetic algorithms and other evolutionary computation metaheuristics, etc.

This model requires an execution plan prepared before commencing the execution. For example, how many resources are needed to run the job, what kinds of resources are capable and how long the execution is going to take? Such information can be extremely hard to provide with any accuracy, especially if the application or the resource environment is dynamic. For example, during execution dynamic applications may significant change the structure and size of the datasets they handle as well as the way they process the data and as a consequence their resource requirements may change significantly if they are to meet specific performance benchmarks. CFD-OG (Computational Fluid Dynamics using Object Graphs) [LNS08, NLS07] is one of the examples in that respect. Such applications do not fit within the batch processing model because their behaviour is unpredictable. Furthermore, the assumption that the resource remains static over long periods of time is unsustainable. New resources may unexpectedly become available and resources may suddenly fail during the execution. This is common in the current grid system [KF98]. An advanced resource allocation plan could be ruined or better choice could be available. Moreover, since resource allocation is statically performed before starting execution, a job that needs multiple resources could have to wait a long time to have enough resources available to run. This weak point is especially prominent when supporting effective, high performance, interactive applications where users require results to be seen in a very short time. For example, some form of visualization for a simulated system.

One solution to the above problems is to make the execution environment more flexible. A flexible execution environment should be able to adapt to changes to the resource situation rather than relying on a static plan based on according outdated information. The execution should be able to introduce more resources if the current allocation is insufficient or replace resources when a more suitable variety becomes available. In this way, execution can be started quickly as long as the minimum resource requirements are met. Resource provision can then be justified according to the needs of the application and the status of the resource pool. For example, if the processing requirements suddenly rise due to steering by a user, the system can respond by immediately allocating more resources to the application to maintain specific performance requirements, such as smooth time progression of execution [LNS07].

In addition to the flexibility requirements, The modern processing environment also needs to be transparent - the user should not need to worry about technical details. In distributed computing, transparency is defined as hiding the distributed nature of the execution of an application from its users so that the application appears and functions as a local desktop application. Current distributed computing support provides some transparency, e.g. resource allocation transparency so the users of a distributed system do not have to be aware of the identity of a resource or where it is physically located. However,

it lacks sufficient resource selection transparency, i.e. users will still need to identify the quantity and quality the resources they reserve. Users must write a work flow description and submit it to the system via some specified interface. Therefore users must be aware of the performance characteristics of the available resources either based on queries to a resource information system (a system provides users resource information such as processor architecture, CPU load, operating system) or through their own experience, and be able to use this information to calculate the number and types of resources they need to perform the execution. Users are also required to be familiar with the various job description languages and submission commands. Many users, especially those who do not have a computer science background will not possess such knowledge.

In a high transparency computing environment, users should not be aware of specific services like grid portals but be able simply to run a distributed application as easily as starting a "helloworld" program on their local machine. Users should also be able to interact with the application anytime they want and have no awareness of how the underlying distributed computing environment is dynamically configured. From the user's point of view, a distributed application should behave exactly like a local desktop application, and the local computer should act like a virtual supercomputer to support such massive computation.

Our mission is therefore to address these problems and to introduce *flexibility* and high *transparency* into distributed computing. The discrete, user-controlled stages of preparing and executing a distributed application should be replaced with an end-to-end software-controlled and flexible process.

1.2 Research Scope

1.2.1 Distributed Memory System

Main memory in a distributed system is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space). In this thesis we focus on distributed memory systems like clusters and computational grids where each computational entity (host) controls its own local memory and entities communicate with each other by message passing.

1.2.2 Computational Distributed Application

There are two main reasons for using distributed systems and distributed computing. First, the very nature of the application may require the use of a communication network that connects several computers. For example, a MVC (Model - View - Controller) application in which the user interface, functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms. Second, there are many cases in which the use of a single computer would be possible in principle, but the use of a distributed system is beneficial for practical reasons. For example, it may be more cost-efficient to obtain the desired level of performance by using a cluster of several low-end computers, in comparison with a single high-end computer. The thesis focuses on the second case, particularly the problem that requires a great number of computer processing cycles to improve the turnaround time, speed, or cost of execution.

1.2.3 User-Level Middleware

Distributed computing involves the integration and collaborative use of computers, networks and scientific software owned and managed by multiple organizations. For this purpose, distributed computing middleware implements protocols and algorithms with the aim to allow users easy access to distributed resources. According to Buyya et al. [RS05], the middleware can be divided into two types. 1. Infrastructure middleware offers services such as job scheduling, allocation of resources, storage access, information registration and discovery, security, and aspects of QoS such as resource reservation. These services abstract the complexity and heterogeneity of the physical resources by providing a consistent method for accessing distributed resources. 2. User-level middleware utilizes the interfaces provided by the low-level middleware to provide higher level abstractions and services to individual applications. These include application development environments, programming tools and resource brokers for managing resources and scheduling application tasks for execution on global resources. Here, we focus on the user-level middleware that operates within the context of existing resource infrastructure middleware, which is represented by DRMs such as Condor [TWML02] and the Sun Grid Engine [Gen01].

1.2.4 Dynamic Applications

Distributed applications can be classified by their resource request behaviours to be either static applications whose resource requirements do not change during executions, or dynamic applications that may require very different resource support level during their executions (see details in Section 2.2). Although our model is designed to support all types of applications, dynamic applications, especially those which have specific execution QoS requirements are the ones supposed to benefit most from our flexible execution model.

1.3 Research Contribution

The goal of this thesis is to use user-level middleware incorporating a new set of algorithms to construct a distributed execution model that is flexible, adaptive, and transparent. In this model, users can easily execute both static and dynamic distributed applications with satisfactory execution performance, without considering how to manage the underlying distributed resources.

It makes the following contributions:

- **Two-layer Model**: Instead of using centrally managed DRMs to control the execution, we design a two-layer model to support the flexible and transparent requirements. The lower layer takes charge of constructing an exclusive execution environment as required; while the upper layer, the application, is responsible for making the best use of the provided execution environment. This design allows application programmers and users to be insulated from the resource management issues and also provides the possibility to manage the execution environment in a self-service basis during the execution.

- **Application Agent (AA)**: The Application Agent (AA), which is the lower layer, is a middleware that cooperates with existing DRMs to dynamically create an exclusive execution environment to

run a distributed application. It supports dynamic resource addition and release during the execution. By using the AA, users are able to invoke an application in their local machines and the execution is transparently performed on remote wide-area resources. We provide the architecture of the AA in the thesis as well as an implementation based on PVM [GBD+94]. This implementation is used to demonstrate, through experiment, that the performance can be enhanced through the flexible execution.

- **On-line Automatic Resource Configuration**: Automatic resource selection is used to provide the high-transparency of distributed computing. We propose a novel On-line Feedback-based Automatic Resource Configuration (OAC) approach based on the AA framework to configure the execution environment without the guidance of users or the application. By learning the application execution behaviour and resource characteristics according to feedback from the application, the AA uses the acquired knowledge to select effective resources on the fly to satisfy the execution requirements (e.g. a deadline, a speed, or a budget). The approach differs from other automatic resource selection approaches in that the AA does not require prior resource/application knowledge and the configuration process is dynamically achieved according to application's requirements. We introduce the Execution Satisfaction Degree (ESD), a measure of the relative satisfaction of execution QoS requirements, for reporting the execution feedbacks. In this thesis we propose two policies - the *Utility Classification* and the *Desired Processing Power Estimation (DPPE)* - to supervise information learning and resource tuning in the OAC.

- **The Utilities Classification Policy**: The Utility Classification policy embedded in the OAC classifies hosts according to their historical performance contributions to the application. According to this classification, the AA chooses high utility hosts and withdraws low utility hosts to configure an optimum environment to satisfy user's QoS requirements. An important feature of the policy is that it does not require users to provide any information regarding applications nor resources in advance. This unique feature helps users who do not have a parallel computing background to execute distributed applications very easily. Moreover, the classification can be used to guide either manual or automatic resource selection in future executions.

- **The Desired Processing Power Estimation (DPPE) Policy**: The DPPE policy embedded in the OAC dynamically configures the execution environment using the least amount of resources according to the estimated desired total processing power needed to satisfy users' execution requirements. Its quick tuning process makes it suitable to support short running applications. It is especially useful to satisfy deadline-constrain job execution without the need for the users to provide information such as estimated job length in advance.

- **Federated Resource Negotiation**: Based on the flexibility and transparency of the standalone execution model, resources can be more effectively used among multiple AA-enabled applications in a federated way. We introduce a federated resource negotiation framework where an application is allowed to ask peer applications to donate resources in order to maintain certain performance.

We demonstrate the effectiveness of the framework by using it to improve the application satisfactions under a competitive resource environment. The framework points towards an autonomous multi-user distributed computing world.

The contributions are partly based on the following original publications:

1. Liu, H. & Sørenson, S.-A., On-line Feedback-based Automatic Resource Configuration for Distributed Applications, *Cluster Computing: the Journal of Networks, Software Tools and Applications*, Accepted, 2010, DOI 10.1007/s10586-010-0123-x

2. Liu, H.; Sørenson, S.-A. & Nazir, A., On-line Automatic Resource Configuration for Distributed Applications, *2nd IEEE International Workshop on Internet and Distributed Computing Systems*, 2009

3. Liu, H.; Sørenson, S.-A. & Nazir, A., On-line Automatic Resource Selection in Distributed Computing, *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 2009, 1-9

4. Liu, H.; Sørensen, S.-A. & Nazir, A., A Lightweight Approach of Automatic Resource Configuration in Distributed Computing, *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, 2009, 506-511

5. Liu, H.; Nazir, A. & Søensen, S.-A., A Software Framework to Support Adaptive Applications in Distributed/Parallel Computing, *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, 2009, 563-570

6. Liu, H.; Nazir, A. & Søensen, S.-A., Preliminary Resource Management for Dynamic Parallel Applications in the Grid, *GridNets*, 2008, 70-80

7. Liu, H.; Nazir, A. & Sørenson, S.-A., Resource Management Support for Smooth Application Execution in a Dynamic and Competitive Environment, *SKG '07: Proceedings of the Third International Conference on Semantics, Knowledge and Grid, IEEE Computer Society*, 2007, 438-441

1.4 Thesis Outline

The outline of thesis is as follows:

Chapter 2 introduces the research background including distributed computing concept, common distributed infrastructure, distributed applications, and the way to run applications on the distributed infrastructure.

Chapter 3 introduces what motivated us to produce the thesis. We investigate the limitation of current distributed computing model for supporting required flexibility and transparency, as well as the necessity to introduce a new distributed computing model by providing new middleware and algorithms.

Chapter 4 introduces the design of a two-layer flexible and transparent distributed computing model. We present the description of components (Applications, the Application Agent, and DRMs) in the model and how they interact with each other.

Chapter 5 introduces the Application Agent software. We present how the AA provides flexible execution services, such as dynamic resource allocation, "run applications anywhere" and wide-area computing. A C++ and PVM implementation is introduced. An experiment with Mandelbrot computation is presented to demonstrate the effectiveness of the flexible execution model supported by the AA.

Chapter 6 introduces the on-line automatic resource configuration approach with the help of the AA. Section 6.1 and 6.2 introduce its concept and framework. The rest of Chapter 6 introduces the Utility Classification policy and two simulation-based experiments.

Chapter 7 continues introducing the OAC approach by introducing the DPPE policy and two experiments for the validation. The rest of the chapter discusses OAC related work and comparison of the two policies.

(A more complex experiment (Lunar Surface Temperature Mapping) that evaluates the combined contributions of Chapter 4 $\sim$ 7 is given in Appendix D.)

Chapter 8 introduces the federated resource negotiation framework. An experiment that aims to improve application satisfactions under a competitive resource environment is presented to show the effectiveness of the framework.

Finally, Chapter 9 concludes the thesis work and outlines our planned future work in this area.

Chapter 2

Background

In this chapter, we introduce the research background, including a brief history of distributed resource infrastructure, common distributed application types and the way to run applications in current distributed computing.

2.1 Distributed Resource Infrastructure

From the middle of the 20th century, unprecedented complex scientific research such as High Energy Physics (HEP) has been looking for extremely large amount of computing resources. Supercomputers have traditionally been used to provide such immense processing capability, but in recent years, it has been more feasible to build "supercomputers" by connecting hundreds of cheap workstations to generate high processing capability. The first example was the Beowulf [SSB+95] which was a supercomputer-like system created from a collection of a number of desktop PCs connected by a high-speed network. The advent of high-speed networks has also enabled the integration of resources which are geographically distributed and administered at different domains. In late 1990s, Foster and Kesselman proposed the plug in the wall approach known today as grid computing [KF98], which aimed to make global geographically dispersed computer power as easy to access as an electric power grid. In the last two years, a new word "Cloud Computing" [Gru08] has been introduced to refer to a technology providing elastic and often virtualized distributed resources over the Internet.

2.1.1 Supercomputer

Supercomputers introduced in the 1960s were designed primarily by Seymour Cray, who was called "the father of supercomputing". Today, supercomputers are typically one-of-a-kind custom designs produced by "traditional" companies such as Cray, IBM and Hewlett-Packard.

In early days, most supercomputers were dedicated to running a vector processor, such the well-known Cray-1 installed in 1970s. Vector processing, normally with instruction pipeline, are designed to efficiently handle arithmetic operations on elements of arrays by performing operations on multiple data elements simultaneously. This is in contrast to a scalar processor which handles one element at a time using multiple instructions. Such machines are especially useful in high-performance scientific computing, where matrix and vector arithmetic are quite common. The Cray Y-MP and the Convex C3880 are two examples of vector processors used today.

Recently, supercomputers are more commonly designed by massively parallel processing (MPP). Today's most powerful supercomputer are all MPP systems, such as Earth Simulator, Blue Gene, ASCI White, ASCI red, ASCI Purple, and ASCI Thor's Hammer. A MPP supercomputer is a distributed memory computer system which consists of many individual homogenous nodes, each of which is essentially an independent computer in itself, and in turn consists of at least one processor, its own memory, and a link to the network that connects all the nodes together. Node communicate by passing messages, using standards such as Message Passing Interface (MPI) [mpi]. The cumulative output of the many constituent CPUs can result in large total peak FLOPS (Floating point Operations Per Second) numbers.

Today, as the cost of commodity computers continues to drop, supercomputers are now highly-tuned computer clusters using commodity processors (COTS) combined with custom interconnects. Clustering can provide significant performance benefits versus price. For example, the System X was a supercomputer assembled by Virginia Tech in the summer of 2003, that was originally composed of 1,100 Apple Power Mac G5 computers. System X ran at 12.25 Teraflops, (20.24 peak), and was ranked #280 in the July 2008 edition of the TOP500 list of the world's most powerful supercomputers. System X was constructed with a relatively low budget of just $5.2 million, in the span of only three months, thanks in large part to using off-the-shelf G5 computers. By comparison, the Earth Simulator, the fastest supercomputer at the time, cost approximately $400 million to build. Clusters now have become the low-cost and standard platforms for distributed computing.

2.1.2 Cluster Computing

A computer cluster is composed of multiple computing nodes working together closely so that in many respects they form a single computer to process computational jobs. Clusters are increasingly built by assembling the same or similar type of commodity machines that have one or several CPUs and CPU cores. Clusters are used in computational science typically when the tasks of a job are relatively independent of each other so that they can be farmed out to different nodes of the cluster. In some cases, tasks of a job may need to be processed in a parallel manner, where tasks may be required to interact with each other during the execution. Specialist interfaces exists for the parallel jobs such as the Parallel Virtual Machine (PVM) [GBD+94] and MPI, which enable the communication among tasks in a distributed memory environment.

The process of allocating jobs to individual nodes of a cluster is handled by a Distributed Resource Manager (DRM). The DRM allocates a task to a node using some resource allocation policy that may take into account node availability, user priority, job waiting time, etc. Examples of popular resource managers include Condor [TWML02], the Sun Grid Engine (SGE) [Gen01] and the Portable Batch Queuing System (PBS) [BHL+99]. They typically provide submission and monitor interface enabling users to specify jobs to be executed and keep track of the progress of execution. A job will be specified by a job description including a collection of resource requirement, such as processor architecture, memory size and operating system type; the files required for the execution, such as the executables and data files; and the execution parameters such as arguments, and environment variables. The DRMs match the requirements with the characteristics of available resources and allocate the eligible resources to run the

job.

DRMs also provide resource information interface, proving users with the means to obtain resource information that contains the resource availability, processor architecture, CPU load, operation system, etc.

Two DRMs, Condor and SGE, will be regularly referred to throughout the thesis as primary examples of DRM system. More details of the two systems can be found in Appendix A.

2.1.3 Grid Computing

The ancestor of the Grid is Metacomputing. The initial idea of Metacomputing was to interconnect supercomputer centers in order to achieve superior processing resources. One of the first infrastructures in this area, named Information Wide Area Year (I-WAY), was demonstrated at Supercomputing 1995 [iwa95]. This project strongly influenced the subsequent Grid computing activities. In late 90s, inspired by the pervasiveness, reliability, and ease of use of the electricity grid, Foster et al [KF98, Fos02], who lead the project I-WAY, began exploring the design and development of an analogous computational power grid for wide-area distributed computing.

Since grid computing is still emerging, the concept of it is evolving. The definition given by Foster is widely accepted and referred: grid computing strives to aggregate diverse, heterogeneous, geographically distributed and multiple-domain resource sharing and problem solving. The resources that grid computing is attempting to integrate are various. They include supercomputers, workstations, databases, storages, networks, software, special equipments, and even people. In this thesis, grid computing refers to large-scale geographically distributed, heterogeneous computational resources that may be from multiple clusters or administrative domains. In this case, grid computing can be seen as multi-cluster computing or wide-area distributed computing.

A typical grid computing architecture [CFK+98, Fel03] includes a meta-scheduler connecting a number of geographically distributed clusters that are managed by local DRMs. The meta-scheduler (e.g. GridWay [grib], GridSAM [gria]) optimizes computational workloads by combining an organization's multiple DRMs into a single aggregated view, allowing jobs to be directed to the best location (cluster) for execution. It integrates computational resources into a global infrastructures in such as manner that users no longer need to be aware of which resources are used to executing their jobs. Existing grid infrastructures are such as e-protein project [epr], eMinerals [BDCT05], and the Tera Grid [Bec05]. E-protein used the JYDE system [SSM+05, MSSJ05] to harness over 500 CPUs from 3 independent Linux clusters from University College London and Imperial College. And eMinerals leverages three Linux clusters located in Bath, Cambridge and UCL, a 40 node PBS cluster in Cambridge, two IBM pSeries parallel computers located in Reading, a Condor cluster that consists of more than 1000 Windows hosts in UCL, and a Apple XServ which is a 5 node cluster running Mac OSX.

One of main purpose of grid computing is to introduce common interfaces and standards that eliminate the heterogeneity from the resource access in different domains. Several grid middleware sytems therefore have been developed to resolve the differences that exist between submission, monitoring and query interfaces of the various DRMs. The Globus Toolkit [FK99, ea03, CFK+98] developed by the

Globus Alliance provides a number of components to standardize various aspects of remote interactions with DRMs. It provides a platform-independent job submission interface, the Globus Resource Allocation Manager (GRAM), which cooperates underlying DRMs to integrate the job submission method; a security framework, the Grid Security Infrastructure (GSI) and a resource information mechanism the Monitoring and Discovery Service (MDS). Interactions with the various components of the Globus toolkit are mapped to local management system specific calls and support is provided for a wide range of DRMs, including Condor, SGE and PBS. Similarly, GridSAM provides a common job submission and monitoring interface to multiple underlying DRAMs. As a Web Service based submission service, it implements the Job Submission Description Language (JSDL) [AA05] and a collection of DRM plug-ins that map JSDL requests and monitoring calls to system-specific calls.

In addition, grid computing is introducing a set of new open standards and protocols (e.g. Open Grid Services Architecture (OGSA), Web Services Resource Framework (WSRF), etc.) as a mean of facilitating interpretability between independent systems. More information can be found in [ogf, FKNT02].

Although the grid originally was considered one of the most promising approaches to harnessing wide area distributed resources, it has a lot of unique characteristics which make the computing in the grid environments difficult. The grand challenge imposed are such as resource heterogeneity, resource dynamics, resource co-allocation/domain autonomy, resource access security, etc. We introduce three challenges related to this thesis:

- **Resource Heterogeneity** Resources in the grid are usually heterogeneous in that these resources may have different hardware, such as instruction set, computer architectures, number of processor, physical memory size, CPU speed and so on, and also different software, such as different operation systems, file systems, and so on. The heterogeneity results in differing capability of processing jobs. The heterogeneity makes the execution performance difficult to access. Standard performance measures such as wall clock and cumulative CPU time do not separate application code and computing platform performance. Thus a resource which has a high SPEC (Standard Performance Evaluation Corporation) rank does not necessarily provide high performance to a specific application. In some case, we need to run the application on the resource to test its actual performance rather than only rely on the resource characteristics provided by resource information system such as MDS [CFFK01] and Network Weather Service (NWS)[WSH99].

- **Resource Dynamic Behaviour** In traditional parallel computing environments, such as a cluster, the pool of resources is assumed to be fixed or stable. In a grid, dynamics exists in both the networks and computational resources. First, a network shared by many applications cannot provide guaranteed bandwidth. This is particularly true when wide-area networks such as the Internet are involved. Second, both the availability and capability of computational resources will exhibit dynamic behaviour. On one hand new resources may join the grid, and on the other hand, resources may become unavailable due to failure or usages from owners (non-dedicated). The capability of resources may vary over time due to the contention among many parties who share the resources. These challenges pose significant obstacles on the application execution in the grid.

- **Resource Co-allocation** The large number of resources available in computational grids leads users to want to execute applications that are distributed across multiple domains; such as running a large simulation on 2 or more clusters. The difficulty with this is that different clusters may have different scheduling systems without any mechanisms to guarantee that an application obtains simultaneous access to the multiple resources. To address this problem, a co-allocator is used to coordinate with local DRMs and bring up the distributed pieces of the application. The Globus Resource Allocation Broker (GRAB) [CFK99] and DUROC [CFK99] (in Globus Toolkit 2.4) were designed for this purpose. An important issue in the resource co-allocation is that the required resources have to be available at the same time otherwise the computation cannot proceed. Globus has been using advanced reservation to guarantee the availability of multiple resources to the applications during specified time period [Fos99], in order to escape un-predicted waits.

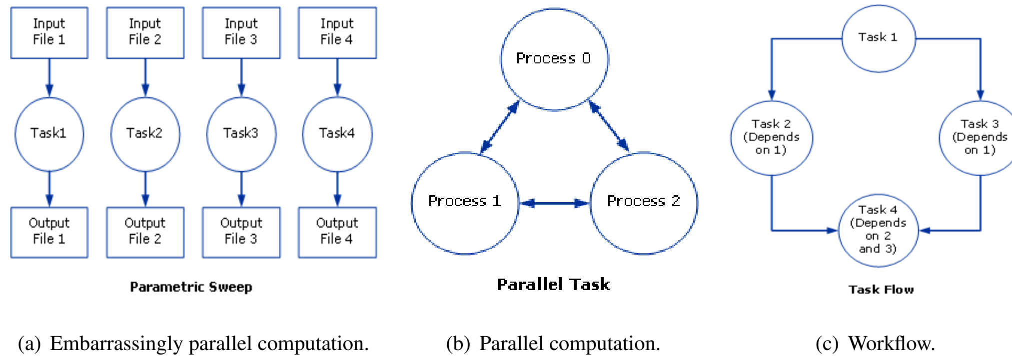
2.1.4 Cloud Computing

Cloud computing [Dan08, clo09], evolved from grid computing, is a style of computing in which dynamically scalable and often virtualised resources are provided as a service over the Internet. Users need not have knowledge of, expertise in, or control over the technology infrastructure "in the cloud" that supports them. Besides virtualisation, Cloud computing has three features that distinguishing it from other distributed computing types:

Elastic on-demand provisioning of resources on a fine-grained, self-service basis. The elastic nature is interesting to some traditional cluster computing customers who have workloads that have tremendous spikes and valleys. When these people need computing resources, they need a lot of it (hundreds, potentially thousands of cores). But their workloads (often CPU-intensive models) do not need to run all the time (like a webserver, or mailserver, or a CFD (Computational Fluid Dynamics) visualization). The self-service nature of cloud computing allows them to create elastic environments that expand and contract based on the workload and target performance parameters. This feature is similar to our flexible execution idea, where resources are provided dynamically according to the demand of the applications as well as the availability of resource. However we focus on high performance applications like CFD, while industry Clouds (e.g. Amazon EC2 [ama]) aims at enterprise applications like SAP, oracle databases, and other business applications. In addition, we focus on constructing a lightweight virtual machine that abstract over the communication mechanisms of computer clusters, while Cloud computing focuses on hardware visualization which hides the physical characteristics of a computing platform from users, instead showing another abstract scalable computing platform.

Device and location independence enable users to access systems transparently such as using a web browser regardless of their location or what device they are using, e.g., PC, mobile. As infrastructure is off-site (typically provided by a third-party) and accessed via the Internet the users can connect resources from anywhere. This feature is like our "transparency" idea which enables users to run an application anywhere without worrying about underlying resource management aspects.

Cost is greatly reduced and capital expenditure is converted to operational expenditure. This lowers barriers to entry, as infrastructure is typically provided by a third-party and does not need to be purchased



(a) Embarrassingly parallel computation. (b) Parallel computation. (c) Workflow.

Figure 2.1: Distributed application types.

for one-time or infrequent intensive computing tasks. Pricing on a *utility computing* basis is fine-grained with usage-based options and minimal or no IT skills are required for implementation.

2.2 Distributed Applications

The application running on the distributed computing infrastructure is different from a desktop application in that it is typically composed of a number of processes that run simultaneously on multiple computational resources with more or less communication in order to solve a big problem. The applications, sometimes called jobs, can be classified by their communication structures into three types: embarrassingly parallel computation, parallel computation and workflow. They can also be classified by their resource requirements to be static and dynamic applications.

2.2.1 Embarrassingly Parallel Computation

In so-called embarrassingly parallel problems, a computation consists of a number of tasks that can execute independently, without communication. Embarrassingly parallel computations can immediately be divided into independent parts which multiple processors can work on simultaneously. Once the work has been allocated, no communication is required between the processors, so a speedup of n is possible, given n processors. Often the problem can be divided into many more parts than there are processors available to work on the parts, so the maximum speedup possible via parallel implementation is constrained by the number of processors.

An example is the Parameter Sweep Application (PSA), in which the same computation must be performed using a range of different input parameters. The parameter values are read from a set of input files, and the results of the different computations are written to an output file. See Figure 2.1(a).

Since tasks are independent of each other and normally do not need user intervention, they are perfectly suitable for current DRMs that are based on queuing system. Various scheduling algorithms can be applied to optimize performance (Minimum Completion Time) for this computation. Popular scheduling algorithms [IK77, DA06, MYWWSZ00] include FCFS, Min-min, Min-max and Sufferage.

2.2.2 Parallel Computation

A parallel computation (Figure 2.1(b)) normally means its tasks must be running simultaneously. It can take a number of forms, depending on the application and the software that supports it. We define the par-

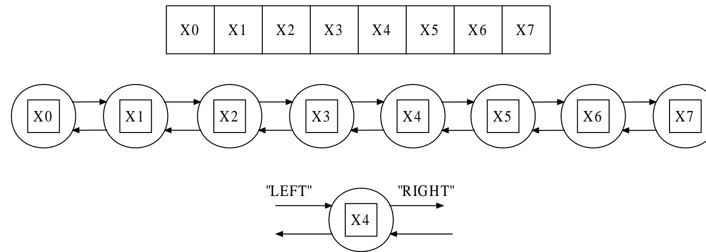


Figure 2.2: A one-dimensional finite difference parallel application. From top to bottom: the one-dimensional vector X, where $N = 8$; the application structure, showing the 8 processes, each encapsulating a single data value and communicating to its left and right neighbours processes; and the structure of a single process, showing its two inports and outports. [Fos95]

allel application as a collection of closely related processes, normally executing the same code, perform computations on different portions of the workload, usually involving the periodic exchange of intermediate results by communication [GBD$^+$94]. It is also called array processing or loosely synchronous computation.

One typical example is the finite differences computation [Fos95]. Here we consider a one-dimensional finite difference problem, in which we have a vector $X^{(0)}$ of size N and must compute $X^{(T)}$, where
$0 < i < N-1, 0 \leq t < T : X_i^{t+1} = (X_{i-1}^t + 2X_i^t + X_{i+1}^t)/4$.
That is, we must repeatedly update each element of X , with no element being updated in step $t+1$ until its neighbours have been updated in step t.

A parallel application for this problem has an SPMD (Single Process, Multiple Data) structure. It has N processes, one for each point in X. The i_{th} process is given the value and is responsible for computing, in T steps, the values $X_i^{(1)}$, $X_i^{(2)}$... $X_i^{(T)}$. Hence, as in Figure 2.2, at each iteration t, for any process P_i, it performs computation and communication according to:
1. sends its data $X_i^{(t)}$ to its left and right process P_{i-1} and P_{i+1}.
2. receives $X_{i-1}^{(t)}$ and $X_{i+1}^{(t)}$ from its its left and right process P_{i-1} and P_{i+1}.
3. uses these values to compute $X_i^{(t+1)}$.

Since all the processes are closely related, they have to be scheduled to suitable resources simultaneously. This may results in long time waiting for multiple resource allocation.

2.2.3 Workflow

A workflow [GF04] can be described by an acyclic graph, where nodes of the graph represent tasks to be performed and edges represent dependencies between tasks. The Directed Acyclic Graph (DAG) can be used to represent a workflow application which is composed of a set of tasks where the input, output, or execution of one or more processes is dependent on one or more other processes. See Figure 2.1(c).

Traditionally, a workflow is represented by a simple script written in Python or Perl or even Matlab. It is often submitted as a "batch" job to a mainframe without user intervention. Condor introduced a meta-scheduler DAGMan [Con] to submit workflows. DAGMan submits the jobs to Condor in an order

represented by a DAG and processes the results. An input file defined prior to submission describes the DAG, and a Condor submit description file for each program in the DAG.

The grid workflow engine provides a negotiation mechanism that allows users to negotiate with services providers/resources owners on a certain service agreement with the requested Quality of Service (QoS). Examples of workflow management engines include Business Process Execution Language (BPEL) [EBC^{+}05] and Vienna Grid Environment (VGE) [BBES05]. The BPEL is currently the standard that supports QoS constrain expression that uses heuristic for scheduling workflow applications.

One limitation with a workflow application is the inflexibility it introduces. The execution must follows the order of workflow therefore all the related resources must be available in a scheduled stage. We propose an alternative flexible *task-based* model where a master application controls when to run the tasks according to their dependencies. Details can be found in Chapter 4.

2.2.4 Dynamic Applications

Most of distributed applications can be seen as static applications, i.e. their resource requirements do not change during their executions. However, many other scientific applications, such as in astrophysics, mineralogy and oceanography, have several distinctive characteristics that differ from traditional applications. They allow significant changes to be made to the structure of the datasets themselves when necessary. Some of the computations may generate additional work during the earlier iterations, while work may be reduced in the later iterations. Consequently they may require resources dynamically during executions to meet their performance requirements. These applications are called dynamic applications, evolving applications, or scalable applications. Computational Fluid Dynamics (CFD) is one of the examples in that respect.

CFD is one of the branches of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. One method to compute a continuous fluid is to discretize the spatial domain into small cells to form a volume mesh or grid, and then apply a suitable algorithm to solve the equations of motion. The approach assumes that the values of physical attributes are the same throughout a volume. If scientists need a higher resolution of the result they need to replace a volume element with a number of smaller ones. Since the physical conditions change very rapidly, high resolution is needed dynamically to represent the flux (amount per time unit per surface unit) of all the physical attributes with sufficient accuracy. Or users can interactively steer the resolution of visualization. Consequently the whole volume grid is changing constantly, which may lead to dynamic resource requirements (Figure 2.3).

The application therefore may have to demand additional resources (processors) to maintain the required performance such as a certain execution time progression. Likewise, the application may want to release redundant allocated resources when low resolution is acceptable. The dynamic resource requirements on the fly is a challenge for current distributed environment, because there is no well-defined interface for applications to communicate with the infrastructure to add/release resources at run-time. Moreover, the unpredictable resource requirement is hardly applicable to the current distributed computing model, which needs the knowledge of application execution behaviour and resource requirements

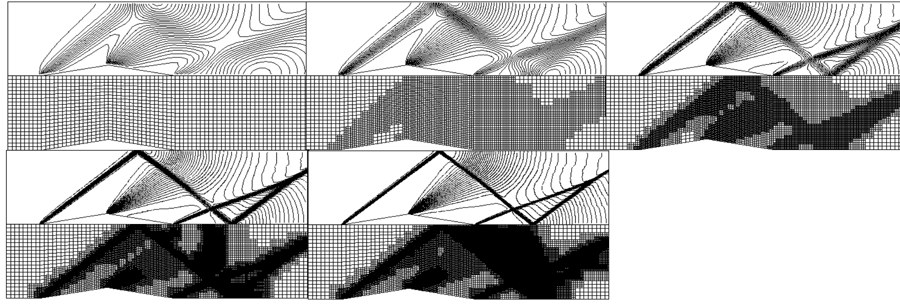


Figure 2.3: A series of slides of the flow across a hump. The grid structure is constantly changing at run-time to adjust the reasonable resolution. The resource requirements in the last slide are around 120 times those in the first slide. [NLS07]

prior to the execution.

2.3 Distributed Application Execution

To execute an application in a current common distributed environment i.e. cluster computing and grid computing, the most common way is to submit the job via grid infrastructure such as a DRM or meta-scheduler, which will schedule eligible resources, deploy the application components (processes and data) and start the execution. To submit a job, users need to write a job description (e.g. JSDL [AA05], classAd [Con] and SGE script [Sge]) and submit it via a grid portal (e.g. a command line interface or a graphical user interface) to request an execution. Once a job is submitted, the resource management is fully controlled by the DRM systems in a central system-level basis and users cannot interfere the resource allocation, job deployment and job startup. Job submission details in the referred Condor and SGE systems can be found in Appendix A.

For most systems, jobs are submitted to a batch queue where they will sit for some unknown length of time, according to the availability of resource and scheduling policies in DRMs. Specially, for the parallel jobs that requires multiple resources, the executions normally have to wait for a longer time since the execution cannot start until all the required resources are allocated. This limitation is one of the reasons that we introduce the flexible execution where no rigid resource requirements are needed to perform the execution.

Alternatively, users can request a reservation to run a job. Advance reservation (AR) [Fos99] is a process of requesting resources for use at a specific time in the future. Globus Architecture for Reservation and Allocation (GARA) [Fos99] was the initial work on AR that defines a basic architecture. AR's objective is to guarantee the availability of resources to users and applications at specific times in the future. It is specially used for co-allocating resources in multiple resource domains, as well as solving the parallel job starving problem: the serial jobs in and out of the execution slots fast enough that there are never enough free slots at any given scheduling interval to satisfy the demands of pending parallel jobs that need multiple slots in order to execute, which results in the larger parallel jobs languishing or "starving" in the pending list for very long periods of time. In a busy environment, resource reservation

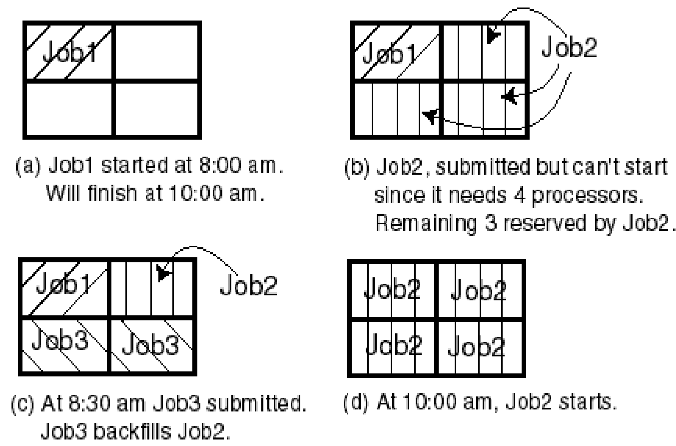


Figure 2.4: Resource reservation with backfilling scheduling.

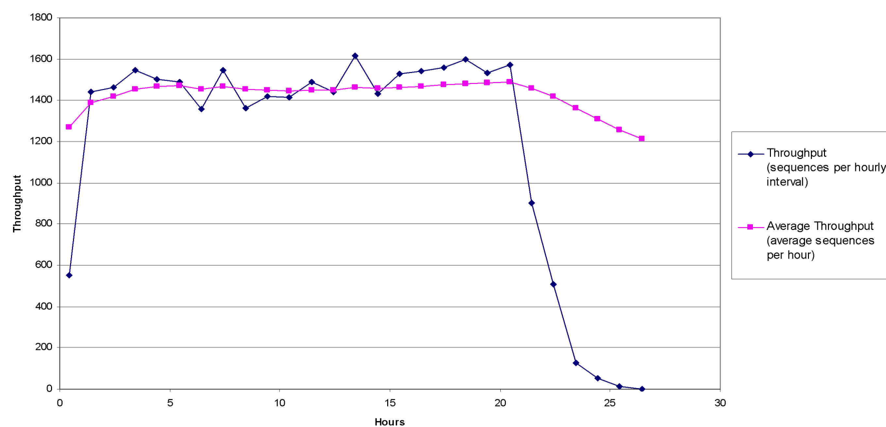


Figure 2.5: The long tail problem. An experiment of annotating the human proteome. This is the throughput per hour measured by number of sequences annotated per hourly interval. [LRK$^+$06]

helps DRMs to schedule big parallel jobs sooner. However, since reserved resources remain idle until the big job starts by default, it greatly degrades the utilization of resources. Backfill scheduling [LZ07] is therefore introduced to allow the reserved job slots to be used by small jobs that can run and finish before the big job starts (Figure 2.4). In order to be effective detailed knowledge with respect to the predicted execution time of each job is required.

The static with advanced reservation type execution has a few rigid conditions: (1) it assumes prior knowledge such resource requirements or execution time prior to executions - users must provide such knowledge to request or reserve a resource allocation; (2) it assumes that reserved resources would not be subject to failure and are to start on time as specified by the user; and (3) for parallel applications, it waits for all the required resources to be available and allocates them at one time. These conditions make it difficult to adapt to the current distributed computing where dynamics is a prominent characteristic.

Firstly, for dynamic applications involving user interaction like computational steering applications, the limitation of condition (1) cannot be ignored. Acquiring prior knowledge of resource requirement

and task execution time is not always possible for computational steering applications as the execution behaviour change at run-time. Hence, it is not possible to acquire knowledge to reserve resources in advance.

Secondly, due to the dynamics nature of resource environment, the availability of reserved resource is hard to be guaranteed. Even if resources can be guaranteed upon initial application start-up, the resource dynamic nature necessitates a constant evaluation of the application needs and resources' availability, and an efficient reallocation strategy. The advance reservations employ re-evaluation of resources with the time constraints when there has been a time conflict or a contract violation, until an optimal agreement is reached. The re-evaluation process is very costly and time consuming, particularly with the restrictions and policies imposed at external domains. In systems with a large number of nodes, the probability of failure significantly increases. This can cause considerable performance degradation for resource allocation. Hence, the management system scales badly when the number of sites and nodes increase.

Thirdly, in a dynamic resource environment, more suitable resources could become available after the resource allocation. Users may want to change to use these resources to improve the execution performance, rather than statically stick to the pre-allocated resources. For example, in Figure 2.5 we can observe the throughput decreases rapidly in the long tails phase at the end of the execution. When the number of nodes exceeds the number of jobs that are left to run, the system should use the fastest processors to maintain high throughput. However the system has lack of adaptivity to migrate jobs from pre-allocated processors to idle faster processors.

Fourthly, since resource allocation is statically performed before starting execution, a job that needs multiple resources (especially resource co-allocation in multiple domains) could have to wait a long time to have enough resources available to run. This weak point is specially prominent when supporting interactive applications where users require results to be seen in a very short time. For example, a graphics rendering application that waits for user input to select an image to render. The rendering may require a immediate burst of computation to produce the image on-line.

2.4 Chapter Summary

In this chapter, we have provided an overview of distributed computing. Current distributed computing was developed from the traditional supercomputing community. Nowadays it is still rooted firmly in the 1970th supercomputer culture with its batch management. This is reflected by dominated queuing system-based DRMs, batch-type applications, and the static job submission approach. Many limitations are inherited from the batch computing culture. The limitations make the computing model more suited to support traditional "batch computing" or "high throughput computing" where users are not sensitive about the execution response, rather than effective, high performance computing, especially the interactive applications in dynamic resource environments. The static and rigid job submission approach also lowers the usability of distributed computing since users must know the steps to submit a job for execution.

Chapter 3

Motivation and Requirements

The initial motivation to produce this thesis came from an astrophysical project which was to map the temperature of the lunar surface over a specified simulation period in order to find the existence of ice on or underneath the moon's surface. In order to attempt to achieve the number of calculations within a reasonable real-time frame, the problem has to be broken down into a series of calculations as well as these calculations being distributed across several CPUs. The simulation therefore has to be built as a distributed application that is able to leverage multiple computing resources.

What our expectation was to run and play with this distributed application as a desktop application with the help of existing infrastructure. We wanted to run the application on our laptop and view the visualization on-line with a reasonable speed in order to see the change of temperature over time. We wanted to dynamically change some parameters of the model (e.g. improve image resolution) on-the-fly and the execution would accordingly reform itself to produce the changed output smoothly - we would at most experience a sudden performance drop but soon have the satisfied speed of execution again. We did not want to touch the complex resource management issues.

However we soon found that to run such an application was not easy. We used a common way to deliver the execution - a SGE cluster for allocating computing nodes and MPI for communication. We must specify a fixed number of resources before we started the application on the cluster. It was difficult for us to steer the application, because the steering might lead to different resource requirement which might cause different execution performance. For example, when we increased the resolution, the application immediately ran much slower but we could do nothing than wait. Therefore we chose to change the resolution off-line. Every time we made a change, we recalculated the potential resource usage and resubmitted it to the cluster. In addition to the manual resource configuration, we needed to start the application on a specific machine (it is usually called the master node that has the role of dispatching and monitoring the different jobs) via SSH. Since the application has a GUI that allows us to view visualization on-line, we needed to setup the Linux remote GUI, such as X11 forwarding. This brought additional workload to us.

We felt that current distributed computing infrastructure was not designed intentionally to support interactive and dynamic applications but only for batch type jobs. Although we may still achieve on-line steering and visualization via various plug-in and tricks, it will require lots of work for the application

developers. The gap between our execution expectation and the actual implementation motivated us to reconsider the distributed computing model, which is static and inconvenient however is enough to support academic batch computing. Further investigation shows that the static way of execution causes problems not only for interactive applications but also nowadays more diverse applications and resource environments (refer to Section 2.3). In addition, the application submission process is a big obstacle to attracting public to use distributed computing. That is why we decided to focus on the research of flexible distributed execution and distributed computing transparency provision.

In the rest of this chapter we will introduce existing work related to the flexibility and transparency in distributed computing, and demonstrate the demand of constructing a new model to satisfy our execution expectation.

3.1 Flexible Execution

Our strategy to address all the possible limitations caused by static resource allocation is to introduce a flexible execution model that enables a running application to re-negotiate the computational resources assigned to it on demand at run-time, rather than only request or book resources in advance. Resource provision in the model is based on a fine-grained, self-service basis instead of one-time, system-level basis. Resource can be dynamically added into or removed from the execution according to the demands of the application. If the computing scale expands the application can always request to add more resource during the execution to maintain certain expected performance metrics. If the computing scale shrinks the application can release redundant resources to obtain high resource utilization. New available resources will be allowed to replace the old resources if they are more suitable for the application. When a resource becomes unavailable due to an unexceptional failure, mechanisms such as checkpointing or load balancing should be taken to grantee the reliability of execution.

The flexible execution notion is similar to the concept of elastic computing in Cloud computing. In the distributed computing context, the flexible execution has the two most important requirements:

- **On-line Resource Allocation**: A resource management system is required to support the on-demand resource allocation. I.e., the application should be able to talk to resource management systems to add or return resources during the execution.

