Tupleware: A Distributed Tuple Space for the Development and Execution of Array-based Applications in a Cluster Computing Environment

by

Alistair Kenneth Atkinson, BComp(Hons).

Submitted in fulfilment of the

requirements for the Degree of

Doctor of Philosophy (Computing)

University of Tasmania

January 2010

Declaration

Statement of Authority of Access

....

Alistair Atkinson, 12 January 2010

Abstract

This thesis describes Tupleware, an implementation of a distributed tuple space which acts as a scalable and efficient cluster middleware for computationally intensive numerical and scientific applications. Tupleware is based on the Linda coordination language (Gelernter 1985), and incorporates additional techniques such as peer-to-peer communications and exploitation of data locality in order to address problems such as scalability and performance, which are commonly encountered by traditional centralised tuple space implementations.

Tupleware is implemented in such as way that, while processing is taking place, all communication between cluster nodes is decentralised in a peer-to-peer fashion. Communication events are initiated by a node requesting a tuple which is located on a remote node, and in order to make tuple retrieval as efficient as possible, a tuple search algorithm is used to minimise the number of communication instances required to retrieve a remote tuple. This algorithm is based on the locality of a remote tuple and the success of previous remote tuple requests. As Tupleware is targetted at numerical applications which generally involve the partitioning and processing of 1-D or 2-D arrays, the locality of a remote tuple can generally be determined as being located on one of a small number nodes which are processing neighbouring partitions of the array.

Furthermore, unlike some other distributed tuple space implementations, Tupleware does not burden the programmer with any additional complexity due to this distribution. At the application level, the Tupleware middleware behaves exactly like a centralised tuple space, and provides much greater flexibility with regards to where components of a system are executed.

The design and implementation of Tupleware is described, and placed in the context of other distributed tuple space implementations, along with the specific requirements of the applications that the system caters for. Finally, Tupleware is evaluated using several numerical and/or scientific applications, which show it to provide a sufficient level of scalability for a broad range tasks.

The main contribution of this work is the identification of techniques which enable a tuple space to be efficiently and transparently distributed across the nodes in a cluster. Central to this is the use of an algorithm for tuple retrieval which minimises the number of communication instances which occur during system execution. Distribution transparency is ensured by the provision of a simple interface to the underlying system, so that the distributed tuple space appears to the programmer as a single unified resource.

It is hoped that this research in some way furthers the adoption of the tuple space programming model for distributed computing, by enhancing its ability to provide improved performance, scalability, flexibility and simplicity for a range of applications not traditionally suited to tuple space based systems.

List of Figures

2.1 The message passing model. 11
2.2 The shared-memory model. 17
2.3 Example of OpenMP's fork/join parallelism (Chapman et al. 2008). 20
2.4 The client/server architecture. 24
2.5 The peer-to-peer architecture. 24

3.1 An example tuple space system. 27
3.2 Disjoint, nested, and overlapping tuple spaces. 40

4.1 Local tuple storage using a hashtable. 49
4.2 Architecture of a Tupleware system. 60
4.3 Example of a tuple production operation. 65
4.4 Single tuple retrieval from both a local and remote segment of tuple space. 66
4.5 Batch retrieval operations. 67
4.6 Grouping of neighbouring nodes based on success factor. 70
4.7 Overview of example system with five nodes. 80

5.1 Updating panel boundary values. 86
5.2 A complete parallel sorting application. 91
5.3 Flow of array segments through the modified quicksort application. . 94

6.1 Total runtimes (secs) for the ocean model. 103
6.2 Overall ocean model speedup. 105
6.3 Sequential runtimes (secs) for varying number of nodes in the ocean model. 106
6.4 Sequential runtimes (secs) in terms of the total number of grid points.107
6.5 Parallel execution runtimes of the ocean model. 108
6.6 Speedup of the ocean model with varying number of nodes. 109
6.7 Total runtimes of the parallel sorting application. 113
6.8 Total speedup of the sorting application. 114

6.9 Sorting application’s parallel execution runtimes. 117
6.10 Speedup of parallel phase of sorting application execution. 117
6.11 Efficiency of the sorting application on a varying number of nodes. . 119
6.12 Communication and computation times (secs) of the parallel phase of the sorting application. 120

List of Tables

2.1 TCP/IP Protocol Stack Layers 12

3.1 Summary of WCL operations. 37

4.1 Summary of the Tupleware API. 64

6.1 Total grid points and data size. 101
6.2 Complete runtimes of ocean model. 102
6.3 Total Runtimes (secs) of ocean model. 103
6.4 Overall speedup of ocean model. 104
6.5 Runtime (secs) of sequential stages of ocean model. 104
6.6 Ocean Model parallel runtimes (secs). 107
6.7 Speedup of the ocean model's parallel phase. 108
6.8 Efficiency of the ocean model. 110
6.9 Total runtimes (secs) of the sorting application. 112
6.10 Total speedup of the sorting application. 113
6.11 Sorting application's initialisation times (secs). 115
6.12 Parallel runtime data (secs) of the sorting application. 116
6.13 Parallel speedup of the sorting application. 118
6.14 Efficiency of the sorting application. 118

Acknowledgements

Completing this thesis has easily been one of the most difficult things I've ever undertaken. During the course of this work, my supervisors, Vishv Malhotra, Julian Dermoudy, and John Hunter have been extremely supportive, and have displayed a great deal of patience in helping me to reach the finishing line. They have my sincere gratitude for their assistance and guidance.

To all those at the School of Computing, particularly those at the Launceston campus, I'd like to thank them for making my years studying there so enjoyable and rewarding. In particular, thanks to my fellow PhD candidate Brad Goldsmith for enduring having to share an office with me for several years, and to the "Dodgy Tackle" indoor soccer team, who, despite suffering frequent demoralising defeats, prompted me to get away from the keyboard on a Wednesday night and get some exercise.

The final phase of this thesis was completed while working at EIT Hawke's Bay, New Zealand, and I thank them for their understanding in allowing me the time and flexibility to complete the write-up process over the course of the last two years.

Most of all though, I'd like to acknowledge my family. My parents, Barry and Helene, have been nothing but loving and supportive for my entire life, and I'd never have gotten this far if it weren't for them. And my deepest gratitude goes to my wife, Bonne, who has been nothing but unbelievably patient and encouraging over these years that I've been a seemingly professional student.

Finally, this research was financially supported by an Australian Postgraduate Awards scholarship, and a cluster was made available by the School of Computing, UTas, to enable development and testing of the software produced.

Contents

Declaration **i**

Abstract **iii**

List of Figures **vi**

List of Tables **vii**

Acknowledgements **viii**

1 Introduction **1**

1.1 Motivation & General Aims of this Research 2
1.2 Limitations of this Research 3
1.3 Overview of Methodology 4
1.4 Contributions of this Research 4
1.5 Thesis Structure 5
1.6 Publications derived from this research 6

2 Parallel & Distributed Computing **7**

2.1 Introduction 7
2.2 Platforms 7
2.2.1 Clusters 8
2.2.2 Wide Area Networks 8
2.2.3 Grid Computing 9
2.2.4 Summary 10
2.3 Models of Distributed Computing 11
2.3.1 Message Passing 11
2.3.1.1 Sockets 12
2.3.1.2 Other message passing implementations 15
2.3.1.3 Summary of the message passing model 16
2.3.2 Shared Memory 16

2.3.2.1 Threads 18
2.3.2.2 OpenMP 19
2.3.2.3 Summary 21
2.3.3 Remote Procedure Calls 22
2.3.4 Distributed System Architectures 23
2.4 Summary 25

3 Coordination Models 26
3.1 Background & Overview 26
3.2 Fundamental Concepts 26
3.2.1 Tuple Space 26
3.2.2 Associative Lookup 27
3.2.3 Loosely-Coupled Interaction 27
3.3 Linda Operations 28
3.4 Programming with Linda 28
3.4.1 Distributed Data Structures 29
3.4.2 Task Parallelism 29
3.5 Related Work 30
3.5.1 MTS-Linda 31
3.5.2 Jada 32
3.5.3 LIME 33
3.5.4 SwarmLinda 34
3.5.5 WCL 35
3.5.6 JavaSpaces 37
3.5.7 Other Java-based Linda implementations 39
3.5.8 Scope 39
3.6 Performance & Scalability Issues 41
3.7 Summary 43

4 Tupleware Implementation 44
4.1 Aims 44
4.2 How these aims will be addressed 45
4.2.1 Scalability & Performance 45
4.2.2 Ease of Programmability & Logical Integration of Distributed Tuple Spaces 46
4.3 Basic Components 47
4.3.1 Tuples & Templates 47
4.3.2 Local Tuple Space 48

4.3.2.1 Tuple Storage & Lookup 48
4.3.2.2 Local Space Synchronisation 50
4.4 Distribution of Tuple Space 54
4.4.1 Inter-node Communication 54
4.4.1.1 Tuple Space Service 54
4.4.1.2 Tuple Space Stubs 54
4.4.1.3 Communication Protocol: Request & Response Objects 55
4.4.1.4 Stub/Service Interaction 56
4.4.2 Runtime System 58
4.4.3 Application Processes: Master & Workers 60
4.4.4 Summary 60
4.5 Runtime System 61
4.5.1 Runtime System API 61
4.5.2 Tuple Production 63
4.5.3 Single Tuple Retrieval 65
4.5.4 Bulk Tuple Retrieval 66
4.5.5 Tuple Search Algorithm 67
4.5.5.1 Rationale 68
4.5.5.2 Request Success Factor 68
4.5.5.3 Searching for a Remote Tuple 69
4.5.5.4 Strengths and Weaknesses of the Search Algorithm 72
4.5.6 Summary 72
4.6 Deployment & Execution 73
4.6.1 Initialisation 73
4.6.2 Task Submission 74
4.6.3 Execution 74
4.6.4 Gathering Results 75
4.6.5 Fault tolerance & dynamic reconfigurability 75
4.7 Application Development 77
4.7.1 Implementation with Tupleware 77
4.7.1.1 Master Process Implementation 77
4.7.1.2 Worker Process Implementation 79
4.7.1.3 Load Balancing & Skewed Access Considerations 80
4.7.1.4 Implementation Summary 82
4.8 Summary 82

5 Applications **84**
5.1 Ocean Model 84
5.1.1 Equations 84
5.1.2 Implementation 86
5.1.3 Application Components & Execution 87
5.1.4 Summary 87
5.2 Sorting 88
5.2.1 Overview & Characteristics 88
5.2.2 Implementation 89
5.2.3 Sorting Summary 92
5.3 Summary 93

6 Performance Evaluation **95**
6.1 Performance Metrics 95
6.1.1 Speedup 95
6.1.2 Efficiency 96
6.1.3 Amdahl’s Law 97
6.1.4 Gustafson’s Law 98
6.1.5 Scalability 98
6.1.6 Summary 99
6.2 Evaluation Methodology 99
6.3 Ocean Model 99
6.3.1 Execution Environment 100
6.3.2 Experimental Details 100
6.3.3 Total Runtimes 103
6.3.4 Sequential Runtime 104
6.3.5 Parallel Runtime 106
6.3.6 Analysis & Summary 109
6.4 Sorting 111
6.4.1 Execution Environment 111
6.4.2 Experimental Details 111
6.4.3 Total Runtime 112
6.4.4 Initialisation 114
6.4.5 Parallel Runtime 115
6.4.5.1 Efficiency 118
6.4.6 Analysis & Summary 121
6.5 Comparison with other systems 121
6.6 Conclusions 122

7 Conclusions & Further Work **123**
7.1 Conclusions . . . 123
7.1.1 Scalability & Performance . . . 123
7.1.2 Ease of programmability . . . 124
7.1.3 Contribution of the research . . . 125
7.2 Further Work . . . 125

References **127**

A Tupleware Source Code **131**
A.1 Tuple . . . 131
A.2 TupleTemplate . . . 132
A.3 TupleSpace . . . 133
A.4 TupleSpaceImpl . . . 133
A.5 TupleSpaceRequest . . . 137
A.6 TupleSpaceResponse . . . 138
A.7 TupleSpaceStub . . . 139
A.8 TupleSpaceService . . . 142
A.9 TupleSpaceRuntime . . . 146

B Application Source Code **157**
B.1 Ocean Model . . . 157
B.1.1 OceanModelMaster . . . 157
B.1.2 OceanModelWorker . . . 159
B.1.3 Panel . . . 161
B.1.4 Original FORTRAN version of Ocean Model . . . 165
B.2 Sorting . . . 169
B.2.1 QSort . . . 169
B.2.2 QSortMaster . . . 172
B.2.3 QSortWorker . . . 175

Chapter 1

Introduction

This thesis describes the research undertaken in order to develop a scalable distributed tuple space-based platform which is able to provide increased performance for certain types of scientific and numerical applications in a cluster computing environment, while retaining a high degree of simplicity for the application programmer.

The fundamental problems being addressed by this thesis is that of simplifying the task of developing parallel applications for a distributed computing environment, and providing these applications with increased performance using a distributed tuple space platform. Despite the fact that distributed computing platforms and technologies have been around for many years, these are still often viewed as inherently complex exercises. This has lead to the aims stated above; put simply, this research aims to provide a platform to distributed system development with a low barrier of entry for the programmer.

Distributed computing is a field of growing importance; it has for a long time been used for solving large problems, such as scientific simulations, digital rendering, and any job of sufficient size that it was more practical or cost-effective to approach the problem using a distributed solution as opposed to a standalone supercomputer.

However, despite a background stretching back over twenty years, and also due to the almost universal coverage of the Internet and the recent popularity of cloud computing, there is even more urgency to develop platforms which can provide the services required by an ever-growing range of applications. Likewise, software developers accustomed to writing programs for standalone computers have been struggling with the challenge of learning to develop software for this emerging environment. Existing platforms are often difficult to use or ill-suited due to their emergence from the scientific computing fields, and quite often one of the more desirable features of a platform is a low barrier of entry while still providing performance benefits. This is the area that this research aims its focus.

It should be noted that we are not trying to present a system which provides the absolute optimal performance for applications; MPI and PVM have long been considered the de-facto standards for achieving this. However, the barrier of entry for the application programmer has been seen as an issue with systems such as these, and as such the tuple space approach has been chosen for use due to the characteristics which enable us to hide many of the issues of distribution and parallelism from the programmer. What this research seeks to explore is how to provide an appreciable level of performance gains while keeping the barrier of entry as low as possible to the application programmer.

1.1 Motivation & General Aims of this Research

The middleware presented in this thesis implements a distributed tuple space which can provide a scalable platform for the development and execution of large, computationally intensive scientific and numerical applications.

Tuple space systems have long been recognised as being an effective tools for parallelising coarse-grained, or embarrassingly parallel applications where there are few dependencies between each parallel task. Such applications tend to have manageable communication requirements, and therefore provide opportunities to deliver speedup benefits by controlling the overhead introduced by transmitting data over the network, and from the computer hosting tuple space having to service requests.

In general, this overhead imposes a serious limit to the scalability of a distributed tuple space system when it is used to run application which are more finely-grained, and have numerous dependencies between tasks and more frequent communication.

Inevitably, a centralised tuple space implementation will become overwhelmed trying to service requests as the number of processes in the system and/or communication frequency becomes large. Therefore, at some point it becomes necessary to decentralise the tuple space, and to spread the load of servicing requests between those computers which host each part of the distributed space.

Distribution of the tuple space presents several challenges which must be addressed in order to maintain and improve the performance of parallel applications which utilise it. Namely, the time taken to search for and retrieve tuples from the space must not increase too dramatically compared to a single centralised space. To this end, tuples need to be stored in the distributed space in such a way that requests are, as evenly as possible, spread amongst the computers which host each part of the space.

The system presented in this paper addresses these issues using techniques outlined in subsequent sections. Further, it aims to not place any extra burden on the application programmer due to the distribution of tuple space. As far as is practicable, the distribution of the tuple space should be transparent to the programmer, and, from the programmer's perspective, behave exactly like a global tuple space.

1.2 Limitations of this Research

The research presented in this thesis was an investigation into how the tuple space model can be adapted for a cluster computing environment so that it can provide a scalable platform for scientific and numerical applications, and so that these applications can obtain performance benefits from using the system. As such, this research did not set out to address any of the following:

- **Geographically distributed systems:** Tupleware was not designed to meet the unique requirements of wide area network environments. It is aimed solely at cluster computing environments (which may also include networks-of-workstations (NOWs)).

- **Security:** The research aims do not extend to any provision of security, and are targetted exclusively at scalability, programmability, and performance. Whilst security is undoubtedly an important consideration for production systems, it is peripheral to the research problem being explored.

- **Use for general-purpose applications:** Tupleware is targetted toward applications which perform array-based processing. This self-imposed limitation allows the system to include optimisations that benefit this particular class of application, rather than try to fulfil the requirements of general-purpose distributed applications.

- **High-performance computing as the fundamental goal:** Established systems such as MPI and PVM are widely used in the scientific community for implementing high-performance computing software. The aim of Tupleware is to provide a *sufficient increase* in the performance of these classes of application, and to do so in such as way that there is not a high burden to the application programmer to implement their software on the system. As we will see, one criticism of the message-passing systems mentioned is that they can be relatively complex. Tupleware aims, as far possible, to preserve the simplicity of the Linda coordination language.