- **Dynamic Load Balancing**: When resources are added or removed during the execution, a mechanism is required to dynamically perform load balancing on the re-configured resources to make full use of them. For example, after adding a resource, the mechanism needs to perform either process migration or data load balancing to make use of the added resource.

3.1.1 On-line Resource Allocation

Existing DRMs normally only provide the static execution way (submit → resource allocate → run) but do not support on-line resource allocation. Once an application is started, DRMs treat later added processes as independent jobs but do not deploy them as a part of the application (they cannot communicate with previous added processes). This is to say, a submitted application is difficult to ask DRMs to

add more resources at run time. The absence of infrastructure support is one of the major obstacles for developers to consider flexible executions.

Parallel Virtual Machine (PVM) [GBD+94] is one of the few resource management systems that support resource provision at run time. Different from DRMs, PVM creates user's own computing environment (the virtual machine) by leveraging dedicated resources. It is widely used and accepted by the scientific community particularly to develop high performance, parallel applications. It is based on a message passing parallel programming model and is provided with features to support flexible process management and virtual machine dynamic configuration, i.e. resources can be added or deleted during the execution of application. Since PVM itself is restricted in single domain computing, several extensions such as PMVM [PZ06], Beolin [Spr01] and ePVM [Fra05] were developed to enable PVM to build multi-domain virtual machines.

All the PVM-family software is limited by the dumb resource management - the user has to manually assign appropriate hosts to the virtual machine. They lack a good collaboration with underlying DRMs for effective resource discovery. Although PVM can run with some DRMs such as Condor (Condor-PVM, details can be found in Appendix A), it is mainly used as a message passing library as an alternative to MPI [SO98] to build parallel applications that can be executed by DRMs. The flexibility of the virtual machine is limited by the DRM static job submission and centralized resource allocation. For example, the user still has to provide a resource requirement to indicate how many resources is expected to need; the process spawn is still fully controlled by the DRM based on a system-level management.

Therefore, there is a demand of software that resides between the application and the DRMs to enable dynamic resource allocation in a user-level to make the application execution benefited from the flexible, dynamic, and on-demand nature. We introduce the Application Agent (AA) under the motivation to make the flexible execution possible on the current resource infrastructure.

3.1.2 Dynamic Load Balancing

Another important requirement of the flexible execution is that some mechanism should be able to performance load balancing once the resources are reconfigured. Current DRMs manage loads in the clusters based on rules associated with job types, load classes, etc. It is a system-level centralized management strategy which works over all users and jobs but do not focus on the performance of a particular application. Such examples are Condor and SGE checkpointing mechanisms.

Grid Application Development Software (GrADS) Project [DSB+04] introduces a rescheduler mechanism to guarantee the performance contract [VAMR01] where expected performance is specified when load in the system or application requirement changes. While the application is running, application sensors embedded in the code monitors application progress by generic metrics such as flop rate or application-intrinsic measures such as the number of iterations completed. Meanwhile, resource sensors are located on the machines on which the application is executing, as well as the other machines available to the user for rescheduling. Application sensor data and the performance contract are passed to the contract monitor, which compares achieved application performance against expectations. When performance falls below expectations, the rescheduler contacts the rescheduling actuators to initiate the

actual rescheduling by either process checkpointing/migration or load balancing via process swapping [SSCC04]. The whole monitor-reschedule process is complex. The re-scheduling only depends on the processor load: when GrADs determines to re-schedule, it moves a process from a over-loaded resource to an eligible under-loaded resource.

DLB [PYE+04] is another software providing application-level load balancing for individual parallel applications. It ensures that all loads submitted through the DLB environment are distributed in such a way that the overall load in the system is balanced and application programs get maximum benefit from available resources. Similar to GrADs, DLB also focuses on the processor load and balances the computational load by moving the application's data "blocks" from over-loaded processors to others.

David Jackson et al. [JH97] introduce an extension to the PVM software system, which acts like a resource monitor to provide dynamic information needed for user-directed load balancing in a multiuser environment. The information, actually the dynamic processor load, is provided to users through an interface to perform a load balancing. Or, users can use the extended pvm_loadspawn() function which performs a load balanced spawn as an alternative to the round-robin spawning process provided by PVM. It spawns by choosing the machines based on either the percentage of idle time only or the percentage of idle time and the architectural speed.

From the above examples, a Service Level Agreement (SLA) approach [PDD05] and a PVM dedicated approach [MCSML07] we can see that present load balancing mechanisms mainly focus on the processor load but do not or deliberately avoid taking account of other factors such as communication delays or communication topology (although few of them claim to have such ability, no experiment has shown the proof. [MCSML07] has considered some communication factors but it is heavily PVM based and only optimizes message transfer time by ad-hoc techniques such as message aggregation.). As we know, besides the processor load, the communication load unbalanced could be another performance obstacle. The load balancing mechanism must take account of this issue. First, algorithms should consider moving the processes that communicate frequently onto the close resources. Second, algorithms should consider using the processing time with the communication time as the criteria to move processes or data. For example, if a fast resource has a large communication delay with other resources, it should still be assigned for fewer data. Such load balancing involves application communication topology which could be so complex that a third-party sensor finds difficult to detect. For example, Figure 3.1 shows an example of a problem requiring unstructured communication. For such applications, a third-party software is hard to perform appropriate data aggregation to avoid communication overhead only based on sensor data. This is the reason that almost all the load balancing mechanisms only focus on the processor load but ignore addressing the communication patterns.

Therefore, a good application-level load balancing should be performed by the application which has the full knowledge of the computing problem. Our system will focus on the on-line and on-demand resource provision, while we let the application perform load balancing to fully make use of the allocated resources to obtain maximized performance.

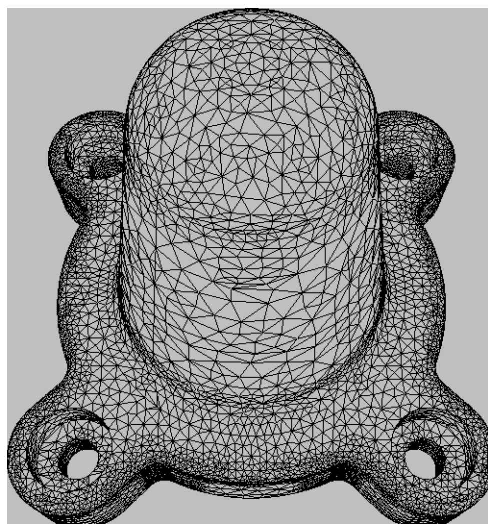


Figure 3.1: Example of a problem requiring unstructured communication. In this finite element mesh generated for an assembly part, each vertex is a grid point. An edge connecting two vertices represents a data dependency that will require communication if the vertices are located in different tasks. [Fos95]

3.2 Providing Resource Management Transparency

In this thesis, transparency is defined as hiding the distributed nature of the application execution from its users so that the application appears and functions as a local desktop application. It can be divided into two aspects. Firstly, *resource access transparency*, which hides the complexity to request and access distributed resources, is usually achieved by certain middleware that replaces the discrete, user-controlled stages of job submission with an end-to-end software-controlled process. Secondly, *resource selection transparency*, is a higher level transparency that hides the complexity to select suitable resources for satisfying performance QoS requirements. It usually involves automatic resource selection polices and associated middleware for supporting such policies. In the following, we will look through existing technologies and investigate their limitations as well as the requirement of introducing better transparency.

3.2.1 Application Frameworks

As we know, the traditional way to run an application requires the user to be familiar with different submission languages and commands. There is a demand of integrating the job submission actions into the application code so that the user can start the application as a desktop application and does not need to worry about the job submission process. Application developers with no assumed background on distributed computing typically wish to devote their time to their own goals and minimize the time spent coding infrastructure functionality. A special type of application framework is therefore developed to insulate application developers from the distributed resource infrastructure.

Grid Application Framework for Java (GAF4J) [Tec04] is a framework that abstracts grid semantics from the application logic and provides a simpler programming model to develop Java grid applications. It abstracts the details of interfacing with a Globus enabled grid infrastructure (GT 2), resulting in a simple programming model that can be helpful for the quicker development of maintainable grid client

applications.

Distributed Resource Management Application API (DRMAA) [TRHD07] is a high-level Open Grid Forum API specification for the submission and control of jobs to one or more DRMs within a Grid architecture. The scope of the API covers all the high level functionality required for Grid applications to submit, control, and monitor jobs to local Grid DRM systems. An application using the standard DRMAA interface can be run on any DRM software that has adopted the DRMAA specification.

Grid Application Toolkit (GAT) [ADG+05] defines a simple, platform-independent API to grid resources and services. The GAT focuses on resource management (job submission and migration), data management (access to files and pipes), event management (application monitoring and control), and information management (application-specific meta data). The GAT is implemented as a runtime library against which applications can be linked.

Similar to the above work, one of the functions of our proposed middleware AA is to replace user-controlled stages of preparing and executing a distributed application with an end-to-end software-controlled process so that users do not need to know how the resources are requested, discovered and allocated.

3.2.2 Automatic Resource Selection

As we know, when users run an application they always wish the execution is performed according to some rate, which is also called Quality of Service (QoS). For example, users want to run an application by 100 iteration/second; or finish the application in 1000 minutes; or finish the application by 100 dollars, or run the application as fast as possible. Requirements can be a single value or a combination. The existence of computational grids allows users to execute their applications on a variety of different resources to satisfy the QoS requirements. An obvious problem that arises is how to select resources to run an application. Many factors go into making this decision: The processing capability of resources that the user has access to, the cost of using different resources, the location of data sets for the experiment, how long the application will execute on different computers, when the application will start executing, and so on.

Since a long time, scientists select resources by themselves. Manual resource selection always relies on the resource information systems (e.g. Globus MDS [CFK+98]) and scientists' experiences to acquire resources. Scientists first need to know the resource's characteristics and calculate what kinds of resources and how many such resources can satisfy their execution requirements. Then they acquire such resources by specifying certain resource requirements (e.g. the number of processor, processor speed, processor architecture, etc.) using job submission languages. According to the specification, resource matchmakers (DRMs or resource broker [Afg04]) select and allocate suitable resources. However, due to the resource environment getting heterogeneous and dynamic, it is very difficult for scientists to know all the resources' execution characteristics. Moreover, because of the complexity of execution behaviour, it is also difficult for them to provide necessary resource specification information to request resources.

A lot of research therefore has been targeting "automatic resource selection/configuration", which aims to help users who have insufficient knowledge about resources to perform the appropriate allocation

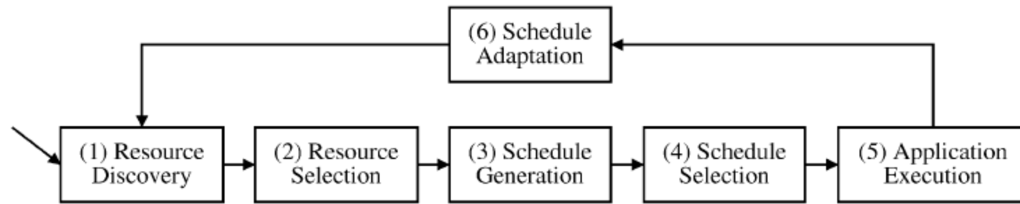


Figure 3.2: Steps in AppLes methodology

of resources for their jobs. Automatic selection is complex as the best choice depends on the application structure as well as the expected availability of computation and communication resources.

Application-Level Scheduling (AppLes) [BWF+96, BWC+03], depending on NWS [WSH99] for resource information, provides automatic resource selection and task scheduling suggestions considering both application intrinsic information and resource characteristics in order to minimize the execution time. Although the implementation details differ for individual examples of AppLeS-enabled applications, all are scheduled adaptively and all share a common architecture. Each application is fitted with a customized scheduling agent that monitors available resource performance and generates, dynamically, a schedule for the application. The agent can account for changes in resource availability by looping back to resource discovery. The steps in AppLes methodology can be depicted by Figure 3.2. It requires users to fill in an application template to provide information about the general functional decomposition of the application, such as the data needed to start the application. AppLeS also developed a few Templates that embody common characteristics from various similar (but not identical) AppLeS-enabled applications. An AppLeS template is a software framework developed so that an application component can be easily "inserted" in modular form into the template to form a new self-scheduling application. Each AppLeS template is developed to host a structurally similar class of applications. To date, two AppLeS templates were developed - APST (AppLeS Parameter Sweep Template) [COBW00], which targets parameter sweep applications, and AMWAT (AppLeS Master-Worker Application Template)[Sha01a], which targets master/worker applications.

Nimrod/G [BAG00, BGA00] introduces an economy-oriented scheduling methodology for parametric computational batch jobs. It collaborates with resource information systems such Globus MDS for resource characteristics (e.g. cost, capability) and selects the most cost-effective resources to meet applications' execution deadline and budget constraints. The scheduling strategy can be changed according to the execution performance response of the application (e.g. it will offer more powerful resources with high cost if it cannot meet the deadline with current ones).

Huang et al. [HCC07] present an empirical model based on a workflow application (Directed Acyclic Graph (DAG) structured) to generate the best number of resources in resource collection in order to minimize the turnaround time. It firstly determines DAG characteristics that are likely to impact the choice of resources and finds the law of the relationship between the best resource collection size and the DAG characteristics by investigating a number of experiments. It then uses the law to determine the best resource collection size given a DAG application. This approach is very ad-hoc and not repeat-

able for other applications. Moreover, users must be familiar with the application to provide the essential DAG characteristics, such as the Communication-to-Computation Ratio, parallelism and DAG regularity that are defined in this paper.

Lindner et al. [LGR07] describe the integration of the performance prediction tool Dimemas [TsGK+03] with the Grid Configuration Manager (GCM) [LGR05] to optimize the resource selection procedure for a given pool of resources. To propose an optimal resource configuration, GCM takes into account the currently available nodes on each machine and the communication pattern of the machine respectively between the machines. Also, the user needs to provide information about the application in form of a trace file which will be used by Dimemas. Based on the resource and application information, GCM creates different "test configurations" for Dimemas, starts the Dimemas simulator, stores and compares the results achieved. The resource configuration with the lowest estimated execution time will be offered to the user. This approach is heavyweight since a lot of experiments have to be tested in advanced. It is specially complex when the resource environment has many types of machines.

From the above example and others such as Dail et al. [DSB+04], Subhlok et al. [SLL99], Kang and Woo [KW05], and Nassif et al. [NNdA07], we can conclude that both manual and automatic resource selections are normally based on a static (off-line) principle, namely users or the middleware select resources prior to commencing the execution. The selection process ends once execution starts. This process works fine in the high performance computing where applications are batch jobs and resources are mainframe or homogeneous computers. However in the modern distributed computing, the application can be dynamic and its execution behaviour could not be predicted in advance. For example, the lunar temperature mapping application we presented at the beginning of this chapter. Moreover, the required resources may be unavailable or other eligible resources may become available during the execution. Therefore it is difficult to plan resource allocation in advance in a grid environment.

Another limitation of current automatic resource selections is that they are heavily based on the information principle, which can be explained by a series of statements in paper [BWF+96]: "Application and system specific information is needed", "Dynamic information is needed for scheduling", "Prediction of application and system performance is needed", and "All resources should be evaluated strictly in terms of the performance to application". Those approaches require users to provide more or less information about the application to obtain optimized performance, such as the communication topology and I/O patterns. Normally, the more information is known by middleware, the better selection is made. However, many users may have no knowledge of either the applications or the resources.

Furthermore, most of existing automatic selection approaches are designed and tested on a dedicated type of applications (ad-hoc), such as CFD, DAG, or bath type applications. They are difficult to be used as generalized approaches.

Therefore, there is a demand of introducing a new automatic resource selection approach that is able to deal with the dynamic distributed execution, and also requires much less collaboration of users. Motivated by the control theory and reinforcement learning, we propose a novel on-line resource selection approach which breaks the traditional rigid selection procedure.

3.3 Chapter Summary

In this chapter we have introduced and discussed what motivated us to address the flexible execution and transparency provision problems, the limitation of current technologies for solving these problems, and the demand of introducing a new distributed computing model. The primary requirements of constructing the new model can be concluded as:

- Flexible Execution
  - On-line resource allocation
    A software that resides between the application and the DRMs to enable dynamic resource allocation in a user-level to make the application execution benefited from the flexible, dynamic, and on-demand nature.
  - Dynamic load balancing
    A good application-level load balancing should be performed by the application which has the full knowledge of the computing problem, when the execution environment is changed.
- Resource Management Transparency
  - Resource access transparency
    A software to replace the user-controlled stages of preparing and executing a distributed application should be replaced with an end-to-end software-controlled and flexible process.
  - Resource selection transparency
    A new automatic resource selection approach that is able to deal with the dynamic distributed execution and also requires much less collaboration of users.

Chapter 4

Model Design

As we introduced in the motivation chapter, we need a new flexible and highly-transparent distributed computing model that is able to adapt to the dynamic and unpredictable behaviours of both the application and the resource environment with the least management from users. According to the requirements summarized in the motivation chapter, both the flexibility and transparency expectations require a change to the way applications are handled in distributed environments.

Instead of using centrally managed DRMs to control the execution, we therefore adopt an alternative approach where each application maintains its own *exclusive execution environment* through direct negotiations with resource infrastructures (Figure 4.1). Each application is supported by its own software, or agent. Rather than submitting the application via a portal and wait for the correct resource combination to become available, the application will start in the normal way and rely on its associated agent to construct the best execution environment possible from currently available resources. If and when the resource environment or application's requirements change during the execution, the agent will adjust the resource configuration to improve performance. Although each application is able to make explicit resource requests to the agent in anticipation of a change in demand, the agent is also able to make autonomous decisions and configure the execution environment without the direct guidance of the application or the user (automatic resource configuration). This process is transparent to the user who will perform the same actions as when running a local application on a personal device. Moreover, since the execution is initiated and controlled locally it is always possible for users to interact with the application and monitor results on the fly via the standard application user interface. Of course, if required, this model will still support the batch processing style. A workflow is now just one possible type of application.

The model is divided into **two layers**, each of which has different responsibilities (Figure 4.2). The lower layer is the user-level software (agent) developed by us, called **Application Agent (AA)**, which takes charge of resource management to construct the exclusive execution environment; while the upper layer, the **application**, is responsible for real problem solving and *load balancing*. Whenever the application asks for a resource, the request is re-mapped to the AA, which then finds a resource from the underlying resource infrastructure and adds it into the application's execution environment. Once resources are added, the AA passes the responsibility to the application to perform load balancing to make

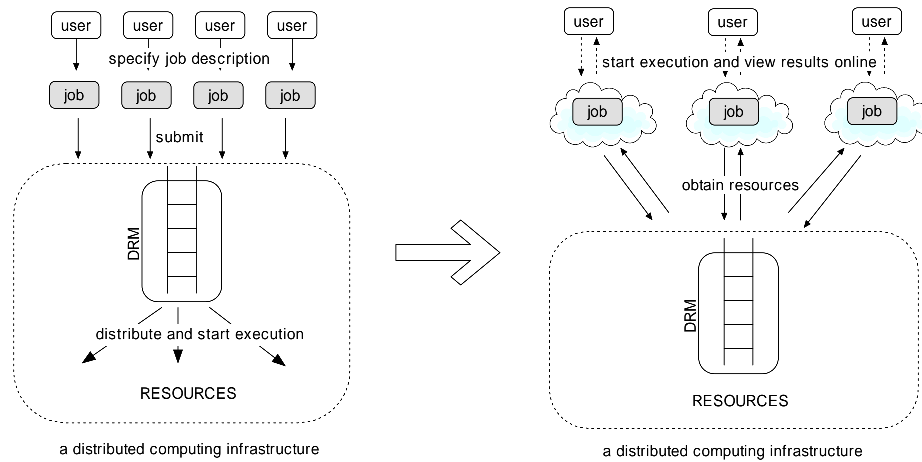


Figure 4.1: Traditional distribution computing model VS Proposed distributed computing model. The execution procedure in the left diagram is an acyclic workflow and each sub-procedure cannot start until the previous one completes. This lead to static execution and low end-user transparency. In the proposed model, the execution environment is configured on a one-to-one basis by a middleware on behalf of the user. The middleware can negotiate resources with the resource infrastructure and adjust the execution environment during the execution.

full use of the new allocated resources. The two layers have explicit division of work. What happens inside of a layer is transparent to another. This design allows application programmers to only focus on high-level problem implementation and to be insulated from the distributed resource management issues which are taken cared of by the AA.

In the following we introduce how we design the AA layer and the application layer respectively; and how we provide high resource management transparency based on the two-layer execution model.

4.1 Application Agent (AA) Layer

The exclusive execution environment to run the application, which we call the **virtual machine**, is a distributed computing platform consisting of a number of physical machines to execute the application as if it was on a single computer. The “virtual machine” is not a hardware/kernel-level virtual machine such as Xen [BDF$^+$03]. It is a lightweight software abstraction of a distributed computing platform consisting of a set of cooperating programs (daemons), which together supply the services (such as communication) required to execute the application.

As we know, a distributed application is composed of several processes communicating with each other by message passing. Each process is responsible for a part of computational workload. In our model, *one process runs on one processor/resource*. During any time of execution, when an application demands more resources to support its execution, it sends requests to the AA to add processes/resources. The AA discovers resources or directly asks for specified resources by contacting DRMs, which will return eligible resources in certain forms. Once a resource is granted by a DRM, the AA deploys a related process on the allocated resource. Subsequently, the AA establishes communication links between the

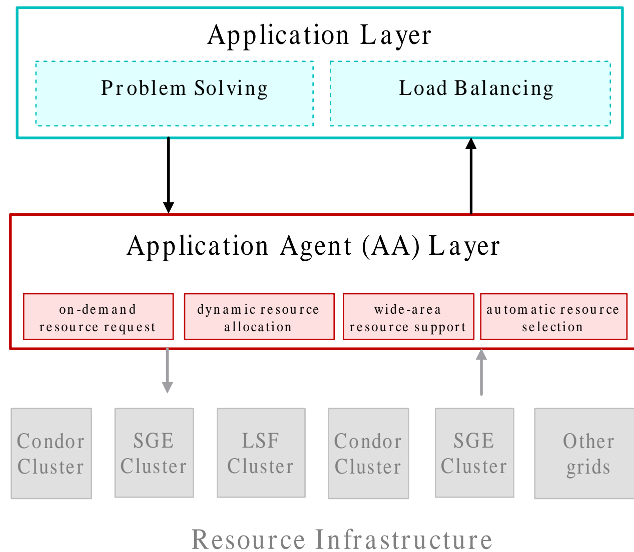


Figure 4.2: An overview of the two-layer model.

new process and old processes of the application. The new added process now becomes a part of the application, which can perform load transferring to the new process to use the new resource. By this way, resources are added into the AA virtual machine. Resource releasing is done by killing a related process and return the resource to underlying resource infrastructure. Before releasing the resource, it has to be vacated first by migrating load to other allocated resources. Both resource addition and releasing are unblocked so the application still runs during the procedures.

4.2 Application Layer

With resource adding or releasing, there is a need of mechanism to perform load balancing to make use of the dynamically allocated resources. As we discussed in Section 3.1.2, a good application-level load balancing should be performed by application programmers who have full knowledge of the problem. Thus we let the application layer to take charge of autonomous load balancing, while the AA concentrates on the resource management issue. To distinguish with traditional applications, we call this kind of applications as **adaptive applications**[1], which is defined as *a distributed/parallel application that is able to autonomously balance the computational load over resources at any time during the execution to furthest make use of allocated resources (reconfigurable)*. When a resource is added, the application should be able to migrate load into the new added process on that resource; while before a resource is released, the application should vacates the resource by migrating load to other resources.

Adaptive applications are different from traditional applications whose partitioning, agglomeration, and process mapping [Fos95] are static. A traditional application can be re-written to be adaptive by using a proper paradigm with dynamic load balancing polices. For example, by using process check-pointing and migration. In this thesis we recommend two easy-implemented paradigms for this purpose: the *distributed object graphs* and the *master-worker*.

[1] Sometimes they are called malleable applications.

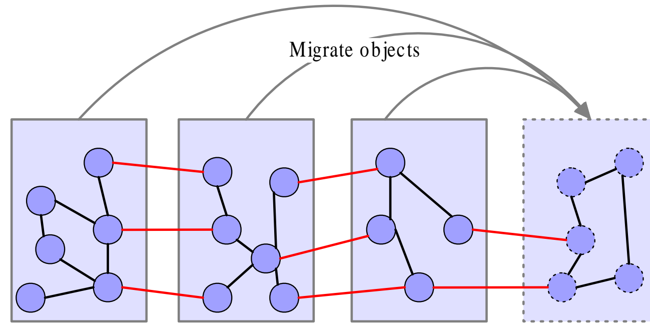


Figure 4.3: Distributed automata graph: "Circle" represents object, black line represents inner communication, red line represents communication between processors. Objects are distributed into "herder" processes. When a new resource is added, objects are migrated to the new added "herder" process.

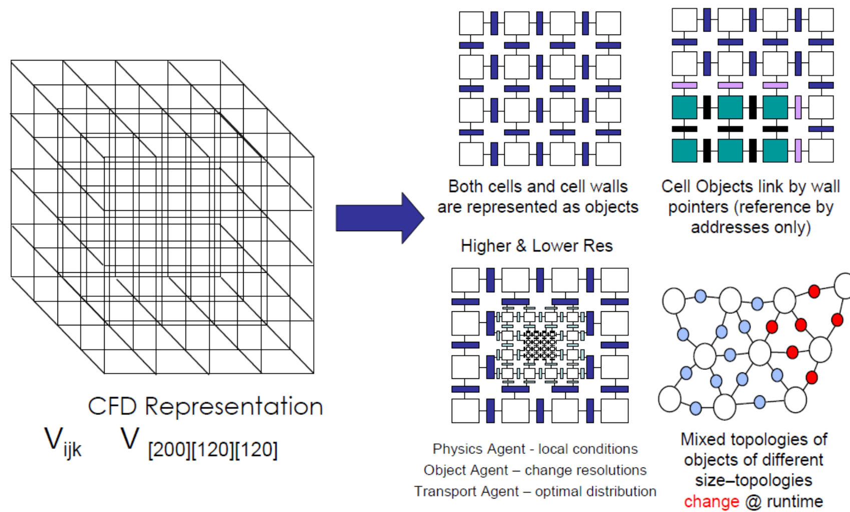


Figure 4.4: The CFD-OG model.

4.2.1 Distributed Object Graphs

For most parallel applications, resource addition/releasing during executions may lead to load migration delivered by process checkpointing, communication preserving and process migration [VD02, LTBL97]. The stop/migration process can involve large data transfer and restarting the application can incur expensive startup costs. The distributed object graphs [NLS07, LNS08, LNS09] (Figure 4.3) which is a flexible extension to the standard cellular automata (CA), providing an easy and efficient way to migrate computation loads among computing resources for such applications. In this paradigm, users create an object interface and connect it to the object through a pipeline that will pass a representation of the interface call to the object. This simple action allows users to distribute the automata objects across any collection of resources. An object will call the interface of the object it wants to communicate with in the normal fashion and the call will take the form of a message being passed to the target object. If the two objects are part of the same process, the transfer will be internal to the process, else it will use some form of message passing service. The automata objects can be moved among resources. After

a migration of an object, its interface will automatically update its reference to the object and make sure future calls to the object can be transferred correctly. By this means, computational load can be easily migrated among resources without addressing kernel level process checkpointing/migration problems. This allows a resource to be added to the application during the execution: the application simply spawns another herder process on the added resource and then moves some automata objects on the empty process. Work [BK99, LK01, FBCL91] introduce similar ideas to support the dynamic application reconfiguration based on the object migration.

One example of the object graphs we have experienced is the CFD-OG [NLS07, LNS08] (Figure 4.4). As we know, in CFD the usual method for introducing high resolution is to replace a volume element with a number of smaller ones. This is a difficult process in traditional CFD because it used a Cartesian of indexed volumes (V_{ijk}). To introduce smaller volumes into such a structure would require further hierarchical indexing leading to complex treatment. An alternative approach is the Object Graph. Objects communicate by exchanging messages with their "neighbours". A neighbour is an adjacent object in the model graph therefore two neighbours do not have to be physically close. A CFD-OG application has two base objects: cells and walls. A cell represents the fluid element and holds the physical values for a volume. It is surrounded by a number of wall objects known to it by their addresses (pointers). A cell simply holds a number of wall pointers. The walls on the other hand only know two objects: the cells on either side of the wall. This is a simple structure that is completely unaware of the physical geometry and topology of the model. The advantage of the object graph is that it uses reference by addressing only. It is therefore possible to change the whole grid topology when smaller volume elements are introduced. This approach also benefits the distributed processing since it is easy to substitute the local addresses (pointers) with global addresses (host, process, pointer) without changing the structure. That being the case, the objects can be distributed on the network computer nodes in any manner. With object graph, a distributed CFD-OG application can be very dynamic and autonomic. The application is composed of a number of processes executing synchronously and normally one process is running on one nodes. In this context, each process holds a number of computational objects (cells and walls) that can migrate from one process to another. As the application progresses, a built-in physical manager monitors the local conditions. If it detects the local resolution is too low, it will ask the built-in object manager to introduce smaller volumes. If on the other hand the built-in physical manager detects superfluous resolutions, it asks to replace several smaller cells with fewer larger ones. This may result in addition or release of resources and the object manager may subsequently attempt to balance processing by moving objects onto light load nodes. A CFD-OG application therefore should be configured with certain optimized algorithm for load balancing.

4.2.2 Load Balancing

Load balancing in this context means *when a resource is added or deleted, load must be readjusted on the new resource configuration to fully exploit the benefit of the provided resources*. The motivation of load balancing is a bit different from the fault-tolerant or performance maximization that current distributed computing uses load balancing for. However, we can use the existing technologies to performance load

balancing once the resource configuration is changed.

Load balancing can be performed globally or locally. A global algorithm needs overall knowledge of computation state to do load re-distribution on processors. Figure 4.3 shows a global load distribution adjustment after a new resource is added: objects are migrated from the old three processes to a new process according to a global plan. In contrast, a local load-balancing algorithms compensate for changes in computational load using only information obtained from a small number of neighboring processors. For example, a processor periodically compares its computational load with that of its neighbours and pushes or pulls computation load if the difference in load exceeds some threshold [Fos95]. Because local algorithms are inexpensive to operate, they are specially useful in situations in which load is constantly changing. However, they are typically less good at balancing load than global algorithms and can be slow to adjust to major changes in load characteristics. For example, if a high load suddenly appears on one processor, multiple local load balancing operations are required before load "diffuses" to other processors.

The application programmer must embed the load balancing algorithm[2] into the application so that the application can autonomously adapt to any resource environments provided. As we discussed before, we do not use the AA to perform load balancing is because we consider only the application layer completely knows how to manage its computation load on the allocated resource to get the maximized performance. For example, with the knowledge of geometric architecture of the dataset, the application can be easier to control and minimise the "red" connectors in Figure 4.3, consequently minimise the communication overhead. There are many well-known load balancing algorithms such as "recursive bisection", "nearest neighbours", "probabilistic methods", "cyclic mapping", "work stealing", etc [Fos95, ZLP95, San99, RSM03, PVL^{+}04, BL94] which can be performed with a comprehensive knowledge of the computing problem. However, programmers are allowed to use existing load balancing frameworks to simplify the development process in some simple situations, such as when load balancing only involves processor load. Such a framework is seen as a part of the application layer.

4.2.3 Master-Worker (Task-based)

Master-worker paradigm [GLY99, Sha01a] (also called task-based) is another programming paradigm to develop adaptive applications. In this paradigm, a master (or multiple masters for fault-tolerant) process dispatches pieces of task to several worker processes, which then compute the tasks, send the results to the master, and request for new piece of tasks (and so on). The number of workers can dynamically adapt to resources in the virtual machine. The computation scale can expand with more workers added when more resources become available or shrink with dismissal of workers when some of the resources are released or fail. The dynamic load balancing hurdle can be circumvented by the automatic adaptive distribution. Figure 4.5 shows the load adaption of the master-worker paradigm when resource configuration is changed (replacing resource A with resource B). Before resource A is released, the master stops sending load to the associated worker and release the resource once the worker completes all the left

[2] We will not investigate novel load balancing algorithms in this thesis. We use existing algorithms to support the flexible execution approach.

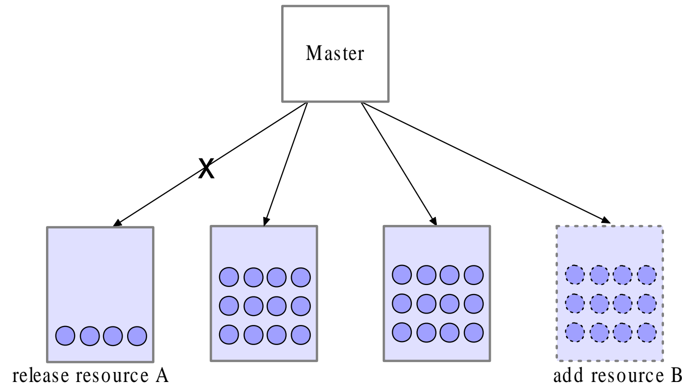


Figure 4.5: The adaption of master-worker paradigm when resource configuration changes - resource A is replaced by resource B.

load. After resource B is added, the master immediately sends load to the associated worker to make use of the resource.

Algorithm 1 Generalized Master-Worker algorithm.

Initialization

Do

  For task = 1 **to** N (one batch)

    Partial_Result = + Worker_Function (task)

  End

  Master_act_on_batch_complete( )

while (end condition not met)

A generalized master-worker paradigm can be used to solve not only a set of independent tasks (embarrassingly parallel jobs) but also the problems that require the execution of several batches of tasks (parallel jobs or workflow). Algorithm 1 shows an algorithmic view of this paradigm. A master process will solve the N tasks of a given batch by looking for worker processes that can run them. The master process may carry out some intermediate computation with the results obtained from each worker as well as some final computation when all the tasks of a given batch are completed. After that a new batch of task is assigned to the master and this process is repeated several times until completion of the problem, that is, K cycles (iterations). For example, for the finite differences computation in Section 2.2.2, the master-worker algorithm can be:

At cycles t,

1. Master sends a calculation task with input $X_{i-1}^{(t)}$, $X_i^{(t)}$ and $X_{i+1}^{(t)}$ to a worker
2. The worker completes the task and sends the result $X_i^{(t+1)}$ back to master.
3. Master updates dataset.

The rewritten finite differences program then becomes dynamic and adaptive. It does not need a fixed number of resource to be allocated simultaneously but can run with any number of resources (workers).

The generalized paradigm is easy to program. All algorithm control is done by one process, the mas-

ter, and having the central control to the schedule of the task. Basically, any problems can be mapped naturally to this paradigm. N-body simulations [GF96], genetic algorithms [CP97], Monte Carlo simulations [BRL99] and materials science simulations [PL96] are just a few examples of natural computations that fit in the generalized master-worker paradigm.

4.2.4 Developing Adaptive Applications

Developing an adaptive application is not much different from developing a traditional distributed application. If programmers choose to use the master-worker paradigm, no extra work will be added. If programmers choose to use the distributed object graph, programmers should design the object graphs according to the above instructions. With applying the object graphs, load balancing policies should also be embedded into the code. In the following chapters we will introduce several adaptive application examples developed based on the two paradigms.

4.3 Transparent Resource Management

We establish the flexible execution model by introducing the AA and adaptive applications. One of advantages of the model is that the execution can adapt to the dynamics of the resource environment. For example, if an application requires 100 processors to run, even though there is only 50 processors, the application does not wait for all the processors but still starts and adds processors on the fly when they become available. The problem is, if there is a QoS requirement, e.g. a deadline, how the application dynamically adds resources to meet the deadline? In this example, the application is supposed to run on 100 processors from beginning to meet the deadline. However, as we start the application with only 50 processors (to save the resource assembling time), during the execution the application must need more than 100 processors to compensate the time lost due to the insufficient resources in the beginning. The on-line resource allocation procedure could be complex with the QoS involved. In the meantime, it must adapt to the resource environment which could change at any time as well as the dynamics of the application behavior.

How to dynamically request and allocate resources to meet certain QoS is a challenge in the flexible distributed computing model. In order to introduce high transparency, the AA must be able to intelligent configure the resources without application's explicit requests. Current automatic resource selections are based on the static (off-line) and information principles (recall Section 3.2.2), therefore we propose a dynamic automatic resource selection approach.

From the overview of the mode (Figure 4.2) we can see, the AA acts like a private agent to autonomously configure the execution environment; and the application is an autonomous system to adapt to any environments that the AA provides. This is similar to the concept of a closed-loop control system in the Control Theory. Motivated by the theory and reinforcement learning, we propose an on-line feedback-based automatic resource configuration to provide full resource management transparency to the application layer. As Figure 4.6 shows, during the execution the application continuously reports its execution feedback (i.e. a certain performance utility value) to the AA, which compares it with the performance reference (a QoS related value) and adjust the input (execution environment) to make the

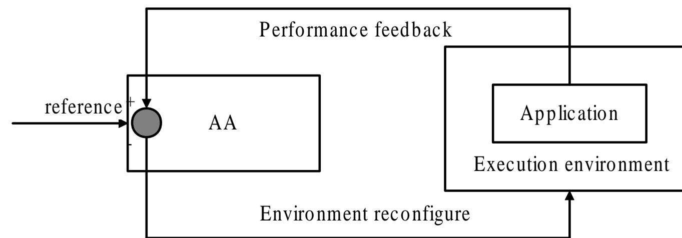


Figure 4.6: An overview of the online feedback-based automatic resource configuration.

performance to approach the reference value. This approach is an autonomous closed-loop and does not need the user's participation.

By this means, with the AA's other services, the AA provides comprehensive resource management including resource selection, resource discovery and resource allocation. While the application only needs to make use of the provided execution environment to get maximized performance. The relationship between the AA, the application, and the user can be interestingly described as a stage manager, a rockstar, and audience, where the manager takes care of decorating the stage (configure the execution environment), and the rockstar performs on this stage (application execution). If the stage is very good the rockstar can perform very well. If the rockstar does not like the stage, the manager must do some replacement (resource reconfiguration). The show's performance does not necessarily depend on the stage but the stage deployment definitely influences the rockstar's performance. Certainly, the audience do not need to do anything but enjoy the show (the user does not need to know anything about underlying management).

4.4 Chapter Summary

In this chapter, we have introduced the proposal of the two-layer flexible and highly transparent execution model. The key questions to implementing this model are: 1. how the AA configures the virtual machine to run applications with collaborating underlying DRMs, including how to contact DRMs to obtain resources, how to deploy processes, how to enable communication, etc; and 2. how the on-line resource configuration is implemented, including how the application reports performance feedbacks, how the AA learns from feedbacks and tunes the resource environment, etc. In the next chapters we give full descriptions of constructing the model.

Chapter 5

Application Agent

The Application Agent (AA) provides the platform and mechanism for the applications to dynamically interact and negotiate resource with the underlying resource environment on a one-to-one basis. In addition, it provides the mechanism for the applications to control computational resources irrespective of the location of the resources. The primary services it provides are:

Flexible Execution:

- **Service 1. On-line resource allocation**: Resources can be added/released on-the-fly according to the demand of application.

Transparent Execution:

- **Service 2. Transparent resource access**: The user or the application can have no knowledge of where and how resources are discovered and allocated. The application can be invoked as easily as starting a desktop program, rather than using job submission in traditional grid computing.
- **Service 3. Run applications anywhere**: An application can be started on any Internet connected machines and its components will be deployed on the remote resources automatically.
- **Service 4. Automatic resource configuration**[1]: The required resources are automatically selected to satisfy performance QoS requirements (e.g. deadline, budget, execution speed, etc), without explicit resource requests from the application.

Except those four, we provide the wide-area computing service to make AA more practical in current distributed computing field:

- **Service 5. Wide-area computing**: An application can be deployed and executed on resources from multiple clusters/domains.

In the following these services will be referred to explain how they are provided. The chapter is further organized as follows: Section 5.1 introduces the overall picture of the AA architecture and how it supports an application execution. Section 5.2 introduces the description of the API and how programmers embed its interface into their application code. Section 5.3 introduces the technical details of the

[1] We introduce this service in Chapter 6 & 7 - Online Automatic Resource Configuration.

architecture of AA. Section 5.4 gives the current implementation of the AA. In Section 5.5 we evaluate the AA's basic performance and demonstrate the ability of AA to support the flexible and transparent execution.

5.1 Overview

The AA is a software framework which is composed of a library of AA interface routines (API) that enable an application to be integrated with the AA, and a set of daemons that reside on the physical machines making up the virtual machine to execute the application. When an AA-enabled application is started, daemons will be started by the AA library. Those daemons are dedicated to this virtual machine, and have the same life cycle as the application. The AA has three basic types of daemon: *LComm*, *PMW-Comm*, and *RD*, which respectively take charge of local communication, process management/wide-area communication, and resource acquisition.

- LComm: **L**ocal **Comm**unication. Each process is associated with a LComm. It is responsible for message passing for the process with other processes that belong to the same domain.
- PMWComm: **P**rocess **M**anagement and **W**ide-area **Comm**unication. Each involved domain has one PMWComm. It is responsible for process startup/stop and cross-domain (cross firewall) message routing.
- RD: **R**esource **D**iscoverer. Each application has one RD. It is responsible for resource discovery and allocation by contacting DRMs. It contains a NameServer component, which is responsible for assigning domain ID to generate process ID.

After an application is invoked, a LComm daemon and a RD daemon will be started on the local machine with the startup of the first process M(normally the master process). The application then requests to add processes/resources via AA API. After receiving a process addition request from the application (actually from the process M), the AA library generates a resource request to the RD, which broadcasts the request to one or multiple DRMs to allocate a resource. RD will make use of the first returned eligible resource and keep the resources returned from other DRMs in a buffer in case that a similar request is made in the near future. After obtaining a resource, RD passes the granted resource information (e.g. host IP address) to the associated PMWComm in the resource domain to deploy a process on the resource. The PMWComm then finally passes the added process information back to the process M. A host thus has been added into the virtual machine for the application. Each process is assigned a Unique Process ID (UPID) which enables them to communicate with each other through LComm and PMWComm daemons.

Figure 5.1 depicts where the three types of daemons are placed in an AA virtual machine and an example of a virtual machine configuration. In this example, the AA leverages eight remote hosts to compose a wide-area virtual machine to run the application. Those hosts are from two environments: four of them are allocated in a SGE cluster. The hosts are discovered and granted through a SGE DRM.

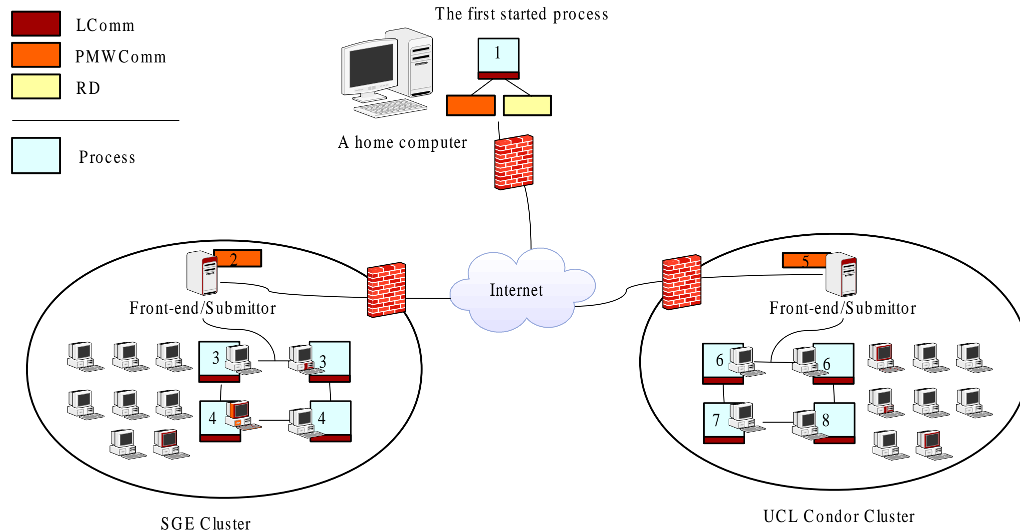


Figure 5.1: The AA daemons create a wide-area virtual machine that contains 9 hosts (including the local machine) from different environments. The numbers in the figure indicate how the virtual machine grows.

Other four hosts are allocated from a Condor cluster. The hosts are discovered and granted through a Condor DRM.

5.2 Application Programming Interface

In order to use the AA's services, application developers must build the application based on the programming interface provided by AA. Application's binary file has to be compiled with AA library. Developers can perform three *basic* operations through the API[2]: add processes, stop processes, and exchange messages among processes. These functions enable an application to request to add or delete resources at any time during the execution without knowing how they are discovered and allocated (**Service 1&2**).

The AA allows application developers to start a named process during the execution by *AddProcess(Name_of_executable)*, which is a non-blocking function. Each function call should be associated with a *Name_of_executable.xml* file, which specifies resource requirements to run this process. The requirements include processor architecture, processor speed, CPU load, free memory, installation of required software and operating system architecture. An example of such a file is provided in Listing 5.1. The XML file will be parsed and transferred to a local DRM related job submission files such as ClassAd and SGE script. If no requirement file is provided, the AA will assume that such process can be started on any resources. As soon as the *AddProcess()* function is invoked, the AA will request a resource from one or multiple DRMs. The application keeps running during the resource discovery process. As in a non-dedicated environment the time it takes for a resource to be allocated is not bound, developers must embed the *GetNotification()* with *PROCESS_ADDED* tag into the execution iteration to keep checking if the process addition has done. If a process is successfully started on a new resource, the *GetNotifica-*

[2]The automatic resource configuration API, such as performance reporting, is introduced in Chapter 6.

tion() will return true. Developers can unpack the notification message by *Upk*()*. The first data of the message is the Unique Process ID (UPID) which is used to identify the added process for communication purpose. The related host information such as clock rate and CPU load are also returned for the purpose of load balancing.

Listing 5.1: A resource requirement XML file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Request>
    <Requirement name="Arch"  type="=" value="Intel"/>
    <Requirement name="Memory"  type=">=" value="64"/>
    <Requirement name="OpSys"  type="=" value="Linux"/>
</Request>
```

Applications can release a process at any time by calling *StopProcess(upid)*. The function call results in the process with the given UPID being killed and the related host being disassociated with the virtual machine. The host will be returned to the underlying resource pool.

Developers should use the AA provided communication interface to pass messages among processes. In order to make developers use the interface easily, the communication interface is deliberately designed similar to PVM, which is a common message passing interface that has been used by many scientific applications. In the interface, a message is a bytestream with an associated pointer. Support functions allow programmers to remove a number of bytes from the buffer stream and return them as one of the basic types (int, float, double) or simply as a null terminated string of chars. A message is built up in the same way. To send a message, programmers must first initialize a send buffer by a call to *NewSendBuffer()*. The message must be "packed" into this buffer using any number and combination of *PK*()* functions. The completed message is sent to another process by calling *Send(upid)*. A message is received by calling either a blocking or non-blocking receive routine and then unpacking each of the packed items from the receive buffer by *Upk*()*.

A full description of the API can be found in Appendix B. An example of using API usage is shown in List 5.2 of Section 5.5.2. The AA also provides a console mechanism for users to configure and monitor the virtual machine. Information can also be found in Appendix B.

5.3 Architecture

In this section, we present the details of the architecture of the AA, including the UPID, the three types of daemons, and how information is recorded and synchronized throughout the AA virtual machine.

5.3.1 Unique Process ID (UPID)

Before introducing the daemons, we firstly introduce the Unique Process ID (UPID). The UPID is the only identifier for processes to communicate with each other. It is generated by the associated PMW-Comm when a process is deployed. It is similar to the tid in PVM and rank id in MPI, but contains more information. It is designed to fulfill the three requirements:

- Inter-VM(virtual machine) communication (see more in Chapter 8)

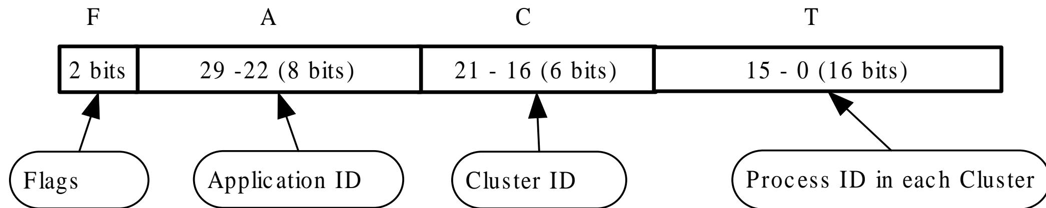


Figure 5.2: Unique Process Identifier

- Wide-area cross domain communication
- Portability

Since the UPID is the only identifier for process communication, it has to contain enough information to satisfy any types of communication. To enable communication among different virtual machines of applications, the UPID has to be globally unique, which means no process shares the same ID even though they belong to different applications. This aim is also the reason for the name - Unique Process Identifier. To satisfy the second requirement, the UPID has to contain domain identifiers that can be mapped to associated PWMComms for cross-domain communication. To satisfy the third, the UPID has to be independent of implementation.

The UPID is made to fit into the largest integer data type (32 bits) available on a wide range of machines. The UPID contains four fields, as shown in Figure 5.2.

- F field: This two bits field is designed as a flag field which indicates the status of a process. Currently it has not been assigned any actual meaning and is left for future use.

- A field: This field contains the application ID which is assigned by *AA-Global-Server* that serves all the running applications in AA world. When an application starts up, the NameServer (in RD daemon) contacts *AA-Global-Server* to get a unique application ID. This ID is synchronized throughout all processes of the application. This field is 8 bits wide, so up to 2^8 applications can exist concurrently.

- C field: This field contains the cluster/domain ID assigned by the NameServer. When a PMW-Comm starts up, it registers itself to the NameServer, which will assign a domain ID to it (a PMW-Comm is associated with one domain). All the process spawned through this PMWComm will have the same domain ID as it. This field is 6 bits wide, so up to 2^6 domains can be concurrently leveraged for each application.

- T field: This field contains the process ID relative to each domain. Each domain is allowed to assign private meaning to the T field, because the LComm daemon in each domain may have different implementation (e.g. based on different message-passing libraries). This field is 16 bits wide, so up to 2^{16} process can be concurrently started in one domain.

Figure 5.3 explains how UPIDs are assigned.

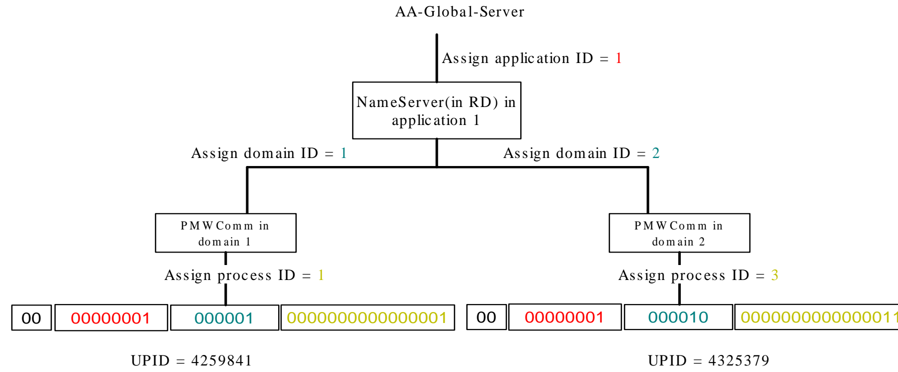


Figure 5.3: Two processes of an application are assigned UPIDs. UPID = 4259841, identifies the 1st process in the domain 1 of the virtual machine of the application 1; UPID = 4325379, identifies the 3rd process in the domain 2 of the virtual machine of the application 1.

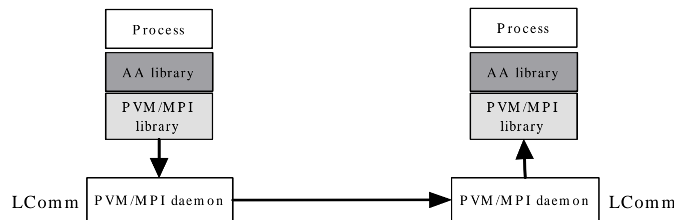


Figure 5.4: Local communication performed by the LComm implemented by PVM or MPI.

5.3.2 LComm

Each running process is associated with a LComm daemon, whose main function is to route messages for the process to others in the local domains. The destination process is identified by its UPID. The LComm daemon borrows the communication function that is provided by existing message passing mechanisms such as PVM or MPI. The library of AA firstly translates a UPID into a PVM/MPI recognizable identifier (map the UPID to the related tid or rank, according to the local process table. See more in Section 5.3.5.), and then the PVM/MPI library is used to request PVM/MPI services to accomplish a communication (Figure 5.4). A LComm can be implemented independently in different clusters/domains according to the setup of cluster. This makes the AA portable for different environments.

The LComm daemon for the first process is started with the invocation of the process when users start the application. Other LComm daemons are started by PMWComm daemons when deploying related processes.

5.3.3 PMWComm

Nowadays many computational clusters are configured with only one IP visible frontend that hides all its internal hosts from the external world. In these architectures, it is difficult to exploit the hosts since they are located in a different network domain that is geographically and administrative separated, usually shielded by firewalls. In addition, some of the clusters are configured to use network address translation (NAT) and have private networks that are not accessible from external hosts. In order to leverage any

hosts on the Internet to build wide-area virtual machines to serve applications, we use the PMWComm daemons which are placed on the frontend host of each domain as proxies to route messages and control processes (**Service 5**). Since the local host always belongs to a different domain from the resource domains, we also place a PMWComm on the local host to route messages between the host and remote resources.

If communication is cross-domain (the sender and the receiver have different domain ID), the sender first needs to extract the domain ID from the receiver's UPID and then to contact the receiver associated PMWComm to ask it to route the message to the receiver's LComm, which will finally accomplish the message passing (Figure 5.5).

Another responsibility of PMWComm is to take charge of process management inside the local domain. This includes deploying new processes and killing unnecessary processes. When a process requests to add a new process by *AddProcess(Name_of_executable)*, AA library first converts this request to a resource request with resource requirements provided by the related XML file, and passes it to the RD daemon, which will then look for a resource. Since the found resource may belong to a domain that is different from the requestor, the process deployment would involve one more PMWComms. Once a qualified resource is found, RD sends the resource information including its IP address with the original process addition request to the resource associated PMWComm. The PMWComm then pulls the process's binary to the destination resource and starts the intended process remotely. Once a process has been started successfully, the PMWComm assigns a new UPID including the domain ID to the new process and stores the process information into the local process table. Finally, the PMWComm reports the new process's information to the requestor. This is done by sending the information to the PMWComm associated with the requestor, which in turn will pass the information back to the requestor when the *GetNotification()* is activated (Figure 5.6).

Upon receiving a process killing request, the AA library delegates the request to the PMWComm associated with the destination process. The PMWComm will finally accomplish the process release. The related host will be returned to the cluster pool.

Since resources in a virtual machine are dynamically discovered, the AA cannot know in advance which clusters it will use during the execution. PMWComms are therefore spawned on-demand by the RD daemon onto the frontend host of domains when it decides to use the hosts from those domains. Each PMWComm is assigned a domain ID starting from 0. When a new PMWComm is spawned, its information will be added into the domain table in the NameServer component(Section 5.3.5).

5.3.3.1 Multiple PMWComms and Domain Tree

Since some domains contain a number of subdomains, each of which has separate firewalls, multiple PMWComms need to be placed to connect to the computing hosts. For example, the CS department of University College London has a firewall and only configures a publicity IP visible frontend named amy.cs.ucl.ac.uk for external machines to access. The HPC cluster in the department is configured by network address translation (NAT) and has a frontend morecambe.cs.ucl.ac.uk. In order to access the cluster hosts from a machine outside the CS department, PMWComms have to be placed on the frontend

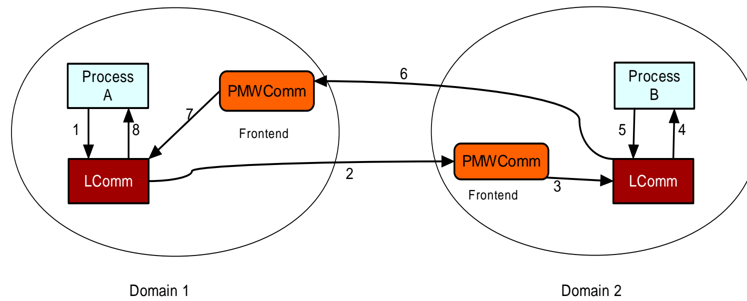


Figure 5.5: The cross-domain communication is delivered by PMWComms. The firewall in each cluster is assumed only to restrict inbound traffic, therefore the sender LComm can directly talk to external PMWComms without contacting its local PMWComm.

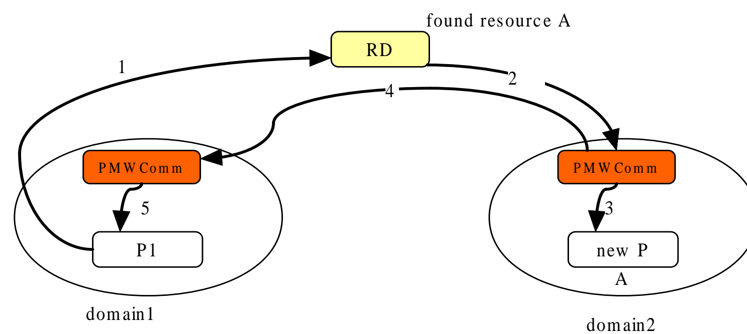


Figure 5.6: A process is added into a different domain for process P1, which is achieved by two PMWComms. 1: request a resource. 2: delegate the process addition request to the resource associated PMWComm. 3: remote process start. 4: pass the new process information to the PMWComm associated with the requestor. 5. the PMWComm notifies P1 the new process has been added.

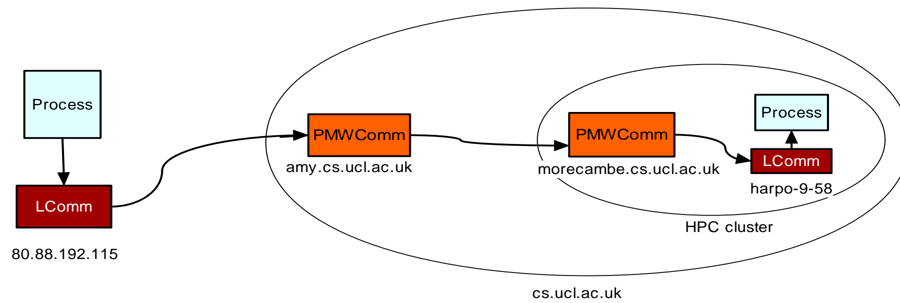


Figure 5.7: Place multiple PMWComms to connect multiple domains

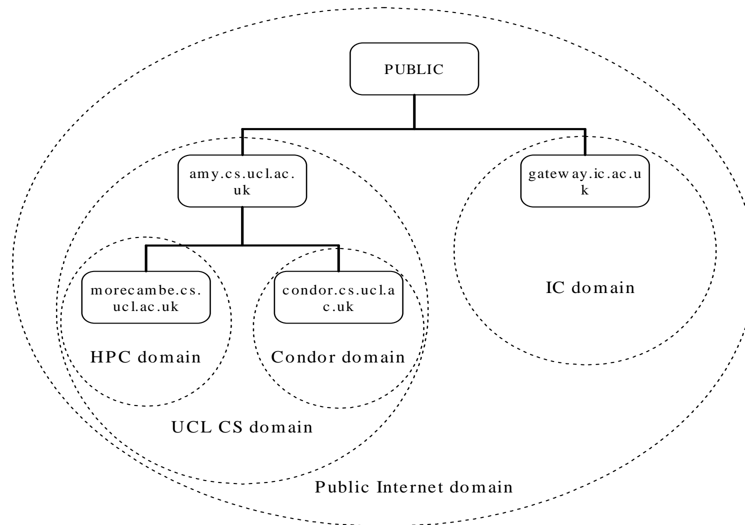


Figure 5.8: A Domain Tree.

of the CS department and also on the frontend of the HPC cluster (Figure 5.7).

One of our objectives is to enable an application to be started from any machines on the Internet and the computing environment to be automatically configured (**Service 3**). This requires that the AA is able to find the path to connect to the resource domain and place PMWComms on the relative frontends, regardless of where the application starts. We therefore introduced the Domain Tree (DT) (Figure 5.8) to store the topology of the domain connections.

Each node in the DT is a frontend address representing a domain. The root node is *PUBLIC*, which represents the public Internet domain that everybody has authority to access. A node may have one or more child nodes, meaning that the related domain may have one or more subdomains. Based on the DT, a fundamental algorithm (algorithm 2) is introduced to find the route to connect hosts of any two domains.

Take the DT in Figure 5.8 as an example, we start an application in the IC domain and the AA discovers resource in the HPC domain. According to the algorithm, the AA finds the path {amy.cs.ucl.ac.uk, morecambe.cs.ucl.ac.uk} to connect to the hosts from the IC domain. It then places two PMWComms on those frontends in the path. The message passing from the IC domain to the HPC domain is then achieved by the $requestor \rightarrow PMWComm_on_amy \rightarrow PWMComm_on_morecambe \rightarrow receiver$ chain.

Algorithm 2 Algorithm to find the connection route in a DT.

Assume: Firewalls only restrict incoming traffic.

Let: The start host belongs to domain A, frontend α, the destination host belongs to domain B, frontend β

Aim: To find the path ρ from the start host to the destination host.

Algorithm:

1. In DT, find the mutual ancestor node γ of α and β
2. calculate δ = β.depth - γ.depth
3. ρ = $\{\beta.parent^{[\delta-1]}$, $\beta.parent^{[\delta-2]}$, ..., β.parent, $\beta\}$

Note message passing from the HPC domain to the IC domain needs involvement of other PMWComms.

Users can customize the DT and potential resource pools via AA's console (Appendix B.4).

5.3.4 Resource Discoverer (RD)

The RD daemon takes charge of resource discovery and allocation for the whole AA virtual machine. It listens to the resource requests and contacts available DRMs to request resources (**Service 2**). A DRM job submission file (e.g. Condor submit description file) is automatically generated according to the user provided resource requirement XML file in this stage. A resource request could be made on-demand when a process addition request is made, or made in advance to hide the resource allocation delay. RD does not submit the actual application processes to DRMs. Instead, it only requires the permission to use certain resource (the actual process startup is performed by PMWComm, as introduced).

Current DRMs (e.g. Condor, SGE) perform resource allocation along with job deployment but do not issue resource permits to third-party software. We therefore propose to submit a special job "probe" to DRMs to simulate the permit acquisition. A probe is a small agent that does not perform computation for the application. It is submitted with the resource requirement of an actual application process by the RD to a DRM to request a resource. Once the probe is started on a host, it will contact RD to issue a permit with the host's information to the AA to use this host. The probe keeps running until the AA kills it, in order to hold that host (in many DRMs, each CPU is only assigned at most one job). In this manner, DRMs only act like resource schedulers but do not perform deployment. Figure 5.9 shows an overview of how the AA works with DRMs.

In some situation, a process spawning would fail even if a host is granted. This is because the granted host becomes unavailable due to sudden change of resource states (e.g. machine shutdown). This failure happens because host information in the AA is outdated by the time the host is returned and the AA attempts to use the host. We reduce the possibility of this failure by making a probe continuously send "HEARTBEAT" messages to indicate that the host is still alive. The load of the host is also synchronized at each time of "HEARTBEAT" sending.

Another issue is that the RD needs to perform remote submission and monitoring on DRMs. Currently only a few DRMs support remote submission via Web Service, such as the Web Services API for Condor [Con] and the OpenDSP project for SGE [ope]. An alternative approach is to place private

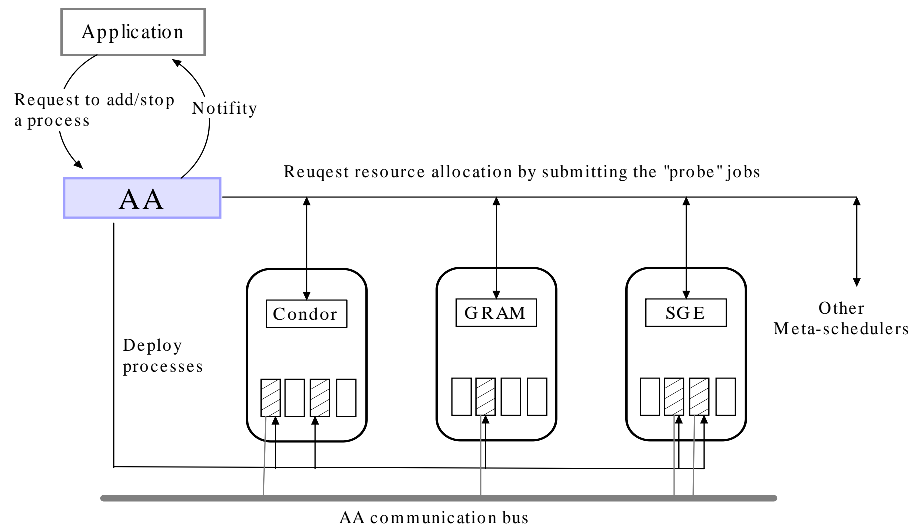


Figure 5.9: How the AA works with DRMs.

servers on submit machines. These servers listen to requests from the RD and perform the submission either via System() call, relevant API provided by DRMs or the standard interface Distributed Resource Management Application API (DRMAA) [TRHD07].

The RD is started on the host where users invoke the application. Sometimes a RD may need PMWComms to communicate with DRMs. In this case the PMWComms will be spawned in advance on the frontend in the connection path, according to the DT algorithm.

5.3.5 Information Record and Synchronization

There are four types of information used in an AA virtual machine: daemon, local process, domain, and host. The latter three are stored in tables. Daemon information includes the contact details of process's parent PMWComm and the RD. The information is passed by its parent PMWComm when spawning the process. If some of the information is dropped somehow, they can be synchronized with the PMWComm daemon again. Both local process table and domain table are synchronized with the process's parent PMWComm daemon on-demand: synchronization is only performed when its local tables do not contain required information. The information synchronization throughout a virtual machine is represented by Figure 5.10.

5.3.5.1 Local Process Table

A local process table contains the information of processes in the local domain. It lists each process's UPID and its identifier in the message passing mechanism (e.g. PVM tid or MPI Rank). Each LComm daemon has a local process table.

Once a process is added, the associated PMWComm will put the process information into the local process table, which can be downloaded from other LComms in later communication. For example, when process A is sending data to process B inside a domain, it first translates process B's UPID into PVM tid using this table, and uses the PVM communication mechanism to complete the communication.

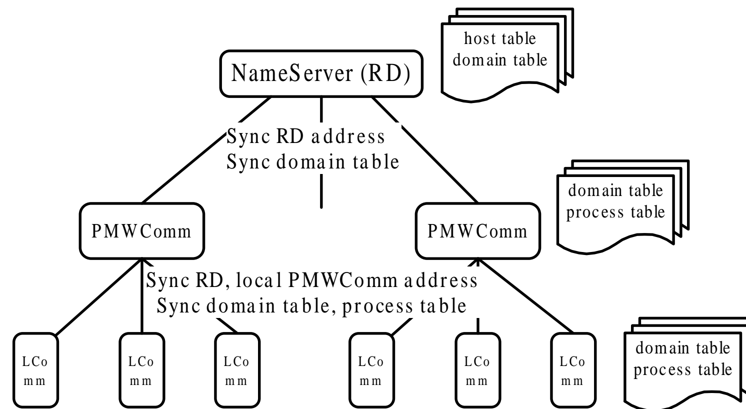


Figure 5.10: Information is synchronized hierarchically.

Table 5.1: A domain table

Domain ID	LComms Implementation	Cluster frontend (PWMComm contact)
1	Unix socket	128.16.6.250
2	PVM	128.21.3.197
3	MPI	128.1.8.12

5.3.5.2 Domain Table

A domain table lists the domain ID and the contact details of related domain. Contact details include the LComm implementation approach of the cluster/domain, and the IP addresses of the cluster frontend where the PMWComm is placed. For example, Table 5.1 lists the information of three clusters, where LComms in cluster 1 is implemented by Unix socket, LComms in cluster 2 is implemented based on PVM, and LComms in cluster 2 is implemented based on MPI. If any process in cluster 2 wants to communicate with processes in cluster 3, it first detects that the destination process is from cluster 3 by extracting its domain ID from its UPID. Then it maps the domain ID to the related PMWComm contact details. It finally contacts the related PMWComm and asks it to forward the message to the destination process via the related protocol (MPI) in cluster 3.

5.3.5.3 Host Table

The host table lists the information of all the processes and related host in the virtual machine, including host name and the related probe job ID. It is used to release the host when killing a process. When a process is killed, the RD will look up the host table to find out the related host and job information. It releases the host by killing the probe job. e.g. qdel 4019372. It also provides users a global view of the virtual machine when users use the AA console. This host table only stays in the NameServer and is not synchronized to other daemons. An example of the host table is shown in Table 5.2.

5.4 Implementation

In this section we introduce how we implemented AA based on the architecture introduced.

In order to allow applications to run on wide-area distributed resources, LComm must adapt to

Table 5.2: A host table

UPID	Domain ID	DRM	Job ID	Host name	Host attributes
4259842	1	Condor	12788.0	calvini.cs.ucl.ac.uk	LINUX, INTEL 32
4259843	1	Condor	12789.0	eco.cs.ucl.ac.uk	LINUX, INTEL 32
4325377	2	SGE	4019372	chico-13-17	LINUX, INTEL 64
4325388	2	SGE	4019373	chico-13-18	LINUX, INTEL 64

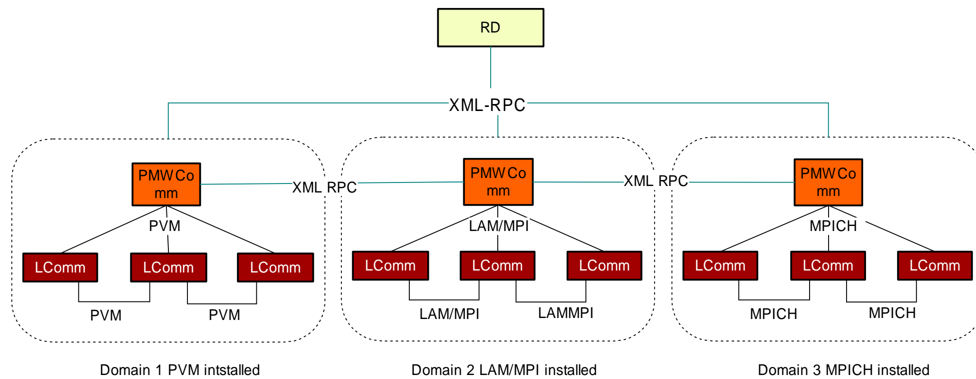


Figure 5.11: The implementation of the AA.

various distributed computing environments configured with different software (Figure 5.11). This is achieved by implementing common interface to various existing software that provide interprocess communication and process management for the LComm daemon. The common communication interface provides flexible access to established communication methods such as Unix socket, PVM and MPI. The common process management interface abstracts a number of function modules to leverage existed methods such as SSH, PVM and LAM/MPI [BDV94]. To port the AA to different environments, the AA programmers only need to re-implement the LComm interface. The current implementation of LComm (Section 5.4.1) is based on PVM.

The PMWComm daemon is implemented as a XML-RPC [Win99, SSL01] server. XML-RPC is a Remote Procedure Calling protocol that uses XML to encode its calls and HTTP as a transport mechanism. We chose XML-RPC instead of other protocols (e.g. CORBA [cor], SOAP [soa]) because it is easy to use and its loose-coupling feature makes it possible to connect a large number of geographically distributed clusters. Also the based HTTP protocol is firewall-friendly and enables the AA to deploy proxies to any parts of the Internet without specific configurations. A PMWComm daemon talks to its external daemons through XML-RPC, while talking to its internal LComms via the LComm communication protocol which is based on the chosen library (Figure 5.11). The remote invocation of PMWComm is performed through SSH.

The RD daemon is also a XML-RPC server. It communicates with other daemons via the XML-RPC protocol. It is spawned via the fork-exec call on the local machine where users invokes an application. The RD contacts cluster DRMs through a special program *RAS (Resource Acquisition Server)* which is pre-placed on the submission machines of clusters. A RAS is another XML-RPC server which

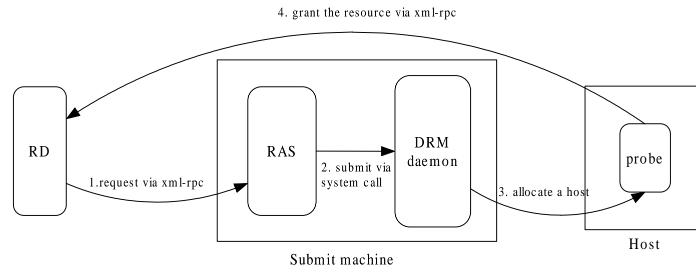


Figure 5.12: The process of obtaining a resource.

serves the host requests from all AA-enabled applications. Currently it uses the system call to submit the probe job for resource allocation. When it receives a request from a RD, it generates a related submission file, and uses the system() call to execute the job submission command that will finally submit the job. For example, in a Condor cluster it generates a probe.submit file and submit it by system("condor_submit probe.submit"). Once a host has been allocated, the probe gains the host information (host name, operating system, CPU architecture, and CPU load) by reading Linux /proc/ directory and reports the information back to the requested AA. Figure 5.12 depicts the process of obtaining a resource from a DRM.

The AA is currently implemented in C++. Communication security is provided by the standard UNIX environment including SSH and RSH. File transfers are performed by SCP and RCP. The AA takes advantage of the GNU LGPL software XmlRpc++ [xml] to implement the XML-RPC mechanism and the AFPL software XMLparser [Ber] to parse the XML file.

5.4.1 PVM Implementation

PVM is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. It has a flexible process management system and a communication system which the AA can make full use of.

After application invocation, the AA library starts a LComm (actually a pvmd) by pvm_start_pvmd(). If a PVM has already been started on that machine, the LComm will attach itself into the system. To add a process in the local cluster, the LComm first adds a granted host into the virtual machine by placing a new LComm on that host via pvm_addhost() and pvm_spawn(LComm), and asks the new LComm to start the process. Since PVM is bounded to join resources that are located in the same network domain, to start a process on an external domain host, the AA places a PMWComm daemon on the frontend of the remote domain. After remote invocation, the PMWComm starts another PVM in the remote cluster, and then adds the granted host to the PVM and starts the process by pvm_spawn(). After successful spawning, processes register themselves to the PMWComm which will then place (UPID - PVM tid) pairs into the local process table.

The communication between LComms is based on PVM's communication mechanism. To send a message to a host in the same cluster, the message is packed into a PVM message buffer (pvm_pk*()) and routed through pvmd by PVM send routine (pvm_send(tid), where tid is interpreted from its UPID).

To send a message to a host belonging to a different cluster, the message needs to be packed into another message buffer which will be sent to the related PMWComm through XML-RPC protocol. The PMWComm then packs the message into a PVM buffer and sends it to the destination process through pvm_send(). Since messages have to be packed before sending, LComms cannot know the addresses of the destination process when performing packing. Therefore at each time LComms pack messages into two buffers: a PVM buffer and a backup buffer. The backup buffer will be ignored if sending is performed through pvmd in the local domain. It will be activated once LComms determine that messages need to be sent through PMWComm. This buffer is cleared every time the NewBuffer() function is called.

When receiving a message from a host in the same cluster, the AA library converts the host's UPID to its PVM tid and uses PVM routines pvm_recv(tid)/pvm_nrecv(tid) to receive the messages arrived from the host. When receiving a message from a host belonging to a different cluster, LComm receives messages from the PMWComm of the local cluster, and uses the message tag in PVM to identify the sender (pvm_recv(PMWComm_tid, Sender_UPID)). In both situations, messages are placed into PVM receiving buffers which will be unpacked through pvm_upk*().

5.5 Evaluation

In the following, we report on three tests: a basic AA communication performance test, a feasibility test to demonstrate that the AA satisfy the requirement of flexible and transparent execution, and a comparison test to demonstrate the performance benefits of flexible execution.

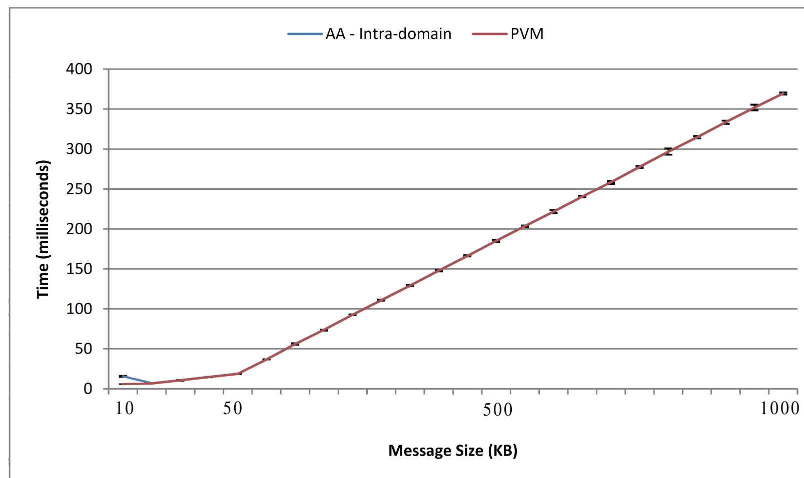
The test resource environment is composed of two clusters (Condor and HPC) in the computer science department of University College London. The department has a firewall and only configures a publicity IP visible frontend named amy.cs.ucl.ac.uk for external machines to access. The Condor cluster is composed of 57 hosts, each equipped with a Intel(R) Celeron(R) CPU 2.80GHz, 502M of RAM, and running Red Hat Enterprise Linux. The cluster is a sub-domain of the department. It has a local firewall and a frontend named condor.cs.ucl.ac.uk. The resource scheduling is managed by Condor. AAs are configured as normal users in this cluster, which means they have to compete for resources with other users. The HPC cluster is composed of 198 hosts, each equipped with a number (1, 4, 8) of Intel(R) Xeon(R) X5355 2.66GHz or Intel(R) Xeon(R) CPU E5462 2.80GHz processors, RAM from 2GB to 30GB, running GNU Linux. The cluster is configured by network address translation (NAT) and has a frontend host (morecambe.cs.ucl.ac.uk) that interfaces the cluster to the outside. The resource scheduling is managed by SGE. AAs use the default queue all.q to submit resource requests to the cluster.

5.5.1 Basic Performance Test

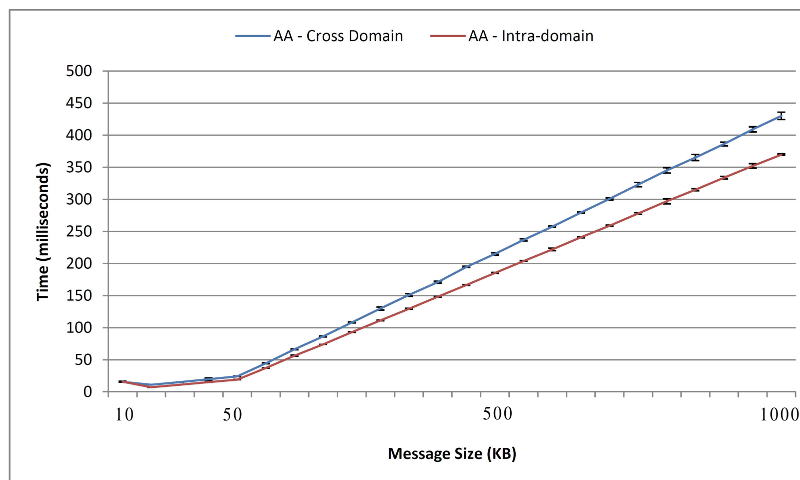
In this test the overhead introduced by AA daemons during the interaction among hosts is measured. The test measures the round-trip time (RTT) for passing different message sizes by the AA and PVM. The test assumes that the difference in latency caused by the network itself (bandwidth, distance, and traffic) can be ignored.

We first test the RTT for intra-domain communication (interaction between two hosts belonging to

the Condor domain/cluster) performed by the AA and PVM. Since the intra-domain communication in an AA is based on PVM, the difference in time between the AA and PVM measures the overhead introduced by the interactions among the AA daemons. As Figure 5.13(a) shows, there are no significant differences between the AA and PVM except when passing the first message(1KB). This is because during the first communication the sender LComm needs to contact the PWMComm in order to download the process table to interpret the UPID of the receiver. Once the receiver information is stored into its local process table, it does not need to contact the PWMComm again, which avoids overheads in the subsequent communication.



(a) Round-trip time in intra-domain communication.



(b) Round-trip time in cross-domain communication.

Figure 5.13: Basic performance test. The result is based on 20 trials. Standard Deviation error bars are also shown on the figures - RRT values have very small variations.

The second test measures the overhead of cross-domain communication in the testbed, i.e. interaction between two hosts belonging to the Condor cluster and the HPC cluster. As Figure 5.13(b) shows, the difference between intra-domain communication and cross-domain communication is within 20%.

Algorithm 3 A local load balancing algorithm

Every 10 seconds:

Send current computing load L_{me}(the number of left objects) and processing capability P_{me} to left and right neighbours.

Receive computing load value L_{neig} and processing capability P_{neig} from neighbour processes.

Calculate proposed load:

$L_{proposed} = \lfloor (L_{neig} + L_{me})/(1.0 + (P_{neig}/P_{me})) \rfloor$;

If $L_{proposed} < L_{me}$

Push $(L_{me} - L_{proposed})$ objects to the neighbour process

Else if $L_{proposed} > L_{me}$

Pull $(L_{me} - L_{proposed})$ objects from the neighbour process

This is mainly due to the overhead of message routing by PMWComms daemons.

5.5.2 Feasibility Test

This experiment demonstrates that an AA-enabled application is easy to run by end-users (transparency), supports dynamic resource allocation (flexible execution), and supports wide-area distributed computing. A more complex experiment that demonstrates the ability of the AA for supporting interactive and dynamic applications is given in Appendix D.

We chose to use a simple application, Mandelbrot computation for the demonstration. The calculation of the Mandelbrot set is an excellent candidate for parallelization. The set of dots that must be calculated can be divided up and allocated to different processors, and no communication is required between the processors to calculate the value for each pixel. The only communication required is the gathering of all of the results into a single processor so that the dots can be plotted.

The Mandelbrot computation is a typical example of traditional static parallel application (e.g. gmandel, http://gmandel.sourceforge.net/). The static partition method is to divide the workspace into n equal rectangles, where n is the number of processors. A contiguous number of rows is computed by each processor. Resources are allocated in advanced of the execution and cannot change during the execution. Here we change the application by introducing a dynamic load balancing algorithm to make the application adapt to the dynamic environment so that the application is suited to flexible execution.

The tested application draws a Mandelbrot set on a 3000 $\times$ 3000 dots canvas with magnification = 1.0. The benchmark iteration is 500,000. The workspace is broken down into 3000 computing objects, and each object takes charge of the computation of a row. The objects are distributed to a number of "herder" processes. Processes compute objects and send results back to a control program, which monitor the execution progress. The canvas is updated from the top down. The application is implemented with a simple dynamic load balancing policy: each process balances the load with its two neighbour processes every 10 seconds, as stated in Algorithm 3. One of the advantages of the local load balancing algorithm is that the adaptation is performed without the participation of a central process, which is normally started on user's local machine that might be far from the computation cluster. This algorithm only

distributes the data to computing nodes from a central node once, reducing the distribution overhead that is introduced in the master-worker algorithm, where a master/central process distributes data to worker processer once they become idle. The maximum number of the “herder” processes are expected to be 30. A fragment of the control program Man_control.cpp is shown in List 5.2.

Listing 5.2: Control program code