1.3 Overview of Methodology

Based on the limitations of tuple space-based systems outlined in Chapter 3 from which the aims outlined above were identified, we sought to implement a distributed tuple space with the goal of providing a scalable yet simple platform for the development and execution of distributed array-based applications. The aims of the system were used to inform the design and implementation decisions made, and which are detailed in Chapter 4.

In order to whether the system was successful in meeting its aims, its performance was evaluated using two applications with contrasting characteristics and requirements: an ocean model and a parallel sorting application. Using these application, the behaviour of Tupleware was evaluated in terms of its runtimes of various relevant parts of its operation, including time spent performing initialisation, computation, communication, and final result gathering tasks. From these measurements we evaluate the performance gains and scalability of the system.

A thorough description of the Tupleware application programming interface is also included in order to outline the usage and semantics of the operations provided, and to highlight their simplicity and consistency with the operations found in Linda.

1.4 Contributions of this Research

In the author's opinion, the tuple space is an understudied topic in the area of distributed computing. As we will see in the review of coordination models presented in Chapter 3, there are many different variations on the tuple space which have been implemented or proposed. However, many of them are intended to be a general-purpose distributed computing platform, focussing on issues such as availability and fault-tolerance. Others are presented as theoretical models which have not been implemented, and offer few clues as to how such an implementation might be approached. Applied research into how the tuple space may be implemented for distributed environments while preserving the simplicity of the original Linda language is lacking.

The overall contribution of this research is to investigate the implementation and behaviour of a distributed tuple space. The Tupleware tuple space is distributed across all of the nodes in a cluster, and all interaction between nodes occurs in a completely decentralised, peer-to-peer fashion. One of the outcomes of this research is to highlight techniques and approaches which may be used to allow a system structured like this to provide a scalable and efficient platform for distributed applications.

Central to this contribution is the development and implementation of a tuple search algorithm which is used to retrieve tuples from remote segments of tuple space. This algorithm aims to minimise the instances of network communication, the main source of poor performance in a distributed system, by targeting requests for remotely hosted tuples to those nodes which have the highest probability of being able to fulfil the request. This probability is represented as a success factor, which is calculated based on the success or failure of previous requests. The search algorithm allows the system to dynamically adapt, at runtime, to the communication characteristics of an application.

The other main contribution of this research is to implement the distributed tuple space in such a way that the complexity is hidden from the application programmer. A layered architecture was used when designing and implementing Tupleware, central to which was a middleware layer which implements the underlying system logic. Applications interact with the lower-level Tupleware system through this middleware, which in turn provides the application with a simple application programming interface on which to develop an application. As we will show, the operations provided by Tupleware are semantically consistent to their original Linda equivalents, and do not add any complexity beyond what would be found in a non-distributed, centralised tuple space such as Java Spaces. This contribution is important: one of the strengths of the original Linda model was its simplicity, and the future adoption of the tuple space as a distributed and/or parallel programming tool will not be aided by adding complexity. It is hoped that this research in some way furthers this cause.

1.5 Thesis Structure

This chapter has provided an overview of the research, and has described the general aims, limitations and contributions of this thesis. The remainder of the thesis is structured as follows:

- Chapter 2 provides an introduction and overview to the field of parallel and distributed computing,

- Chapter 3 discusses the area of coordination languages, including Linda,

- Chapter 4 describes Tupleware, and includes a complete discussion of its aims, design, implementation, operation and its application programming interface,
- Chapter 5 details the applications, and ocean model and a parallel sorting application, which were implemented on Tupleware in order to evaluate its scalability and performance,
- Chapter 6 contains the results and analysis of the performance evaluations which were undertaken,
- Chapter 7 summarises the research and contributions of this thesis, and identifies some further work which could be undertaken in the future.

1.6 Publications derived from this research

The following refereed conference papers have been published as a result of the work undertaken during this research:

> Atkinson, A 2008, ’Tupleware: A Distributed Tuple Space for Cluster Computing’, *Proceedings of the Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, University of Otago, Dunedin, pp. 121-126.

This paper provided a general overview of the Tupleware system, and relates to the entirety of this thesis.

> Atkinson, A 2009, ’A Dynamic, Decentralised Search Algorithm for Efficient Data Retrieval in a Distributed Tuple Space’, to appear in the *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing*, Queensland University of Technology, Brisbane.

This paper described in detail the design and operation of the search algorithm used in the Tupleware runtime system.

Chapter 2

Parallel & Distributed Computing

2.1 Introduction

Distributed computing is the study of the methods involved in enabling networked autonomous computers to cooperate in order to carry out a common task. It is closely related to the field of parallel computing, which is concerned with running separate parts of a program on different processors in order to achieve increased execution speed. It follows, then, that a distributed computing system may also be a parallel computing system, where each discrete part of a program is executed on a different computer on a network. It is this intersection of the two fields which is the main focus of this chapter.

This chapter firstly provides an overview of the fundamental models used in constructing a distributed system, and concludes with a survey of some of the more notable distributed computing platforms.

2.2 Platforms

Distributed systems may be deployed on a wide variety of platforms. Each of these platforms have the following in common: discrete parts of the system will execute on individual computers (or nodes), and communicate via a network to carry out some common task. Apart from this, a distributed system platform may have a number of differences, often centred around the type of network used to connect each node. These variations often dictate the suitability of particular applications for each platform. For example, applications which require large data transfers will perform better when deployed on a platform with large amount of available network bandwidth.

The following section describes some of the common distributed computing platforms, including clusters and wide-area networks, briefly covers grid computing and finally details two common distributed application architectures: client/server and peer-to-peer.

2.2.1 Clusters

A cluster is defined by Pfister (1998, p. 72) as a collection of interconnected individual computers which are used as a single, unified computing resource. This definition may encompass clusters which are specifically designed and assembled with specialised hardware and high speed interconnects, through to systems which consist of inexpensive personal computers and which are connected via an Ethernet LAN.

The common characteristic of a cluster is that parallel applications which are executed on them treat the cluster as though it is a single system rather than a collection of individual computers. To achieve this, usually a piece of software known as a middleware is required. Middleware software controls access to the cluster's resources by applications, usually in a transparent way, so that the application logic does not need to account for the physical location of individual hosts or other system resources.

The main advantages of using a cluster as a platform for distributed computing are that they are relatively cheap and simple to construct, especially if commodity components are used, and can nonetheless provide high performance computational capability using a range of easily obtained cluster middleware software.

2.2.2 Wide Area Networks

Wide Area Networks (WANs), such as the Internet, consist of geographically distributed nodes connected by relatively low speed networks. Nodes are generally not under the administrative control of the person deploying the distributed system, and are voluntarily ”donated” by their owner to take part in the activities of the distributed system when they are idle. This approach has been shown to be successful by projects such as SETI@Home (University of California 2008) and distributed.net (Distributed.net 2008), each of which use the processing capacity of idle Internet-connected computers to carry out, respectively, analysis of observational data from a telescope, and solving large-scale cryptographic problems. In the case of SETI@Home, as of August 15th, 2008, there were over 327,000 active nodes with a combined capacity of approximately 472 TeraFLOPS (Boincstats 2008).

Whilst Internet-scale distributed systems clearly provide access to vast numbers of nodes at minimal financial cost, they are not without their limitations. The wide geographic distribution of nodes, lack of centralised administrative control and use of an open network such as the Internet mean that WAN-based distributed systems are inherently less reliable than clusters, and security becomes an important consideration. Furthermore, the use of low speed networks is a limiting factor in the types of applications which are suited to a WAN environment; communication and data transfer are more expensive and thus need to be minimised in order for the system to scale. This means that tightly-coupled applications which require frequent communications will not experience the same speedup as loosely-coupled, embarrassingly parallel applications such as those deployed on the SETI@Home and distributed.net platforms.

Despite the limitations outlined above, WANs are an inexpensive way to to harness the unused processing capacity of idle computers, on the condition that people are willing to volunteer their idle computers to join the system.

2.2.3 Grid Computing

Grid computing is a form of distributed computing which makes it possible for geographically distributed resources, such as clusters and data storage devices, to be accessed as a single computing resource (Foster & Kesselman 2003). The term initially referred to a pervasive, ”on-demand” model of distributed computing which provided access to vast amount of processing power and data. However, another important feature of grid computing is its ability to federate the computing resources of separate organisations into a single *virtual organisation* (Foster & Kesselman 2003, p. 38). As such, grids can not only allow access to high-performance computing resources, but also collaborative computing and resource sharing between organisations.