```
#include <AA.h>
...
//start an AA.
AA aa = new AA(arge, 0);
...
//Request to add 30 ''herder'' processes.
for(int i=0 ;i <30; i++){
    aa->AddProcess( "Herder" );
}
//Distributes objects to the first added process.
while(true){
    if((aa->GetNotification(PROCESS_ADDED)) > 0 ) {
        //Get process's UPID
        int upid = aa->UpkInteger();
        //Distribute 3000 objects to the first process
        ...
        break;
    }
    ...
}
...
while(true){
    //More processes are being added for the computation
    if((aa->GetNotification(PROCESS_ADDED)) > 0 ) {
        int upid = aa->UpkInteger();
        //assign neighbour processes' UPID to the the added
        //process for local load balancing.
        ...
    }
    //Receive results, draw Mandelbrot set
    if(aa->NReceiveFrom( WILDCARD )>0){
        ...
```

We started the application on a laptop which was connected to the Internet through a BT® home router. The invocation was simply achieved by $./Man_control$, rather than any job submission languages. Once the control process was started, the AA was automatically invoked to deploy the virtual machine to run the application. It started related daemons on the local laptop, and automatically placed PMWComms on the hierarchical domain frontends: amy.cs.ucl.ac.uk, condor.cs.ucl.ac.uk and morecambe.cs.ucl.ac.uk, in order to connect the cluster resources. Figure 5.14 depicts how the AA discovered the path to allocate resources and deploy processes in the two clusters.

According to the demand of the application, the AA contacted the Condor and SGE clusters to obtain resources and deployed processes on the allocated resources. The AA could not collect 30 resources immediately, due to resource competitions from other users. Instead of reserving and waiting for resources, the AA started the execution with available resources immediately and added resource to the execution on the fly. The dynamic resource configuration is shown in Figure 5.15. From Figure 5.16 we can see after the first process was added from the condor cluster, the master sent all the 3000 objects to it and started the execution (left objects was decreasing). Later 4 more resources (1 from Condor cluster and 3 from HPC cluster) became available (probably other users finished their work) and the AA immediately deployed 4 processes on them. The load in the first process was immediately migrated to the 4 processes for load balancing. In the rest of the execution more resources/processes were gradually added for the computations and the load was continually migrated to the under-loaded processes. The whole execution finally leveraged 30 hosts including 12 from the Condor cluster and 18 from the HPC cluster. The execution took about 10 minutes.

The execution speed changed when more processes involved in the computation. During the execution, the control program drew the Mandelbrot set on-the-fly when it received finished dots. The user setting in front of his laptop viewed the Mandelbrot set drawing but had no idea how the underlying resources were managed.

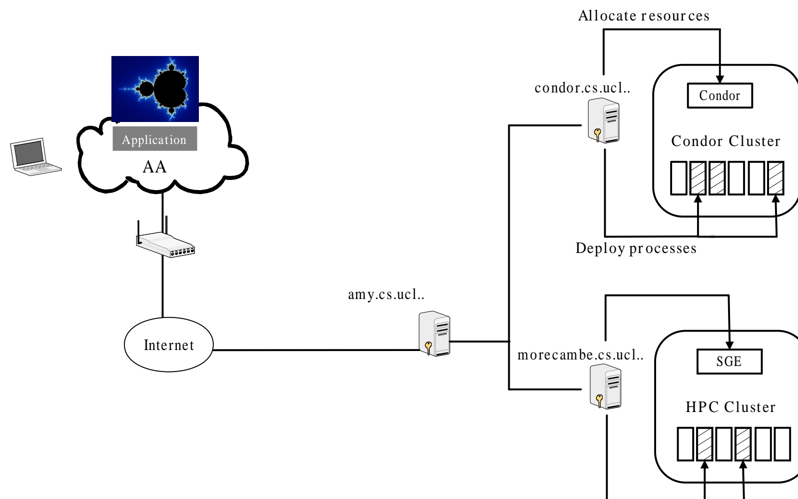


Figure 5.14: The AA deploys an application in the Condor and HPC cluster from a laptop.

5.5.3 Flexible Execution VS Static Execution

Table 5.3: Flexible execution VS static execution. Testbed: the Condor cluster with no resource competition. The result is based on 10 trials.

	Invoke time	Exec. start time	Exec. complete time	Assembled hosts	Time saved
Static	0	40	542	30	–
Flexible	0	4	445	30	20%

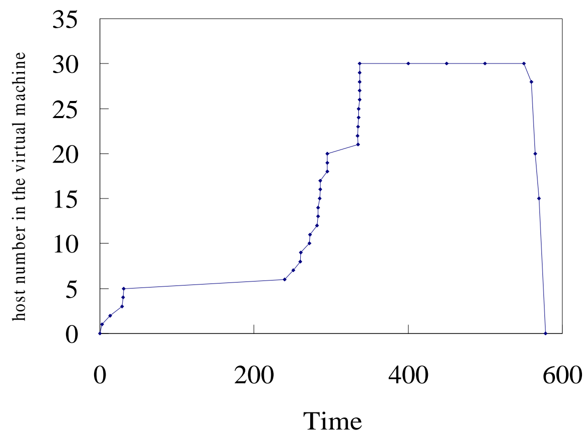


Figure 5.15: The AA virtual machine existed for 578 seconds. The number of hosts in the virtual machine changed dynamically according to the resource availability.

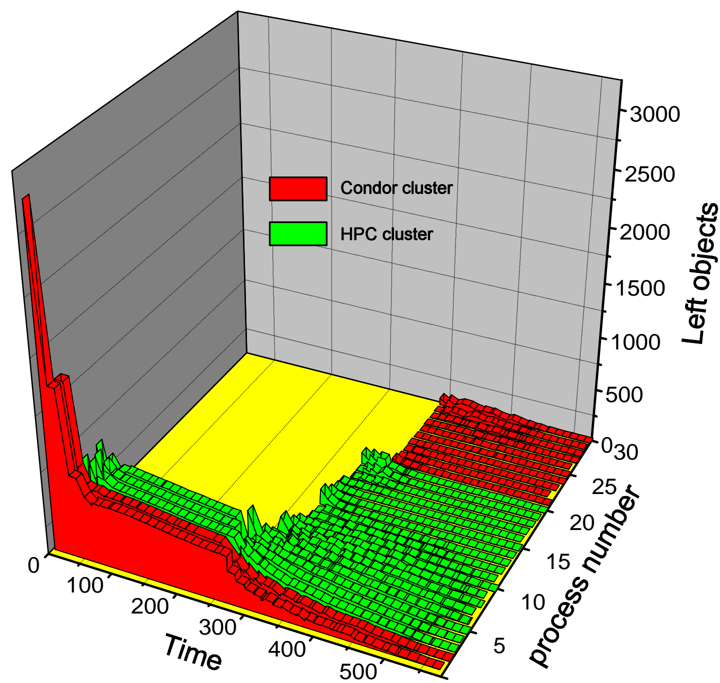


Figure 5.16: Dynamic process addition for the Mandelbrot application. “Red” processes were added in Condor cluster, and “green” processes were added in HPC cluster. Load (left objects) was continuously migrated to under-loaded processes according to the local load balancing policy when more processes were added.

The below experiment shows how the flexible execution is able to enhance execution performance by eliminating the waiting time from multi-resource allocation. In Section 7.1.3.1 we introduce an experiment to demonstrate that the flexible execution with our automatic resource configuration can also improve the resource usage. In Appendix D we introduce an experiment to show how the flexible execution supports an interactive application that has dynamic resource requirements during the execution, which static execution feels difficult to support.

The traditional Mandelbrot application (static partition) requires all the processes to be placed on resources otherwise the execution cannot proceed. In this experiment, we use the same Mandelbrot application to demonstrate the effectiveness of the flexible execution, comparing with the traditional static execution approach. In order to see the difference in execution time and resource usage, we only use the Condor cluster for the experiment to make sure both execution approaches will use the same resources. The usage of the cluster was set to zero in the beginning, i.e. no other jobs. We have run the experiment ten times and obtained the average performance.

For the static execution, even though there was no resource competition, it still took 40 seconds on average to assemble 30 computers before commencing the execution. The assembling time was due to the intrinsic scheduling time of Condor and other communication overhead. It finally took 542 seconds on average to complete the execution.

For the flexible execution, it started to run once the first resource was allocated at second 4 on average. During the execution, more resources were added and normally at second 45, all the 30 resources were added. It finally completed the execution by an average of 445 second. The 20% time saving comparing with the static execution was because of two reasons. 1. Flexible execution eliminates the assembling time. This advantage is especially prominent when the assembling time is long in a competitive resource environment. 2. With the static parallel algorithm, since the Mandelbrot computation amount distributed to each process is not identical, there are problems whereby some processor will finish their allocation of work and remain idle while other processors are still working. While flexible execution fully makes use of every allocated resource by load balancing.

5.6 Chapter Summary

In this chapter, we have introduced the agent AA, including its API, architecture and implementation. The AA provides the platform and mechanism for an application to interact and negotiate resource with the standard grid resource environment. We have demonstrated that the flexible execution supported by the AA can enhance the execution performance by eliminating the long waiting time from multiple resource allocation. In addition, we have demonstrated that with the help of the AA, users can transparently start a distributed computing on any computers without knowing how resource are discovered and where they are allocated.

By using the AA, users are able to interact with a running application anytime (This is demonstrated by Appendix D). There is some existing work related to the application interaction in a grid environment. The glogin [RV04] provides a direct connection to the grid-node where submitted job runs to enable users to interact the application. Similar work includes Virtual Network Computing (VNC) and X11

forwarding. The AA differs with them by providing the interaction through an application framework rather than a user tool. The UK RealityGrid project [Har03] provides a steering library that enable calls which can be embedded into its three components that are simulation, visualization, and a steering client. The collaborative online visualization and steering (COVS) [REF+07] enables users to observe the intermediate processing steps and also allows for computational steering to influence the computation of the simulation during its run-time on a supercomputer or cluster. The AA differs with them in that the AA is based on general opportunistic resource environments supported by DRMs while their work is dedicated to certain grid platform such as Globus and UNICORE [Erw01]. Most importantly, the AA focuses on addressing the dynamic resource requirement problem that the computing steering may cause. However none of the above work has considered this aspect.

The AA is the fundamental software for our research. Based on the flexible execution principle, we propose an online automatic resource configuration approach, which further helps users to run a distributed application without requiring them to provide any resource specifications.

Chapter 6

Automatic Resource Configuration and the Utility Classification Policy

The Application Agent (AA) provides a mechanism that allows the dynamical creation of a virtual machine to execute an application without requiring users know how the resources are integrated. However, users still have to know what kind of and how many resources they need to run the application to satisfy the performance QoS requirements, e.g. a deadline, a speed, or a budget. For example, programmers need to specify the number of processes/resources to add in the code, or provide an interface for users to specify. In order to provide full transparency, i.e. a system where users and programmers have no responsibility for the configuration of the distributed machines, the AA needs to handle all aspects of the virtual machine management. This does not only include resource requesting, process deployment and communication providing, but also automatic resource selection (**Services 4**). The prospect is that the programmers should only provide the binary files and the AA will automatically choose *sufficient* and *appropriate* resources to satisfy given performance requirement.

In this and the following chapter, we will introduce the **Online Feedback-based Automatic Resource Configuration (OAC)** in an attempt to break with the traditional resource selection approach referred to in the Section 3.2.2. In this alternative approach, the AA adds, replaces and deletes resources during the execution of application rather than statically selects resources before commencing execution. The application is treated as a *black-box* able only to provide feedbacks expressing its level of satisfaction. The AA listens to the feedbacks and according to the feedbacks it learns the execution behaviour and intelligently re-configures the execution environment that is able to support desired execution performance.

We will introduce two policies for the AA to learn from the feedback information and tune the execution environment: the Utility Classification policy and the Desired Processing Power Estimation (DPPE) policy. Each policy will be validated by an iterative application and a non-iterative application to demonstrate that both policies are general for most kinds of applications.

The chapter is further organized as follows: Section 6.1 introduces the definition of the Execution Satisfaction Degree which is proposed for reporting the execution feedbacks. Section 6.2 introduces the framework of the OAC. Section 6.3 introduce the Utility Classification policy applied to the approach.

In Section 6.4 and Section 6.5 we introduce two experiments based on a two-dimension heat equation application and an economy-based task farming application used to validate the OAC with the Utility Classification policy. Chapter 7 will continue this chapter by introducing the DPPE policy as well as other OAC related work.

6.1 Execution Satisfaction Degree

Before introducing the approach, we first give the definition of the Execution Satisfaction Degree (ESD), which is proposed by us. The $ESD(t)$ is a measure of the relative *satisfaction* of execution QoS requirements at time t. It is inspired by and similar to the concept of utility in economy [Ing87], which is a measure of the relative satisfaction from, or desirability of, consumption of various goods and services. Therefore it can be regarded as the *happiness* of the users with respect to the resource services. The ESD is defined as a single real number starting from 0.0. A value of 1.0 indicates that the QoS is excellent and thus the application is highly satisfied. The ESD can be a representation of a single type of QoS requirement or can be summarised as the satisfaction of multiple types of execution requirements. For the latter case, $ESD = 1.0$ means that all the requirements are met. Generally, ESD is a function of a set of observed values and related desired values:

$$ESD = f(f_1(observed_value_1, desired_value_1), f_2(observed_value_2, desired_value_2), ..., \\ f_n(observed_value_n, desired_value_n))$$

For example, consider users that wish to run an iterative applications by 100 iteration/second and also on a budget of 1 pound/second. If the actual running speed of the application is 120 iteration/second with a cost of 2 pound/second, the ESD is defined as 0.5 based on the following function:

$$ESD_1 = observed_speed/desired_speed(if ESD_1 > 1.0 : ESD_1 = 1.0) \\ ESD_2 = desired_cost/observed_cost(if ESD_2 > 1.0 : ESD_2 = 1.0) \\ ESD = min(ESD_1, ESD_2)$$

We can also define the $ESD = 0.5 \times ESD_1 + 0.5 \times ESD_2 = 0.75$. The difference between the two functions is that users are greedier in this first case than the second case. It is like asking users "Is half a cup of water half empty or half full?". In the first case users will answer half empty while in the second case they will answer half full.

Users or application developers need to define the ESD function by themselves to represent their execution satisfaction on certain execution stage. There are many ways to define it, as long as the ESD result represents the satisfaction of the execution requirements. Normally, there are three ways to combine ESDs, according to the indifference curve [BL06] in microeconomic theory :

- Cobb-Douglas function[1]: $ESD = ESD_1^{\alpha} \times ESD_2^{1-\alpha}$ $0 \leq \alpha < 1$. α is the user preference for

[1] In economics, the Cobb-Douglas functional form of production functions is widely used to represent the relationship of an

ESD_1. For example, users have two QoS requirement: execution speed ESD_1 and cost ESD_2. The marginal utility of speed, holding cost constant, is decreasing with the increase of speed. It means that if only the speed gets faster, users would never be satisfied. To make the ESD to reach 1.0, we have to increase the satisfaction of both speed and cost, with different increasing rate probably. If the speed is the user's primary goal, the satisfaction degree of speed should have bigger Elasticity of Substitution, i.e. satisfying the speed is more important in order to satisfy users, but not the only requirement. In this case, $\alpha > 0.5$.

- Perfect substitutes function[2]: $ESD = \alpha ESD_1 + \beta ESD_2$. α is the substitutability degree of ESD_1 for ESD_2. β is the substitutability degree of ESD_2 for ESD_1. For example, ESD_1 represents the satisfaction of speed and ESD_2 represents the cost, $\alpha = \beta = 0.5$. We do not need to satisfy them at the same time. We can satisfy one of them to compensate the unsatisfactory of another. That is to say, the user can be satisfied if the speed is very fast (twice the requirement) regardless of the cost factor.

- Perfect complements function[3]: $ESD = min(\alpha ESD_1, \beta ESD_2)$. α is the complementarity degree of ESD_1 for ESD_2. β is the complementarity degree of ESD_2 for ESD_1. By using this function, we must satisfy both measures at the same time. Increasing the value of only one of them will not increase the whole satisfaction degree. Moreover, by increasing Δe for ESD_1, we must increase $\frac{\alpha}{\beta}\Delta e$ for ESD_2. However if ESD_2 is much larger than ESD_1, e.g. $ESD_2 > \alpha(1 + \Delta e)ESD_1$, nothing needs to be done for ESD_2.

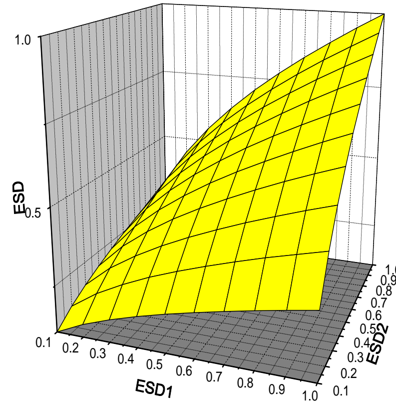
For each way, users' preferences of requirements are quite different. Figure 6.1 shows the ESD plots of the three functions. We can see that, even under the same execution behaviour (the same ESD_1 and ESD_2) the user's total satisfaction degrees are different in the three types of function.

The advantage of ESD is that it combines all the requirements to a single value that can represent the execution satisfaction of users on any execution stages. A series of ESD values also reflects the ESD function, i.e. the trend of user's demands (which requirements should be satisfied first if users have multiple QoS requirements) in certain execution stage. This ESD signal can be easily passed to a third party to investigate the execution behaviour. This way is different from traditional performance measurements which use application instrumentation to capture performance data such as communication frequency or instruction/second [RVR+98, MCSML07, KG96, LDM+01]. We use the human utility value to measure the performance rather than using specific performance metrics. By this way it is possible to include more general performance information such as cost which traditional measurements find it difficult to involve. A similar utility-based performance measurement has been applied to network

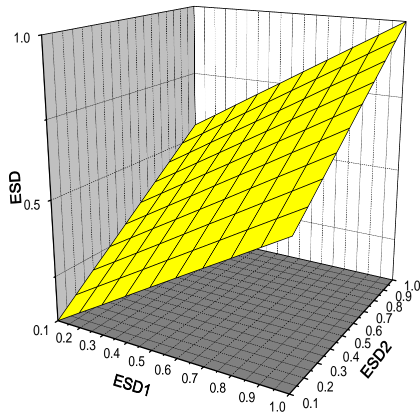
output to inputs. For capital K, labor input L, and constants a, α, and β, the Cobb-Douglas production function is: $Y = bK^{\alpha}L^{\beta}$. If $\alpha + \beta = 1$ this production function has constant returns to scale.

[2] In economics, one kind of good (or service) is said to be a substitute good for another kind in so far as the two kinds of goods can be consumed or used in place of one another in at least some of their possible uses.

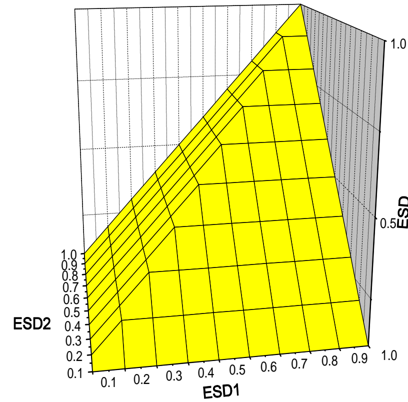
[3] A complementary good or complement good in economics is a good which is consumed with another good. It is two goods that are bought and used together. This means that, if goods A and B were complements, more of good A being bought would result in more of good B also being bought.



(a) The Cobb-Douglas function ($ESD_1^{\alpha} \times ESD_2^{1-\alpha}$) with $\alpha = 0.5$.



(b) The perfect substitutes plot ($\alpha ESD_1 + \beta ESD_2$) with $\alpha = \beta = 0.5$.



(c) The perfect complements plot ($min(\alpha ESD_1, \beta ESD_2)$) with $\alpha = \beta = 0.5$.

Figure 6.1: The plots of ESD functions.

buffer management [FS05].

6.2 Online Feedback-based Automatic Resource Configuration

The Online Feedback-based Automatic Resource Configuration (OAC) approach relies on the flexible execution model that we have introduced in Chapter 4 and the middleware AA we have introduced in Chapter 5. The application needs to be an adaptive application, which is able to autonomously balance the computational load over resources when the execution environment changes. The AA uses this approach to automatically configure the execution environment to run the application.

Figure 6.2 shows how the AA collaborates with underlying resource management infrastructure and interacts with the application to automatically configure the virtual machine, or the **Resource Configuration (RC)** (a set of hosts where an application runs). The application and the AA have different level functions. The application is responsible for making full use of the allocated resources (RC) by load balancing or migration, as well as reporting ESDs. Meanwhile, the AA takes charge of the intelligent

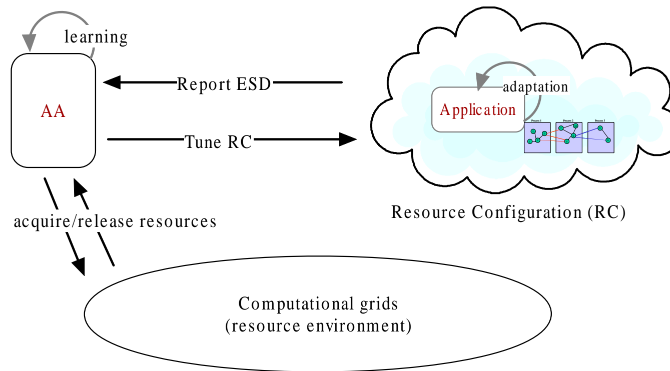


Figure 6.2: How the AA performs the OAC.

resource management to create an optimal execution environment. Initially, the AA randomly selects some hosts to run the application. During the execution, the application periodically (e.g. every n iterations) reports a ESD value to the AA. The AA learns the relationship between the execution behaviour and the resource characteristics based on the reported ESD feedbacks and resource allocation history. The AA keeps updating the information every time a report is made and uses the history gathered to adapt its resource management policy. Once resources are added/replaced/deleted, the application will be notified and will reform itself to adapt to the current RC in order to fully exploit the benefits of provided resources. In most cases, if the RC is better (e.g. including more fast hosts), the performance will improve. Gradually, the AA tunes the RC to a satisfaction level that is able to fulfil user's execution QoS requirements.

The AA uses the embedded *learning and resource tuning policy* to intelligently configure the RC. In the following we will introduce two policies (the Utility Classification and the DPPE) for the AA to gather the necessary information and tune the RC based on the on-line and interactive way.

6.2.1 Implementation

In this section we introduce how we extended the AA architecture and API to support the OAC approach.

6.2.1.1 The AA Side

In order to provide the automatic resource tuning service, we introduce another daemon called the **A**utomatic **R**esource **T**uner (**ART**) to replace the control process of the application and take charge of all the resource management issues (Figure 6.3). If programmers set the second parameter of the function $AA(arge, \mathbf{1})$ as 1, the AA will start the ART daemon on the computer where the master process is - normally the local computer. The ART relies on the reported ESD values to automatically generate resource requests. The requests will be sent to the RD daemon which in turn will contact the underlying DRMs to request resources. Once a process is deployed on a new resource, the host addition notification will be reported first to ART, which will use the host information to calculate further RC tuning. ART will then generate a process addition notification and pass it to the application. When ART decides to kill a certain resource, it will first send a *PROCESS_KILLING* notification to the related process. Upon receiving the *READY_FOR_KILLING* notification from the process, ART will ask its associated PMWComm to kill

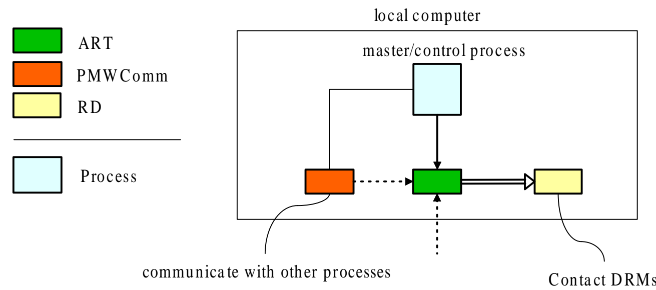


Figure 6.3: This diagram shows how the ART collaborates with other daemons to perform the automatic resource configuration. A solid arrow indicates an ESD reporting. A double-line arrow indicates a resource request to the RD. A dotted arrow indicates other processes or PMWComm daemons notifying a process addition/releasing to the ART.

the process. Note users can also manually add or delete resources by $AddProcess()$ or $KillProcess()$. Manual actions will also be notified to ART to calculate future RC configuration. Different from other daemons, ART is implemented as a special user-level process based on the AA library. The communication between ART and other daemons and processes is managed by the AA communication services (LComm, PMWComm).

6.2.1.2 The Application Side

As we stated above, in the AA model only a single process runs on each resource. The addition/release/replacing of a resource is associated with the addition/release/replacing of a process. After each action, the AA only needs to inform the result of the action to the application. On the other hand, the application checks if any actions are performed by using *GetNotification(int tag)*. Two types of tag *PROCESS_ADDED* and *PROCESS_KILLING/PROCESS_KILLED* are respectively used to check if a new process is added or a process is killed. Adding a host is straightforward: after added a host, the AA sends the *PROCESS_ADDED* notification to the application, which can unpack the notification message by Upk*() and then migrate load to the new host to make use of the host. Releasing a host is a three-phase process:

- 1. Before killing a process, the AA sends a *PROCESS_KILLING* notification to the application to indicate its desire to kill the process.
- 2. Upon receiving the notification, the application vacates the process, and sends a *READY_FOR_KILLING* notification to the AA by invoking ReadyforKilling() to indicate that the process is ready to be killed.
- 3. Upon receiving the *READY_FOR_KILLING* notification, the AA kills the process and releases the related host. The AA sends a *PROCESS_KILLED* notification to the application to indicate that the process has been removed.

The replacement process is decomposed to an addition of new process and a killing of the replaced process. A notification message which includes the target process UPID and basic host information such

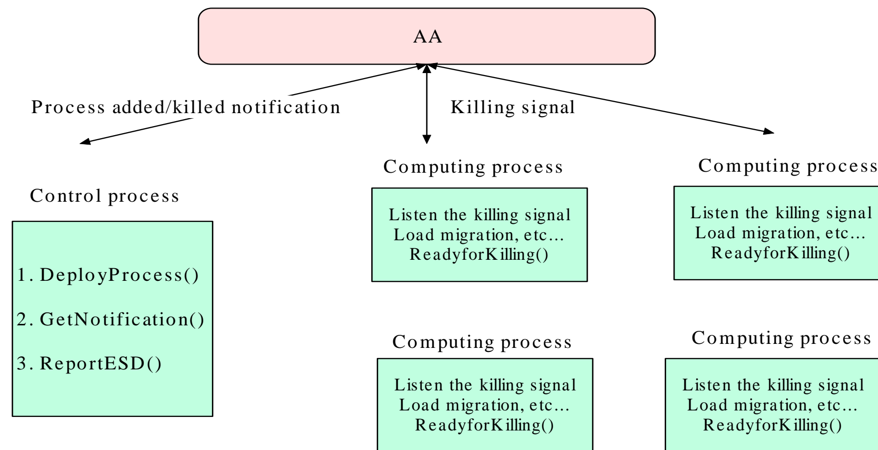


Figure 6.4: The automatic resource configuration: application code structure.

as processing speed and CPU load are returned when *GetNotification()* returns true. Application reports ESDs via API *ReportESD()*.

An AA-enabled application should have a control/master process, which is responsible for listening for the process addition notification and reporting the ESD; and a number computation processes which computes the real problem. The master process could also be one of the computation processes. For example, here we have an adaptive application which is developed based on a local load balancing algorithm. When a new process is generated, the master tells the related process that a new neighbour has been added. The process then migrates its load to the new added neighbour. Gradually, the load of all the processes will be balanced through the periodical balancing. Figure 6.4 shows the code structure. Listing 6.1 and 6.2 show code fragments.

Listing 6.1: Master process code

```
//start the AA and enable the automatic resource configuration service.
AA aa = new AA( arge , 1 ) ;
//Tell the AA the process name to add/delete.
aa->DeployProcess("Worker");
while(true){
    //A process/resource is added.
    if((aa->GetNotification(PROCESS_ADDED)) > 0 ) {
        int upid = aa->UpkInteger();
        float p  = aa->UpkFloat();
        float load = aa->UpkFloat()
        //tell related processes a new neighbour (UPID) is added.
        ...
    }
    //A process/resource is released.
    if((aa->GetNotification(PROCESS_KILLED)) > 0){
    }
    //Monitor ESD according to a ESD function.
    ...
    //report ESD value to the AA.
```