Several software toolkits exist to allow developers to create computing grids, an example of which is the Globus Toolkit created by the Globus Alliance (Globus 2008). Globus performs the integration of all of the resources on the grid, transparently providing the appearance of a single unified computing resource.

Grid computing has had several successes in recent years, mainly in the area of scientific research. The use of computational grids in this field is often referred to as *e-Research*. A notable recent example of this is the grid computing platform used for the Large Hadron Collider (LHC) experimental particle physics facility. This grid platform, known as the *Worldwide LHC Computing Grid* (WLCG), is an example of a data grid: its main purpose is to store and distribute the massive quantities of data produced by the LHC, which is expected to generate up to 15 Petabytes of data annually when it is fully operational (CERN 2008). The grid also allows scientists from other organisations to access the data so that they can analyse the data produced.

As an example of the scope of the inter-organisation resources that are a part of the WLCG, consider the following facts taken from (CERN 2008):

- The grid's computing resources are located in more than 140 computing centres in 33 countries,
- It is able to harness the capacity of 100 000 processors,
- It is expected to be used to run over 100 million computer programs by 2008,
- It will provide access to data for 5000 participating scientists in approximately 500 research organisations worldwide.

As we can see from this example, grid computing is capable of providing a very large scale distributed computing platform, and is able to integrate geographically distributed computing resources from across the world.

2.2.4 Summary

In this section we have described the three broad categories of distributed computing platform: clusters, wide area networks, and grids. As discussed, each have their own unique properties, strengths, and weaknesses, which makes each platform suited to particular types of application. Clusters can provide access to large amounts of processing capacity at a relatively low cost if commodity hardware is used, and are suitable for a broad class of applications. Distributed systems, such as as SETI@Home, that utilise wide area networks can provide very inexpensive access to geographically distributed computing resources. However the high latency and relatively low network throughput of these networks means they are more suited to coarse-grained applications which do not require frequent communication between nodes. Finally, grid computing can integrate the computing resources of geographically distributed organisations in order to provide access to vast amount of computing capacity, including data storage and processing capability. However, grids are suited more to large-scale inter-organisational settings, and are relatively complex to set up and administer, and so are suited mostly for areas such as scientific research settings.

2.3 Models of Distributed Computing

There are numerous distributed computing technologies which enable the creation of a distributed system, ranging from simple TCP sockets through to sophisticated Grid computing systems such as Globus. Each individual technology is unique in some way, whether it be domain-specific or general purpose, be designed for a specific architecture or be cross platform, or whether or not it is easily transferable to a particular programming paradigm.

Nonetheless, these technologies often share some common properties, which allow us to group technologies based on the general model of distributed computing they support. Leopold (2001, p. 20) defines a distributed computing model as having a ”high degree of abstraction” and are used to ”express the important concepts of a field”. This section presents three of the fundamental models of distributed computing: message passing, shared memory and remote procedure calling.

2.3.1 Message Passing

The message passing model involves two or more processes running in parallel, and communicating by sending and receiving messages. Each process maintains its own local memory, and must coordinate their communication so that every send operation has a corresponding receive operation. Message passing may be used to carry out exchange of data or the synchronisation of actions between processes. The message passing model is shown in Figure 2.1.

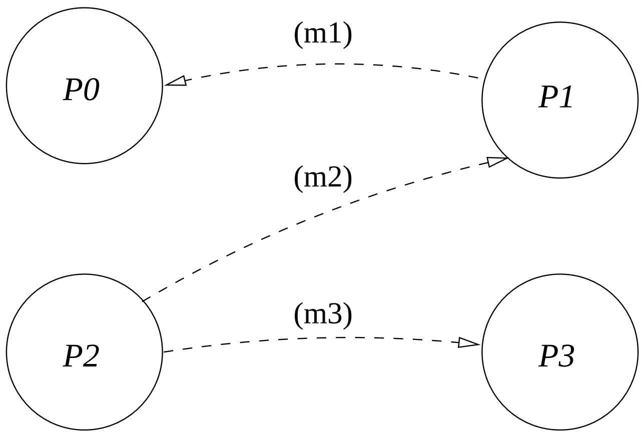


Figure 2.1: The message passing model.

Each message passing instance must be explicitly specified by the application programmer. In order for a message to be sent, the programmer must specify not only the message itself, but also the address or identifier of the receiving process.

Thus, processes must be known to each other before messages can be exchanged, and, as the message is transmitted directly between the two processes, both processes must be in existence at the time the message is sent.

2.3.1.1 Sockets

At perhaps the lowest level, the message passing model can be implemented using Internet Protocol (IP) sockets. IP, which operates in conjunction with either the higher-level Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP), is the dominant network protocol for LANs and WANs at the time of writing, significantly due to it being the protocol used on the Internet. TCP, UDP, IP and their other associated protocols are sometimes referred to as the *Internet Protocol Suite*. Note that, for the purposes of this section, discussion will be limited to the more common IP version 4, and ignore the newer IP version 6. For information on IP version 6 see Deering & Hinden (1998).

Overview of the Internet Protocol Suite

IP is a network layer protocol which encapsulates data from a higher level protocol into packets. Each host in an IP network has a unique 32 bit IP address. An IP address is represented using dotted-decimal notation (four octets separated by a dot ('.') character).

IP is a connectionless protocol which does not guarantee the delivery of packets to their destination. Rather, it is a store-and-forward protocol which simply attempts to route a packet to a node nearer to its intended destination. It is not uncommon for packets to be dropped on the way to their destination, as IP allows this to happen when network congestion is encountered. Also, packets may arrive out of order, or their data payload may be corrupted while being transmitted. Finally, IP is a host-to-host packet delivery protocol, rather than process-to-process or application-to-application, as the IP address refers only to an individual host on the network. All of these reliability issues are handled by a higher level protocol, such as TCP.

OSI Layer	**Internet Protocol Suite**
Application	HTTP, FTP, SMTP etc
Transport	TCP, UDP
Network	IP
Data Link	Ethernet, 802.11a/b/g, ATM etc
Physical	Optic fibre, Twisted Pair etc

Table 2.1: TCP/IP Protocol Stack Layers

TCP exists at the transport layer, and is a connection-oriented, stream-based protocol used by most of the Internet's application protocols such as HTTP. Data to be transmitted from an application is packaged into a TCP segment, which, among other header fields, also contains a port number denoting the port of the receiving application. The use of a port number allows for communication between networked applications, rather than simply between hosts as is the case at the IP level. TCP guarantees the reliable, ordered delivery of a stream of data to its destination. These features, however, do come at the cost of additional communications overhead. The receipt of each TCP segment is acknowledged by the receiving application, and packets are retransmitted in the event of a packet being lost. This makes TCP unsuitable for some applications, including real-time applications such as streaming media. For applications such as these, UDP is often the preferred choice of transport layer protocol.

Like TCP, UDP provides application-to-application communications using ports. However, unlike TCP, it does not guarantee the reliable delivery of a message (called a datagram in relation to UDP), nor does it provide for ordered delivery of messages. The only error detection included in UDP is a 16-bit checksum header field used to detect the corruption of any of the datagram's header fields or data payload. UDP is not a connection-oriented protocol, and datagrams are stateless (ie. not part of a specific connection or data stream. As such, the receipt of delivered datagrams are not acknowledged, and undelivered datagrams are not retransmitted. Also, UDP datagrams do not contain as much overhead as TCP segments in the form of header fields. This results in UDP being a more efficient protocol, and it is suited for use in real-time applications such as VOIP, DNS and networked games.

For further information on TCP, UDP or IP refer to Comer (2003).

The Socket Interface

Sockets provide a full-duplex communications channel between two processes executing on computers connected by an IP network. Peterson & Davie (2000, p. 43) state that sockets can be though of as ”the point where a local application process attaches to the network”. Sockets are an abstraction provided originally by the 4.2BSD UNIX operating system, and now in the majority of modern operating systems including Microsoft Windows, Mac OS X and GNU/Linux. Libraries for accessing sockets are available in most programming languages.

A socket consists of the following components:

- A protocol (TCP or UDP),

- Local IP address,
- Local port number,
- Remote IP address (for established connections), and
- Remote port number (for established connections).

When a socket is first created by a process it is in an unconnected, or unestablished state. A connection must be established to another socket (usually on a remote host) in order to create a *socket pair*. The establishment of a socket pair involves one host opening a socket and waiting for a connection to be initiated by some other host. The former is referred to as the *server*, and the latter as a *client*. The following examples (using Java) illustrate the process of creating a socket pair.

Firstly, the server must instantiate a `java.net.ServerSocket` object and bind it to a local port. Then, it uses the accept() method to wait for an incoming connection request from a client.

```
ServerSocket srvSock = new ServerSocket();
Socket incoming = srvSock.accept();
// socket pair has now been created
```

Secondly, the client creates a `java.net.Socket` object which it uses to connect to the server.

```
Socket clientSocket = new Socket(SERVER_ADDRESS, PORT);
```

Once the connection has been established, both server and client may obtain input and output streams for the socket and begin communication. For example, the client would perform the following:

```
ObjectOutputStream out =
    new ObjectOutputStream(clientSocket.getOutputStream());
out.flush();
ObjectInputStream in =
    new ObjectInputStream(clientSocket.getInputStream());
```

Sockets Summary

Sockets are a programming interface to the operating system's underlying TCP/IP protocol implementation, assisting a programmer to develop networked applications. Compared to other message-passing models, sockets are relatively low-level. It should also be noted that sockets are the underlying network interface upon which most other distributed computing technologies are built. An understanding of sockets and the protocols involved is therefore imperative when designing or implementing any distributed computing system.

2.3.1.2 Other message passing implementations

The Message Passing Interface (MPI) (Snir et al. 1996) is a standard for message passing on computing clusters; the most common implementation of the standard being the MPICH library for C, C++ and FORTRAN, although libraries for other programming languages have been developed. For the purposes of this section, we will not make a distinction between the MPI standard and its specific implementation.

At its core, MPI provides a mechanism to allow processes running in parallel to exchange messages via *send* and *receive* routines (`MPI_Send()` and `MPI_Recv()` respectively). These operations are quite low level in nature, and require the programmer to specify details such as the type and size of data being sent or received, addresses of buffers being used, and identity of the process being communicated with. Also, it is necessary for these parameters to match in order for communication to be successful (Leopold 2001, p. 102).

As mentioned in Section 2.3.1, in the message passing model for every send operation there must be a corresponding receive operation, or else the send operation will block indefinitely. MPI handles this problem by providing a number of blocking modes which can be chosen by the programmer. These modes allow both send and receive operations to return before they have been fully completed, before being completed at a later stage, using `MPI_ISEND()` and `MPI_IRECV()` respectively (Snir et al. 1996, pp. 49-60). These operations post the initial data included in the message, before being completed with either `MPI_WAIT()` or `MPI_TEST()`.

Parallel Virtual Machine (PVM) is a software tool for developing parallel applications for computing clusters (Geist et al, 1994). It consists of a complete framework, including libraries and runtime system, for the development of applications in C, C++, and Fortran.

Like MPI, PVM is an implementation of the message passing model, and it contains the operations `pvm_send()` and `pvm_recv()` to allow a message to be sent and received respectively. PVM also contains operations for spawning new processes, group (multicast) communication, and synchronisation.

Overall, MPI and PVM have been very popular tools for the development of scientific and numerical applications, particularly MPI, which provides a high level of speedup for both loosely- and tightly-coupled applications (Wernstein et al. 2003).

2.3.1.3 Summary of the message passing model

The main benefits of message passing models such as MPI and PVM are that they can provide a high level of speedup for broad classes of applications, and that they give the programmer control at a low level over the communication characteristics of a system. This allows applications to be optimised in order to maximise the performance of a system.

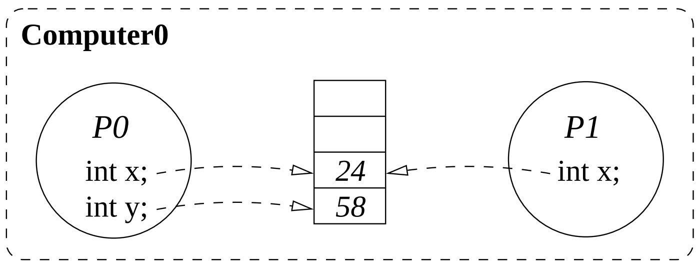
However, this low level of control can also be a drawback: applications which utilise message passing can become very complex and difficult to understand and debug. Gorlatch (2004) argues that message passing imposes too much difficulty for developing a parallel system, and suggests that message passing be replaced by a more structured form of parallel programming in much the same way that the *goto* statement was phased out of sequential programming languages in favour of a more structured approach.

Much of the complexity inherent in the message passing model stems from the tightly-coupled interaction between processes. In order for two processes to communicate, the address or name of each process must be known, and processes must both exist at the same point in time. Nonetheless, the performance gains provided by these systems must be weighed against these disadvantages when evaluating their suitability for a given parallel application.

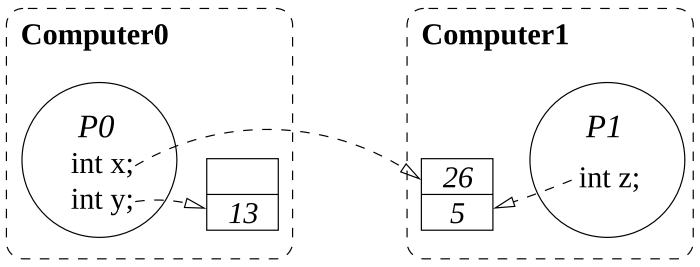
2.3.2 Shared Memory

The shared memory model involves two or more processes running in parallel, and communicating by reading from and writing to variables in a shared memory. In a traditional shared memory system, processes execute on the same computer, and share this computer's memory. In a *distributed* shared memory (DSM) system, the shared memory may be accessed by remote processes, or even be comprised of areas of memory located on more than one computer in the distributed system. These two approaches to shared memory are illustrated in Figure 2.2, which show firstly two processes sharing the a memory region on a single computer, and secondly a distributed shared memory across two computers. Each process contains one or more variables which reference a particular memory location.

Leopold (2001, pp. 74-75) discusses several advantages and disadvantages of the shared memory model. Advantages include the fact that this memory model is the same as in common imperative programming languages, and that the physical location of the shared memory and the details of communication with other processes do not need to be handled by the programmer.



Shared memory on a single computer.



Distributed shared memory on two computers.

Figure 2.2: The shared-memory model.

Possible disadvantages inherent to this model are those associated with synchronisation and locality. In the former case, it is possible for parallel processes to access a shared variable simultaneously, possibly leading to corruption of the variable's data, or for a race condition to occur. These problems introduce the need for a synchronisation mechanism to coordinate and sequentialise process' access to shared variables, such as, for example, Dijkstra's semaphores (Dijkstra 1968) or Java's synchronized keyword (Lea 2000, pp. 75-79). This, however, introduces the need for the programmer to include explicit synchronisation in their program whenever a shared variable is accessed, increasing the complexity of the program and also the difficulty in tracing bugs. Synchronisation also introduces the possibility of deadlock, whereby two or more processes may be waiting for the other to release a resource (in this case, a shared variable), leading to both (or all) processes blocking indefinitely.

Data locality refers to the proximity, or speed of access, of stored data in a DSM relative to the processes which access it. Obviously it is desirable for data in a DSM to be able to be accessed as quickly and efficiently as possible, and so, ideally, data that is to be accessed will be stored in a location which facilitates this. Data that is to be shared between two or more processes should ideally be stored at an optimal location, taking into consideration the locations of the executing processes. The efficiency of accessing shared data is obviously highly dependent on the underlying network of the distributed system, in particular its latency. Of course, these factors are not known to the programmer, who has limited control of the locality of data. Therefore it is the responsibility of the DSM's runtime system to ensure that shared data is stored in a location as close to optimal as possible.

The shared memory model is a form of inter-process communication more commonly used between processes executing on the same physical machine, mostly due to the problems associated with data locality. Nonetheless, platforms do exist which allow processes on separate machines to communicate and synchronise their activities using a shared-memory approach. The following is an overview of two of the common technologies used in the creation of a distributed system using shared memory: threads and OpenMP.

2.3.2.1 Threads

Threads represent a sequence of program commands, much like a process. However, a process may consist of multiple threads, each of which shares the process' address space. A process with two or more threads is said to be *multithreaded*. In single-processor computers, a multithreaded process will run concurrently, with each thread being independently scheduled by the operating system to execute on the processor. On multiprocessor or multi-core computers, a multithreaded process may run in truly parallel fashion, with threads executing simultaneously on separate processors or cores.

The main advantage of using threads rather than multiple processes for concurrency is that threads can communicate via their shared address space, whereas processes running concurrently must use some other form of inter-process communication such as UNIX pipes or sockets. Also, due to threads sharing an address space and other process state information, context switching between threads is generally faster than switching between processes.

Threads may be made available to the programmer by a library, such as the standard POSIX threads (Pthreads), or be natively supported by the programming language, such as in Java.