```
        ReportESD(value);
    }
```

Listing 6.2: Computation process code

```
    while(true){
        //computation
        ...
        //Periodically load balancing with its two neighbour processes.
        ...
        //Receiving the process killing signal.
        if((aa->GetNotification(PROCESS_KILLING)) > 0 ) {
            //Migrate load to two neighbour processes.
            ...
            if(process_is_vacated)
                aa->ReadyforKilling();
        }
        ...
    }
```

6.3 The Utility Classification Policy

Resources in the candidate resource environment (e.g. a grid) are usually heterogeneous in that these resources may have different instruction sets, computer architectures, number of processor, physical memory size, CPU speed and so on, and also different operation systems, file systems, and so on. The heterogeneity can result in significant variations in the processing capabilities and cost of executing applications. The resources can be classified by these characteristics. For example, Figure 6.5 shows a testbed composed by 11 types of hosts classified by their characteristics such as location, MIPS ranking, and price.

A Resource Configuration (RC) is composed of a number of the same or different types of hosts from the resource environment. In order to find an optimum RC[4]which can satisfy the execution requirement ($ESD \geq 1.0$), we must know which types of hosts are effective for the execution and how many such hosts are needed to fulfill the requirement. The heterogeneity makes the execution performance difficult to assess. Traditionally, people select hosts according to resource characteristics obtained from the resource information systems such as MDS and Network Weather Service (NWS). For example, people prefer to select hosts that have higher processor speed or MIPS ranking. However, standard performance measures such as wall clock and cumulative CPU time do not separate application code and computing platform performance. Thus, a resource which has a high SPEC (Standard Performance Evaluation Corporation) rank does not necessarily provide high performance to a specific application. The performance indices could depend on many factors including communication delay, memory size, operating system, cost, or even personal preferences. It is difficult to evaluate all these factors and find the “good” hosts before execution.

[4]The optimum RC may not be unique and may change according to the change of execution behaviour or resource environment.

Simulated Resource Characteristics Vendor, Resource Type, Node OS, No of PEs	Equivalent Resource in Worldwide Grid (Hostname, Location)	A PE SPEC/ MIPS Rating	Resource Manager Type	Price (G$/PE time unit)	MIPS per G$
Compaq, AlphaServer, CPU, OSF1, 4	grendel.vpac.org, VPAC, Melb, Australia	515	Time-shared	8	64.37
Sun, Ultra, Solaris, 4	hpc420.hpcc.jp, AIST, Tokyo, Japan	377	Time-shared	4	94.25
Sun, Ultra, Solaris, 4	hpc420-1.hpcc.jp, AIST, Tokyo, Japan	377	Time-shared	3	125.66
Sun, Ultra, Solaris, 2	hpc420-2.hpcc.jp, AIST, Tokyo, Japan	377	Time-shared	3	125.66
Intel, Pentium/VC820, Linux, 2	barbera.cnuce.cnr.it, CNR, Pisa, Italy	380	Time-shared	2	190.0
SGI, Origin 3200, IRIX, 6	onyx1.zib.de, ZIB, Berlin, Germany	410	Time-shared	5	82.0
SGI, Origin 3200, IRIX, 16	Onyx3.zib.de, ZIB, Berlin, Germany	410	Time-shared	5	82.0
SGI, Origin 3200, IRIX, 16	mat.ruk.cuni.cz, Charles U., Prague, Czech Republic	410	Space-shared	4	102.5
Intel, Pentium/VC820, Linux, 2	marge.csm.port.ac.uk, Portsmouth, UK	380	Time-shared	1	380.0
SGI, Origin 3200, IRIX, 4 (accessible)	green.cfs.ac.uk, Manchester, UK	410	Time-shared	6	68.33
Sun, Ultra, Solaris, 8,	pitcairn.mcs.anl.gov, ANL, Chicago, USA	377	Time-shared	3	125.66

Figure 6.5: A WWG testbed composed of 11 types of resources. [BM02]

One solution to this problem is to test the hosts to see if they could bring positive effects or not. By using our on-line resource configuration technology, we are able to classify hosts on the fly according to the actual historical performance effects they brought to the application. In this section, we introduce the Utility Classification policy to automatically and dynamically configure the optimum RC. It aims to accomplish two goals: 1. To classify hosts according to whether they are appropriate or unsuitable for the application. 2. To configure sufficient effective hosts by adding or releasing until the RC satisfies the execution requirement.

6.3.1 The Policy

The theory behind this policy is to use host utilities to classify hosts. A high utility host is regarded as a good host, which means that adding such a host to the execution should improve the performance. The utility of host type X to an execution context a[5], denoted by $AU(X, a)$, can be calculated as the average value of the historical ESD *rewards* that the host type has made to the execution context. To simplify, *AU(X, a)* is written as ***AU(X)*** and the term of utility is abbreviated as **AU (average utility)**. In the rest of the chapter, unless explicitly stated, the utility to the application execution context equals the utility to the application. The reward a host type X brings to an execution context a at time t is denoted by

[5] An execution context includes an application, the application's setup (input, users) and QoS requirements.

$R(X, a, t)$. To simply, it is written as $R(X)$.

$R(X)$ is calculated as:

$$R(X) = \begin{cases} ESD_{after_adding_X} - ESD_{previous} & \text{Adding X} \\ ESD_{previous} - ESD_{after_deleting_X} & \text{Deleting X} \end{cases} \tag{6.1}$$

Note that $R(X)$ is evaluated when a host is being added or deleted but replaced. This is because $R(X)$ only reflects the value of one type but replacing involves two types of hosts (the one replaced and the substitute).

The $AU(X)$ records the average rewards the host has brought to the application all the time. *It tells how good the host is in general*. It is calculated by:

$$AU(X) = \sum R(X)/n \ (n = evaluation\ times) \tag{6.2}$$

For example, if after adding a host type A the ESD increases 0.2 ($R(A)_1 = 0.2$), this means the host type A made a 0.2 contribution to the application at that time. $AU(A) = 0.2$. Next time, after adding another type A the ESD increases 0.1 ($R(A)_2 = 0.1$), then $AU(A) = (0.2 + 0.1)/2 = 0.15$. The third time, adding a type A causes the ESD decrease by 0.1 ($R(A)_3 = -0.1$), then $AU(A) = (0.2 + 0.1 - 0.1)/3 = 0.067$. The AU is updated in this manner. For each ESD report, the AA evaluates the AU of the host class that was added/deleted in the last update; and use the evaluated AU values as the basis for adding/replacing/deleting a class of hosts in order to make the ESD approaching 1.0. The AA keeps updating the AU values iteratively and use them to deal with the resource configuration in the coming turns - basically, the higher AU a class of resource holds, the higher the probability that the AA will choose and keep its members.

A host AU table is recorded as:

Host Type	AU	Selection Probability	Evaluation Times
A	1.001	48%	5
B	-1.19	1%	1
C	1.122	51%	10

The AU can be negative, indicating that resources of this type generally bring negative effects to the application. Another factor in the AU table is the selection probability (SP), which is calculated based on the AU to decide which class of hosts to be added. In the above example, type C has the greatest possibility to be selected. Although type B currently provides negative utility to the application, we still consider that this type may be helpful in the future therefore assign 1% selection probability to it. We will explain this later. The final factor in the table is the Evaluation Times, which tells how many times the related hosts have been evaluated. This information however is not taken into account for current resource selection process but may be used for future upgrading of the policy.

Below, Algorithm 4, 5, and 6 will describe the rules for AU update, SP calculation, and the RC tuning.

Algorithm 4 AU Update

1. In the first execution, before the first time of updating, each host type has an initial AU $\varepsilon = 0.5$ (*default utility*). At the first time of updating, ε changes to zero.

2. Except the first execution, the initial AU at the beginning of execution equals the value from the last execution.

3. Update AU according to the equation 6.1 and 6.2.

Algorithm 5 Selection Probability (SP) Calculation

1. Copy all the AU values to AU'.

2. Shift all the AU' values by adding all numbers with the absolute of the most negative (minimum value) such that the most negative one will become zero and all other number become positive.

3. Shift all the AU' value by adding a very small number if any AU' value = 0.

4. $SP(X)$ = $AU'(X)/\sum AU'$.

For rule 1 of algorithm 4, each resource type is initially assigned a default utility ε. In this way every type will have the same selection probability and will be attractive to be chosen and evaluated at least once. The rule 2 of algorithm 4 makes use of the information obtained from last executions (the AU table, representing the host classification) to guide host selection in order to save tuning time. For rule 3 of algorithm 5, a small number is added because we still want to give a small chance to the current worst type. In algorithm 6, there are three parameters Ψ, Δ, and Ω. They can be customized according to the actual execution condition. The default configuration of $\Omega = -1$ aims to give chances to more "bad hosts".

In this policy we have chosen the average utility AU as the benchmark to determine how good a host is. There is another utility value in this context - the **marginal utility (MU)**. $MU(X) = R(X)$. It reflects the instant utility the host brings to the application at a specific time. Different from AU, *It tells how good the host is in the current situation.* However, as we know, the reward R(X) is not only influenced by the previous action taken but also the current state, i.e. the current allocated hosts, the MU(X) cannot always represent the true utility of a host. For example, given an application has been provided 5 hosts with very large communication latency, the addition of any host may bring down the performance because of the latency bottleneck. If host Y which has not previously been evaluated is added now, even though Y is supposed to be a good host, its MU will still be negative. In this situation, its MU cannot represent its true utility. Moreover, due to the change in execution behaviour and resource environment, the contribution a host brings to the application could dynamically change, with the MU changing accordingly. The MU could oscillate dramatically throughout an execution, and this could make the host classification difficult. Therefore we did not choose the MU as our benchmark to classify hosts.

Although the reward a host type brings to the application can vary, its AU will gradually approach its true value as more evaluations are performed. Consider the above example. If host Y is really a good host, its AU will increase in other situations (e.g. no latency bottlenecks). If it is not, its AU will still be

Algorithm 6 Tuning Algorithm

Ψ = $Tuning\ Amplitude$, the number of hosts to be added/replaced/deleted at each time. Default $\Psi = 1$.

Δ = $Tuning\ Interval$, the number of ESD reports between two action attempts. Default $\Delta = 1$.

Ω = $AU\ Criterion$, the borderline AU to determine a good host - $AU(good_host)$ $>= \Omega$. Default $\Omega = -1$.

Add Action: According to the SP, the AA chooses to add Ψ type X hosts. If $A(X) > \Omega$ and all X are available, the AA adds X to the execution. If not re-do the add action.

Replace Action: Replace Ψ lowest AU host type X in current RC with Ψ available host type Y whose $AU(Y) = \max AU(all_the_other_available_hosts)$, $AU(Y) >= \Omega$, and $AU(Y) > AU(X)$.

Delete Action: Delete Ψ lowest AU hosts.

Procedure:

The AA starts with the "add" action.

While $ESD < 1.0$:

In each Δ reports

1. Update: The AA updates AU, SP.

2. Attempt: The AA tries one of the three actions "add", "replace" or "delete" per time. If a certain action improves ESD, the AA keeps performing such action in the next turn. If not, it tries an alternative action. The AA changes the action according to the order "add$\rightarrow$ replace$\rightarrow$ delete$\rightarrow$ add".

negative. Basically, the more evaluations, the more accurate the AU will be. This is the reason we do not use the exact utility value to select hosts. Motivated by the Certainty Factors in the expert system, we choose to use the selection probability to make the decision. It is also the reason we retain a floor for the worst utility hosts. This protect us from losing these hosts due to a possible misleading evaluation.

6.3.2 Policy Evaluation

In the experiments below we will investigate the effectiveness of the Utility Classification policy. We have chosen to use a simulated framework for this purpose. There are three reasons for doing so. Firstly, we need a large number of heterogeneous hosts to fairly evaluate the ability of host classification. Secondly, we need to be able to deliberately make changes to the resource environment (such as resource availability) in order to prove the adaption of the policy. Thirdly, we need to test different applications, different application requirements, and different resource environments to show that the policy is general to suit any situation. Overall, by using simulation we have more control over the applications and the resource environments.

We will evaluate the policy by two simulated experiments, based on two types of applications - an iterative parallel computation and a non-iterative embarrassingly parallel computation.

6.4 Experiment I - Two Dimension Heat Equation

A large number of scientific and engineering applications exhibit an iterative nature. Examples include partial differential equation solvers, particle simulations, and circuit simulations [CMK$^+$94]. We have chosen to experiment at first with the class of iterative applications for two reasons: this class is important in the scientific and engineering communities, and its iterative behaviour is easy for the AA to learn. In this experiment, we use a two-dimension heat equation application as a typical example of iterative applications. Obviously, the heat equation is not only restricted to heat. Many other physical phenomena share the same form of the heat equation such as mass transfer, momentum transfer, and Schrodinger's equation. Mathematical detail of heat equation can be found in [Ro04].

6.4.1 Simulation Setup

We first simulate 5 clusters of host (A, B, C, D, and E). In each cluster the hosts are homogeneous and classified by their processing capability (P) and location (L). A RC to execute the application can then be represented by $\{\{P_0, L_0\}, \{P_1, L_1\}, ...\{P_i, L_i\}\}$. The latency among hosts inside a cluster is fixed, denoted by La. The delay for sending d data units inside a cluster is assumed to be $d \times La$. For cross-cluster communication, we assume that the delay is only affected by hosts' distance. We use h to denote a host and h_i denote the ith host in the RC. The communication delay for sending d data between host i and j in a cross-domain environment = $d \times |h_i.L - h_j.L|$. In the testbed, the P of the 5 host type are respectively $P(A)$ = 100 UD(unit data)/UT(unit time), $P(B)$ = 200 UD/UT, $P(C)$ = 300 UD/UT, $P(D)$ = 400 UD/UT and $P(E)$ = 500 UD/UT. Their locations are changeable according to specific experiments.

For the application, a 2D (1000×1000) mesh of cells is used to represent the problem data space. At each iteration, the new value of each grid cell is defined to be a function of itself and its four nearest neighbours during the previous iteration (Figure 6.6). Typically, the grid computation is parallelized by partitioning the grid into rectangular regions, and then assigning each region to a different host. This decomposition strategy is favourable because a processor need only exchange the value of the border cells for its region during each iteration. The amount of computational work scales as the area of each region, whereas the amount of delay due to communication is not affected by the scale of the region. In the application, the grid is decomposed into 1000 rectangular regions, called computational elements. The application distributes the elements onto hosts according to their processing capability. A load balancing algorithm 7[6] is used to make the application adapt to the provided RC (Figure 6.7). Once the RC is changed, the master process, which is the first process in RC, autonomously gathers values of all cells in other processes, and re-distributes the cells with their current values according to the algorithm to maximally make use of the allocated hosts.

The simulated application processes are created by Java threads. A thread is spawned on one "host". A sleep() function is used to simulate the computation time, which is calculated with the allocated host's

[6] This algorithm is not an optimum algorithm since it does not take into account the communication delay. As we introduced, load balancing is beyond our research scope. Our goal is to provide the best-effort execution environment to run the application no matter what algorithms it uses.

Algorithm 7 A simple global load balancing algorithm

P_i = the processing capability of host i in RC.

N_i = the number of elements distributed on host i.

for n from 1 to 100

find k in (1,...i) such that $(N_k + 1)/P_k$ is minimum;

$N_k = N_k + 1;$

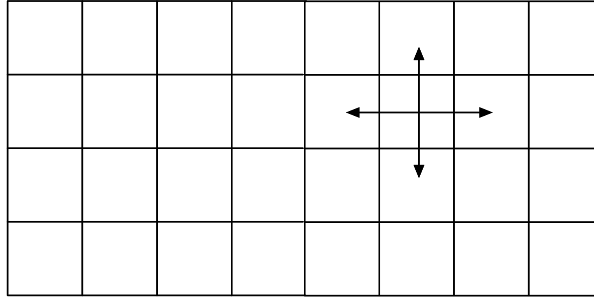


Figure 6.6: 2D heat equation computation.

processing capability. Updating a cell needs to compute 0.01 UD. Updating a rectangular region then needs to compute 10 UD. The communication delay is also simulated by sleep(), which is performed by a separate "send" thread. Passing a cell value needs to send 0.01 UD data. It will be totally sending 10 UD data for 1000 cells in the border region in one time of message passing. For a unit time (UT), the simulation sleeps for 1000 milliseconds[7]. The simulation details can be found in Appendix C.

6.4.2 Simulation Validation

We first validate the simulation by setting $La = 0$. We repeated run experiments allocating 5, 10, 15, 20, 25 hosts of type E ($P = 500$) to execute the application. The theoretical execution speed (iterations/UT) is 0.25, 0.5, 0.75, 1.0, 1.25. Figure 6.8 shows the average experimental speed. We can see that although the actual speed is relatively lower than theoretical values due to simulation and speed monitoring overheads, it still displays a linear speedup because we have neglected the communication delay.

We then set $La = 0.1$. Due to communication latency, the execution speed is supposed to restricted in $(1.0/(0.1 \times 10) = 1.0$ iterations/UT. We now again allocate 10, 20, 30, 40 hosts of type E to execute the application. Figure 6.8 shows the experiment result. As we can see, the speed increases linearly according to the host number when allocating less than 20 hosts. In this situation the computation time is longer than the communication delay (1 UT), so the speed has a linear relationship with the total processing capabilities. When allocating more than 20 hosts, the communication delay starts to dominate the execution time. The execution speed therefore is limited by 1.0 iterations/UT. Due to simulation overheads, the maximum speed is around 0.98.

In the simulation the execution speed oscillates iteration by iteration due to unsynchronized pro-

[7] Higher sleep time can avoid the overhead introduced by simulation, e.g. CPU overload caused by simulation.

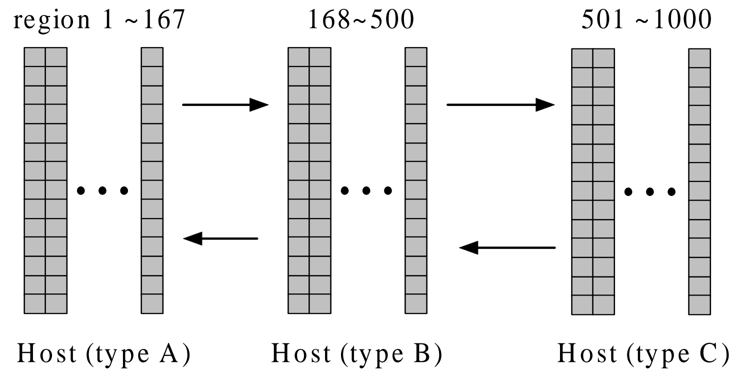


Figure 6.7: The grid cells are distributed into a type A host, a type B host, and a type C host.

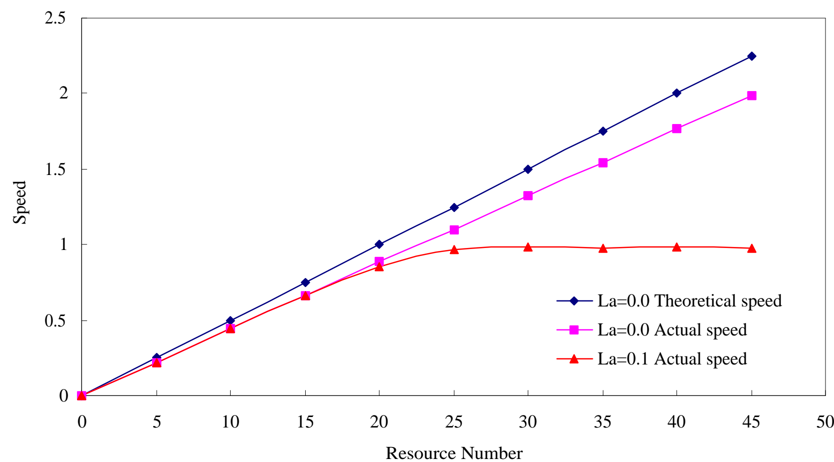


Figure 6.8: Simulation validation.

cessing speed (thread sleep simulation). Sometimes the oscillation gets very strong. The oscillation also happens in the real world due to unstable processing capability and network factors. Because of the unstable execution behaviour, the reported ESD will therefore fluctuate. The fluctuations may lead to AA's overreacting. For example, a temporary ESD drop may cause the AA to add redundant resources. In order to remove the ESD noise signals, the AA uses the average ESD as the representative benchmark to determine the tuning. The AA will not perform any actions until the average filtered ESD is stable, which can be detected through the variability or dispersion of a series of filtered values. The AA renews the average filter for every new RC. Figure 6.9 shows an actual ESD record and its filtered value.

6.4.3 Experimental Setup

In order to demonstrate the adaption of the Utility Classification approach, we build different resource environments based on the testbed in the following 4 sub experiments to run the application. The resource setup for the 4 experiments is shown in Table 6.1.

The user's QoS in all these experiments is a single requirement - execution speed, given in iterations per UT. Such execution rate requirements are common in real world especially when performing on-line

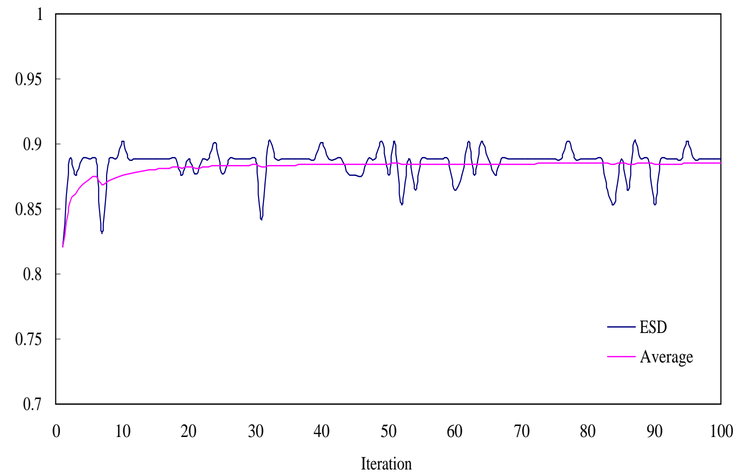
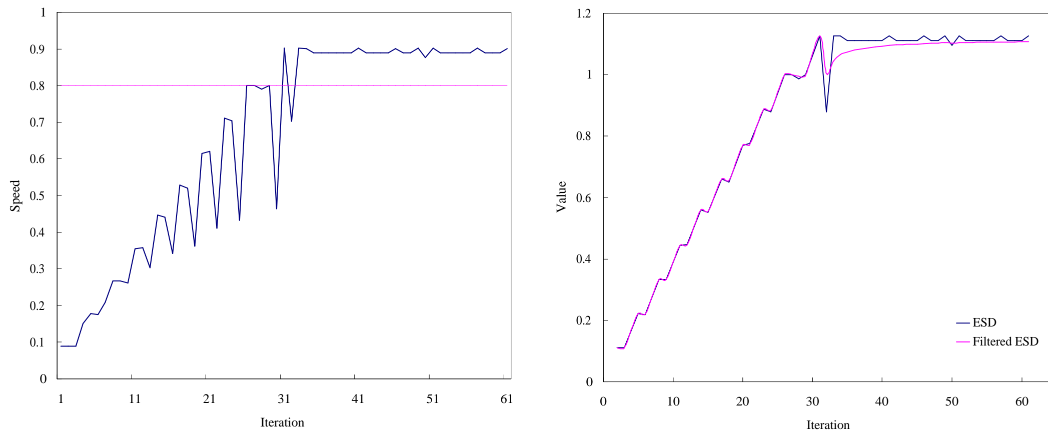


Figure 6.9: The filtered ESD is much smoother.



(a) The execution speed was tuned to reach the QoS requirement.

(b) The ESD was filtered to avoid overreacting.

Figure 6.10: Results of experiment I.1 - homogeneous environment, QoS = 0.8 iterations/UT

parallel rendering. The execution rate is important for smooth rendering. The ESD is defined as:
$ESD = observed_speed/desired_speed.$
The application reports ESD to the AA every iteration, except for the first iteration after each RC's reconfiguration because of the overhead from the time cost to gather cells and re-distribute to processes.

The AA's job is to tune the RC to make the execution satisfy the QoS requirement. Neither the application nor the resource manager provides the AA with information. It only knows that there are 5 host types but has no idea how they will work for the application. The AA configures the RC from scratch, and relies fully on the ESD feedback to perform resource classification and RC tuning.

6.4.4 Experiment I.1 - Homogenous Environment

In this experiment we only use cluster E as a resource environment (Table 6.1). We aim to show the ability of the AA to automatically add the right number of resources (type E) to satisfy the QoS require-

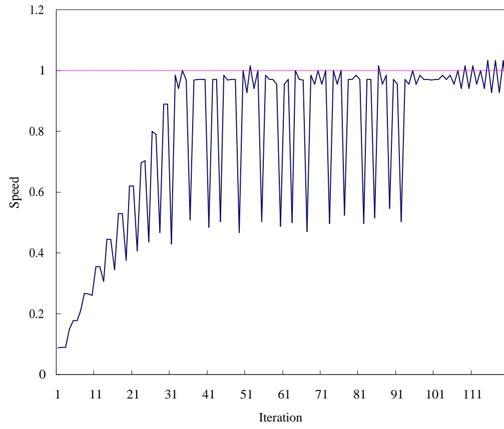
Experiment Number	Cluster	Host Type	*P*	*La*	Location	Availability Probability
Experiment 1	E	E	500	0.1	0.0	100%
Experiment 2	A	A	100	0.1	0.0	100%
	B	B	200	0.1	0.0	100%
	C	C	300	0.1	0.0	100%
	D	D	400	0.1	0.0	100%
	E	E	500	0.1	0.0	100%
Experiment 3	A	A	100	0.1	0.1	100%
	B	B	200	0.1	0.2	100%
	C	C	300	0.1	0.3	100%
	D	D	400	0.1	1.0	100%
	E	E	500	0.1	0.4	100%
Experiment 4	A	A	100	0.1	0.1	100%
	B	B	200	0.1	0.2	80%
	C	C	300	0.1	0.3	60%
	D	D	400	0.1	1.0	40%
	E	E	500	0.1	0.4	20%

Table 6.1: Simulated resource environment for experiment I.1 $\sim$ I.4.

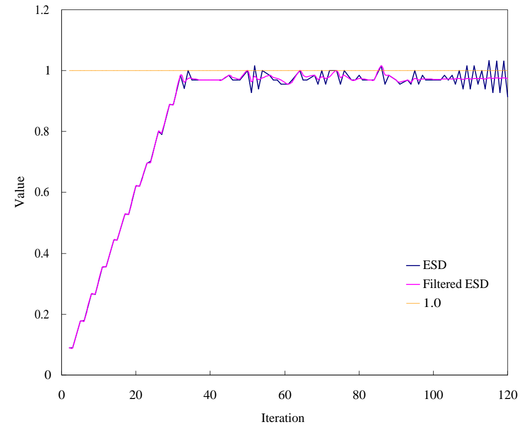
ment. We configure the tuning policy with $\Psi = 2$ (add/delete 2 resources at a time in order to shorten the tuning time), $\Delta = default$, $\Omega = default$. The desired speed QoS = 0.8 iterations/UT.

Figure 6.10(a) shows the application execution behaviour. The short and heavy drops during the tuning phase were due to RC reconfiguration overheads. After about 32 iterations, the execution speed was tuned to the satisfied level 0.8. The final resource number to satisfy the requirement was 20. Figure 6.10(b) shows the history ESD and the average-filtered ESD. The filtered ESD was much smoother than the raw ESD. We can see that even though there was a temporary ESD drop due to simulation overhead around iteration 34, the AA did not add any hosts because the tuning was done according to the filtered ESD and the AA only took action when filtered ESD was stable. The AU of the host during the execution was relatively stable, around 0.11. This was because of the homogeneous resource environment and the iterative execution behaviour. As we know, the execution speed increases linearly according to the number of hosts when allocating less than 20 hosts. Therefore the AU, which was the average of the ESD rewards, remained the same.

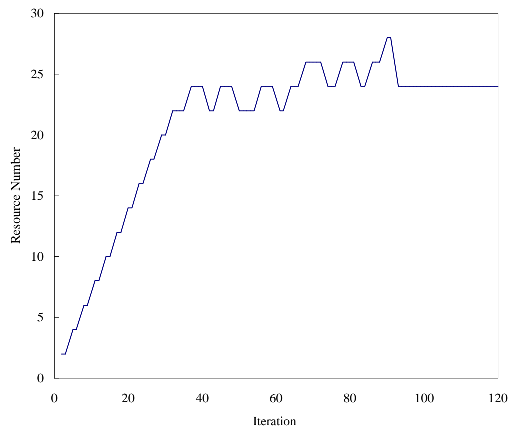
When we raised the QoS requirement to 1.0, interesting results showed up. As we showed in validation, “0.98” is the maximum speed the execution can reach due to various overheads. As Figure 6.11(a) shows, the final tuned speed vibrated around 0.98. The filtered ESDs could not reach 1.0 and that made the AA keep taking actions to tune the RC. Figure 6.11(c) shows how the AA continuously added and deleted resources to try to find the optimum number to make ESD = 1.0. However this was impossible



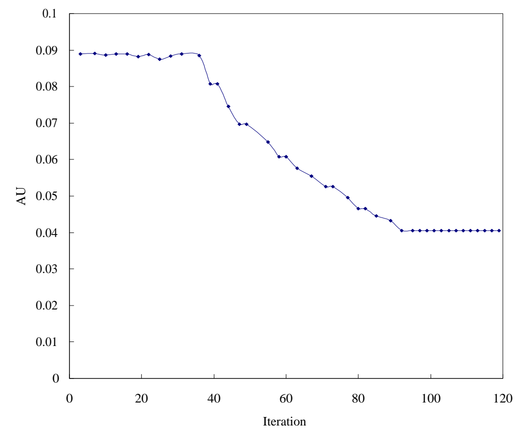
(a) The execution speed was tuned to approach the QoS requirement: 1.0 iteration/UT.



(b) The ESD was filtered to avoid overreacting.



(c) The number of resources in RC.



(d) The historical AU of type E.

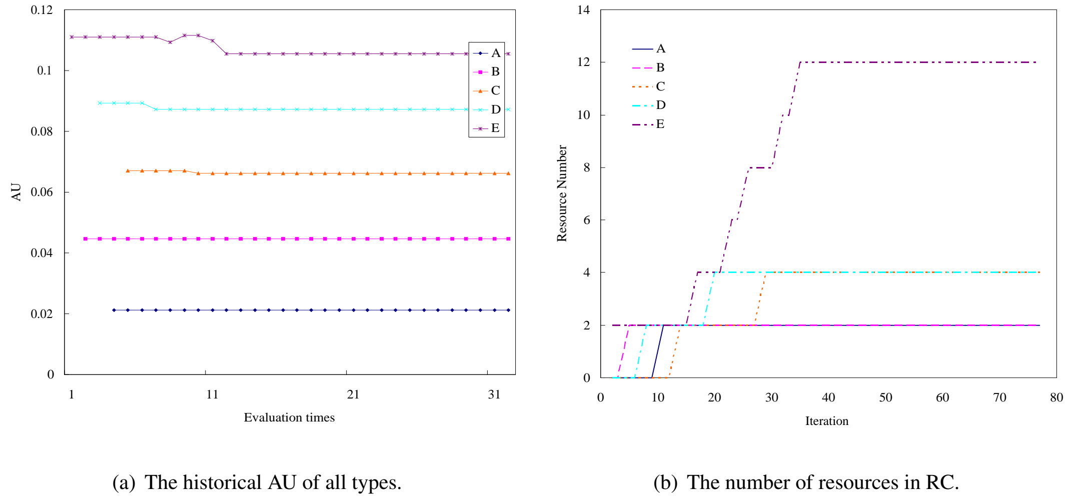
Figure 6.11: Results of experiment I.1 - homogeneous environment, QoS = 1.0 iterations/UT

because of communication overhead. After a couple of repeated reconfigurations, the AA stopped tuning and remained at a host number 24. The interesting result is the AU of type E. As we know, the speedup is limited by the communication delay when allocating more than 20 hosts E. The AU of type E started to decline after the resource number increased to 20 (Figure 6.11(d)), because adding a type E would make no difference for the execution speed (MU(E) = R(E) =0).

Comparing the two experiments, $AU(E)(QoS = 0.8) > AU(E)(QoS = 1.0)$. This does make sense. Type E is not a desirable host when the QoS =1.0 because even though it could bring positive performance effects it cannot completely satisfy the requirement.

6.4.5 Experiment I.2 - Heterogeneous Environment

In this experiment, the resource environment contains 5 host types A, B, C, D, E. Their locations are set to be all the same (Table 6.1). There is only an internal latency = 0.1 among them. This experiment aims to demonstrate that the AA is able to automatically classify the hosts according to their utilities to the application without application and host information provided in advance, and use the learned



(a) The historical AU of all types.

(b) The number of resources in RC.

Figure 6.12: Results of experiment I.2 - heterogeneous environment, QoS = 0.8 iterations/UT

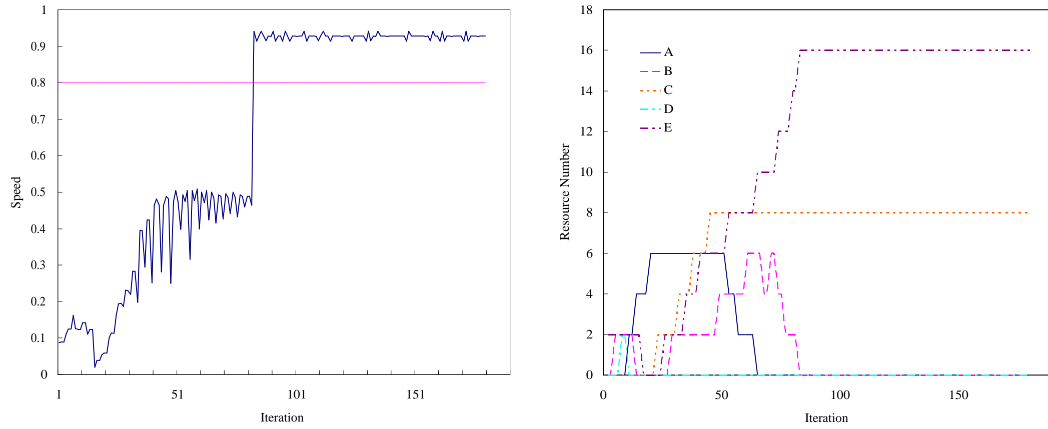
information to guide the resource tuning. We configure the tuning policy with $\Psi = 2$, $\Delta = default$, $\Omega = default$. The desired speed = 0.8 iterations/UT.

In the first execution, the speed was tuned to the satisfied level after about 35 iterations. During the execution, the AU of each type of hosts were evaluated, as shown in Figure 6.12(a). As we can see, all the values were stable throughout the execution. This was because the speedup was only effected by the processing capability but not communication latency due to relatively small QoS requirement we set. Their AUs were assigned according to their processing capabilities. For example, $AU(E) \approx 5 \times AU(A)$, which is consistent with the fact that $P(E) = 5 \times P(A)$. Note that the evaluation of each type started at different time, because the AA only added and evaluated one host type at a time. Figure 6.12(b) shows the total number of hosts used during the execution. We can see that in the beginning almost the same number of each type was used. However, after tuning a larger number of the E type were used. This was because all the hosts had the same default $\varepsilon = 0.5$, and therefore the same selection probability to be chosen in the beginning. With the evaluation of AU, type E received a higher selection probability therefore the AA preferred this type. The RC tuning process ended with 12E, 4D, 4C, 2B, 2A, totally 24 hosts to satisfy the 0.8 speed requirement.

After the execution, the AU table became:

Host Type	AU	Selection Probability
A	0.023	7%
B	0.044	13%
C	0.066	20%
D	0.088	27%
E	0.109	33%

By obtaining the AU values, the AA classified these hosts into 5 levels. Type E is the preferred type, having a 33% selection probability. This is in line with the fact that type E is the fastest and most



(a) The execution speed was tuned to satisfy the QoS requirement: 0.8 iteration/UT.

(b) The number of resources in RC.

Figure 6.13: Results of experiment I.3 - heterogeneous environment with cross-domain delay, QoS = 0.8 iterations/UT, central locus = cluster E.

useful host in the current execution situation. In the subsequent execution the AA can either configure the optimum RC found from last execution, or use the existing AU table to tune the RC from scratch to the optimum level by using the best hosts rather than selecting them randomly. The experimental result was that the AA took 28 iterations[8] (fewer than the first time) and used 14E, 4D, 4C to satisfy the QoS requirement. This time the AA did not use relative bad type A, B but used more type E.

6.4.6 Experiment I.3 - Heterogeneous Environment with Cross-domain Delay

In this experiment, the locations of the five host types we set as shown in Table 6.1. The aim of the experiment is to show that the AA is able to classify hosts and remove the bottleneck hosts to satisfy the QoS in a relative complex environment. The classification will take into account both the processing capability and the communication delay (location). We configure the tuning policy with $\Psi = 2$, $\Delta = default$, $\Omega = default$. The desired speed = 0.8 iterations/UT.

Figure 6.13(a) shows how the execution got tuned to the required level. We notice that the speed does not increase steadily as more resources are added. This is due to the communication delay some of the hosts introduces. For example, around iteration 10 the speed drops significantly due to the introducing of type D, which is located far from the allocated type E hosts therefore brings large communication delays. Between iteration 40 $\sim$ 90, the speed gets stuck around 0.5 and does not increase. This is because the RC during this time includes both type B and type E. The communication delay between these two types (theoretically $10 \times (0.4 - 0.2) = 2.0$) dragged the speed down to 0.5. Fortunately by learning and updating the AU table the AA finally removed the bottleneck type B and made the speed raise rapidly to 0.9. From Figure 6.13(b) we can clearly see how the AA initially tried each type and gradually removed the unsuitable hosts D, A, B and only added the positive hosts C and E. Finally the AA successfully tuned the RC to satisfy the requirement by using 16E and 8C. In this execution the AA

[8]The tuning process (30 iteration) was also affected by the Tuning Amplitude Ψ - bigger Ψ would lead quicker tuning process.

chose cluster E as the central locus [BWF+96], and hosts in the RC were gradually assembled to be close to cluster E to avoid long communication delay. After the execution, the AU table became:

Host Type	AU	Selection Probability
A	0.008	5%
B	0.013	9%
C	0.045	29%
D	0.000	1%
E	0.086	56%

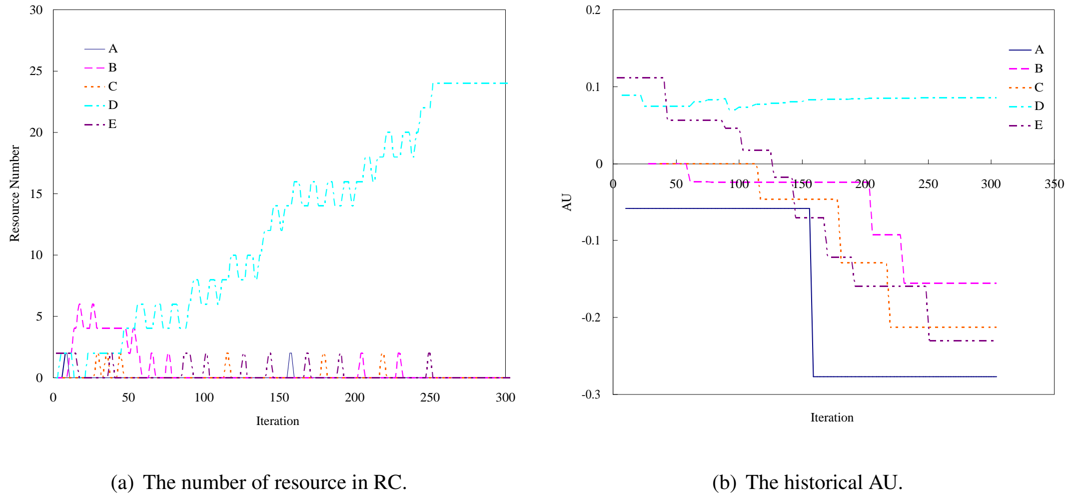
The table tells us that type E is the most suitable hosts and type C is the second best choice. In later executions, the AA will prefer adding type E and C. The result is a reflection of that applications need close hosts. In this execution, the AA chose cluster E as the central locus and other hosts were evaluated based on that choice. Even though type D is the second fastest, it is regarded as the worst host because of its location.

Without any prior knowledge, the AA successfully found the satisfied RC and classified the resources by autonomously learning and tuning. By using the existing classification, the AA is able to tune the RC faster. During the second execution, the AA only took 50 iterations to satisfy the requirement.

The selection of a central locus is automatically done by the AA. As type E is the fastest host and is relative close to other hosts, it always has the biggest AU. The AA therefore choose cluster E as the central locus. However this is not always the case because of the randomness in the policy. Interestingly in one of the experiments the AA also chose cluster D (D is the second fastest) as the central locus, although this situation was very rare. In this experiment the AA used a total of 24 type D to satisfy the requirement. All the other types were abandoned. Figure 6.14(a) shows the RC tuning process. It took around 250 iterations to finish the tuning. This was mainly because type E had a very high AU in the beginning. It took a long time with repeated adding and releasing for the AA to learn that type E was actually bad when choosing cluster D as the central locus. Figure6.14(b) shows the AU updating history. The AU of type D is the only one increasing. For all other types the AU decreased. After the execution, type D had the highest AU, and a selection probability around 60%.

In cases where it is difficult to determine the characteristics of an application and how well resources will perform for it, this approach is undoubtedly great help to users who do not need to provide any configuration information. The feature is very different from the current techniques (e.g. AppLes) that calculate resource selection according to the information provided in advance.

There is another situation the current resource selection techniques find difficult to deal with. As we know, the allocation delay in a grid environment is always not bound. It is difficult to predict which resources will be available at a specified time. If we make the resource selection in advance and only use the candidate hosts it may take a long time to assemble sufficient hosts. In contrast, we here try any available host as long as it is able to improve the performance of the application, even if it is not necessarily the best. In the next experiment, we will see how the AA deals with a dynamic resource



(a) The number of resource in RC.

(b) The historical AU.

Figure 6.14: Results of experiment I.3 - heterogeneous environment with cross-domain delay, QoS = 0.8 iterations/UT, central locus = cluster D.

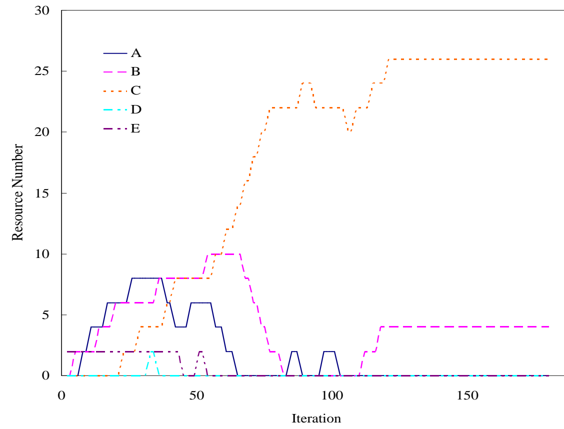


Figure 6.15: The number of resource in RC in experiment I.4 - dynamic environment, QoS = 0.8 iterations/UT

environment.

6.4.7 Experiment I.4 - Dynamic Resource Environment

In this experiment, we change the resource environment to be dynamic. We assume that due to competition from other jobs, the required hosts are not always available to the AA. The availability probabilities are set to 100%, 80%, 60%, 40% and 20% respectively for type A, B, C, D, and E (Table 6.1). For example, the 80% availability probability means that if the AA requests to add type B 10 times, the request will on average fail twice. In a real-world scenario the availability probability can also change dynamically. Here we use a fixed probability because it makes it easier to demonstrate the effectiveness of the approach. We configure the tuning policy with $\Psi = 2$, $\Delta = default$, $\Omega = default$. The desired speed = 0.8 iterations/UT.

In the dynamic environment, through about 120 iterations tuning, the AA finally used 25 type C

and 4 type B to satisfy the requirement. Figure 6.15 shows the historical host number in RC. We can see now the AA gradually abandoned other types and started selecting type C and B. This result reflects the availability probabilities in the clusters. Because it was difficult to obtain type E, the AA chose the alternative type C as the central locus. Other hosts were then assembled because they were close to type C. Since type C has lower processing capability, the number of total hosts in the final RC was larger than that in experiment I.3. After the execution, the utility table changed to:

Host Type	AU	Selection Probability
A	-0.036	20%
B	0.043	26%
C	0.069	29%
D	-0.3	1%
E	0.015	24%

As we expected, cluster C has the highest AU because of its relative high processing capability and high availability probability. Type E however is not regarded as good as experiment I.3 because resources of this type were too difficult to obtain. Type B replaced type E to be the second preference because resources of this type are close to those of type C, has a reasonable performance and a high availability probability. The utility table thus does not only represent the static attributes of hosts but also the dynamic characteristics such as availability as demonstrated here.

Based on the AU table, the AA will prefer to choose hosts in cluster C and B, because they are good, and also easy to obtain.

6.5 Experiment II - Economy-based Task Farming

In the last experiment the Utility Classification policy showed its effectiveness in supporting iterative applications. The execution requirements of iterative applications are always some kinds of execution rates (iteration/sec, cost/iteration). The ESDs are independent of previous iteration behaviour and do not change frequently under the same RC.

For task farming applications which are usually non-iterative, the QoS requirements will involve the whole execution (e.g. execution deadline and budget). The ESD is therefore more heavily affected by previous behaviour and could dynamically change even under the same RC. This brings additional difficulty to the tuning. In the following experiments we will demonstrate that the AA is able to support non-iterative applications and can also support multiple QoS requirements.

6.5.1 Experimental Setup

The Parameter sweep application [BGA00, BMA02] is a common task farming application. This model contains a large number of independent tasks operating on different data sets (Single Program Multiple Data (SPMD) model). A range of scenarios and parameters to be explored are applied to the application as the input values to generate different data sets. Each task therefore take similar time to compute. In order to compare the effectiveness of our approach with existing techniques, we build the experiment based

Host/Type	**Process ability** (UD/UT)	**Cost** (unit cost(UC)/UT)
0	1	10
1	1	12
2	1	14
3	1	16
4	1	18
5	1	20
6	1	22
7	1	24
8	1	26
9	1	28

Table 6.2: Simulated resource environment composed of 10 heterogeneous hosts.

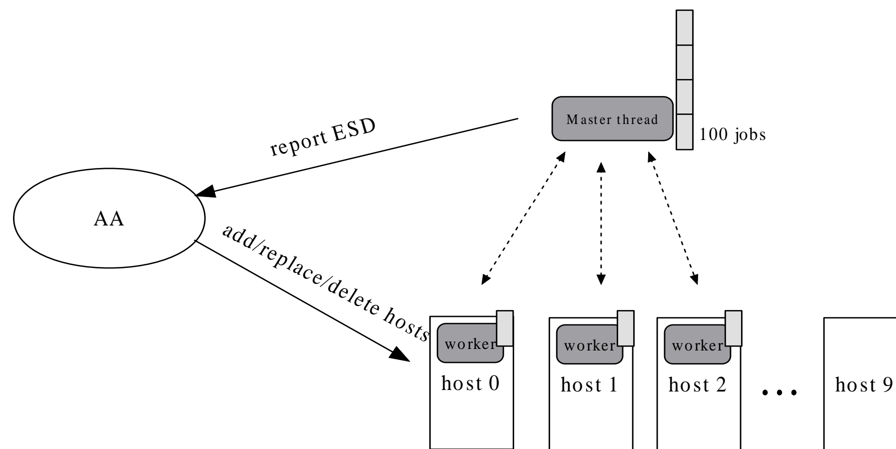


Figure 6.16: Simulated parameter sweep application, resources and the AA. The master thread reports the ESD and the AA thread decides to add/replace/delete worker threads.

on paper [BGA00], which introduce the well-known deadline and budget constrained (DBC) scheduling algorithms.

We simulate the 10 hosts classified by their costs in Table 6.2 according to paper [BGA00].

The adaptive application is created by a master Java thread and a number of worker Java threads. The master thread has 100 tasks and each tasks has 90 units of data to process. The master thread continuously distributes the tasks to idle workers to process. Worker threads simulate the processing delay by sleeping 100 millisecond for 1 UT. For example, if a worker is spawned on host 0 then its processing capability is 1 UD/UT. To compute 1 UD, it will take 100 millisecond in real time. One worker is spawned on one host. The maximum number of workers is therefore 10. The job of another Java thread of the AA is to add/replace/delete hosts (worker threads) for the application according to the policy to satisfy its execution requirements - the deadline and budget (Figure 6.16).

The optimal deadline for this experiment is archived when all the hosts are allocated for the appli-

cation, which should be ideally 900. Due to simulation overhead, the simulation always takes more time to finish (our model took 92641 millisecond and 926 UT when allocating all the hosts). We therefore set the optimal deadline to be $900 \times (1 + 10\%)$ UT. The lowest budget of 126000 UC is the budget required to execute 20 tasks on each of the 5 cheapest queues. For this value, the deadline of 990 UT is not feasible and a deadline of 1980 (990×2) is required.

In the experiments the QoS specified by the user is 1980 UT deadline with a budget of 126000 (the reason we have chosen this QoS is that it is relatively difficult to achieve according to paper [BGA00]). As there are two requirements (deadline and budget) we need to define two sub ESDs and combine them in some way. Here we use the Cobb-Douglas function to create the ESD function.

α_1 = Actual computing rate (UD/UT) at time t.
ε_1 = Desired computing rate (UD/UT) at time t = $(Total_data - computed_data)/(Deadline - t)$.
α_2 = Actual cost rate (UC/UD) at time t.
ε_2 = Desired cost rate (UC/UD) at time t = $(Budget - used_cost)/(Total_data - computed_data)$
$ESD_1 = \alpha_1/\varepsilon_1$. if $ESD_1 > 1.0, ESD_1 = 1.0$
$ESD_2 = \varepsilon_2/\alpha_2$. if $ESD_2 > 1.0, ESD_2 = 1.0$
$ESD = ESD_1^{\alpha} \times ESD_2^{1-\alpha}$

ESD_1, which represents the deadline satisfaction degree, is displaced in Figure 6.17. ESD_2 representing the cost satisfaction degree, is influenced by ESD_1. The budget requirement is more difficult to fulfill because the budge is the lowest. The budget ESD therefore has a bigger influence on the total ESD. So $(1 - \alpha) > 0.5$. Meanwhile, we still want to satisfy the deadline as much as possible, so α cannot be too small, otherwise the ESD feedback is difficult to represent the deadline requirement. So we set $\alpha = 0.3$.

The application reports ESD to the AA every 10 UT. The tuning policy is configured by $\Psi = default, \Delta = default, \Omega = 0$. The reason we set $\Omega = 0$ is that we do not want to give too many chances to the “bad” hosts in order to save the tuning time, which has a great impact on the completion time. This is different from iterative applications where the performance is rarely related to the completion time.

6.5.2 Experiment II.1

As the policy has some randomness at the start of the tuning process, we run 5 tests to get an average result. Table 6.3 shows the results of the 5 tests. For the first executions (executions without learning the AU tables), the average completed percentage is 91%. We can see there are some differences among tests for the first executions. This is due to the randomness of resource selection in the beginning. For example, in test 2 which has the worst performance, the AA unfortunately chose several bad hosts like 6, 7, 8, and 9 in the beginning. Since those hosts still brought positive effects to the application (increased the deadline ESD) in the early stage of execution, the AA initially regarded them as good hosts. Gradually, the AA realized that those hosts were not so good (damaged the budget ESD) and re-evaluated them. The re-evaluation process extended the tuning process and additional costs were

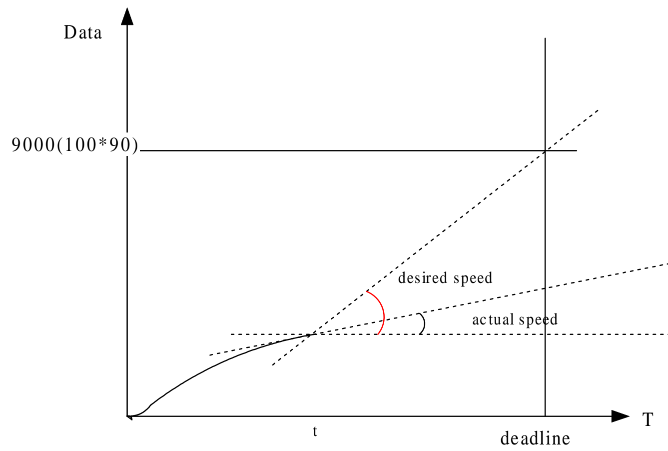


Figure 6.17: The application is expected to run as the "desired speed" from time t in order to meet the deadline. We calculate the satisfaction degree of the deadline at time t as $actual_speed/desired_speed$.

	First Execution			Second Execution		
	Completed	Time	Cost	Completed	Time	Cost
Test 1	91%	1797	126000	97%	1858	126000
Test 2	82%	1027	126000	97%	1765	126000
Test 3	87%	1410	126000	95%	1884	126000
Test 4	100%	1860	126000	100%	1862	126000
Test 5	93%	1783	126000	99%	1980	125724
Average	91% (SD = 6.7%)			98% (SD = 1.9%)		

Table 6.3: Test results in experiment II.1. The "first executions" are performed without any information provided before execution, and the "second executions" are performed based on the AU tables obtained from the "first executions". Completed: the percentage of the completed tasks in the total tasks within the Deadline and Budget. Time: total competition time, which cannot go beyond the deadline. Cost: total cost, which cannot go beyond the budget.

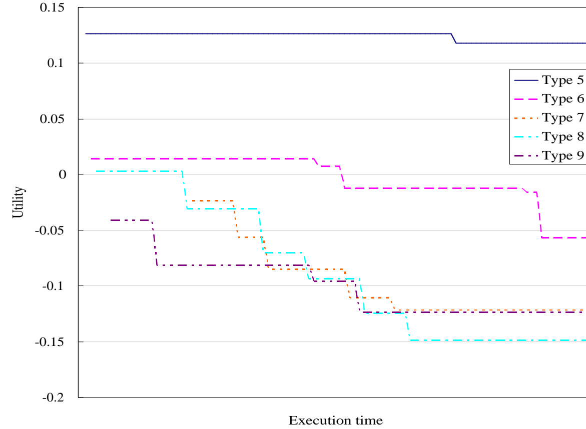


Figure 6.18: The historical AU of type 5 $\sim$ 9 in the first execution of test 2.

incurred during the “try and release” process. Even so, the AA finally realised their real utilities with respect to the application. Figure 6.18 shows the AU evaluation process for type 5, 6, 7, 8, and 9 in the initial execution of test 2. As we know, to satisfy the 126000 budget, the application can only use 5 cheapest hosts, i.e. type 0, 1, 2, 3, and 4. Other types should have negative AUs to the application. From the figure we can see that although type 6, 7, 8, and 9 gave positive AUs in the beginning, their AUs gradually declined to be negative when more evaluation were performed. The AU values were very useful for later executions - during the second execution of test 2, based on the learned AU table, the execution performance was improved to 97%.

The opposite example to test 2 is test 4, in which the AA luckily selected the very suitable hosts 0, 1, 2, 3, and 4 from the beginning. Since the ESD was already satisfied at that stage, the AA stopped tuning and just used the 5 hosts to execute the application. Therefore the tuning process was very short and the execution performance was very good (100% success in the first execution).

Based on the AU table produced by the first execution in each test, the performance of the second execution in each test is much better and more stable. The average completed percentage rises up to 98%. We can see that no matter how good or bad the first executions are, the AA can still produce AU tables that will enhance the performance of subsequent executions.

Let us take test 5 as an example (other tests have similar behaviour) and use it to investigate some detailed information from the tuning process. In this test, the application completed 93% tasks within the deadline and budget (1783 UT, 126000 UC). Figure 6.19(a) shows the historical ESDs. In this Figure ESD_1 was higher than ESD_2 on average, because the deadline that ESD_1 represents is a looser requirement. The ESD kept changing due to the tuning. In the end stage of execution, the ESD dropped quickly no matter how the AA tuned. This was because the processing and tuning had already ruined the satisfaction regarding deadline and budget. Figure 6.19(b) shows the history of host number in the RC. Even though the number oscillated, the average number was about 5. After the first execution, the AU table became:

Host Type	AU	Selection Probability
0	0.608	22%
1	0.674	24%
2	0.151	12%
3	0.621	23%
4	-0.192	5%
5	-0.259	3%
6	-0.138	6%
7	-0.289	2%
8	-0.323	2%
9	-0.408	1%

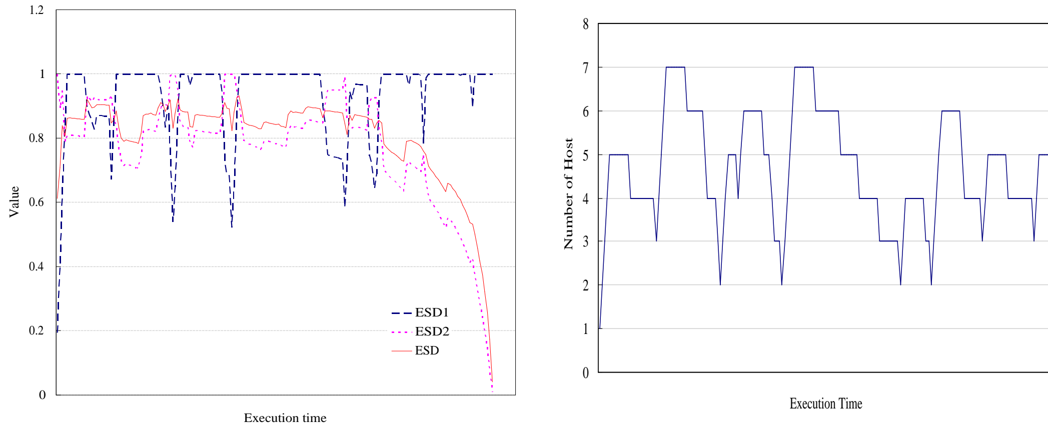
Inspiringly, without any knowledge of the resources, the AA managed to find the desirability of resources from the on-line learning process. We can see now the AA assigned high AU values to types $0 \sim 3$. This is consistent with the fact that these hosts have the highest cost performance and highest AU to the application under the requirement (as we know, in order to meet the requirement the application only needs the 5 cheapest hosts: host $0 \sim 4$). Type $5 \sim 9$ were assigned negative AU values because they could only bring negative effects to the application under such requirements. Their utilities were descending and host 9 had the lowest AU, because it is the most expensive host. One small error is associate with the host 4, which was supposed to have a higher utility value. One of the reasons for this discrepancy is that the AA had randomly selected too many hosts in the beginning and this reduced the host requirements in terms of completion time afterwards, which made host 4 less important.

Having established the AU table, it is much easier for the AA to tune the RC during subsequent executions. In the second execution, the AA completed 99.9% tasks by 1980 UT, with 125724 UC. Figure 6.20(a) and Figure 6.20(b) show the ESDs and RC history. As we know, the number of hosts in the optimum RC is supposed to be 5, since the budget of 126000 units is only enough to complete the application if the cheapest 5 nodes are used. However because of the initial spike in the RC tuning, the AA ran out of money in the later stages of execution. The AA therefore was forced to release one host to safeguard the budget. This action slightly slowed down the execution and the application could not be 100% completed before the deadline.

6.5.2.1 Comparing with the DBC Scheduling Algorithms

Table 6.4 shows a summary of performance comparison between the Utility Classification and the DBC scheduling algorithms. Here the Utility Classification policy shows its clear effectiveness. For the QoS of 990 and 17100, to complete 100% tasks all hosts much be allocated. Since the Utility Classification algorithm only deals with one type of host at a time, it takes time to add all the hosts to the execution - the adding procedure reduced the performance about 0.1%.

We must stress that the Utility Classification policy is performed without any prior resource knowledge, while the DBC algorithm relies on resource information like cost performance being made available and also needs to perform an evaluation in advance. However in some cases, the cost performance



(a) The historical ESD_1, ESD_2, and ESD in the first execution. $ESD = ESD_1^{0.3} \times ESD_2^{0.7}$).

(b) The historical number of hosts during the first execution.

Figure 6.19: First execution in Experiment II.1 .

of a host to an application could not be calculated just based on general information like SPEC rank, because a host may have different execution performance to different applications. This knowledge can only be obtained from previous execution experiences. This indirectly increases the workload of the user. In contrast, the Utility Classification policy learns the knowledge automatically during the tuning process and the users does not need to do any resource calculation beforehand.

6.5.3 Experiment II.2 - Small Grain Size

In experiment II.1, although the second executions completed 98% job on average, the initial executions were not very inspiring (completed only 91%). We believe that the reaction of the AA to a resource selection has not been optimised. Because the data grain size is one task (90 unit data) and no migration policy applied in the application, when the AA decides to delete a host, the application cannot kill the related process until the under processing task has been completed. The slow reaction resulting from this causes the AA to over-estimate or under-estimate the AU of deleted hosts. For example, deleting host 9 is supposed to make the ESD climb. However, due to the deletion delay combined with the ESD dropping trend, the ESD might be smaller than the ESD before host 9 has been deleted and this has a decremental effect on the utility evaluation for host 9. Moreover, the slow reaction will extend the tuning process, and thereby affect the deadline and the budget satisfaction.

An easy way to speed up the application's reaction time is to decrease the grain size. In this experiment, we slightly change the data set: the master has 9000 tasks and each task only has 1 unit data. The total data amount is the same as last experiment but the grain size is much smaller.

We use the same ESD function and the same QoS requirement: Deadline = 1980, Budget = 126000. Table 6.5 shows the results. As we expected, the average performance of the first executions increases to 98%, which is much better than in experiment 1. In the second executions all the tests were 100% completed within the grain size restrictions.

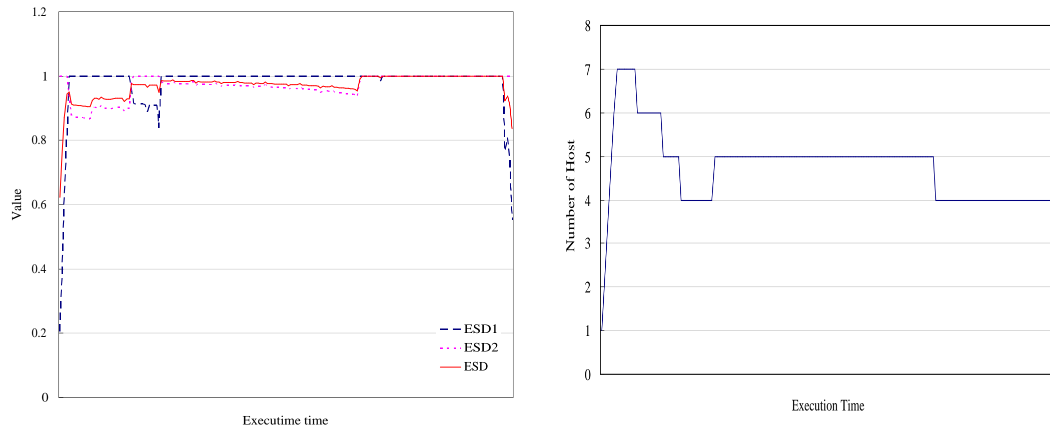
Figure 6.21(a) and Figure 6.21(b) show the history of ESDs and RCs of Test 5. Comparing with

QoS Requirement	*Algorithm*	*Completed*	*Time*	*Cost*
D = 1980, C = 126000	Utility Classification (first execution)	91%	1575	126000
	Utility Classification (second execution)	98%	1870	126000
	DBC Cost Minimisation	98%	1939	125440
	DBC None Minimisation	97%	1912	125900
	DBC Time Minimisation	82%	1968	102350
D = 990, C = 171000	Utility Classification (first execution)	99.9%	990	169660
	Utility Classification (second execution)	99.9%	990	169850
	DBC Cost Minimisation	86%	982	139800
	DBC None Minimisation	99%	930	168690
	DBC Time Minimisation	100%	933	171000

Table 6.4: Comparing the Utility Classification algorithm with the DBC scheduling algorithms [BGA00]. D = Deadline, C = Budget.

	First Execution			Second Execution		
	Completed	Time	Cost	Completed	Time	Cost
Test 1	98%	1967	126000	100%	1855	126000
Test 2	95%	1980	121908	100%	1843	125814
Test 3	99%	1836	126000	100%	1851	126000
Test 4	100%	1860	126000	100%	1862	126000
Test 5	97%	1725	126000	100%	1859	126000
Average	98% (SD=1.9%)			100% (SD=0.0%)		

Table 6.5: Test results in experiment II.2.



(a) The historical ESD_1, ESD_2, and ESD in the second execution. $ESD = ESD_1^{0.3} \times ESD_2^{0.7}$). (b) The historical number of hosts during the second execution.

Figure 6.20: Second execution in Experiment II.1 .

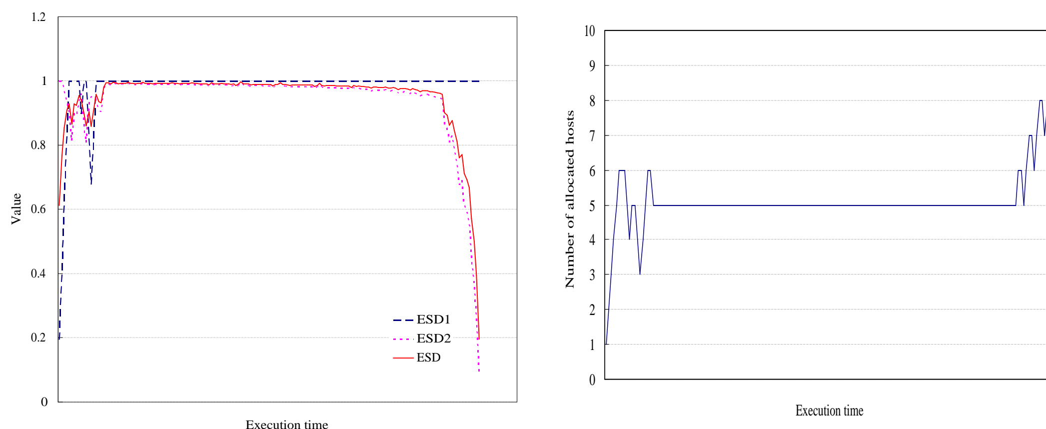
Figure 6.19(a), the tuning process in the experiment is fine while the process in experiment II.1 is coarse. After the execution, the AUs are much closer to the true values because the AA had more times to evaluate the utility and was given more accurate ESD reactions because of the smaller grain tasks:

Host Type	Utility	Selection Probability
0	0.623	36%
1	0.170	15%
2	0.120	14%
3	0.098	12%
4	-0.003	8%
5	-0.156	1%
6	-0.020	7%
7	-0.076	4%
8	-0.124	2%
9	-0.176	1%

We can see the AU values decrement from host 0 to host 9. Host $0 \sim 3$ are assigned the absolute advantages, and the utilities of other hosts are sorted according to their real cost performance. Unsurprisingly, relying on the AU table, in the second execution the AA completed the application 100% by 1859 UT; 126000 UC.

6.6 The Utility Classification Policy Summary

The Utility Classification policy classifies hosts according to their historical performance contributions to the application. According to this classification, the AA chooses high utility hosts and withdraws low utility hosts to configure an optimum RC to satisfy user's QoS requirements. An important feature of the



(a) The historical ESD_1, ESD_2, and ESD in the first execution.

(b) The historical number of hosts in the first execution.

Figure 6.21: First execution in Experiment II.2 .

policy is that it does not require users to provide any information regarding applications nor resources in advance. This unique feature helps users who do not have parallel computing background to execute distributed applications very easily. Moreover, the classification can be used to guide either manual or automatic resource selection in future executions. The information systems like MDS and NWS only provide a host's benchmark performance, which might not be the real performance to a specific application.

The policy can also collaborate with grid information systems. For example, the CPU load on a host can be dynamic. The utility a host provides to an application will therefore also be dynamic due to the load change. In this case, we need to use information systems to monitor the load and calculate the utility taking account of the load. Moreover, resource information provided in advanced can improve the tuning performance. For example, in experiment I, the AA can initially choose the highest CPU rank hosts, which may provide the highest performance to the application. The tuning process can be shortened by selecting the right resources in the first place.

The utility (AU) used by this policy is similar to the reputation notion [AM02] which has been considered in a wide variety of systems. The online auction system eBay (www.ebay.com) is an important example of successful reputation management. In eBays reputation system, buyers and sellers can rate each other after each transaction, and the overall reputation of a participant is the sum of these ratings over the last six months and can be used to determine the trust. PeerTrust [XLS04] developed a trust mechanism for system in which peers can quantify and compare the trustworthiness of other peers and perform trusted interactions based on their past interaction histories without trusted third parties. We have adopted this idea to be used in the resource selection of distributed computing.

Chapter 7

Automatic Resource Configuration and the DPPE Policy

This chapter continues introducing the Online Feedback-based Automatic Resource Configuration (OAC). In Section 7.1 we introduce another learning and tuning policy, the Desired Processing Power Estimation (DPPE), and two experiments to validate it. In Section 7.2 we present the related work of OAC. In Section 7.3 we summarize the OAC approach and present the comparison of the two tuning polices.

7.1 The DPPE Policy

This policy was initially designed to support embarrassing parallel applications, whose execution performances are only influenced by the processing capability of host. The policy calculates desired total processing capability according to the current execution behaviour, and adds hosts to reach the desired total processing power. It is based on an assumption that the ESD has a linear relationship with the total processing capability:

Assumption

N_i = *the number of computational objects on host i;*

P_i = *the processing capability (instruction per time unit) of hosts i*;

M = *the number of instruction of an object, given that each object has the same amount of instruction*

T = *the total time for completing the application.*

Since an adaptive application can balance the load at any time:

$\frac{N_1 \cdot M}{P_1} \approx \frac{N_2 \cdot M}{P_2} \approx ... \approx \frac{N_i \cdot M}{P_i} \approx T$

We can get:

$\sum N_i = N_1 + N_2 + ...N_i \approx \frac{T}{M} * (P_1 + P_2 + ...P_i) \approx \frac{T}{M} * \sum P_i$

$T \approx \frac{\sum N_i \cdot M}{\sum P_i}$

Therefore, ideally the reported ESD, which is related to T, has an approximate linear relationship with the $\sum P_i$. According to the ESD and the current $\sum P_i$ which is monitored by the AA, the AA can estimate the desired $\sum P_i$. We use Φ to denote the desired $\sum P_i$ that is supposed to satisfy the

performance requirement. For each report, $\Phi_t = (1.0/ESD_t) \times (\sum P_i)_t$ (direct estimated). However, in reality small variation in the many parameters that have influence on the performance (e.g. adaptive overhead, load imbalance or operation system variation), a single estimation is prone to errors. For example, a sudden spike in the CPU load may lead to a drastic drop in the ESD and therefore a sharp rise in the value of Φ. In response to this rise the AA will immediately make a large number of processors available to the application. Such a reaction is not only superfluous, it can severely harm the performance because of the remedial actions the AA needs to take afterwards and the delays they impose on the system. The AA therefore need to ignore the noise and concentrate on the long term trends.

We apply a Kalman filter [WB06] to remove the effects of noise and get a better estimate of the current Φ. Kalman filters are essentially a method of discrete signal processing that provide optimal estimate of the current state of a dynamic system described by a *state vector*. The state (Φ) is updated using periodic observations of the system or recursive estimation equations. One of the main advantages of Kalman filters is that there is very little computational and storage overhead as they are expressed through recursive equations. The entire history of the system does not have to be maintained and it is sufficient to record the value of the current inner-state and the parameters of the recursive equations, updated at every step [CMEM07].

7.1.1 Applying Kalman Filter

7.1.1.1 State model

We apply Kalman filter theory to address the one-dimensional discrete-time controlled process that is governed by the linear stochastic difference equation:

$$\mathbf{x}_k = \mathbf{A}_k \mathbf{x}_{k-1} + \mathbf{B}_k \mathbf{u}_k + \mathbf{w}_k$$

where the $\mathbf{x}_k$ is the true value of Φ at time k which is derived from the state at (k-1). $\mathbf{A}_k$ is the state transition model which is applied to the previous state $\mathbf{x}_{k-1}$. Since the execution behaviour is difficult to be predicted, it is difficult for the AA to determine the value of $\mathbf{A}_k$ in advance. To simplify this process, we assume that the state does not change from step to step (or at least it does not change frequently) so $\mathbf{A} = 1$. $\mathbf{B}_k$ is the control-input model which is applied to the control vector $\mathbf{u}_k$. There is no control input so $\mathbf{u} = 0$. $\mathbf{w}_k$ is the process noise which is assumed to be drawn from a zero mean Gaussian distribution with variance $\mathbf{Q}_k$. The determination of $\mathbf{Q}$ is generally difficult as we typically do not have the ability to directly observe the process we are estimating. Here we assumed the process is relatively stable, so we let $\mathbf{Q} = 1e - 5$[1]. Sometimes a relatively simple process model can produce acceptable results [WB06].

At time k an observation (or measurement of Φ) $\mathbf{z}_k$ of the true state $\mathbf{x}_k$ is made according to

$$\mathbf{z}_k = \mathbf{H}_k \mathbf{x}_k + \mathbf{v}_k$$

where $\mathbf{H}_k$ is the observation model which maps the true state space into the observed space. Our measurement is of the state directly so $\mathbf{H} = 1$[2]. $\mathbf{v}_k$ is the observation noise which is assumed to be

[1] We could let $\mathbf{Q} = 0$ but assuming a small but non-zero value gives us more flexibility in "tuning" the filter [WB06]; this can also be reflected in the later "predict" formula (to make $\mathbf{P} \neq 0$).