The advantages of resource sharing between threads does come at the cost of synchronisation: if multiple threads can access shared variables simultaneously, then there needs to be some sort of coordination to ensure that race conditions or data corruption do not occur (as discussed in Section 2.3.2). Both Pthreads and Java's threading model provide mechanisms such as object locks and mutexes for this purpose. For more on Java's synchronisation mechanisms see Lea (2000).

2.3.2.2 OpenMP

OpenMP is an API developed for the purpose of enabling the development of portable shared memory parallel applications (Chapman et al. 2008, p. 23). The API is defined by the OpenMP Architecture Review Board (OpenMP Architecture Review Board 2008), which consists of a group of computer software and hardware vendors, who came together to address their common cause of defining a standard parallel programming platform. OpenMP consists of compiler directives, environment variables and library functions for the FORTRAN, C and C++ programming languages. It is important to note that OpenMP is not intended as a distributed computing platform; it is a tool for multithreaded programming on shared memory multiprocessor computers. However, it a higher-level language than plain threads, as we will see below.

Parallelism is most commonly expressed in OpenMP using the fork/join model, which lends itself to the development of SPMD parallel applications. OpenMP applications begin with a single sequential thread, known as the *initial thread* (Chapman et al. 2008, p. 24). Sections of code, such as loops, may be marked as being parallelisable using (for C syntax) the compiler directive `#pragma omp parallel {...}` to indicate the parenthesised code block is to be parallelised. An example of the usage of this is demonstrated in the code listing below, which shows a parallel section in which each thread will obtain its number and print out the resultant value:

```
#pragma omp parallel {
  n = omp_get_thread_num();
  printf("I am thread number %i", n);
}
```

The OpenMP system will create an appropriate number of threads to carry out each parallel task; the exact number of threads created is determined by the OpenMP system, unless explicitly specified by the programmer. During the parallel section, the initial thread becomes the master of the newly created threads, which will execute either via multitasking on single CPU computers, or in truly concurrent fashion on multiprocessor or dual-core computers. Upon completion of the parallel section, the program will execute sequentially until the next parallel section is encountered. It is up to the OpenMP implementation to handle the low level details of creating and coordinating the threads used to complete the parallel computation. An illustration of a fork/join style of application can be found in 2.3.

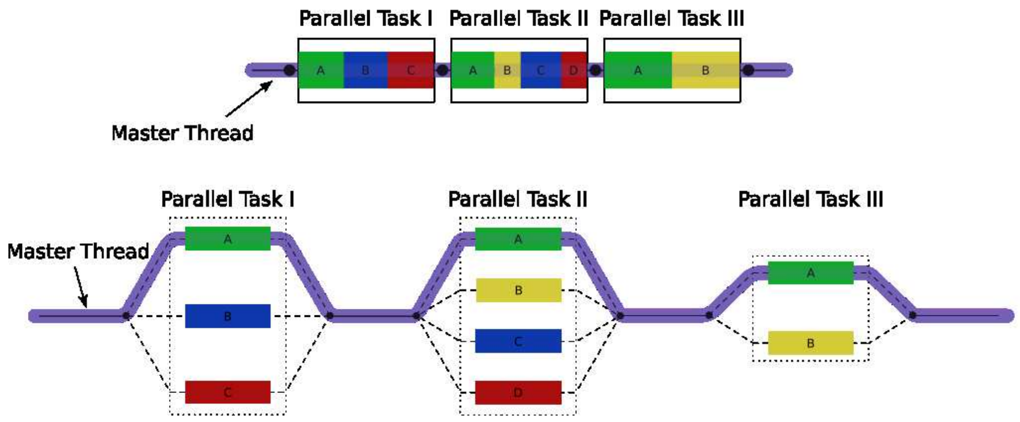


Figure 2.3: Example of OpenMP's fork/join parallelism (Chapman et al. 2008).

This approach allows the programmer to express parallelism in a high-level, abstract way, and it also gives the advantage of parallelism to be added to a sequential program in an iterative way. That is, a program need not be completely parallelised at inception, but rather sections of code may be parallelised as they are identified as being suitable for doing so. This also allows the programmers to maintain a single codebase for the application, rather than separate sequential and parallel versions.

OpenMP also caters for the MPMD style of parallelism using the `#pragma omp sections {...}` compiler directive in addition to `#pragma omp parallel`. Each specific task is specified inside the parentheses using `#pragma omp section {...}` directive. This usage of OpenMP will result in each thread executing a different section of the code. As an example, consider the code listing below:

```
#pragma omp parallel {
  #pragma omp sections {
    #pragma omp section
      foo();

    #pragma omp section
      bar();
  }
}
```

In this example code listing, two parallel sections are specified with the end result being that one thread will execute the `foo()` function, and the other will execute the `bar()` function.

Finally, OpenMP can be used to parallelise loops where no synchronisation is required within the loop. This is useful when performing some sort of array processing, where each thread may compute individual segments of the array. The following code listing, which computes the product of two arrays, demonstrates the use of the `#pragma omp parallel for` directive for parallelised loops:

```
#pragma omp parallel for
for(i = 0; i < 20; i++)
  a[i] = b[i] * c[i];
```

This loop will be parallelised by assigning a chunk of iterations to each thread to be executed. How these chunks are assigned will depend on which scheduling algorithm is being used; there are four algorithms available to the programmer, but for the purposes of this discussion we will classify them as either static or dynamic (Leopold 2001, p. 87). The programmer is able to choose which algorithm and/or chunk size is used, or they may leave it up to the OpenMP system to decide.

To explain the way the choice of scheduling algorithm influences the assignment of chunks of iterations, assume we have four threads that have been created to execute this loop. If a static scheduling algorithm were being used, then the assignment of chunks would be determined at compile-time. Therefore, a chunk size of five would assign five iterations of the loop to each thread: thread_0 would perform iterations 0 to 4, thread_1would perform iterations 5 to 9 and so on. If a dynamic scheduling algorithm were to be used, then the chunk size is determined at run-time, by the OpenMP system. The chunk size may be determined at the beginning of the loop and not alter, or alternatively, it may vary during execution of the loop based on the application characteristics.

In summary, OpenMP allows both SPMD- and MPMD-style parallel programs to be written using high-level programming constructs for shared memory parallel computers. Like threads, it is not a mechanism to distributed shared memory, but it does allow multithreaded programs to be expressed in a more abstract manner.

2.3.2.3 Summary

The shared memory model of parallel computing has several advantages, which can be summarised as follows (Leopold 2001, p. 74):

- The memory model used for these programs is the same as for ordinary sequential programs, thus being easier to understand for the programmer,
- The programmer is not burdened with the problems of specifying any details related to communication nor the location of data storage.

Compared to message passing systems such as MPI, shared memory is a higher level abstraction of a parallel system, however it generally relies on a runtime system to carry out the underlying operation of the system and hide complexities from the programmer.

Disadvantages of the shared memory model relate to synchronisation and the locality of stored data (Leopold 2001, p. 75). Firstly, as there is no direct communication between processes in a shared memory system, all interaction between processes occurs via creating and modifying data stored in the shared memory. In order to ensure consistency of data and that operations on a shared variable do not overlap, it is necessary for these systems include synchronisation constructs to serialise access to shared variables. However, this synchronisation introduces an overhead to the system due to this serialisation, and may also introduce problems such as deadlock and race conditions by making it difficult to ensure the correctness of a program.

The other issue of data locality has a large bearing on the performance of a shared memory system. There is a time penalty for accessing data, and to ensure optimum performance data that is required frequently should be stored near to where it will be used. In an ordinary shared memory computer, we wish for frequently used data to remain in cache to the fullest extent possible: in a distributed shared memory system, we wish for frequently used data to be stored on the node which is accessing it to avoid any inter-node communication from taking place. Because the shared memory model is relatively high level compared to message passing, often the programmer does not have complete control over these issues, and instead must use techniques such as program restructuring to minimise the negative effects of this problem.

2.3.3 Remote Procedure Calls

Remote procedure calling (RPC) is a form of inter-process communication which aims to make the act of communicating with remote processes as transparent as possible to the programmer. It does this by retaining the usual semantics of procedure calling in imperative programming languages when invoking the procedures of remote processes.

The idea of RPC was first proposed by Andrew Birrell and Bruce Nelson (1984). The basic concept is to allow programs to call procedures on another computer. They reasoned that procedure calls were a well understood concept for the transfer of control in a program, and therefore could be extended for use over a communications network. RPC successfully hides the underlying message passing from the programmer, making calls to remote procedures indistinguishable from calls to local procedures, and thus transparent.

Tanenbaum & Steen (2002, p. 69) suggest, however, that "subtle problems exist" with this idea. Firstly, the calling and called procedures in an RPC are on different machines, and hence in different address spaces. This complicates the passing of parameters and accessing return values to and from procedures. For example, consider parameters that are pointers. Also, it is inevitable that either one of the computers will crash or the network connection will be lost during a remote procedure call, thereby causing the system to experience partial failure.