[2] We dropped the subscript k in several places because the respective parameters remain constant in our simple model.

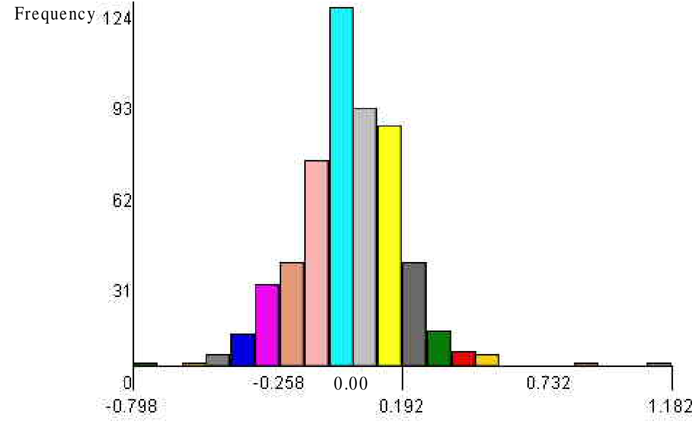


Figure 7.1: A histogram of observed Φ noise. N = 500, Mean = 0.00, Std.Dev = 0.19.

zero mean Gaussian white noise with variance $\mathbf{R}_k$. Figure 7.1 shows a histogram of observation noise, which is consistent with the Gaussian distribution. Because we do not evaluate the measurements before the execution (the AA does tuning on the fly), we need to set the measurement noise variance $\mathbf{R}$ in advance. Bigger $\mathbf{R}$ value means the measurements have larger variation. The system will then more "slowly" to believe the measurements. On the contrary, smaller value will make the system respond to measurements "quickly", therefore the filtered values can quickly adapt to the dynamics of the execution, but also "trust" more noisy measurements. We assume $\mathbf{R}$ is constant in this model.

7.1.1.2 Kalman filtering

The Kalman filter estimates a process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of (noisy) measurements. As such, the equations for the Kalman filter fall into two groups: time update equations and measurement update equations. The time update equations are responsible for projecting forward (in time) the current state and error covariance estimates to obtain the a priori estimates for the next time step (Prediction). The measurement update equations are responsible for the feedback, i.e. for incorporating a new measurement into the a priori estimate to obtain an improved a posteriori estimate (Update). The time update equations can also be thought of as predictor equations, while the measurement update equations can be thought of as corrector equations. In the following, the notation $\hat{\mathbf{x}}_{n|m}$ represents the estimate of $\mathbf{x}$ at time n given observations up to, and including at time m.

Predict

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{A}_k \hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_k \mathbf{u}_k = \hat{\mathbf{x}}_{k-1|k-1}$$
$$\mathbf{P}_{k|k-1} = \mathbf{A}_k \mathbf{P}_{k-1|k-1} \mathbf{A}_k^{\mathrm{T}} + \mathbf{Q}_k = \mathbf{P}_{k-1|k-1} + \mathbf{Q}$$

Update

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^{\mathrm{T}} (\mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^{\mathrm{T}} + \mathbf{R}_k)^{-1} = \mathbf{P}_{k|k-1} (\mathbf{P}_{k|k-1} + \mathbf{R})^{-1}$$
$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1}^{-} + \mathbf{K}_k (\mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k|k-1}) = \hat{\mathbf{x}}_{k|k-1}^{-} + \mathbf{K}_k (\mathbf{z}_k - \hat{\mathbf{x}}_{k|k-1})$$
$$\mathbf{P}_{k|k} = (I - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1} = (1 - \mathbf{K}_k) \mathbf{P}_{k|k-1}$$

The Kalman gain $\mathbf{K}_k$ will eventually converge. With a smaller $\mathbf{R}$, $\mathbf{K}_k$ will have a bigger value, therefore the observation $\mathbf{z}_k$ will have a bigger influence on the true value $\mathbf{x}_{\hat{k|k}}$. Since initially the AA does not have knowledge of the true value, it would hope that the measurements are reliable. We normally assume $\mathbf{R}$ is small.

The Kalman filtered desired processing power, which is $\hat{\mathbf{x}}_{k|k}$, is denoted by Φ^k.

7.1.2 The Policy

Upon receiving a ESD report, the AA first calculates Φ^k. Further it checks if the ESD is inside the satisfactory range. The default satisfactory range is $1.0 \sim 1.1$. If it is not and the current $\sum P_i \neq \Phi^k$, the AA reconfigures the RC by adding, releasing or replacing hosts to make the $\sum P_i$ reach Φ^k.

There are two additional factors we need to consider when making a reconfiguration decision:

- The effect of RC reconfiguration of the execution may take some time due to the time taken by the application to adapt. For example, it may take a few seconds after a resource addition to see the ESD increasing. Therefore when the AA attempts to add hosts, it must initially check whether the ESD is increasing. If it is, it probably indicates that the application is adapting to the RC and the ESD may gradually reach the desired 1.0 value after some time. In this situation the AA will not perform any reconfiguration operation. The same policy is applied when the AA releases a host.

- The ESD may fluctuate under the same RC. For example, under the same RC, $ESD_{t=1} = 1.05$, and $ESD_{t=2} = 0.90$. For the second report, the AA will try to reconfigure the RC since the ESD is not in the satisfactory range. Fortunately we have already considered this problem by applying Kalman filter to filter the fluctuation. In this case, even though $ESD_{t=2} < 1.0$, the AA would not do any actions since the current $\sum P_i$ should have reached Φ^k.

Table 7.1 summaries the conditions to perform the three actions.

Note that, although the policy is initially designed to support embarrassingly parallel applications, it is also suitable to support parallel applications that involve communication. In the following we use two experiments based on two types of applications to validate this policy - a non-iterative embarrassingly parallel application and an iterative parallel application.

7.1.3 Experiment I - Mandelbrot

We first validate the policy by executing the Mandelbrot computation we used earlier in a homogeneous environment. This experiment aims to show the policy's quick tuning process and its ability to support highly dynamic applications.

Different from the application developed by a local load balancing algorithm in Section 5.5.2, we use the master-worker paradigm to build the adaptive application. The master distributes objects to each of the workers. When a worker finishes an object, it sends the result back to the master and then the master sends another object to it to compute. As we know, since not all the dots have the same computation amount (the dots belongs to the Mandelbrot set have more computation, because they take more iterations to reach the benchmark), the application has a variation of computation amount during

Actions	How to perform	When to perform
Adding	Add hosts until $\sum P_i > \Phi^k$	$ESD < 1.0$ $ESD <= ESD_{-1}$ $\sum P_i < \Phi^k$
Releasing	Release hosts until $\sum P_i - \min P_i < \Phi^k$	$ESD > 1.1$ $ESD >= ESD_{-1}$ $\sum P_i - \min P_i > \Phi^k$
Replacing	Request to add a host whose P $\Phi^k + \min P_i - \sum P_i < P < \min P_i$ If such a host is added, delete the $\min P_i$ host.	$ESD > 1.1$ $ESD >= ESD_{-1}$ $\sum P_i > \Phi^k$ $\sum P_i - \min P_i < \Phi^k$
ESD_{-1} : *the ESD value of last report.*		

Table 7.1: Summary of three actions: addition, releasing, and replacing.

the execution. Because we update the Mandelbrot canvas from the top down, the dynamic computing amount will be low $\rightarrow$ high $\rightarrow$ low. The resource requirement therefore is dynamic accordingly.

The resource pool for the experiment is the 57 hosts Condor cluster. Some hosts may have light load and the load does not change frequently. In order to see the change in $\sum P_i$, we set the standard P of a host with load 0 as 1.0. The P of a host with CPU load l is calculated by $1.0/(1 + l)$.

We set the execution deadline as the performance requirement. The ESD is defined as:

n_e = *the number of desired finished objects at time* t $= 3000 * t/deadline$

n = *the number of actual finished objects at time* t.

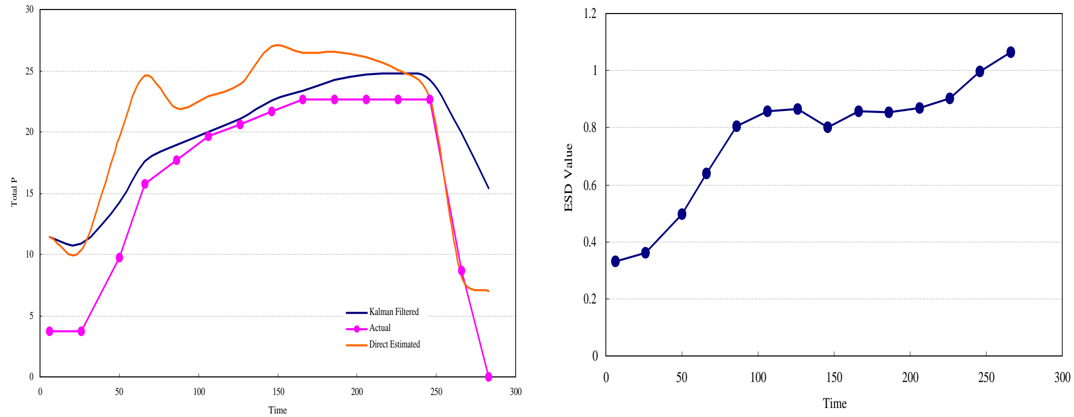
$ESD = n/n_e$

The application reports the ESD to the AA every 20 seconds. The AA's job is to make $ESD \geq 1.0$, with the *least* resources, to satisfy the execution deadline. Noise variance $\mathbf{R} = 10^{-4}$, so that the system can quickly respond to the dynamic behavior of the execution and can also remove small noise signals.

We did two tests, with the QoS requirement as a 300 seconds and 600 seconds deadline, respectively. For both tests the AA initially assigned 4 hosts. For deadline 300, the AA completed the application after 283 seconds and used 23 hosts at most. For deadline 600, the AA completed the application by 558 seconds, using 14 hosts at most. For both deadline the AA made accurate resource estimation and both deadlines were satisfied.

First of all, from Figure 7.2(a) and 7.3(a) we can see that Kalman filtered estimations were much smoother than direct estimations by eliminating small variations. And the filtered estimations also followed the dynamic resource requirement trend.

Figure 7.2(a) shows how the AA tuned the RC during the execution to meet the 300 seconds deadline. The Φ^k was continuously updated during the execution in response to the changing resource requirements of the application. In the beginning, the AA kept adding hosts according to Φ^k. The ESD value therefore kept increasing with the resource addition. Between time 100 $\sim$ 200, we notice the ESD



(a) Φ, Φ^k, and the actual $\sum P_i$, representing the change of RC. $\sum P_i \approx$ the number of hosts. Some hosts had mild CPU load.

(b) The historical ESD values during the execution.

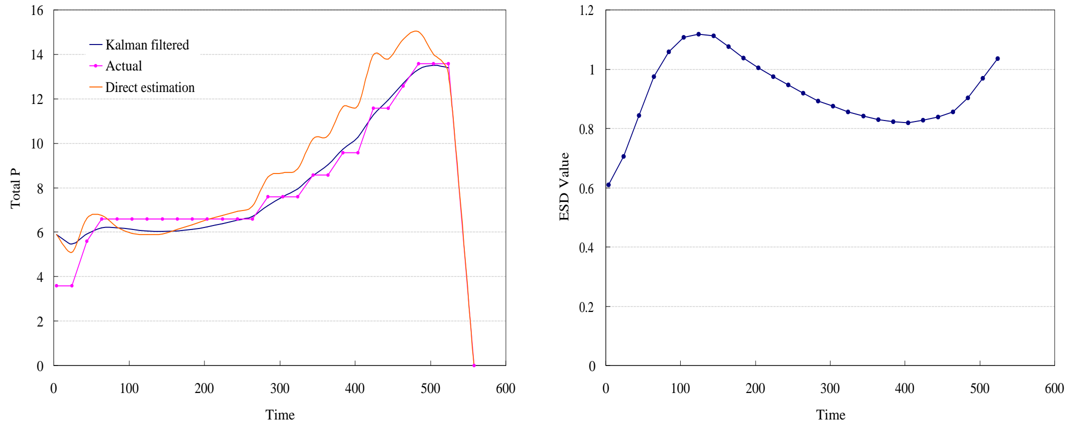
Figure 7.2: Satisfying the 300 seconds deadline.

has stopped increasing and remained around 0.8. This is because the increment of processing power was neutralized by the increment of the resource requirement of the application at this stage. At the end of the execution, due to a fall in computational load, Φ^k was dropping. As we anticipated, the AA did not only satisfy the deadline but also aimed to use the least number of resources to get high resource utilization. We can see how the AA started to release hosts after 250 seconds due to the Φ^k being updated. Also, because of the decline in the computational load the ESD started to increase and finally reached to 1.0, even though resources were being released.

Figure 7.3(a) shows how the AA tuned the RC during the execution to meet the 600 seconds deadline. The AA acquired 7 hosts in the initial stage (time 50 to 250) to satisfy the requirement since the computation amount was small. The ESD slowly increased to 1.0 due to the time the application took for adaption, as Figure 7.3(b) shows. Although Φ^k was smaller than the actual total P at this stage, the AA did not release hosts because it detected that releasing one host would result in a lack of resources. At a later stage in the execution, the AA detected that the ESD was dropping due to the increase of the computational load. The value of Φ^k was increasing accordingly and the AA added more hosts to reach the changed Φ^k to meet the performance requirement.

7.1.3.1 Compare with Manual Static Resource Selection

As we can see, the above policy is effective for automatically configuring resources to satisfy a certain deadline without the supervision of users. Moreover, the dynamic resource management makes it possible to use the least amount of resources to satisfy the deadline. Figure 7.4 shows the resource usages for using the automatic selection approach and two manual resource selections to satisfy the 600 seconds deadline. For the first manual selection, we statically partitioned the workspace into 15 parts and each part took charge of the computation of 200 rows. We allocated 15 hosts to compute each of them. It finished the execution in 655 seconds, which exceeded the deadline. The poor performance was because of the static resource allocation and also the imbalanced load distribution (some hosts had to take



(a) Φ, Φ^k, and the actual $\sum P_i$, representing the change of RC. $\sum P_i \approx$ the number of hosts. Some hosts have mild CPU load.

(b) The historical ESD values during the execution.

Figure 7.3: Satisfying the 600 seconds deadline.

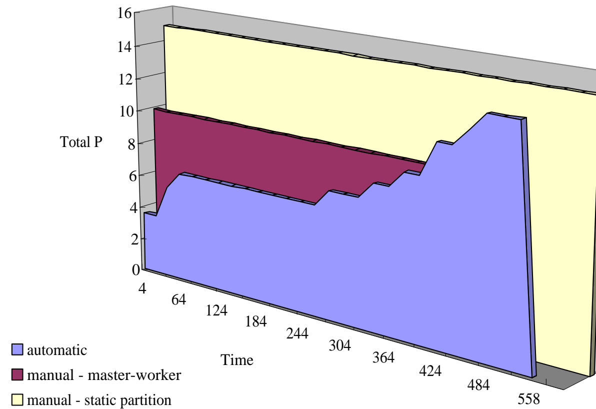


Figure 7.4: Resource usage comparison between automatic and manual resource selection

more time to finish a part). For the second manual selection, we used the master-worker distribution algorithm and allocated 10 hosts (10 workers) to execute the application. It completed the execution in 486 seconds. Although it satisfies the deadline because of its better load distribution, the resource usage ($\int_0^T (\sum P_i)_t \, dt$, i.e. the area in Figure 7.4) in the manual selection is about *20%* higher than the automatic selection. This indicates, by using our automatic selection with dynamic resource tuning according to the execution behaviour, we are able to optimize the resource utilization - using the least resources to satisfy the requirement so other users can utilize the saved resources.

7.1.4 Experiment II - Bulk Synchronous Parallel

In this section we validate the policy by a more complex execution situation - an iterative Bulk Synchronous Parallel (BSP) model application in a heterogeneous resource environment. This experiment aims to show the ability of the policy to support heterogeneous environments, and also iterative parallel applications that involve communication and sequential fraction.

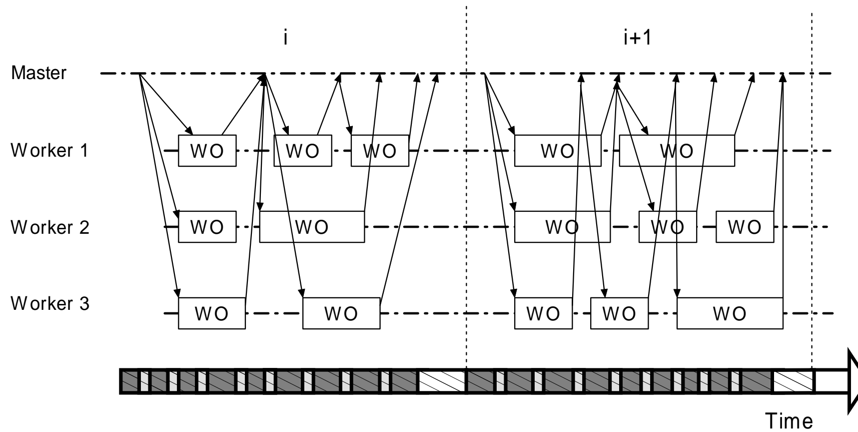


Figure 7.5: The BSP master-worker model. WO = work object. In the time line, dark gray area denotes computing time; light gray area denotes data sending/receiving time; white area denotes coordination time.

The BSP model [SHM96, MT04] divides parallel applications into a series of supersteps (iterations), each of which consists of parallel running threads/processes with any number of operations. The most important aspect of the BSP model is the barrier synchronisation which takes place at the end of each superstep. The individual parallel threads perform only local communication until the barrier. At this point all global communication takes place. Each superstep can be subdivided into three ordered phases consisting of: 1. computation locally in each process, using only values stored in the local memory of each processor; 2. communication actions amongst the processes, involving movement of data between processors; 3. a barriers synchronisation, which waits for all the communication actions to complete, and which then makes the data that was moved available to the local memories of the destination processors. The BSP model differs from many other parallel computing models in that it considers communication actions en masse to simplify the parallelization.

Since the processes are independent within the supersteps, we can package the work to be done in a superstep as separate work packets and farm them out to worker processes using a master-worker model, in order to make a BSP application into an adaptive application. In each iteration, the master continuously assigns work objects to the idle worker processes, which compute the work and return results to the master. When all work objects are done, the master goes through the result work objects stored in a work pool, and performs the information exchange operations by moving data from one work object to another. We call this coordination phase, which is sequential. After all the information exchange operations have been performed, each object in the work pool is replaced by its updated version. The next iteration begins as the master starts giving workers new work objects. Figure 7.5 illustrate the master-worker BSP working process. We use W to denote the computing time, C to denote the data sending and result receiving time and L to denote the coordination time. The time to finish one iteration is then $T = W + C + L$.

In this experiment we use a synthetic BSP application. The application has two stages. In stage one (from iteration 1 to iteration 20) the application uses 100 work objects per iteration and in stage

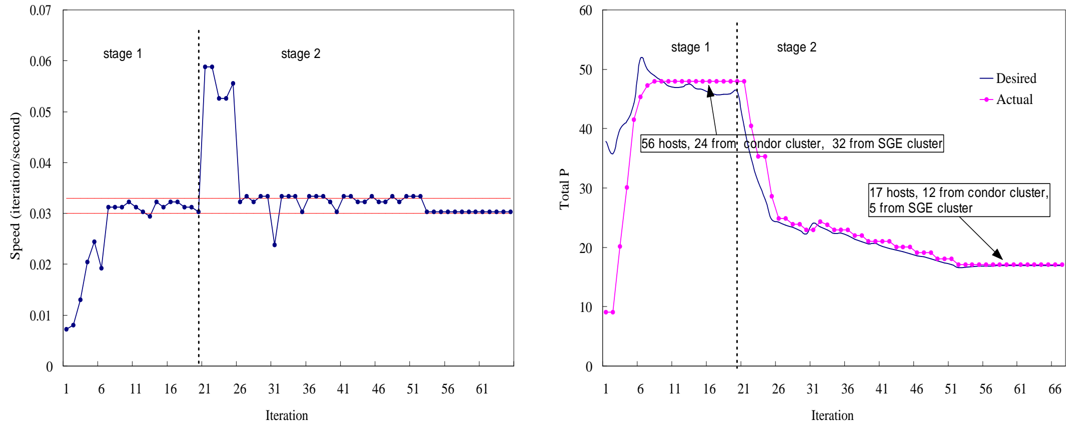
two (after iteration 20) it uses 40 work objects per iteration. Each object tasks 8 seconds on a Intel(R) Celeron(R) 2.80GHz processor. For each iteration the master takes $L = 5$ seconds for coordination. The resource environment is composed of two clusters. A Condor cluster composed of 57 hosts, each equipped with a Intel(R) Celeron(R) 2.80GHz processor and a heterogeneous SGE cluster composed of 198 hosts. The hosts in the SGE cluster are equipped with different number of (1, 4, 8) processors with processor types Intel(R) Xeon(R) E5462 2.80GHz and Intel(R) Xeon(R) X5355 2.66GHz. We use the processor clock as the benchmark to set up P. We set the P value of a processor with 2.80GHz with load 0 to 1.0. The P value of a processor with 2.66GHz can be calculated to 0.95. The P value with CPU load l is calculated by $P(load = 0)/(1 + l)$.

The QoS is a desired execution progression speed of 0.03 iterations/seconds, which means it requires to complete each iteration in 33 seconds. The application reports the $ESD = observed_speed/0.03$ in every iteration. $\mathbf{R} = 10^{-4}$, so that the system can respond quickly to the stage change.

The result is shown in Figure 7.6. The speed was quickly tuned to the desired range $(0.03 \sim 0.033)$ after the application startup. The AA employed 56 hosts from both clusters to compose the RC at this stage. After iteration 20, the execution went into the stage 2. Due to the sudden drop in the processing load, the execution speed increased to around 0.055. The AA immediately realised this and adjusted Φ^k. The AA then released some redundant hosts according to the updated value of Φ^k to tune the progression speed back to the desired range. Since the policy is based on the assumption that the ESD has a linear relationship with the total processing capability but in the experiment the ESD was also influenced by the times C and L, a single Φ^k estimation included errors that could not be neglected. Fortunately the policy is dynamic and the AA can always adjust the value of Φ^k according to ESD updates. That is one of the reason why Φ^k was decreasing from iteration $30 \sim 50$ - because the AA overestimated the resource demand in the first place and fine-tuned it. Another reason was that during this time the load of some hosts became light the AA therefore needed to recalculate the value of Φ^k. Finally the AA tuned the execution environment to 17 hosts to satisfy the 0.03 requirement.

7.1.5 Policy Summary

This policy dynamically configures the execution environment using the least amount of resources according to the estimated desired total processing power needed to satisfy users' execution requirements. Its quick tuning process makes it suitable to support short running applications. Although the assumption used in the policy is not valid for applications that involve communication or other overheads, the policy can also support them, because a single estimation of Φ^k does not need to be accurate. The value just gives the AA a rough idea on how many hosts needs to be added or deleted. As we demonstrated in experiment II, since the policy is dynamic, it can always adjust Φ^k according to the new ESD feedbacks with the help of the Kalman filter. If the assumption is valid, such as for embarrassingly parallel applications, the value of Φ^k can be accurate and the tuning process can be short. If the assumption is not valid, such as for parallel applications, the AA needs to take more time to adjust Φ^k and find the optimum configuration.



(a) The execution speed.

(b) The Φ^k, and the actual $\sum P_i$, representing the change of RC.

Figure 7.6: Resource configuration for the BSP application. QoS = 0.03iterations/second

This policy is especially useful to satisfy deadline-constrain job execution without the need for the users to provide information in advance. Current DRMs (scheduling systems) such as LSF [ZZWD93] and SGE require users to provide the estimate job length before they reserve resources according to the information to guarantee the deadline. The assumption is that users would be motivated to provide accurate estimates of job length, because (1) other jobs would have a better chance to backfill if their estimates are tight, but (2) would be killed if they are too short [TEF07]. Nevertheless, empirical studies of traces show that user estimates are generally inaccurate [MF01]. A possible reason is that users find the motivation to overestimate - so that jobs will not be killed - much stronger than the motivation to provide accurate estimates and help the scheduler to perform better packing. Moreover, the complexity of deadline scheduling makes it uncommon in practical job submission circumstance. In contrast, our approach requires no information from users as it automatically provides sufficient resources to satisfy the deadline. The AA simply relies on the FCFS policy in the DRMs rather than advance job scheduling polices, therefore the AA can integrate any DRMs to provide the deadline guarantee service to users. When users are greedy in terms of execution requirement (e.g. tight deadline), the AA will try to accommodate the request and since we do not rely on advanced reservation and backfilling, the greedy application will not be killed but it will run as fast as possible according to available resources.

7.2 OAC Related Work

The OAC approach involves the collaboration of many other research areas besides the automatic resource selection (Section 3.2.2). In this section we introduce other related work and the differences between them and our approach.

7.2.1 Control Theory and Reinforcement Learning

The OAC approach is initially motivated by the Control Theory [FPEN01]. In a closed-loop control system, a sensor monitors the output and feeds the data to a computer which continuously adjusts the

control input as necessary to keep the control error to a minimum. Feedback on how the system is actually performing allows the controller to dynamically compensate for disturbances to the system, such as changes in slope of the ground or wind speed. An ideal feedback control system cancels out all errors, effectively mitigating the effects of any forces that may or may not arise during operation and producing a response in the system that perfectly matches the user's wishes.

The Utility Classification policy has borrowed the exploration/exploitation techniques from reinforcement learning [KLM96]. Reinforcement learning is suitable for an environment that typically can be formulated as a finite-state Markov Decision Process, which is a discrete time stochastic control process characterized by a set of states; in each state there are several actions from which the decision maker must choose. In our research context, since resource environments and applications could be dynamic, it is difficult to define a reasonable state. For example, the reward may not be fixed when changing from state A to state B at different times if the application itself is dynamic. Even if we can create such a definition, the huge number of states could be difficult to track. Therefore we introduced the Utility Classification approach which classify resources based on a general basis rather than using specific states.

7.2.2 Feedback-based Scheduling

Our approach uses feedback to tune the execution environment. The approach is similar to the feedback control real-time scheduling that has been researched in recent years. The real-time scheduling project by University of Virginia [SHA+01b, LWK05] uses the theory to develop a resource scheduling algorithm to give quality of service guarantees in unpredictable environments to applications such as online trading, agile manufacturing and web services. Similar work can be found in paper [LSD08], which takes as input a target execution rate for an application and adjusts the applications' schedules to achieve those rates. Our work differentiates from them by not directly tuning the input (RC) because we do not know how to tune initially. We focus on learning information from the feedback and then performing the adjustment.

7.2.3 Performance Measurement and Tuning

Traditional performance measurement uses application instrumentation to capture performance data during execution. The performance data is recorded and used either during the execution or later in post-mortem analysis to determine performance bottlenecks and opportunities for optimization. Examples are such as MATE [MCSML07], where an application is monitored, its performance bottlenecks are detected, and their solutions are given and the application code is modified on the fly to improve its performance. This software is dedicated to PVM applications and aims to change the application code but not the resource environment to guarantee the performance. The Autopilot [RVR+98] project is based on a closed loop and allows parallel applications to be adapted in an automated way. It contains a library of run-time components needed to build an adaptive application. Autopilot provides a set of distributed performance sensors, decision procedures and policy actuators. The developer can decide which sensors/actuators are necessary and then manually insert these in the application source code. The toolkit includes a fuzzy logic engine that accepts performance sensor inputs and selects resource management policies based on observed application behaviour. Paper [VAMR01] introduces the Performance Con-

tract where commitments are specified, and put forward an application signature model for predicting application performance. It describes a monitoring infrastructure that detects when behaviour does not fall within the expected range and find out the bottlenecks.

Our approach is different from theirs in two aspects. 1. We rely on the QoS satisfaction feedback ESD rather than performance data gathered by sensors. The utility-related ESD is more human oriented and also has the flexibility to contain more QoS information such as budget and user preference. 2. The approach does not only focus on improving performance by eliminating the bottlenecks but also assume we can manage resources to recompense the performance bottlenecks. For example, if an application is running slowly due to the overload of one processor, traditional approaches will try to discover the bottleneck processor by various performance measurement approaches and replace this processor with a new one. In contrast, our approach will simply attempt to add a new processor to improve the performance (since the application can adapt to the change). We replace the bad resources only when adding new resources is not effective. This way avoids the complexity introduced by performance analysis such as the Performance Contract [VAMR01].

7.3 Chapter 6 & Chapter 7 Summary

In this chapter and the previous chapter, we have introduced the on-line feedback-based automatic resource configuration (OAC) framework based on the AA's flexible execution model to provide full resource management transparency to the application and users. In this approach, the AA automatically tunes the execution environment on the fly to satisfy users' execution requirements according to application's performance feedback ESD. We introduced two learning and tuning policies to support this approach. We used two experiments (iterative and non-iterative) for each of the policy to demonstrate both policies are sufficiently general to support most types of applications. Using the OAC, even a user without a parallel computing background can easily start and run a distributed application in an unknown resource environment and obtain the required level of performance.

The two OAC polices have different advantages and disadvantages summarized in Table 7.2. The Utility Classification policy provides very high transparency since it does not need any information regarding either application or resources. The DPPE policy on the other hand needs the relative P information, although the information can be just approximate. The high transparency of Utility Classification leads to a relatively slow tuning process because it needs to learn the classification first and in each tuning operation it only deals with one type of resource. In contrast, DPPE can quickly add/release resources to reach the estimated resource level at one time. Although both policies support parallel applications, Utility Classification is the better choice. This is because DPPE has difficulties finding the communication bottleneck and only attempts to add resources to compensate the performance loss, which may not be effective for some execution situations. Moreover, Utility Classification satisfies multiple QoS requirements, while DPPE can only support single execution rate QoS, such as deadline or speed.

A potential limitation of the OAC approach is that the intelligence and automatics is based on sacrificing some time for resource tuning and especially application load balancing. The overhead introduced by load balancing varies according to application types. Normally, adaptive applications that use ob-

	Utility Classification	DPPE
Transparency	High	High
Tuning speed	Relatively slow	Fast
Supporting parallel applications	Yes	Yes
Finding communication bottlenecks	Yes	No
Supporting multiple QoS requirements	Yes	No

Table 7.2: Comparison of two OAC polices

ject graph approach will introduce overhead according to the size of objects that need to be migrated. For example, in the experiment of 2D heat equation, a sharp performance drop can be observed during the tuning process because of the object redistribution. We give another example of such overhead in the lunar temperature mapping experiment in Appendix D. However, the overhead only exists when the application performs reconfiguration and it is always negligible comparing with the total execution time. For the master-worker type applications, there is no such overhead generated, because the master-worker paradigm is instinctively adaptive. For example, reconfiguration overhead is not observed during the tuning process of the Mandelbrot application and the BSP application, both of which are based on the master-worker paradigm.

Chapter 8

Federated Resource Negotiation

In Chapters 4 $\sim$ 7, we have constructed a flexible and transparent standalone distributed computing model where a distributed application is dynamically and automatically provided resources by the AA to ensure the performance QoS requirement. The resource management is transparent to the user, who needs no knowledge about how resources are selected and allocated. Using this model, we can create an autonomous multi-user distributed computing world where each application is taken care of by its own AA, which in turn obtains the resources to run its client application. Figure 8.1 illustrates this notion. Since the resources in the resource pool may be insufficient to support all the applications, the AAs need to compete to get the best facilities for their own applications.

In a traditional grid system, there is a centralized controlling dictator taking over and deciding exactly who gain access to what resource at any one time, for example, the meta-scheduler. The rule can be simply First Come First Served (FCFS) strategies, where all resource requests are considered equally important and applications are executed in the order in which they enter a queue, or, priority-based approaches may be used to advantage specific users depending on various conditions, such as the importance of the request or historical usage. For example, Condor uses job checkpointing/migration to satisfy the high-priority jobs according to user's resource usage history. However, the centralized resource management would involve a lot of rules and administration especially when the grid contains many organizations who have their own polices.

By moving away from the traditional top-down and centric meta-scheduling model, we propose an alternative, more flexible approach called **Federated Resource Negotiation**, which lets the AAs negotiate resource allocations amongst themselves during the execution. Each AA will try its best to acquire resources from the underlying resource environment to satisfy the QoS requirement. The resource provision is simply based on local scheduling rules of a resource environment, normally FCFS. When resources are insufficient due to resource competition from other users, the AA can ask its peers to transfer resources (Figure 8.1). The peer AAs can accept or reject such a request according to the embedded polices. Each AA could have different polices. The whole resource management therefore does not require specific rules imposed by DRMs - it is performed autonomously by the AAs themselves.

In this chapter, we construct a federated resource negotiation framework based on the AA platform and present a pilot study about its usage. This chapter aims to demonstrate a potential extension based

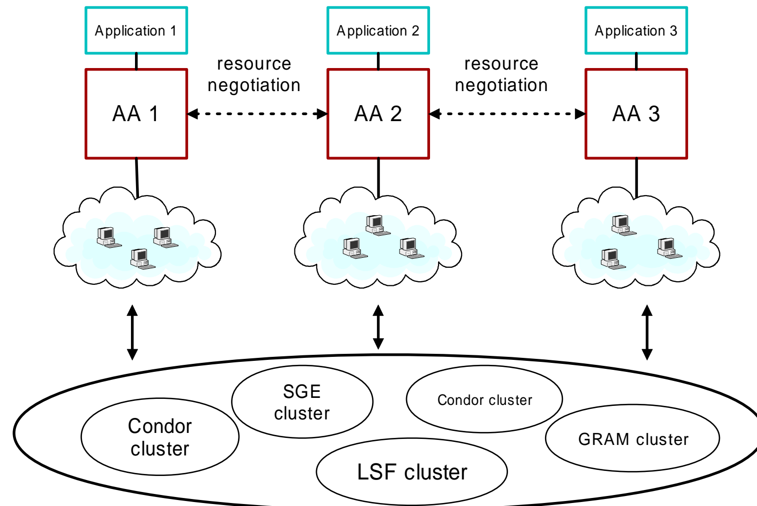


Figure 8.1: A multi-AAs distributed computing world containing 3 applications, 3 AAs, and 3 virtual machines.

Application ID	ART UPID	Gateway Daemon
1	4194306	golding.cs.ucl.ac.uk
91	381681666	leolau.cs.ucl.ac.uk

Table 8.1: An example of the AA-World table

on the flexible and transparent model.

8.1 Framework

As introduced in Chapter 6, the ART (Automatic Resource Tuner) is the AA component that takes charge of the resource management issues for the application. It decides how to configure the execution environment, and asks the RD daemon to contact underlying DRMs to request resources. After sending the request, it waits for the resource permits from the DRMs. When the waiting time of a request exceeds a given boundary[1] due to resource unavailability in the resource environment, the ART will send a resource transferring request to its peer AAs to rent a resource. The request transmission is achieved by contacting the peer AA's ART daemon. As we know, each ART is a user-level process and has a global unique UPID. The UPID of ART and other contact information of each AA can be gained from the *AA-World Table* (Table 8.1), which can be download from the *AA-Global-Server*, a global server that serves all AAs.

The request is actually a cross-VM (virtual machine) communication. In order to deliver the cross-VM communication, each AA must have a *gateway* as an entry point for processes in other virtual machines talking to processes in its own virtual machine. We use the first spawned PMWComm in the virtual machine (normally the one started on the local computer) as the gateway. When one ART talks to another ART, according to the AA-World Table, the requestor ART contacts the receiver's gateway

[1]Default 10 seconds. The resource allocation process in some clusters normally takes $1 \sim 10$ seconds.

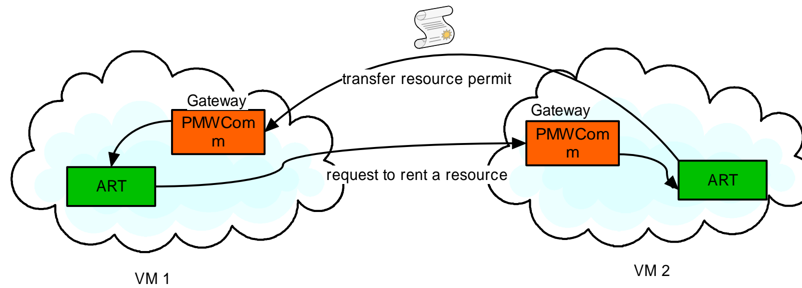


Figure 8.2: The federated resource negotiation is delivered by ARTs' cross-VM communication.

which then routes the request to the receiver ART. Figure 8.2 depicts how an ART contacts another ART to ask for resource transferring.

Once a peer AA receives the request, it needs to decide to accept or reject the request. If it accepts, it will become the resource lessor and the requestor will become the resource lessee. The lessor then transfers the resource permit to the lessee AA: it first releases the application process on that resource, and then passes the resource permit (resource address including cluster-gateway address if it is behind the firewall) to the lessee. Note that it does not return the vacated resource to the resource pool but transfers the right to use the resource directly to the lessee. In the resource pool, the probe job submitted by the lessor still runs to hold that resource. Upon receiving the resource permit, the lessee deploys a process and uses the resource in a normal way. Once the lessee finishes using the resource, it will return the permit back to the lessor which can continue the usage.

8.1.1 Negotiation Policies

The rules guiding how to decide to accept or reject a resource transferring request is a key problem to the approach. Each AA needs to be embedded a policy to make this decision. The policy could involve many factors. For example, we can borrow the free market concept - every resource transferring involves a cost. The requestor must pay to rent the resource. The request will only be accepted if the requestor can afford the cost. If there are multiple requestors, the one that has the highest offered price can win the resource. Another more intelligent policy is that the AA makes the decision based on whether the remaining resources and the resources that could be used in the future are able to fulfill its own QoS requirement. If the AA predicts it can still meet user's requirement even if it lends the resource, it will accept the request. Alternatively, the resource transfer can be set to only happen in the AA peers who trust each other, such as AAs that belong to the same organization. The accept/reject policy can be so complex that it can be a separate research topic. In this thesis we will not investigate advanced policies to negotiate resources. Here we introduce a simple policy that aims to improve the application satisfactions under a competitive resource environment to demonstrate the effectiveness of the federated resource negotiation framework.

8.2 Using the Federated Negotiation to Improve Multiple Application Satisfactions

Under resource competition, certain applications may request specific resources that are currently allocated to other applications, due to the lack of resources for that specific resource requirement. In this scenario, we have a situation in which the request has to be held in the queue until that particular host is being released by the other application. In many DRM systems migration is performed to attempt to solve this problem. Migration is decided according to some priorities. For example, in Condor, the priority is set according to user's resource usage history. This approach could have the fairness of resources allocations in terms of the user, but it has no specific concern for satisfying an application's requirements. Moreover, the centralized approach is not scalable in the grid systems since each domain's DRM could has its own migration policies.

On the other hand, sometimes requests are being held in the queue not because of the lack of resources but because the resources have not been efficiently allocated. For instance, in the Condor pools at UCL, many jobs are submitted with no explicit details of execution QoS requirements, such as deadline. These jobs are normally batch jobs such as parameter sweep applications, and they always need large amounts of computing resources over long periods of time. Users are satisfied as long as such jobs are running but not sensitive to the execution times. The problem is, once those jobs are assigned resources, other jobs which have certain resource requirements may have to be held in the queue until those particular resources are being released. For rigid QoS requirement applications, the delay can significantly affect the execution satisfaction of the user. The rigid QoS applications are normally short-term high performance computing, especially the interactive applications where users require results can be provided in a very short time. In that case, a job that has no or soft QoS requirements should sacrifice execution performance and transfer some of owned resources to the rigid jobs to make as many applications satisfied as possible. To address this situation, we introduce the so-called Sacrifice Policy.

Sacrifice Policy:

1. Any applications that have no or soft QoS requirements should transfer resources to the applications that have rigid QoS requirements once the latter request federated resource negotiations.
2. The lessor application must remain at least one resource to continue the execution.
3. The lessee must return resource permits to the lessor once the rented resources become superfluous.

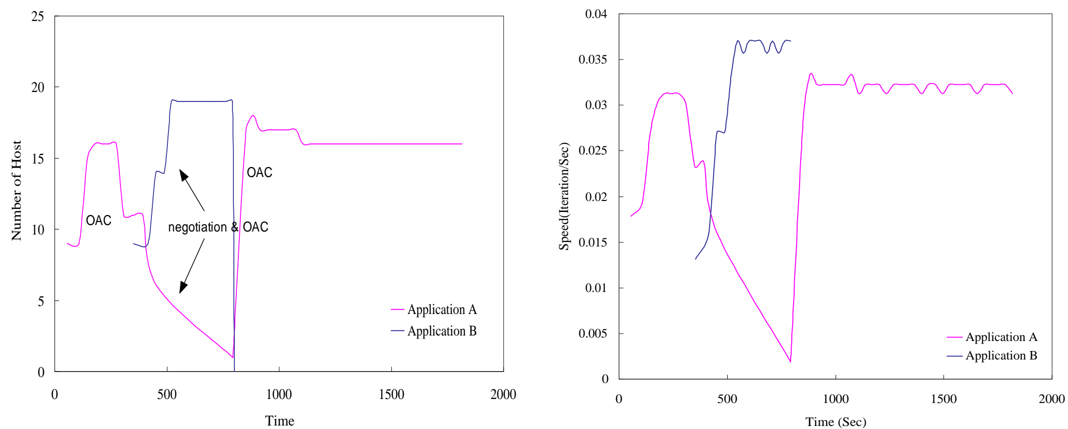
Users can set the soft/rigid parameter by using ESD reporting function $ReportESD_P2P(esd, 0/1)$. 0 means soft; 1 means rigid. The parameter is associated with the current ESD and resource requirements. Users can change the parameter during the execution.

8.2.1 Experiment

We use the resource negotiation between two applications to validate the federated resource negotiation framework and the Sacrifice policy. We built the two applications based on the BSP model introduced in Section 7.1.4. Each application has 100 work objects and each objects takes 3 seconds to run on a Intel(R) Celeron(R) 2.80GHz processor. The first application, called application A, is supposed to run

for very long time and has a soft speed requirement = 0.03 iterations/second, which means it can tolerate slow execution speed. The second application, called application B, is supposed to run a short time (15 iterations) and has a rigid speed requirement = 0.035 iterations/second. The test resource environment is a Condor pool contains limited 20 Intel(R) Celeron(R) 2.80GHz processors. Each application is taken cared by an AA which is configured with the OAC DPPE policy and the Sacrifice policy for resource transferring.

We have a scenario where application A runs first, and after 5 minutes application B starts. Because the system has insufficient resources to satisfy the demands of application B, application A should transfer resources on the fly to satisfy B. Once application B no longer needs the resources, application A can continue to use the returned resources to satisfy its own requirement.



(a) The resource historical usage of application A and B.

(b) The execution speed of application A and B.

Figure 8.3: An experiment of the federated resource negotiation.

As Figure 8.3(a) shows, the federated resource negotiation happened between time 400 $\sim$ 600. Since application A initially occupied 16 hosts to satisfy its own requirement, there were not sufficient resources left to satisfy application B. When the AA of application B (AA-B) started performing the OAC, it realized the resource insufficiency and requested resource transferring from AA-A. According to the Sacrifice policy, AA-A immediately vacated 15 resources and transferred these permits to application B. Note that the resource transferring was performed more than once. This was because AA-B's OAC tuning process lasted 5 iterations. After the transfer, application B had access to 19 hosts and ran at the satisfying speed 0.035 iterations/second, as Figure 8.3(b) shows. While the execution speed of application A gradually dropped to 0.001 iterations/second, because it was supported by only one host. After application B terminated the execution around time 900, AA-B returned the permits back to AA-A. AA-A then immediately performed the OAC using the returned resources to satisfy the speed requirement. As we can see the resource usage of application AA immediately raised to 18 (AA soon re-adjusted the number to 16). The speed of application A was also tuned back to the satisfied performance level.

This simple policy has demonstrated effectiveness to support negotiations among two or a small number of applications. However, when there are a large number of applications, the sacrificed applica-

tion may starve with only one resource supporting - other applications may keep taking advantage of it. In this situation, this policy may need to work with other fair-use or user priority policies.

8.3 Chapter Summary

In this chapter we have introduced the federated resource negotiation framework based on the AA's OAC service. This framework allows the AAs to use resources in a service federated fashion: applications can transfer their own resources to satisfy peer applications when necessary. This approach is similar to the peer-to-peer grids [ABCM07] which is used to build grid infrastructures with a larger number of sites. These grids sacrifice support for complex sharing policies in exchange for ease of deployment, management, use, and growth, using P2P technology. For example, Condor flocking [DLP+96] allows a Condor scheduler to submit allocation requests to another cluster when no resources are available locally. A similar notion has been applied to Internet Peering [Nor00], where Internet service providers (ISP) provide connectivity to each other's clients in order to reduce both cost and latency. Different from them, our federated resource negotiation is achieved at the application-level rather than at the system level. It is therefore more lightweight and scalable. It requires no additional sharing polices set into clusters.

Our research of the federated resource negotiation is preliminary. There are some issues we still need to address in the future, such as sophisticated negotiation policies, user account and resource authority problem. However, we have demonstrated the possibility of achieving autonomous federated resource negotiation based on the AA platform. The multi-application resource negotiation points towards a multi-user, autonomous distributed computing world which is different from the current grid computing notion in that each application is provided resource based on an individual resource management without a central manager such as a meta-scheduler. In this computing world, each AA fights for the best resources for their applications and a state of balance results. It is in constant flux as applications enters and exits.

Chapter 9

Conclusion

The experience of working with the Condor and SGE clusters has motivated us to extend the batch computing model to a more flexible and transparent distributed execution model. The <submit → resource allocate → execution> approach in current grid/cluster computing works well for batch type applications that do not necessarily have specific execution requirements. For other high performance applications with dynamic resource requirements and QoS expectations, this approach is inadequate or needs significant degree of cooperation from users. It is this requirement that has led to the contribution presented in the thesis.

We have offered a flexible execution model, being supported by a novel middleware system - the Application Agent (AA). Based on the fundamental flexible execution services of the AA, we have introduced an automatic resource configuration approach that greatly simplifies the user experience. The standalone, flexible and transparent execution model with the federated resource negotiation framework points towards a multi-user, autonomous distributed computing world. These contributions are unique in that we rely on the existing grid computing technology (DRMs) to aggregate resources but break the rigid execution process provided by it. We have demonstrated that by using the model we can provide improved execution performance, distributed computing transparency and resource utilization.

It should be noted that the primary focus of the thesis has been on the possibility of resource providing for the application in a flexible and transparent way. There are many potential issues that may be associated with the resource provision that are not specifically addressed by the contribution of the thesis, such as software reliability and security. These domains need further exploration in the context of the issues covered here.

9.1 Contributions

Contributions can be recognised in three aspects. They are

- Flexible distributed execution - Flexible execution enables a running application to re-negotiate computational resources assigned to it on demand at run-time, rather than only request or book resources in advance. It is able to adapt to dynamic and unpredictable behaviours of both the application and the resource environment. This is supported by:
  - **Two-layer Model**

- **Application Agent**

- Resource management transparency - This is defined as hiding the distributed nature of the execution of an application from its users so that the application appears and functions as a local desktop application. It is divided into resource access transparency which hides the complexity to request and access distributed resources, and resource selection transparency which hides the complexity to select suitable resources to satisfy performance requirements. This is supported by:
  - Two-layer Model
  - Application Agent
  - **Online Automatic Resource Configuration**
    - **The Utilities Classification Policy**
    - **The DPPE Policy**
- Federated multi-user resource management - Based on the flexibility and transparency of the standalone execution model, resources can be more effectively used among multiple AA-enabled applications in a federated way. This is supported by:
  - Two-layer Model
  - Application Agent
  - Online Automatic Resource Configuration
  - **Federated Resource Negotiation**

9.1.1 The Two-layer Model and the Application Agent

We have introduced a two-layer model (Chapter 4) to provide the flexibility and high transparency into distributed computing. The two layers have explicit division of work. The lower layer AA takes charge of on-demand exclusive execution environment construction; while the upper layer, the application, is responsible for making the best use of the provided execution environment. A significant assumption laid out was that the served applications must be adaptive applications able to autonomously balance their load over allocated resources. This assumption was made according to the discussion on load balancing in Section 3.1.2.

The middleware AA, which is mainly introduced in Chapter 5, is able to cooperate with the existing grid and cluster infrastructure to discover, allocate resources dynamically on behalf of the application. An application can use the AA API to control computational resources at any time during the execution. Whenever there is a resource request from the application, the request will be re-mapped to the AA which will complete the resource management. Users do not have to know how resources are discovered, allocated, and deployed.

In addition to the "dynamic resource allocation for flexible execution", the AA has also provided the "run applications anywhere" for distributed computing transparency and "wide-area computing" to enable applications to be deployed and executed on the resources from multiple clusters/domains. The AA is the fundamental software to support all the contributions.

9.1.2 Online Automatic Resource Configuration and Two Tuning Policies

In Chapters 6 & 7, we completed the two-layer execution model by introducing the online feedback-based automatic resource configuration (OAC) based on the AA's flexible execution framework to provide high resource management transparency to the user. We extended the AA's API and architecture to implement this approach. The approach selects resources according to the performance feedback (ESD) reported by the application. It differs from other automatic resource selection approaches in that the execution environment is configured on the fly and the configuration needs minimum collaboration from the user (ESD reporting). With the introduction of the OAC, a user can run a AA-enabled application without considering the resource management issue which includes resources selection, resource discovery, resource allocation and resource deployment.

In Chapter 6, we introduced the Utility Classification policy to enable the resource to learn and tune under the OAC approach. The policy classifies hosts according to their historical performance contributions to the application. According to the classification, the AA chooses high utility hosts and withdraws low utility hosts in order to configure an optimum execution environment and satisfy the QoS of users. We used two simulated experiments to validate the policy: a 2D heat equation application and an economy-based task farming application.

The DPPE policy for the OAC approach was introduced in Chapter 7. This policy dynamically configures the execution environment with the least resources according to the estimated desired total processing power in order to satisfy user's execution requirements. We used two real-world experiments to validate the policy: a Mandelbrot computation and a BSP model application. Comparing with the first policy, it is more lightweight however has narrower applicable scope. The comparisons were detailed in the summary section of Chapter 7.

9.1.3 Federated Resource Negotiation

With the help of the two-layer execution model - including the flexible execution and the transparent execution, we proposed a federated resource negotiation framework based on AA's services in Chapter 8. This framework allows multiple AAs to work together in a federated way: applications can request resource transfer from peer applications, thereby constructing a multi-user autonomous distributed computing world. We used a specific scenario which is to improve application satisfactions in a competitive environment to validate the framework. Although this chapter only provided a preliminary investigation, it has made a step towards the autonomous distribution computing world which needs much less administration efforts than grid computing.

9.2 Discussion of Adaptive Application Related Work

As we mentioned in several places, our model is based on the assumption that the served applications are adaptive. There is much work performed related to adaptive or malleable applications in recent years. In this section we distinguish our contribution by comparing it with the existing research.

Charm++ [LK01] provides a framework to create adaptive application based on the parallel objects, which is similar to our Distributed Object Graphs paradigm introduced in Chapter 4. Programs written in

Charm++ are decomposed into a number of cooperating message-driven objects called chares. When a programmer invokes a method on an object, the Charm++ runtime system sends a message to the invoked object, which may reside on the local processor or on a remote processor in a parallel computation. The chares in a program are mapped to physical processors by an adaptive runtime system. The mapping of chares to processors is transparent to the programmer, and this transparency permits the runtime system to dynamically change the assignment of chares to processors during program execution to support capabilities such as measurement-based load balancing, fault tolerance, automatic checkpointing, and the ability to shrink and expand the set of processors used by a parallel program. Adaptive MPI (AMPI) [BBK+00] is an implementation of the Message Passing Interface standard on top of the Charm++ runtime system and provides the capabilities of Charm++ in a more traditional MPI programming model. AMPI encapsulates each MPI process within a user-level migratable thread implemented as a Charm++ object.

Amber [FBCL91] is another object-oriented parallel-distributed programming framework supporting application reconfiguration to adapt to the change of resource environments. Applications written by Amber are decomposed into objects and threads distributed among the logical nodes distributed across a set of physical nodes. A running parallel-distributed program can respond to three types of node reconfiguration: the node set can shrink in size, stay the same size but change member-ship, or grow. The runtime system of Amber provides the mechanism for object virtual addressing and migrating objects, while the application takes charge of object load balancing polices to resolve load imbalances caused by growth and shrinkage of the node set.

Barker et al. [BCCP04] introduces a framework and runtime software system PREMA for supporting the development of adaptive applications on distributed-memory parallel computers. The runtime system supports a global namespace, transparent object migration, automatic message forwarding and routing, and automatic load balancing. These features can be used at the discretion of the application developer in order to simplify program development and to eliminate complex bookkeeping associated with mobile data objects.

Desell et al. [DMV07] introduces a new type of application reconfiguration, called malleability, which dynamically change the granularity and data distribution to adapt to dynamic changes in resource availability. The data redistribution however is based on process checkpoiting/migration rather than object migration. The change of granularity is implemented by splitting or merging processes. The malleability has been implemented in Internet Operation System (IOS) [MDSV06], a modular middleware for reconfiguration of distributed applications.

Systems such as VDS [Dec00] provide support for automatic dynamic load balancing through the dispersion and migration of computation threads. VDS is a general load balancing library for irregular applications. The library shields the programmer from the need to implement functions that do not strictly concern the application. Load balancing, termination detection, and data management are done automatically. VDS supports various parallel programming paradigms such as parallel independent tasks, fully strict multi-threaded computations, and directed acyclic task graphs. Moreover, VDS

	Adaptive Application Development				Flexible/Transparent RM		
	OO Framework	Load Balancing	Process Migration	Message Passing	Resource Selection	Resource Allocation	Process Deployment
Charm++	•	•		•			
Amber	•						
PREMA	•	•		•			
IOS		•	•	•			
VDS		•	•				
AA				•	•	•	•

Table 9.1: Software related to adaptive applications.

allows the concurrent use of different paradigms in the same application.

Our system AA differentiates greatly from the above work in that we focus on the **resource management for flexible execution** based on adaptive applications, which none of the existing frameworks has considered. Table 9.1 summaries the differences. Firstly, the purpose of introducing adaptive applications in their work is mainly fault-tolerance for dynamic resource environments. While in our research, the motivation of using adaptive applications is to construct a flexible execution model that is able to eliminate the disadvantage of the static submission approach in grid computing. Secondly, their work concentrates on how to ease users to develop adaptive applications, including the parallel objects framework, object load balancing, or process checkpointing and migration. They assume that the computational resources are dedicated but do not consider integrating their work into common grid computing resource environments. While our work completes them by introducing a resource management system residing between adaptive applications and resource infrastructures to automatically select, discover and allocate resources to run the applications, in order to exploit the benefit of the flexible execution. The work is important since the trend in distributed computing is to build a global resource clouds composed of many geographically distributed resources. To run an application, users need to obtain resources from the public resource clouds managed by various DRMs rather than use a local dedicated resource pool.

9.3 Future Directions

We have proposed in this thesis a number of complementary approaches, brought together to form a flexible and transparent distributed computing framework. All of these are worthy of further exploration either independently or as an integrated set.

9.3.1 Development of the Application Agent

We have developed the architecture of the AA and provided a C++ implementation. To make it more practical, further development will include fault recovery, job description standardization, resource buffer service, application development framework cooperation, security, and Java implementation.

9.3.1.1 Fault Recovery

Currently the AA is implemented with a basic fault recovery mechanism. If a process exits or crashes, the local LComm will send a process crash notification (based on PVM PvmTaskExit notification) to its PMWComm. The PMWComm will notify the master that this process has been killed, and then spawn a new process on that host. If a host or a LComm fails, the LComm-LComm or PMWComm-LComm will time out. The PMWComm will discard that host, and notify the master that the associated process has been killed. Sometimes the failure is because of communication lost but not the host having actually failed. In this case, if a LComm loses its PMWComm, the LComm will shut itself down. This setup ensures that the virtual machine does not become partitioned and run as two partial machines. A PMWComm daemon can spawn a backup PMWComm on another frontend machine in case that this PMComm fails. The backup synchronises information with the original. Once the original fails (detected by communication timeout), the backup will immediately take over the job. Similar setup can be applied to the RD daemon.

The current fault recovery mechanism only deals with process/daemon recovery. Any data/message loss due to the failure has to be taken cared by the application. For example, the application may consider re-computing one or more iterations. In the future, we intend to evaluate the recovery system with large-scale applications, and introduce more sophisticated fault tolerant algorithms.

9.3.1.2 Job Description Standardization

In some situations users may need to tell the AA the resource requirements via *Name_of_executable.xml* to run applications. In the thesis we provided a simple requirement definition method based on XML (List 5.1). In future development we intend to replace this method by using the grid job description standard - Job Submission Description Language (JSDL) [AA05], which many users are already familiar with. The AA will borrow the Resource Elements of JSDL to let users define the resource requirements. An example is shown in List 9.1. A JSDL specified resource requirement will be converted by the AA into various DRM recognisable submission languages.

Listing 9.1: JSDL based resource requirement

```
<jsdl:JobDefinition xmlns="http://www.example.org/"
    xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
    xmlns:jsdl-posix="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <jsdl:JobDescription>
        <jsdl:Resources>
            <jsdl:CPUArchitecture>
                <jsdl:CPUArchitectureName>x86_32</jsdl:CPUArchitectureName>
            </jsdl:CPUArchitecture>
        </jsdl:Resources>
    </jsdl:JobDescription>
</jsdl:JobDefinition>
```

9.3.1.3 Resource Buffer Service

The Resource Buffer (RB) is a plug-in design in the AA. It was initially designed dedicated to an application that cannot tolerate resource allocation delays. This service can be further developed and expanded as a standard service of the AA.

In non-dedicated distributed computing environments, the amount of time it takes until a host is allocated is sometime not bound. This is either because of the resource competition of other applications or the overload of the resource submitter machines. In order to maintain the smooth execution of the application which has dynamic resource requirement during the execution, the RB stores a number of granted hosts that can be returned to the application immediately. We have chosen a simple periodic policy to manage the RB in this case. The policy periodically reviews the RB to ensure that the RB is kept to a predetermined host amount RL at each predefined interval. For every time interval, the number of N host requests is:

$$N = \begin{cases} +(RL - R) & : \quad R < RL \\ -(RL - R) & : \quad R > RL \end{cases}$$

where RL is the request (threshold) level and R is the current number of granted hosts in the RB. "+" means that the AA requests to obtain N new hosts; and "-" means that the AA requests to release N granted hosts. An evaluation of this policy is provided in [LNS08].

The RB can also be useful in the following circumstances:

1. When a request comes, the AA may contact a number of DRMs to request a resource allocation. In this situation, more than one granted hosts would be returned by DRMs for this request. Rather than rejecting redundant granted hosts, the RB keeps them for later possible requests from the application.

2. After a process is released, the idle host will be returned into the RB for later possible requests from the application.

3. Fault-tolerant: The RB keeps several backup hosts in case of some hosts in execution fail.

In future work, more policies will be introduced according to different circumstances. Users can choose certain policy before they run an application. For example, for the first circumstance, one policy is to keep all the redundant hosts until the AA detects that the application does not request to add processes for a long time. For the third circumstance, the RB can calculate the number of backup hosts according to the reliability of the resource environment.

9.3.1.4 Cooperation with Other Application Development Frameworks

Future work will enable the AA to be compatible with other application development frameworks to further help programmers to develop an adaptive application. The adaptive application development frameworks such as Charm++ and PREMA allow users to easily develop adaptive applications based on their object-oriented frameworks and migration services. Future work will enable the Charm++ or PREMA based applications to run on the AA platform thus they can be benefited from the flexible and transparent execution services provided by the AA. Future work will also allow load balancing frameworks similar

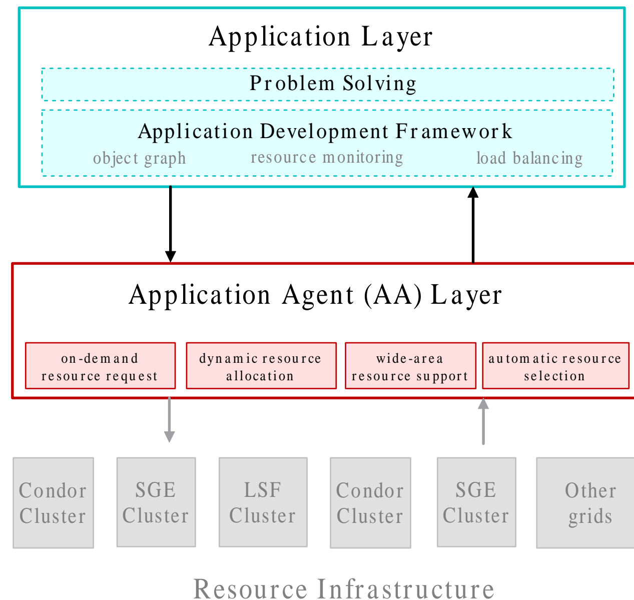


Figure 9.1: The two-layer model in the future.

to DLB [PYE+04] and work [JH97] to be plugged into the AA platform. These frameworks will be able to monitor the resources (CPU load, network delay, etc) in the exclusive execution environment configured by the AA and perform simple load balancing for the application. This work will need to involve implementation modifications of both the AA and the frameworks.

The two-layer model introduced in Chapter 4 therefore can be extended as Figure 9.1.

9.3.1.5 Security

We intend to provide a GUI for user registration and authentication. Both registration and authentication will be accomplished by connecting the *AA-Global-Server* which stores all the user information. The authentication determines user's authority to use certain sets of resources. According to the authority, the AA creates virtual machines distributed across a number of authorized resource clusters managed by local DRMs. The AA virtual machine is recognised by local clusters as a single virtual user called "AA". This user is the only entity known to participating clusters and each host must recognise this user. The "AA" user can be further assigned a separate queue by local DRMs but it can also be treated as a general user and forced to compete with local resource demand. By restricting host access for "AA" locally the cluster manager is also able to control how many hosts are to be shared.

9.3.1.6 Java Implementation

Currently the AA is implemented in C++ for Unix machines. In order to connect a heterogeneous collection of Unix, Windows and Mac computers to compose a single large virtual machine, the AA has to be compatible with all the possible computer architectures. Moreover, since the AA has the "run anywhere" service, a more ambitious prospect is to run the AA-enabled application on any devices, such as PDA or smart phones. The Java programming language allows us to achieve these goals. The Java based AA will be able to run on any Java virtual machine (JVM) regardless of computer architecture.

The Java Native Interface (JNI) technology [SM] provides the possibility for the AA to communicate with underlying C/C++ libraries such as PVM and MPI.

9.3.2 OAC Investigation

With respect to the online feedback based automatic configuration (OAC) approach, we believe there is much potential for more complex policies to be implemented by cooperating machine learning. One possible extension of the Utility Classification policy is that after learning the application's resource preference we can use the pattern classification technology to decide whether a new resource belongs among the "good" resources.

The investigation of ESD can be another future direction. In this thesis we use ESD functions to combine multiple QoS requirements into a single value. In the future, we intend to investigate the possibility of using multiple ESD values, each of which will represent a single QoS requirement. Multiple ART agents will be used to take care of each ESD report. For example, there is a ART to take care of the deadline requirement and another one to manage the budge requirement. We will compare the two ESD management approaches and investigate whether the later one is better by introducing more software complexity.

As we mentioned, one limitation of the OAC is that it is not suited to the applications which cannot tolerate the execution environment reconfiguration overhead as well as application adaption overhead. Future work will give a detailed overhead evaluation for different types of applications and provide a guidance of which situations are more suitable to use the OAC approach.

9.3.3 Autonomous Distributed Computing

In this thesis we have introduced the federated resource negotiation framework to support multiple AAs working together. We have made a step forward towards an autonomous distributed computing world where each AA takes care of its own application and the resources are autonomously distributed among AAs/applications according to the application's QoS requirements and the resource negotiation policies embedded in AAs. Further investigation will include the federated negotiation policies such as the proposals we have made in Section 8.1.1, and validating the framework on large scale grid systems.

Appendix A

Condor and SGE Systems

A.1 Condor

Condor is a batch scheduling DRM which was first developed in 1988 at the Computer Science Department, University of Wisconsin-Madison. It targets at harvesting unused computational cycles from a set of user workstations. Condor provides scheduling with different policies and prioritization, resource monitoring and job management. Resource owners maintain full control of their hardware and can set policies on their acceptable use. Condor uses a proprietary ClassAd language [Ram00], which allows hardware owners to describe their resources, and users to specify resource requests. A matchmaker component compares them to match job requirements with appropriate execution hardware. Currently the UCL Condor Pool contains about 1400 Windows PCs, and the UCL CS Condor pool has about 60 lab machines running Linux.

A.1.1 Run Applications in Condor

A job is submitted to Condor using the *condor_submit* command. *condor_submit* takes as an argument the name of a file called submit description file. This file contains items such as the name of the executable to run, the initial working directory, command-line arguments to the program, and resource specifications. *condor_submit* creates a job ClassAd based upon the information, and Condor works toward running the job. Condor plays the role of a matchmaker by continuously reading all the job ClassAds and all the machine ClassAds, matching and ranking job ads with machine ads. Condor makes certain that all requirements in both ClassAds are satisfied.

For example, Listing A.1 specifies 150 batch tasks called foo which has been compiled and linked using *condor_compile* specifically for Silicon Graphics workstations running IRIX 6.5. This job requires Condor to run the program on machines which have greater than 32 Megabytes of physical memory, with the specified platform. It expresses a preference to run the program on machines with more than 64 Megabytes of physical memory, if such machines are available. The *Image_size* command advises Condor that the program will use up to 28 megabytes of memory when running. Since these tasks are independent with each other, Condor distributes them one by one as long as available resources exist.

Listing A.1: Condor simple job submission sample

```
Executable      = foo
Requirements    = Memory >= 32 && OpSys == "IRIX65" && Arch =="SGI"
Rank        = Memory >= 64
Image_Size      = 28 Meg

Error    = err.$(Process)
Input    = in.$(Process)
Output   = out.$(Process)
Log      = foo.log

Queue 150
```

Condor's parallel universe supports parallel jobs which need to be co-scheduled. A co-scheduled job has more than one process that must be running at the same time on different machines to work correctly. Condor must be configured such that resources (machines) running parallel jobs are *dedicated*. Listing A.2 specifies the universe as parallel, letting Condor know that dedicated resources are required. The *machine_count* command identifies the number of machines required by the job. When submitted, the dedicated scheduler allocates eight machines with the same architecture and operating system as the submit machine. It waits until all eight machines are available before starting the job. When all the machines are ready, it invokes the /bin/foo command on all eight machines more or less simultaneously.

Listing A.2: Condor parallel job submission sample

```
universe = parallel
executable = /bin/foo
machine_count = 8
queue
```

Listing A.3: Condor-PVM job submission sample

```
universe = PVM
# The executable of the master PVM program is "master.exe".
executable = master.exe
Requirements = (Arch == "INTEL") && (OpSys == "LINUX")
machine_count = 2..4
queue
```

Condor has a special parallel environment (Condor-PVM) to support dynamic PVM applications. In Condor-PVM [Con, GKLY00], when a PVM program asks for nodes (machines), the request is re-mapped to Condor. Condor then finds a machine in the Condor pool via the usual mechanisms, and adds it to the PVM virtual machine. If a machine needs to leave the pool, the PVM program is notified of that as well via the normal PVM mechanisms. Listing A.3 shows a example submitted file for Condor-PVM. By using machine_count = $< min >$ .. $< max >$, the submit file tells Condor that before the PVM master is started, there should be at least $< min >$ number of machines of the current class. It also asks Condor to give it as many as $< max >$ machines. During the execution of the program, the application may request more machines of each of the class by calling pvm_addhosts() with a string

specifying the machine class. Condor provides support to PVM applications only based on the master-worker programming paradigm and is limited to single domain computing.

Condor also provides a Computing On Demand (COD) environment to support interactive, compute-intensive jobs. It allows the interactive jobs to come in with high priority and run instead of the batch job on any given resources. A user must be granted authorization to a create COD claims on a specific machine. In addition, when the user uses these COD claims, the application binary or script they wish to run (and any input data) must be pre-staged on the machine. Therefore, a user cannot simply request a COD claim at random. The user specifies the resource on which to make a COD claim by *condor_cod_request*. If an application needs multiple resources, users must make multiple claims. Once the claim is made successfully, a claim ID is returned to address this claim. The next step is to spawn a COD job using the claim. The way to do this is to activate the claim, using the *condor_cod_activate* command with the claim ID. Once a COD application is active on a COD claim, the COD claim will move into the Running state, and any batch Condor job on the same resource will be suspended.

The setup of a COD execution is complicated. Moreover, since the resources still need to be claimed in advance, it is difficult to support interactive applications that have different resource requirements in different execution stages, with the steering of users.

A.2 Sun Grid Engine

Sun Grid Engine (SGE), earlier known as CODINE (COmputing in DIstributed Networked Environments) or GRD (Global Resource Director) is an open source batch scheduling DRM system, supported by Sun Microsystems. SGE is built on an agent master-slave model in which one server computer (master) controls other clients (slaves). Master node (Master Host) serves as the only portal to the system and offers a command line as well as graphical user interface to the cluster. Slave nodes (Execution Host) are running an agent responsible for executing jobs, monitoring their status and report it back to master hosts. Besides, they report to master host the resource information about underlying hardware, software and execution environment from which complex resource selection can be made. Each execution host in the cluster represents one queue and each CPU on that host is one slot. Basically each CPU will only be assigned at most one job regardless of its utilization. Parallel processing by PVM or MPI can be executed, with one job spanning across a number of nodes. The queue scheduling is based on first-come-first-serve (FCFS) model with customized policies and priorities. The UCL CS HPC cluster managed by SGE currently contains 485 Linux machines.

A.2.1 Run Applications in SGE

A job submit to SGE is described by a shell script, which allows all options and the program file to placed in a single file. The shell script can be written using the C shell, the BASH shell, or other shells. Users submit the shell by *qsub* or a graphic interface QMON. Users can specify resource requirements when submitting the job. For example, Listing A.4 requires a host with solaris64 architecture and 750M memory to run a job specified in permas.sh.

SGE also supports parallel job submission by configuring PVM or MPI environment. Similar to

Condor, it requires users to specify the number of required resources in advance and only starts the application after all the resources has been allocated.

SGE supports resource advance reservation, which allows resources to be reserved for a specified user, administrator, or job. For example, Listing A.5 reserves an slot in the all.q queue on host1 or host2 or host3. The reservation is supposed to start at 12:00 on 1st of December and lasts for 1 hour.

Listing A.4: SGE submission sample

```
% qsub -l arch=solaris64,h_vmem=750M,permas=1 permas.sh
```

Listing A.5: SGE reservation sample

```
% qrsub -q "*@host1,*@host2,*@host3" -u leolau -a 01121200 -d 1:0:0
```

Appendix B

Application Agent (AA) User Manual

B.1 Feature

- C++ & Linux: This library is to incorporate into C++ applications running on Linux-version OS.

- Dynamic resource allocation: Resources can be added / released on-the-fly. Computing scale can expand or shrink on-demand during the execution.

- Transparent resource configuration: Remote distributed resources are automatically selected and configured to run the application in order to satisfy performance QoS requirements (e.g. deadline, budget, execution speed, etc), without resource management from the application or the user.

- Run applications anywhere: An AA-enabled Application can be started on any Internet-connected machines. The AA will automatically find the path to connect resources and place proxies on the key machines to make application processes communicable.

- Wide-area computing: The AA supports wide-area meta-computing, i.e. an application can be deployed and executed on the resources from multiple clusters (geographically or administratively separated, behind firewalls, NAT).

B.2 Installation

1. You must install PVM beforehand, and setup related PVM environment variables.
2. Unzip AA.tar.gz.
3. Go to the AA directory, normally $Home/AA.
$ make

After installation, three AA daemon binaries will be installed into the $Home/pvm3/bin/$PVM_ARCH directory. An AA console binary (aa) will be installed under the AA directory.

In addition, a demonstration application's binary files including a master and a slave will be also installed. The master binary will be placed under the AA directory. The slave binary will be placed into the $Home/pvm3/bin/$PVM_ARCH directory.

B.3 Run

1. Domain Tree (DT) configuration.

Add your local computer where you will start the application into the DT. For example, if your local computer has a public IP 80.88.192.115 which belongs to the PUBLIC domain, you can use the console command: add_node [parent] [address] to add it to the DT:

```
aa> add_node PUBLIC 80.88.192.115
done!
aa> DT
0. PUBLIC
    1. newgate.cs.ucl.ac.uk
        2. condor.cs.ucl.ac.uk
        2. morecambe.cs.ucl.ac.uk
    1. 80.88.192.115
```

2. Resource environment configuration.

Two default resource pools have been configured into the demonstration - a Condor cluster and a SGE cluster, both of which are from UCL Computer Science department. Users can add more clusters via the AA console. Moreover, the AA allows users to add authorized dedicated hosts to run an application. Adding dedicated hosts is also done via the console.

3. Run the demonstration application.

If you want to use the default clusters to run the application, you need to register first - generate your ssh public key using ssh-keygen and email the public key to h.liu@cs.ucl.ac.uk. Alternatively, you can use your own authorized clusters.

A sample output is shown below:

```
$./Master 30
HOST=80.88.192.115
PVM_ROOT=/usr/share/pvm3
HOSTTYPE=i386-linux
USER=haoliu
PVM_ARCH=LINUX
AAROOT=/home/haoliu/AA/
Application ID: 1
parent PUBLIC
Starting local PWMComm on 80.88.192.115...Done
Starting RD.......Done.

Connected clusters (represented by frontend):
local: 80.88.192.115
remote: condor.cs.ucl.ac.uk
remote: morecambe.cs.ucl.ac.uk

********************
*                  *
* Application Agent *
*                  *
```

```
********************
Process 4259842 added !
Send to 4259842
Process 4259843 added !
Process 4259844 added !
completed 1.35%
...
```

B.4 AA Console

The AA provides a console for users to configure and monitor an AA virtual machine.