The concept introduced by RPC to address these problems and enable RPC to to work effectively are *stubs*. Stubs on a client machine act as a proxy for a remote procedure, providing a local interface for the programmer to use to make a remote procedure call. The client stub will handle all of the details of sending messages over the network, including packing parameters into valid network messages (known as *marshalling*) and unpacking (*unmarshalling*) any values returned. Once the remote procedure call has been made, the stub will wait for a response from the corresponding server stub (Tanenbaum & Steen 2002, pp. 71-72).

A server stub is analogous to a client stub, characterised by Tanenbaum & Steen (2002, p. 71) as "a piece of code that transforms requests coming in over the network into local procedure calls." A server stub unmarshalls any parameters needed for the procedure call, makes the actual procedure call, and then marshalls any values returned before sending them over the network back to the client stub.

The client and server stubs provide conversion facilities if they are situated on incompatible systems; for example, if one system interprets bytes using the big endian format, and the other uses little endian. These compatibility issues are handled by the stubs on either machine. Stubs also handle the problem of passing pointers as parameters mentioned earlier; usually the pointers are dereferenced and passed by value.

RPC was an innovative technology that simplified the complex task of developing a distributed system. It introduced many concepts that are still in use in many current distributed technologies, including remote method invocation systems such as Java RMI (Sun Microsystems 2008) and CORBA (Object Management Group 2008).

2.3.4 Distributed System Architectures

Distributed systems may be classified as belonging to one of two broad categories based on their architecture: client/server or peer-to-peer. These categories are based on the role performed by each node in the system, and the way in which these nodes interact and communicate.

In the first instance, a client/server, or *centralised*, architecture implies the existence of at least one server node, which provides some service for one or more client nodes. In order to access this service, the client will send a request to the server, which will process the request and send the server back an appropriate response. The particular service being provided by a server can vary widely, but may include things such as data storage or transfer, processing, or a centralised point of communication with other nodes. The client/server architecture is illustrated in Figure 2.4.

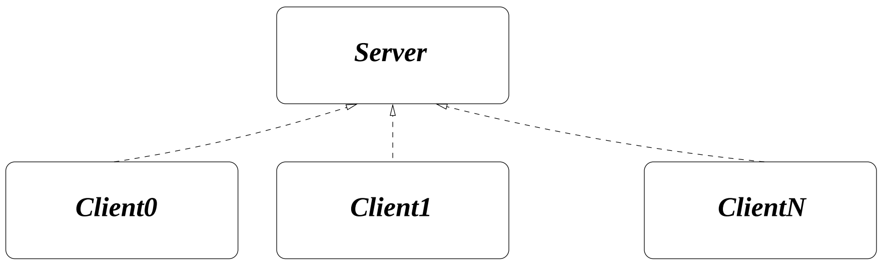


Figure 2.4: The client/server architecture.

In the second instance, peer-to-peer, or *decentralised*, architectures do not make the distinction between client and server nodes, but rather each node is treated as being equal, and most often acts as both a client and a server at different times. A good example of this is peer-to-peer file sharing programs such as Gnutella and Bittorrent (Bittorrent.org 2008), whereby each node in the system may be simultaneously downloading and uploading data from and to other nodes. The peer-to-peer architecture is illustrated in Figure 2.5.

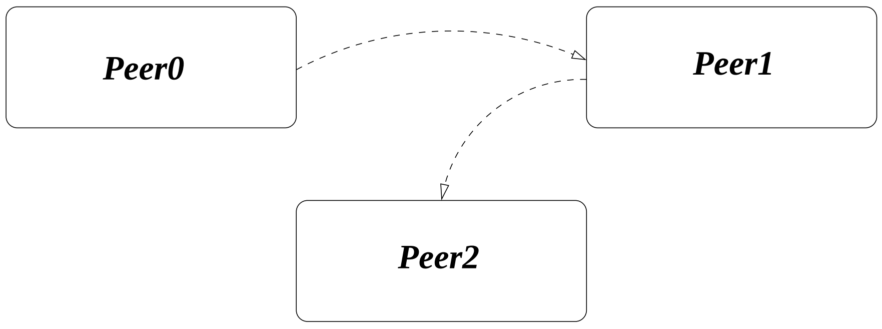


Figure 2.5: The peer-to-peer architecture.

The relative advantages and disadvantages of each architecture depend on factors such as the nature of the application, and the capabilities, reliability, and quantity of nodes in the system. As a general rule, client/server systems tend to experience scalability issues as the number of client requests becomes large and the server struggles to service the requests in a timely fashion. However, some applications, such as the World Wide Web, are inherently centralised, and so techniques such as load balancing must be used to spread client requests amongst multiple servers thus reducing the load on each individual server. On the other hand, peer-to-peer systems will generally perform more poorly in small systems, but may scale more effectively to larger sizes.

Another point of difference between the two architectures is that of fault tolerance. Client/server systems are centralised; that is, there are one or more servers servicing multiple clients. The server(s) in such a system can become a point of failure, and the loss of a server can cause the failure of the entire application. On the other hand, peer-to-peer systems are decentralised and do not rely on any central servers, and as such tend to be more fault tolerant. Most peer-to-peer applications can continue to function after the failure of even several nodes.

Finally, the administration of client/server systems tends to be simpler than for peer-to-peer networks, because the application logic can be contained in a small number of servers rather than distributed across all nodes participating in the system. Thus, the server has more control over matters such as task assignment and load balancing.

2.4 Summary

In this chapter we have described the main models and architectures of distributed and parallel systems. The broad categories of platform have been discussed, including clusters, grids and wide area networks. The main communication models have also been presented, including message passing, shared memory and remote procedure calling. The models covered here will inform much of the later discussion of the design and operation of Tupleware.

In the following chapter, the other main parallel computing paradigm is described in detail, that being coordination models.

Chapter 3

Coordination Models

3.1 Background & Overview

Coordination models were first proposed by Gelernter (1985) in his coordination language, Linda. Coordination models were proposed as a method of ”building programs by gluing together active pieces” (Carriero and Gelernter 1990, p. 8). As this implies, these models provided distinction between the computation which occurs in an individual process, and the interaction which occurs between processes, whether they are running on the same computer or on different computers on a network.

3.2 Fundamental Concepts

Coordination models are based on two concepts which distinguish them from other distributed programming abstractions: the idea of a *tuple space*, and *generative communication* as the means of process interaction.

3.2.1 Tuple Space

Tuple space is an an associative shared memory, by which processes may communicate, store data and coordinate their actions. More specifically, tuple space stores tuples; these are ordered lists of typed data of arbitrary length. For example, the tuple <’’`data`’’, `1.3`> contains two fields. The first field is a string, and the second a floating point number. Processes in a tuple space system act as producers and consumers of these tuples. An example of a tuple space system is shown in Figure 3.1.

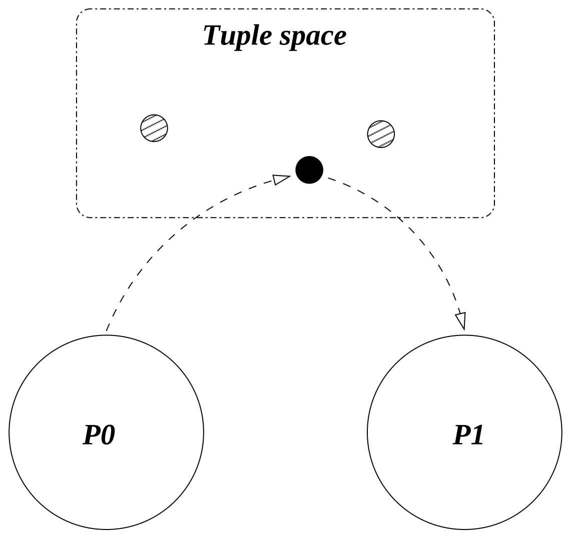


Figure 3.1: An example tuple space system.

3.2.2 Associative Lookup

Unlike most other parallel shared memory systems, tuple stored in tuple space are not explicitly addressed. Rather, they are accessed via *associative lookup*. This lookup process makes use of a *tuple template*, which is the same as a tuple except that some of its fields may be assigned a **null** value. These null values may act as wildcards, matching against any given value in the corresponding field of a tuple. A template is said to match a tuple provided the following two conditions are true: a) the template is the same length as the tuple, and b) any values specified in the template are equal to the tuple's values in their corresponding fields.

As an example, consider the tuple template `<''data'', null>`. This template would match the tuple `<''data'', 6.3>` due to the equality of their length and of their first field. It would not, however, match the tuples `<''hello'', 2.1>` or `<''data'', 6.3, ''rw''>` due to field equality and unequal lengths in each respective instance.