```
aa> help
commands:
DT               View the Domain Tree
add_node         Add a node into the Domain Tree
delete_node      Delete a node from the Domain Tree
clusters         View current registered resource clusters
add_cluster      Add a cluster into the resource environment
delete_cluster   Delete a cluster from the resource environment
dhosts           View dedicated hosts
add_dhost        Add a dedicated host
delete_dhost     Delete a dedicated host
vm               View the hosts being used in current VM
quit             Quit the console
```

```
aa> clusters
Name              Type     Submit_Machine           Frontend
<UCL-CS Condor>   Condor   condor.cs.ucl.ac.uk      condor.cs.ucl.ac.uk
<UCL-CS HPC>      SGE      morecambe.cs.ucl.ac.uk   morecambe.cs.ucl.ac.uk
```

```
aa> dhosts
Name                     Frontend
golding.cs.ucl.ac.uk     condor.cs.ucl.ac.uk
hardy-1-18               morecambe.cs.ucl.ac.uk
```

```
aa> vm
Virtual Machine of Application[1]:
Domain  Type     Hostname               UPID
1       Condor   kerouac.cs.ucl.ac.uk   4259842
1       Condor   golding.cs.ucl.ac.uk   4259843
1       Condor   eco.cs.ucl.ac.uk       4259844
2       SGE      graucho-3-38.local     4325377
2       SGE      graucho-3-37.local     4325378
2       SGE      graucho-3-37.local     4325379
```

If no AA-enabled application is running:

```
aa> vm
No application is running!
```

B.5 Main API

AA(char arge, bool ifART)**

Constructor for master/control process. **arge**: the standard UNIX environment parameters. **ifART**: To turn on the Automatic Resource Tuner (ART) service. 0: No, 1: Yes.

AA(char arge)**

Constructor for computation/worker/slave processes. **arge**: the standard UNIX environment parameters.

void AddProcess(char* pn)

The unblocked routine requests to add a process named **pn**.

void KillProcess(int upid)

The unblocked routine requests to kill a process **upid**.

int GetNotification(int tag)

The routine checks if the notification with label **tag** has arrived. If a matched notification message has arrived it immediately places the message into a new active receive buffer, and returns 1. Programmers can unpack the receive buffer to get the targeted process UPID.
Three tags:
PROCESS_ADDED: a process has been added.
PROCESS_KILLED: a process has been killed
PROCESS_KILLING: the *local* process needs to be vacated and prepares to be killed.

void DeployProcess(char* pn)

Automatic resource configuration: Tell the AA which processes to be added/deleted. The AA will automatically decide to add a number of **pn** processes on selected resources or delete **pn** processes. It is different AddProcess(), which only adds one process at a time.

void ReportESD(float value)

Automatic resource configuration: The routine reports the ESD value to the AA.

void ReportESD_P2P(float value, 0/1)

Automatic resource configuration with the federated resource negotiation: The routine reports the ESD value to the AA. 0 means the QoS requirement is soft; 1 means the QoS requirement rigid.

void ReadyforKilling(int upid)

Automatic resource configuration: The routine notifies the AA the process (caller) **upid** is ready to be killed.

void StopAA()

Turn off all the daemons, turn off the AA virtual machine and release all the allocated resources.

void NewSendBuffer()

The routine clears the send buffer and prepares it for packing a new message.

void PkByte(int l, char* s)

Pack the active message buffer with a string **s** with length **l**.

void PkDouble(double)

Pack the active message buffer with a double data.

void PkFloat(float)

Pack the active message buffer with a float data.

void PkInteger(int)

Pack the active message buffer with a integer data.

void Send(int upid)

Send the data in the active message buffer to the process **upid**.

void ReceiveFrom(int upid)

The routine blocks the process until a message has arrived from process **upid**.

int NReceiveFrom(int upid)

The routine checks to see if a message has arrived from process **upid** and also clears the current receive buffer if any. If a matching message has arrived it immediately places the message in a new active receive buffer, and returns 1.

int TimedReceiveFrom(int upid, int32_t s, int32_t m)

The routine blocks the process until a message has arrived from tid. If no matching message arrives within the specified waiting time, it returns 0, otherwise returns 1. **s**: second; **m**: minute.

Below routines unpack messages from the receive buffer, referring to corresponding PK* routines.

char* UpkByte()

double UpkDouble()

float UpkFloat()

int UpkInteger()

int WhoAmI()

Returns the upid of the calling process.

int GetParent()

Returns the upid of the process that spawned the calling process. If no parent return -1.

Appendix C

Simulating the 2D Heat Equation Experiment

We have chosen the Java thread and the sleep() function to simulate the 2D heat equation experiment. Figure C.1 shows the abstract simulation framework. We used 7 Java classes to simulate a heterogenous resource pool, the AA, and the application which is composed by a master and a number of worker processes.

HostType class: This class holds the information of a host type. The information includes processing capability (P), location (L) and cost.

Resource_Pool class: This class generates a resource pool (RP) which contains a number of different host types created based on the HostType class. We generate 5 types according to Table 6.1.

AA class: The thread class simulates the AA. It generates an execution environment (EE) to run the application. The EE is an array of HostType objects. The EE is generated by adding/deleting/replacing hosts from the EE in the previous iteration. The actions are made according to the Utility Classification policy. This class has a function called reportESD(), which is called externally by a master thread to report the ESD values.

Work_Element class: This class holds the information of a data grid cell. The information includes the grid cell's index, the cell's current interaction number, and the iteration numbers of the cell's four neighbour cells. It also contains a process ID, which is the ID of the worker thread that processes the cell.

Master class: The thread class simulates the master process of the application. It holds a "dataGrid" 2D array which is created based on the Work_Element class. The array stores the updated information of all the 1000*1000 cells. This class has a function called reconfigure() which is called externally by an AA class instance to receive a new EE configuration.

Worker class: The thread class simulates the execution of a worker process. It holds a local "dataGrid" array (a part of the entire dataGrid), which stores the data distributed by the master. It also holds the information of the host where it "stays". The host information includes processing capability P and location L. It uses the sleep() function to simulate the processing time. For example, computing 10UD data is simulated by sleep(10/P * UT), where UT = 1000, meaning that for a unit time (UT) the simulation sleeps for 1000 milliseconds.

Send class: The thread class simulates the message passing delay by sleep() function. For example,

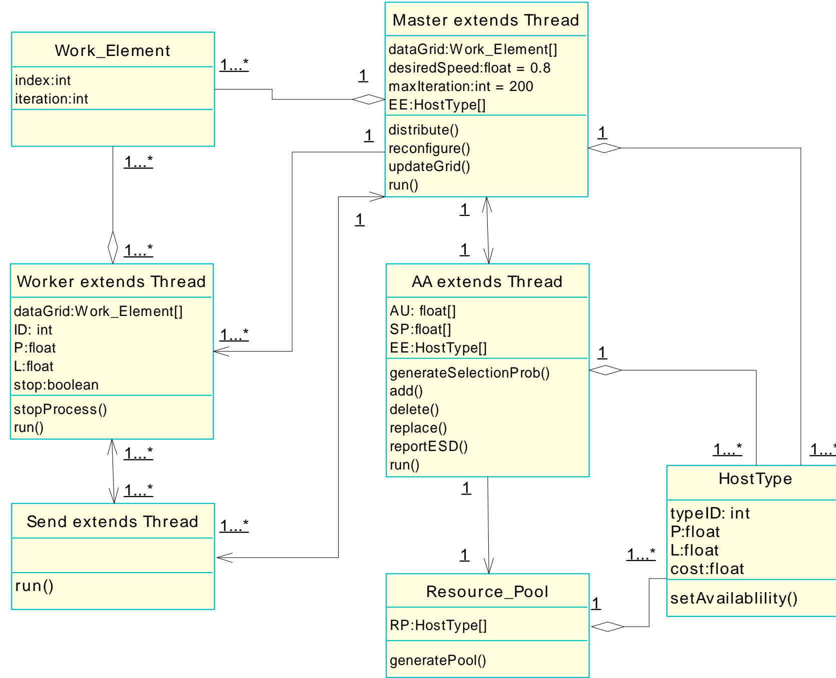


Figure C.1: Object-oriented 2D heat equation experiment simulation framework

to send 10UD data from worker A to worker B with a 0.1 unit delay, it will sleep(10*0.1*UT). The class delays 10*0.1*UT milliseconds then calls the related function of worker B to pass the data.

C.1 Simulation

A master thread and an AA thread are created in the first place. The AA thread generates a Resource_Pool instance. The AA thread then chooses hosts from the Resource_Pool according to the Utility Classification policy. After the selection, it generates an execution environment (EE) represented by an array and passes it via reconfigure() function to the master thread. Upon receiving the new EE, the master stops all the current worker threads by invoking the stopProcess() functions of the worker threads. It then starts a number of new worker threads according to the new EE and distributes data to them by distribute() function according to algorithm 7. A new worker thread receives related host information and data from the master when it is started. After processing a cell, each worker thread needs to pass the result to the cell's four neighbour cells. If the neighbour cells stay on the same worker thread, the updating is performed locally. Otherwise the worker starts a Send thread to pass the result to other worker threads which hold the neighbour cells. After updating an iteration, each worker thread sends all the local results back to the master, which updates the central dataGrid.

Once updating an iteration, the master calculates the ESD and reports it to the AA thread by invoking the reportESD() function. Upon receiving the value the AA thread enters into a new tuning turn.

Appendix D

Lunar Surface Temperature Mapping

This appendix gives an overall evaluation of the primary contributions (the flexible execution and the distributed computing transparency) introduced in the thesis. We used the Lunar Surface Temperature Mapping experiment for the evaluation because it was the project which initially motivated us to produce this thesis. Referring back to the motivation chapter, our expectation was that we could run the temperature visualization from our desktop without worrying about underlying distributed resource management. We could dynamically change the resolution, which may cause significant change of resource requirements. We wish we could continuously view smooth image output with minimal disruption even if the resolution has been changed. To us (the users), the distributed application should be not different from a desktop application. In the following, we will prove that we have made this dream finally come true.

D.1 A Little Experiment Background

Although the Moon is more than two hundred thousand miles away it affects the earth in a myriad of ways. In recent times, with the dawn of the Space Age, some of the original mysticism surrounding the moon has been dispelled by Armstrong's and Aldrin's 1969 landing of the moon. Despite this, human has been always dreaming to someday colonise the moon, terraforming it to a habitable and appealing alternative to earth. Colonising the moon is currently not possible since water is needed for life and survival. The most viable solution, seems not to be bringing litres upon litres of water from earth, costing an exorbitant amount in terms of logistic fees, but rather, to find water within the moon, extracting it, and purifying it for consumption, within a reasonable budget and timeframe. Finding water in the moon is the first step, and that is the step that this project deals with. Given accurate data of the lunar surface, and precise data concerning the path and energy of the sun, we can calculate a map of the predicted temperature of the lunar terrain. Comparing this with actual data sampled from the moon, any discrepancies will need to be analysed, noting that a body of water beneath the surface may manifest itself as one of those discrepancies. It is the wish of this project to bring man that much closer to inhabiting the moon. [STB$^+$07]

D.2 Mathematical Algorithms

The lunar surface is normally represented as interlocking triangular facets in 3D space. Each consisting of its own set of sub-facets that shall be used to mode the heat conduction through a certain specified

depth. In this experiment we choose a 2D space to simplify the mathematical computation.

D.2.1 Surface Temperature

As a first approximation, consider the surface of a smooth Moon illuminated by the Sun which is a constant 1.495985 1011 m (1 astronomical unit) away. A single facet situated at latitude 0° and longitude 90° will see the Sun rising on the eastern horizon (longitude 0°) at t=0, passing directly through zenith 5.9014 105 sec (6.83035 days) later and setting on the western horizon (longitude 180°) after 1.180305×10^6 sec (13.6607 days). The luminosity of the Sun is 3.826×10^{26} W and the energy impacting on the lunar surface when the Sun is in Zenith is therefore 1.34×10^3 W m^{-2}. Because we have assumed that the Moon that is smooth as a billiard ball, reflections and blackbody radiation emitted by a facet are radiated straight back into space.

The temperature on the surface is determined by the balance between the energy absorbed from the impacting sunlight, the energy lost through blackbody radiation and the energy conducted into the subsoil. The surface temperature is thus given by:

$$T_S^{n+1} = T_S^n + \frac{\delta t}{\rho C_v} Q_S$$

where Q_S is the energy flux across a unit area of the surface:

$$Q_S = \frac{L_o}{4\Pi r^2}(1 - A_b) sin^+(\Psi) - J_0 - \varepsilon_{IR}\sigma_B T^4$$

Here L_o is the solar luminosity (3.90 1026 W), A_b is the bolometric albedo (here 0.1), ε_{IR} is the infrared emissivity (here 0.98) and σ_B is the Stefan-Boltzman constant (5.669 10^{-8} $W\ m^{-2}\ K^{-4}$). The effective area is proportional to $sin^+(\Psi)$ where Ψ is the Sun's angle with the facet horizon; contributing only when the Sun is above the horizon. The term J_0 is a constant ($\sim$2 $W\ m^{-2}$) originating from internal energy sources. Under constant illumination, the surface should reach a steady state with the surface temperature

$T = \sqrt[4]{\frac{\frac{L_o}{4\Pi r^2}(1 - A_b) sin^+(\Psi) - J_0}{\varepsilon_{IR}\sigma_B}} \approx 387K$, which is the maximum temperature at the lunar equator at noon.

D.2.2 Heat Conduction

Below the lunar surface, energy is transported by heat conduction. The energy flow across a unit area (Q) in the presence of a temperature gradient is:

$$Q = -\kappa \frac{\partial T}{\partial z} \frac{W}{m^{-2}}$$

where $\kappa \frac{W}{Km}$ is the thermal conductivity.

Now consider a one-dimensional column of subsoil extending down from the lunar surface. If the column is divided into N volume elements of equal depth δx and horizontal surface area S, the energy

change in the volume element during the interval δt is the difference between the amount entering and the amount leaving averaged over the volume:

$$\frac{E_j^{n+1} - E_j^n}{\delta t} = \frac{Q_{j+\frac{1}{2}}^n - Q_{j-\frac{1}{2}}^n}{\delta z}$$

Introducing the specific heat $C_v = (\frac{\partial Q}{\partial T})_v \frac{J}{kg\ K}$ (the energy required to cause an increment in temperature for a fixed volume), this formula becomes the one-dimensional thermal conduction equation as $\delta t \to 0$ and $\delta z \to 0$:

$$\frac{\partial T}{\partial t} = \frac{1}{\rho C_v} \frac{\partial}{\partial z} (\kappa \frac{\partial T}{\partial Z})$$

where T is the temperature and ρ the density. The specific heat is calculated using an approach by Golden [Gol77]. Based on analysis of lunar soil samples obtained by the Apollo mission, it represents the specific heat (measured in $J\ kg^{-1}\ K^{-1}$) as a polynomial of the form:

$$C_v(T) = 785.6 + 498.6(\frac{T - T_{300}}{t_{300}}) + 73.7(\frac{T - T_{300}}{t_{300}})^2 + 1139.2(\frac{T - T_{300}}{t_{300}})^3 + 782.5(\frac{T - T_{300}}{t_{300}})^4$$

for temperatures above 300 K and below 350 K. Above this limit we use an expression by Wechsler et al. [WGF72]:

$$C_v(T) = 849.5 + 160.4(1 - e^{\frac{T - T_{350}}{T_{100}}})$$

The notation uses subscripts to indicate values so $T_{100} = 100K$. The conductivity is given by the expression:

$$\kappa = \kappa_c[1 + \chi(\frac{T}{T_{350}})^3]$$

where κ_c is the phonon conductivity (9.22 $10^{-4}\ Wm^{-1}\ K^{-1}$) and χ is the ratio of "radiative conductivity" to phonon conductivity.

D.2.3 Energy Flux Fraction

The energy a facet receives from other facets due to blackbody radiation becomes a primary factor to its temperature when the facet does not receive sunlight. The surface fraction between facet i and facet j is:

$$F_{i-j} = (cos(\angle_i)cos(\angle_j)S_i)/(4\Pi(D_{i-j})^2)$$

where $\angle_i$ is the angel of facet, S_i is the area of facet and D_{i-j} is the distance between them.

The energy a facet i receives from all the other facets therefore can be calculated as:

$\sum F_{i-j}(\frac{L_o}{4\Pi r^2}A_b sin^+(\Psi) + \varepsilon_{IR}\sigma_B T^4)_j$

D.3 Application Development

The application is developed based on the Distributed Object Graphs approach (refer to Section 4.2.1).

Facet Object: A facet object calculates the temperature of a facet surface according to the mathematic algorithms introduced above. It communicates with other objects to exchange the blackbody radiation data. In every time step, each object broadcasts its outflux data to all the other facet objects, which calculate influx fractions based on the fraction formula. If the objects are in the same process, data transfer will be internal to the process, else it will use the message passing service of the Herder processes. The number of facets is decided by the geographic profile and visualization resolution. The higher the resolution is, the more facet objects are generated.

Herder Process: The process holds a number of facet objects and provides a communication bus for object data transfer with the help of the AA's communication service (LPWComm, PMWComm). It is also responsible for object migration once the Master process decides to redistribute objects. An application normally has several Herder processes to improve execution performance.

Master Process: The process is responsible for data (facet objects) distribution, load balancing, and receiving result from the Herder processes. It is the process which starts and communicates with the AA's RD (Resource Discovery) and ART (Automatic Resource Tuner) daemons.

Control/View Process: The process developed based on GTK+ is started on our local computer and only communicates with the Master. It generates temperature plots according to the temperature data passed by the Master. It also provides a console for users to change the resolution.

D.3.1 Load Balancing

The Master performs load balancing based on Algorithm 7. Load balancing is performed when the execution environment is changed due to the AA's automatic tuning as well as when the data structure is modified due to a resolution increasing or decreasing.

Load balancing involves object redistribution and object migration, i.e. move facet objects from a processor to a less loaded processor. Once the Master calculates a new object distribution, it broadcasts the object-process mapping (object index $\rightarrow$ process UPID) to each Herder process. The Herder processes then contact related processes to transfer objects. For example, a new mapping is:

Object Index	Process UPID
$0 \sim 99$	4259842
$100 \sim 150$	4259843
...	...

If process 4259842 currently holds objects ($0 \sim 150$), according to the new mapping it will contact process 4259843 to migrate objects $100 \sim 150$.

An object *O* is transferred by: firstly, an empty facet object *O'* is created on the receiving Herder process. The sending Herder then passes all the attributes of *O* to the Herder, which fills object *O'* with those attributes. Finally, the sending Herder deletes object *O*.

D.3.2 ESD Report

The Master process calls ReportESD() at the end of each timestep. The load balancing overhead, which is the time taken to perform load balancing, needs to be eliminated before reporting an ESD to the AA. The overhead is tracked by adding two gettimeofday() before and after the load balancing function. By doing so the AA can receive more accurate performance$\leftrightarrow$ resource information to help its tuning.

D.4 Application Execution

D.4.1 From the User's Point of View

We started the application on a laptop by simply invoking "./LunarTemp", without any manual resource configuration. We set the execution QoS requirement to be **0.2 second/timestep**, which means that we wish it takes 0.2 seconds to process one timestep so that the View process will have enough data buffer to generate visualization images smoothly.

We started the application with resolution scale = 10. After about 2 seconds wait, the application started to produce images smoothly. At the 500th step, we increased the resolution to 100. The image updating speed immediately became very slow. Soon after the speed had a quick raise. The speed then vibrated for about 100 seconds before finally returned to the satisfied 0.2 second per timestep.

During the entire running process, the temperature data was steadily generated by 0.2 (or less) second per step except a temporary spike due to resolution change. The satisfaction interruption was negligible comparing with the whole execution which had 50,000 timesteps and took about 3 hours. From beginning to end, we had no worry about resource management issues and were highly satisfied by the smooth visualization.

Figure D.1 shows the execution speed and the speed with reconfiguration overhead eliminated. The latter was used to generate the ESD. Figure D.3 shows the temperature mapping updating.

D.4.2 Under the Hood

The resource management was taken over by the AA with the DPPE policy configured. When the application was invoked, the AA was immediately activated and began to discovery resources from underlying resource infrastructure. Here the candidate resource environment were two LINUX clusters in UCL computer science department. The resource configuration during the execution is shown in Figure D.2.

With the resolution of 10, the application generated 196 facets. Since the AA did not know how many resource it needed to satisfy the application, it initially added 5 nodes. With learning execution feedbacks, it calculated the most efficient resource configuration and released 4 redundant nodes after 20 steps. As shown in Figure D.1, the speed had a little change after 20 steps (4 nodes was released) but was still far above the requirement.

After the resolution was changed, the number of facets had a sharp increase (1847 facets) which caused very slow execution speed. In order to maintain the QoS requirement, the AA immediately began to add more nodes into the execution according to the updated resource requirement under the new application configuration. The adding procedure was taken a couple of times because the AA tried

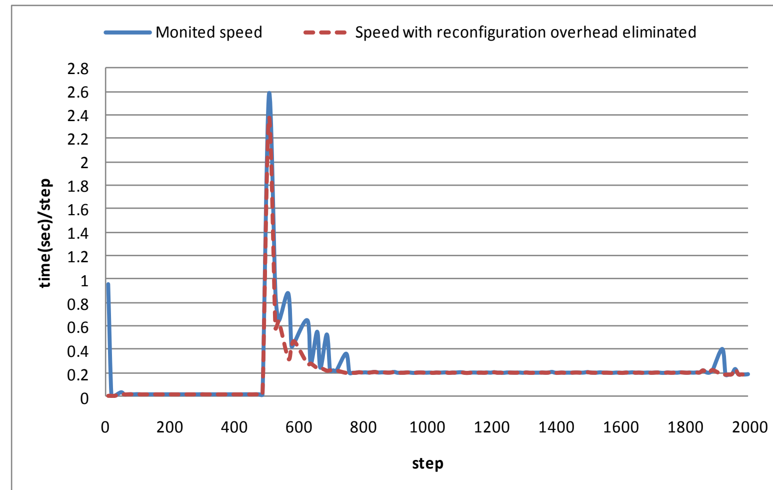


Figure D.1: Temperature mapping application execution speed. The vibration during the speed recovery process is due to application reconfiguration overhead.

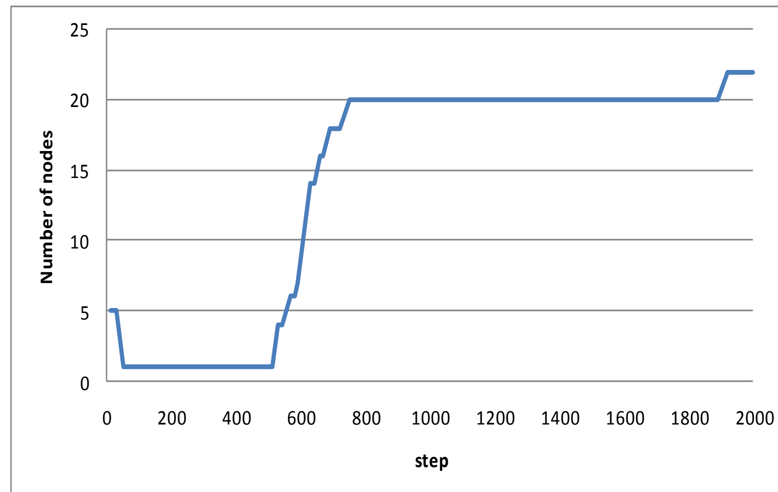


Figure D.2: The number of resource configured during the temperature mapping execution.

to calculate the least number of nodes that can satisfy the requirement based on the feedback of each reconfiguration. After this period, the AA still kept adjust the number of nodes. Once the speed deviated from the QoS level (either too high or too low), the AA would add/delete nodes to make sure it was the most efficient configuration to guarantee the QoS requirement. As we can observe in Figure D.2, there was another increase of node number around the 1800th step. This was because some nodes being used were suddenly overloaded due to job startups from other users. This resulted in unsatisfied speed therefore more nodes were needed.

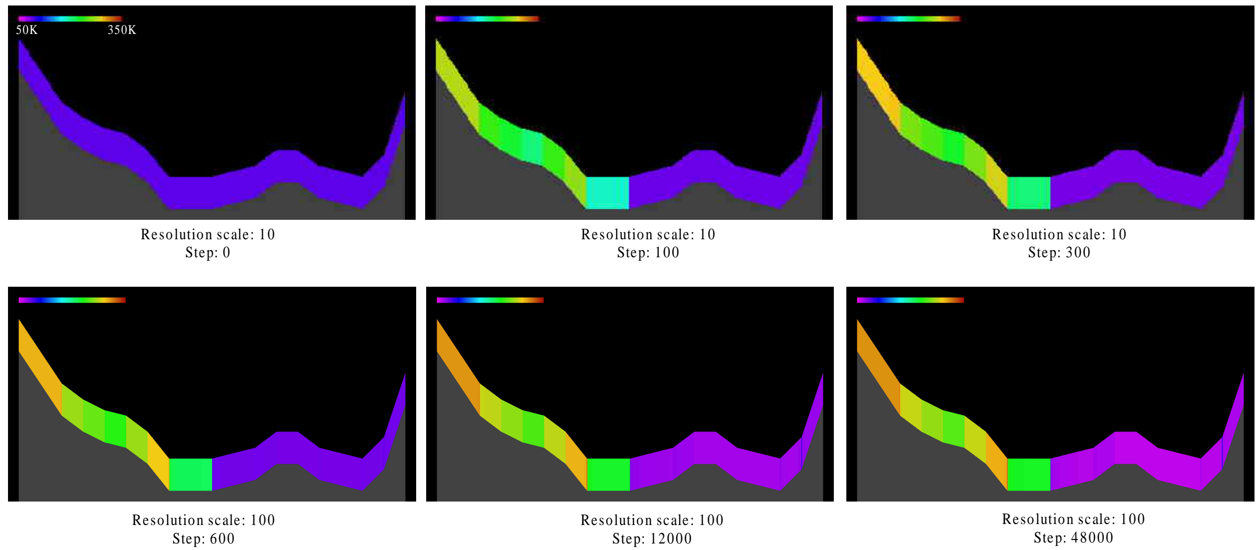


Figure D.3: Lunar temperature mapping. The first row shows low resolution images that were generated in the beginning 500 steps. The second row shows higher resolution images that were generated after we improved the resolution. The temperature changed over time with the influence of sunlight, blackbody radiation and energy conduction.

2D Lunar crater model - Latitude: 8.5e+01, Shadowpoint (Distance): 9.0e+00, Timestep: 1.0e+00, Geographic mode (Distance, Altitude): {(0.000000e+00, 1.738500e+03), (1.000000e+00, 1.738200e+03), (2.000000e+00, 1.737900e+03), (3.000000e+00, 1.737750e+03), (4.000000e+00, 1.737650e+03), (5.000000e+00, 1.737600e+03), (6.000000e+00, 1.737450e+03), (7.000000e+00, 1.737200e+03), (8.000000e+00, 1.737200e+03), (9.000000e+00, 1.737200e+03), (1.000000e+01, 1.737250e+03), (1.100000e+01, 1.737300e+03), (1.200000e+01, 1.737450e+03), (1.300000e+01, 1.737450e+03), (1.400000e+01, 1.737300e+03), (1.500000e+01, 1.737250e+03), (1.600000e+01, 1.737200e+03), (1.700000e+01, 1.737400e+03), (1.800000e+01, 1.738000e+03)}.

Subsoil model - Number of sub-facets: 200, Thickness of top layer<m>: 0.0013, Depth of subsoil model<m>: 10., Depth of soil model transition<m>: 0.02, Shallow layer density <kg m^{-3}>:1300., Deep layer density<kg m^{-3}>: 1800., Solid conductivity for upper layer<$W\ m^{-1}\ K^{-1}$> : 9.22E-04, Solid conductivity for deep layer<$W\ m^{-1}\ K^{-1}$>: 9.3E-03, Ratio of radiative to solid conductivity at 350K for upper layer: 1.48, Ratio of radiative to solid conductivity at 350K for deep layer: 0.0073, Start temperature<K>: 90., Flux from lunar interiour <$W\ m^{-2}$>: 0.033, Infrared surface emissivity<n/a>: 0.95, Lunar surface albedo <n/a>: 0.10.

Bibliography

[AA05] Michel Drescher Ali Anjomshoaa, Fred Brisard. Job submission description language (jsdl) specification, version 1.0. Technical report, Global Grid Forum, 2005.

[ABCM07] Nazareno Andrade, Francisco Brasileiro, Walfredo Cirne, and Miranda Mowbray. Automatic grid assembly by promoting collaboration in peer-to-peer grids. *J. Parallel Distrib. Comput.*, 67(8):957–966, 2007.

[ADG+05] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The grid application toolkit: Toward generic and easy application programming interfaces for the grid. *Proceedings of the IEEE*, 93(3):534–550, March 2005.

[Afg04] Enis Afgan. Role of the resource broker in the grid. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 299–300, New York, NY, USA, 2004. ACM.

[AM02] F. Azzedin and M. Maheswaran. Evolving and managing trust in grid computing systems. In *Electrical and Computer Engineering, 2002. IEEE CCECE 2002. Canadian Conference on*, volume 3, pages 1424–1429 vol.3, 2002.

[ama] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[BAG00] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. In *Proceedings of The Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region*, pages 283–289. IEEE Computer Society Press, 2000.

[BBES05] I. Brandic, S. Benkner, G. Engelbrecht, and R. Schmidt. Qos support for time-critical grid workflow applications. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8 pp.–115, July 2005.

[BBK+00] Milind Bhandarkar, Milind Bh, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-based adaptive load balancing for mpi programs, 2000.

[BCCP04] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. A load balancing framework for adaptive and asynchronous applications. *Parallel and Distributed Systems, IEEE Transactions on*, 15(2):183–192, Feb 2004.

[BDCT05] R.P. Bruin, M.T. Dove, M. Calleja, and M.G. Tucker. Building and managing the eminerals clusters: a case study in grid-enabled cluster operation. *Computing in Science & Engineering*, 7(6):30–37, Nov.-Dec. 2005.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[Bec05] Peter H. Beckman. Building the teragrid. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 363:1715 – 1728, 2005.

[Ber] Frank Vanden Berghen. Small, simple, cross-platform, free and fast c++ xml parser. http://www.applied-mathematics.net/tools/xmlParser.html.

[BGA00] Rajkumar Buyya, Jonathan Giddy, and David Abramson. An evaluation of economy-based resource trading and scheduling on computational power grids for parameter sweep applications. In *Sweep Applications, The Second Workshop on Active Middleware Services (AMS 2000), In conjunction with HPDC 2001*. Kluwer Academic Press, 2000.

[BHL+99] A. Bayucan, R. L. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten. Portable batch system: External reference specification. Technical report, MRJ Technology Solutions, November 1999.

[BK99] Robert K. Brunner and Laxmikant V. Kale. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, 1999.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS*, pages 356–368, 1994.

[BL06] Bruce R. Beattie and Jeffrey T. LaFrance. The law of demand versus diminishing marginal utility. *Review of Agricultural Economics*, pages 263–271, 2006.

[BM02] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. 2002.

[BMA02] Rajkumar Buyya, Manzur Murshed, and David Abramson. A deadline and budget constrained cost-time optimization algorithm for scheduling task farming applications on global grids. In *In Int. Conf. on Parallel and Distributed Processing Techniques and Applications, Las Vegas*, 2002.

[BRL99] J. Basney, B. Raman, and M. Livny. High throughput monte carlo. In *the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[BWC+03] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive computing on the grid using apples. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003.

[BWF+96] Francine D. Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, Washington, DC, USA, 1996. IEEE Computer Society.

[CFFK01] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, page 181, Washington, DC, USA, 2001. IEEE Computer Society.

[CFK+98] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459:62–82, 1998.

[CFK99] Karl Czajkowski, Ian Foster, and Carl Kesselman. Resource co-allocation in computational grids. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 37, Washington, DC, USA, 1999. IEEE Computer Society.

[clo09] Introduction to cloud computing architecture. Technical report, Sun Microsystems, 2009.

[CMEM07] C. Chapman, M. Musolesi, W. Emmerich, and C. Mascolo. Predictive resource scheduling in computational grids. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[CMK+94] J. R. Cruz, R. E. Mineck, D. F. Keller, M. V. Bobskill, Juan R. Cruz, Raymond E. Mineck, Donald F. Keller, and Maria V. Bobskill. *Parallel Computing Works*. 1994.

[COBW00] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS parameter sweep template: User-level middleware for the grid. pages 75–76, 2000.

[Con] Condor. Condor online manual version 7.0. http://www.cs.wisc.edu/condor/manual/v7.0/.

[cor] Corba. http://www.corba.org/.

[CP97] Erick Cant'u-Paz. Designing efficient master-slave parallel genetic algorithms, 1997.

[DA06] Fangpeng Dong and Selim G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical report, School of Computing, Queens University, 2006.

[Dan08] Krissi Danielson. Distinguishing cloud computing from utility computing. 2008.

[Dec00] Thomas Decker. Virtual data space — load balancing for irregular applications. *Parallel Computing*, 26(13–14):1825–1860, 2000.

[DLP+96] R. Van Dantzig, Miron Livny, Jim Pruyne, D. H. J. Epema, D. H. J. Epema, M. Livny, M. Livny, X. Evers, and X. Evers. A worldwide flock of condors: load sharing among workstation clusters, 1996.

[DMV07] Travis Desell, Kaoutar El Maghraoui, and Carlos A. Varela. Malleable applications for scalable high performance computing. *Cluster Computing*, 10(3):323–337, 2007.

[DSB+04] Holly Dail, Otto Sievert, Fran Berman, Henri Casanova, Asim YarKhan, Sathish Vadhiyar, Jack Dongarra, Chuang Liu, Lingyun Yang, Dave Angulo, and Ian Foster. Scheduling in the grid application development software project. *Grid resource management: state of the art and future trends*, pages 73–98, 2004.

[ea03] Luis Ferreira Viktors Berstis Jonathan Armstrong et al. *Introduction to Grid Computing with Globus*. IBM Redbook, 2003.

[EBC+05] Wolfgang Emmerich, Ben Butchart, Liang Chen, Bruno Wassermann, and Sarah L. Price. Grid service orchestration using the business process execution language (bpel). *J. Grid Comput.*, 3(3-4):283–304, 2005.

[epr] e-protein project. http://www.e-protein.org/.

[Erw01] Dietmar Erwin. Unicore - a grid computing environment. In *Lecture Notes in Computer Science*, pages 825–834. Springer-Verlag, 2001.

[FBCL91] Michael J. Feeley, Brian N. Bershad, Jeffrey S. Chase, and Henry M. Levy. Dynamic node reconfiguration in a parallel-distributed environment. In *In Proceedings of the 1991 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 114–121, 1991.

[Fel03] Joshy Joseph; Craig Fellenstein. *Grid Computing*. IBM Redbook, December 30, 2003.

[FK99] Ian Foster and Carl Kesselman. The globus toolkit. *The grid: blueprint for a new computing infrastructure*, pages 259–278, 1999.

[FKNT02] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, Jun 2002.

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Fos99] C.; Lee C.; Lindell B.; Nahrstedt K.; Roy A. Foster, I.; Kesselman. A distributed resource management architecture that supports advance reservations and co-allocation. *Quality of Service, 1999. IWQoS '99. 1999 Seventh International Workshop on*, pages 27–36, 1999.

[Fos02] Ian Foster. What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[FPEN01] Gene F. Franklin, David J. Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Fra05] Franco Frattolillo. Running large-scale applications on cluster grids. *Int. J. High Perform. Comput. Appl.*, 19(2):157–172, 2005.

[FS05] Cedric Angelo M. Festin and Søren-Aksel Sørensen. Utility-based buffer management for networks. In *ICN (1)*, pages 518–526, 2005.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[Gen01] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:35, 2001.

[GF96] Vasudha Govindan and Mark A. Franklin. Application load imbalance on parallel processors. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 836–842, Washington, DC, USA, 1996. IEEE Computer Society.

[GF04] Dennis Gannon Geoffrey Fox. Workflow in grid systems, 2004.

[GKLY00] Jean-Pierre Goux, Sanjeev Kulkarni, Jeff Linderoth, and Michael Yoder. An enabling framework for master-worker applications on the computational grid. In *HPDC*, pages 43–50, 2000.

[GLY99] J. Goux, J. Linderoth, and M. Yoder. Metacomputing and the master-worker paradigm, 1999.

[Gol77] L.M Golden. *A microwave interferometric study of the sub-surface of the planet Mercury*. PhD thesis, California University, 1977.

[gria] Gridsam. http://gridsam.sourceforge.net/.

[grib] Gridway: Metascheduling technologies for the grid. http://www.gridway.org/.

[Gru08] Galen Gruman. What cloud computing really means. *InfoWorld*, 2008.

[Har03] Brooke Coveney Harting. Computational steering in realitygrid, 2003.

[HCC07] Richard Huang, Henri Casanova, and Andrew A. Chien. Automatic resource specification generation for resource selection. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[IK77] Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, 1977.

[Ing87] Jonathan E. Ingersoll. *Theory of Financial Decision Making*. Rowman & Littlefield Publishers, Inc, 1987.

[iwa95] I-way project develops ”internet of the future”. *SUPERCOMPUTING '95*, 1995. http://www.sdsc.edu/SDSCwire/v1.15/7037.iway.html.

[JH97] David J. Jackson and Chris W. Humphres. A simple yet effective load balancing extension to the PVM software system. *Parallel Computing*, 22(12):1647–1660, 1997.

[KF98] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[KG96] James Arthur Kohl and G. A. Geist. The pvm 3.4 tracing facility and xpvm 1.1. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, pages 290–299, 1996.

[KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[KW05] Kyung-Woo Kang and Gyun Woo. An automatic resource selection scheme for grid computing systems. *Computational Science and Its Applications ICCSA 2005*, pages 29–36, 2005.

[LDM+01] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis using hardware counters. In *International Conference on Parallel and Distributed Computing Systems*, 2001.

[LGR05] Peggy Lindner, Edgar Gabriel, and Michael M. Resch. Gcm: a grid configuration manager for heterogeneous grid environments. *Int. J. Grid Util. Comput.*, 1(1):4–12, 2005.

[LGR07] Peggy Lindner, Edgar Gabriel, and Michael M. Resch. Performance prediction based resource selection in grid environments. In *HPCC*, pages 228–238, 2007.

[LK01] Orion S. Lawlor and Laxmikant V. Kal. Supporting dynamic parallel object arrays. In *In Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, pages 21–29, 2001.

[LNS07] Hao Liu, Amril Nazir, and Soren-Aksel Sorenson. Resource management support for smooth application execution in a dynamic and competitive environment. In *SKG '07: Proceedings of the Third International Conference on Semantics, Knowledge and Grid*, pages 438–441, Washington, DC, USA, 2007. IEEE Computer Society.

[LNS08] Hao Liu, Amril Nazir, and Søren-Aksel Sørensen. Preliminary resource management for dynamic parallel applications in the grid. In *The Second International Conference on Networks for Grid Applications (GridNets)*, pages 70–80, 2008.

[LNS09] Hao Liu, Amril Nazir, and Sren-Aksel Srensen. A software framework to support adaptive applications in distributed/parallel computing. In *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, pages 563–570, June 2009.

[LRK+06] McGuffin LJ, Smith RT, Bryson K, Sorensen SA, and Jones DT. High throughput profile-profile based fold recognition for the entire human proteome. *BMC Bioinformatics*, 2006.

[LSD08] Bin Lin, Ananth I. Sundararaj, and Peter A. Dinda. Time-sharing parallel applications through performance-targeted feedback-controlled real-time scheduling. *Cluster Computing*, 11(3):273–285, 2008.

[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[LWK05] Chenyang Lu, Xiaorui Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 16(6):550–561, June 2005.

[LZ07] Bo Li and Dongfeng Zhao. Performance impact of advance reservations from the grid on backfill algorithms. *Grid and Cooperative Computing, 2007. GCC 2007. Sixth International Conference on*, pages 456–461, Aug. 2007.

[MCSML07] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. Mate: Monitoring, analysis and tuning environment for parallel/distributed applications: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(11):1517–1531, 2007.

[MDSV06] Kaoutar El Maghraoui, Travis Desell, Boleslaw K. Szymanski, and Carlos A. Varela. The Internet Operating System: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications (IJHPCA), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms*, 20(4):467–480, 2006.

[MF01] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[mpi] Mpich2. http://www.mcs.anl.gov/research/projects/mpich2/.

[MSSJ05] Liam J. McGuffin, Richard T. Smith, Soren-Aksel Sorensen, and David T. Jones. Multi-site structural annotation of the human proteome in 24 hours using jyde, 2005.

[MT04] Jeremy M. R. Martin and Alex V. Tiskin. Dynamic BSP: Towards a Flexible Approach to Parallel Computing over the Grid. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 219–226, sep 2004.

[MYWWSZ00] H. Min-You Wu; Wei Shu; Zhang. Segmented min-min: a static mapping algorithm for meta-tasks on heterogeneous computing systems. *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 375–385, 2000.

[NLS07] Amril Nazir, Hao Liu, and Søren-Aksel Sørensen. Powerpoint presentation: Steering dynamic behaviour. In *Open Grid Forum 20*, Manchester, UK, 2007.

[NNdA07] L.N. Nassif, J.M. Nogueira, and F.V. de Andrade. Distributed resource selection in grid using decision theory. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 327–334, May 2007.

[Nor00] William B. Norton. Internet service providers and peering, 2000.

[ogf] Open grid forum. http://www.ogf.org/.

[ope] Fedstage open drmaa service provider. http://sourceforge.net/projects/opendsp/.

[PDD05] James Padgett, Karim Djemame, and Peter Dew. Grid service level agreements combining resource reservation and predictive run-time adaptation. In *UK e-Science All Hands Meeting*, Nottingham, UK, 2005.

[PL96] Jim Pruyne and Miron Livny. Interfacing condor and pvm to harness the cycles of workstation clusters. *Future Gener. Comput. Syst.*, 12(1):67–85, 1996.

[PVL+04] Johan Parent, Katja Verbeeck, Jan Lemeire, Ann Nowe, Kris Steenhaut, and Erik Dirkx. Adaptive load balancing of parallel applications with multi-agent reinforcement learning on heterogeneous systems. *Sci. Program.*, 12(2):71–79, 2004.

[PYE+04] R.U. Payli, E. Yilmaz, A. Ecer, H.U. Akay, and S. Chien. Dlb - a dynamic load balancing tool for grid computing, 2004.

[PZ06] Mario Petrone and Roberto Zarrelli. Enabling pvm to build parallel multidomain virtual machines. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 187–194, Washington, DC, USA, 2006. IEEE Computer Society.

[Ram00] Rajesh Raman. *Matchmaking frameworks for distributed resource management*. PhD thesis, 2000. Supervisor-Miron Livny.

[REF+07] Morris Riedel, Thomas Eickermann, Wolfgang Frings, Sonja Dominiczak, Daniel Mallmann, Thomas Dussel, Achim Streit, Paul Gibbon, Felix Wolf, Wolfram Schiffmann, and Thomas Lippert. Design and evaluation of a collaborative online visualization and steering framework implementation for computational grids. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 169–176, Washington, DC, USA, 2007. IEEE Computer Society.

[Ro04] Harald Rock. Parallel solving of the heat equation with mpi. Technical report, Department of Scientific Computing University of Salzburg, 2004.

[RS05] Buyya R. and Venugopal S. A gentle introduction to grid computing and technologies. *CSI Communications*, pages 9–19, 2005.

[RSM03] Primoz Rus, Boris Stok, and Nikolaj Mole. Parallel computing with load balancing on heterogeneous distributed systems. *Adv. Eng. Softw.*, 34(4):185–201, 2003.

[RV04] Herbert Rosmanith and Jens Volkert. glogin - interactive connectivity for the grid. In *Proc. of DAPSYS 2004, 5th Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 3–11. Kluwer Academic Publishers, 2004.

[RVR+98] Y L. Ribler, Jeffrey S. Vetter, Randy L. Ribler, Je S. Vetter, Huseyin Simitci, Huseyin Simitci, Daniel A. Reed, and Daniel A. Reed. Autopilot: Adaptive control of distributed applications. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, pages 172–179, 1998.

[San99] Peter Sanders. Analysis of nearest neighbor load balancing algorithms for random loads. *Parallel Comput.*, 25(8):1013–1033, 1999.

[Sge] N1 grid engine6 administration guide. Technical report, Sun Microsystems,Inc.

[Sha01a] Gary Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, 2001. Chair-Berman, Francine.

[SHA+01b] J.A. Stankovic, Tian He, T. Abdelzaher, M. Marley, Gang Tao, Sang Son, and Chenyang Lu. Feedback control scheduling in distributed real-time systems. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 59–70, Dec. 2001.

[SHM96] D. B. Skillicorn, Jonathan M. D. Hill, and W. F. Mccoll. Questions and answers about bsp, 1996.

[SLL99] Jaspal Subhlok, Peter Lieu, and Bruce Lowekamp. Automatic node selection for high performance applications on networks. In *In Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 163–172. ACM Press, 1999.

[SM] Inc Sun Microsystems. Java native interface. http://java.sun.com/j2se/1.4.2/docs/guide/jni/.

[SO98] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[soa] Soap. http://www.w3.org/TR/soap/.

[Spr01] Paul L. Springer. Pvm support for clusters. In *CLUSTER '01: Proceedings of the 3rd IEEE International Conference on Cluster Computing*, page 183, Washington, DC, USA, 2001. IEEE Computer Society.

[SSB+95] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[SSCC04] Otto Sievert, Otto Sievert, Henri Casanova, and Henri Casanova. A simple mpi process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications*, 18:341–352, 2004.

[SSL01] Edd Dumbill Simon St. Laurent, Joe Johnston. *Programming Web Services with XML-RPC*. O'Reilly, 2001.

[SSM+05] Soren-Aksel Sorensen, Stefano Street, Alan Medlar, David Jones, and Liam McGuffin. Cross-domain cluster computing. 2005.

[STB+07] Patrick Sumby, Eleanor Tengku, Mark Buckingham, Timothy Leung, Lesley Jonas-Nartey, Ying Li, Chermai Man, and Edward OŚhaughnessy. Temperature mapping of the lunar surface (3c02 group project). Technical report, University College London, 2007.

[Tec04] IBM AlphaWorks Technology. Grid application framework technical white paper. Technical report, 2004.

[TEF07] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *In IEEE TPDS*, 18:789–803, 2007.

[TRHD07] Peter Troger, Hrabri Rajic, Andreas Haas, and Piotr Domagalski. Standardization of an api for distributed resource management systems. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 619–626, Washington, DC, USA, 2007. IEEE Computer Society.

[TsGK+03] Damien Tool-set, Edgar Gabriel, Rainer Keller, Matthias S. Mller, Peggy Lindner, Matthias S. M/iller, Michael M. Resch, and Innovative Computing Laboratories. Software development in the grid: The damien tool-set, 2003.

[TWML02] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor: a distributed job scheduler. pages 307–350, 2002.

[VAMR01] Fredrik Vraalsen, Ruth A. Aydt, Celso L. Mendes, and Daniel A. Reed. Performance contracts: Predicting and monitoring grid application behavior. In *GRID*, pages 154–165, 2001.

[VD02] Sathish S. Vadhiyar and Jack J. Dongarra. Srs - a framework for developing malleable and migratable parallel applications for distributed systems. In *In: Parallel Processing Letters. Volume*, pages 291–312, 2002.

[WB06] Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical report, 2006.

[WGF72] A.E. Wechsler, T.E. Glaser, and J.A. Fountain. Thermal properties of granulated materials. *Thermal Characteristics of the Moon*, 1972.

[Win99] Dave Winer. Xml-rpc specification. On-line, 1999. http://www.xmlrpc.com/spec.

[WSH99] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[XLS04] Li Xiong, Ling Liu, and Ieee Computer Society. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Transactions on Knowledge and Data Engineering*, 16:843–857, 2004.

[xml] Xmlrpc++. http://xmlrpcpp.sourceforge.net/.

[ZLP95] Mohammed Javeed Zaki, Wei Li, and Srinivasan Parthasarathy. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, 43:156–162, 1995.

[ZZWD93] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Softw. Pract. Exper.*, 23(12):1305–1336, 1993.