3.2.3 Loosely-Coupled Interaction

Processes in a tuple space system do not interact directly. Rather, all communication is performed via tuple space, which provides persistent tuple storage space independent of other processes in the system.

This distinctive feature of Linda-style systems allows for processes to be uncoupled in both space and time. In the former case, the use of tuple space as a communications mechanism allows processes to be distributed across different computers on a network. In the latter case, processes do not have to exist at the same point in time in order for them to communicate, due to the persistent nature of tuple space. It is possible for a process to place a tuple in tuple space and then terminate, only for another process to be created and consume the tuple at some later point in time.

3.3 Linda Operations

Linda is comprised of just a few fundamental operations, which are detailed below, and these operations were intended to be embedded into other conventional programming languages. Originally, Linda was embedded into the C programming language to create a variant named C-Linda. Linda consists of six operations: **out()**, **rd()**, **in()**, **rdp()**, **inp()** and **eval()**. The functionality of each is listed below.

- **out()** places a new tuple into tuple space,
- **in()** retrieves a tuple from tuple space which matches the given tuple template, permanently removing it from the space. If a matching tuple does not exist, then the operations blocks until one is placed into the space,
- **rd()** behaves exactly like **in()**, except that it retrieves a copy of a matching tuple rather than removing the tuple permanently,
- **inp()** and **rdp()** behave as **in()** and **rd()** respectively, except they do not block if a matching tuple is not available, and **null** values are simply returned.
- Finally, **eval()** is used to spawn a new process. This operation, though included in the original Linda model, tends to be excluded from most other implementations in favour of more traditional process creation mechanisms, usually largely influenced by the host implementation language.

3.4 Programming with Linda

Linda was designed to provide a clear separation between a program's computation and coordination code. The fundamental concept was that parallel system could be constructed by ”gluing together active pieces” (Carriero & Gelernter 1990, p. 181), where the active pieces are processes written in the host language, and the ”glue” is the coordination and communication mechanisms provided by Linda.

This section outlines some of the common patterns found in Linda programs, and gives examples of their usage.

3.4.1 Distributed Data Structures

A distributed data structure is a data structure whose values are stored in tuple space, with each value or element encapsulated in a tuple (Carriero & Gelernter 1990, pp. 49-50). The most common example of this are so-called ”position-accessed structures” (Carriero & Gelernter 1990, pp. 53-59), which are used to represent a the elements of an array in tuple space.

Distributed data structures such as these consist of a series of tuples which represent each element of the array, and use certain fields within the tuple for access. Typically the first field of the tuple is the name of the array, the following fields specify the index or indices of the array element, and the remainder of the tuple contains the data values stored in the array element.

So, as an example, if we wish to store the two-dimensional array *A* as a distributed data structure in tuple space, we would use a series of tuples such as the following:

```
<"A", 1, 1, (element data)>    <"A", 1, 2, (element data)>
<"A", 2, 1, (element data)>    <"A", 2, 2, (element data)>
...
```

If we wish to retieve a copy of one of the elements of *A*, then a template such as the one below would be used:

```
rd("A", 2, 2, ? data);
```

This would retrieve a copy of the element at $A[2][2]$ and assign its value to the variable `data`. The use of distributed data structures allows all processes participating in the system to share access to the data stored in them, and remove the need for data to be stored locally within the process itself. This gives a clean separation between processes and highlights the loosely-coupled nature of tuple space systems. A distributed data structure approach such as this maps well to a data parallelism style of system. Each process may be performing the same task, yet operating on different parts of the data simultaneously.

Position-access structures such as these are relevant to this thesis in that they are employed for the applications presented in Chapter 5.

3.4.2 Task Parallelism

These distributed data structures relate easily to a task-based parallelism approach, whereby each process obtains tasks from tuple space, carries out the task, and returns the result to tuple space. Instead of specifying individual data elements, we can create task tuples to specify the tasks which need to be completed, such as:

```
out("task", <task description>)
```

A process would retrieve this task, compute its result, and return the result to tuple space as follows:

```
in("task", ? task);
result = compute(task);
out("result", result);
```

The tasks completed by each process may be different or the same, depending on the specification of the task given in the task tuple.

3.5 Related Work

Tuple space systems have long been recognised as being an effective tools for parallelising coarse-grained, or embarrassingly parallel applications where there are very few dependencies between each parallel task. Such applications tend to require less communication, and therefore do not suffer from the overhead introduced by transmitting data over the network, and from the computer hosting tuple space having to service requests.

However, this overhead imposes a serious limit to the scalability of a distributed tuple space system when it is used to run applications which are more finely-grained, and as such have greater dependencies between tasks and more frequent communication. Inevitably, a centralised tuple space implementation will become overwhelmed trying to service requests as the number of processes in the system and/or communication frequency becomes large. Therefore, it is desirable to decentralise the tuple space, and to spread the load of servicing requests between those computers which host each part of the distributed space.

Distribution of the tuple space presents several challenges which must be addressed in order to maintain, and hopefully improve, the performance of parallel applications which utilise it. Namely, the time taken to search for and retrieve tuples from the space must not increase too dramatically compared to a single centralised space. To this end, tuples need to be stored in the distributed space in such a way that requests are, as evenly as possible, spread amongst the computers which host each part of the space.

These scalability issues have most often been addressed by adding extra tuple spaces to the system, the idea being that the load of requests can be spread amongst each individual tuple space and therefore increase peak performance. For example, in a JavaSpaces system, it is straightforward to start additional JavaSpace services, creating a system with multiple instances of these services. However, the major drawback of this approach is that each of these services exist completely independently. They do not coordinate their actions to provide optimal servicing of client requests, and therefore it becomes the programmer's responsibility to add extra functionality to the application in order to utilise the additional services.

A more beneficial and appropriate approach to this situation would be to aggregate all of the JavaSpace services into a single, distributed space. This distributed space would appear to the programmer to behave as though it were a single, centralised space; that is, the distribution of the space should be completely transparent at the application level. A distributed space such as this would result in less complexity in the application code, and, hopefully, a more adaptable and scalable distributed system.

There have been quite a few implementations of the Linda model which have attempted to address similar issues. Some of the more notable systems are explored in the remainder of this section.

3.5.1 MTS-Linda

Multiple Tuple Space Linda (Nielsen & Slrensen 1994) (usually abbreviated to MTS-Linda) was one of the earliest attempts to add multiple tuple spaces to the original Linda model. MTS-Linda incorporates tuple spaces which are treated as first class objects, and can be manipulated by the programmer to fulfil applications' specific needs. The use of multiple tuple spaces allowed data (represented as passive tuples), and processes (active tuples) to be grouped and manipulated as a whole. As tuple spaces are treated as first class objects, each tuple space is simply conceptualised as a "local data structure within a process" (Nielsen & Slrensen 1994, p. 12), which goes some way towards raising the level of transparency of the system's distribution.

Tuples which reside in other (non-local) tuple spaces may also be accessed, provided they are within the "context tuple space" of the process making the access request. That is, multiple tuple spaces may belong to the same context, and processes may opt to retrieve tuples from either its local tuple space, or from a tuple space contained in the same context. In this way, multiple tuple spaces are added in a hierarchical manner, rather than the flat, or disjoint way they have been incorporated in some other systems.

Another attempt at implementing multiple tuple spaces for Linda was by (Rowstron & Wood 1996), who adapted the Linda model to networks of heterogeneous workstations. This system did not propose a new way of adding multiple tuple spaces, but simply assumed that they existed. The main contribution this system made was the addition of new tuple space access primitives, namely bulk retrieval operations `collect()` and `copy-collect()`. The former operation moves a set of matching tuples from one tuple space to another, and the latter performs a similar function, except matching tuples are copied from one tuple space to another (Rowstron 1998). This implementation classifies tuple space as either local or remote, the main difference being that tuples stored in a local tuple space are not accessible by remote nodes in the system, whereas those stored in a remote tuple space do not have this restriction. Further, local tuples are stored locally, in the local processes address space, whereas remote tuples are stored on remote *tuple space servers*, which generally reside on separate, dedicated nodes on the network. The decision as to whether a given tuple is classified as being local or remote is performed dynamically at runtime by the system kernel.

The bulk operations allowed the movement of multiple tuples using only a single operation, whereas in the original Linda model this would have required multiple invocations of the tuple spaces' access operations. This factor, along with the optimisation of the locality of stored tuples, allowed the system to make more efficient use of the network, and to realise some significant performance gains compared to traditional implementations (Rowstron & Wood 1996, pp. 7-13).

The two systems discussed in this section identified the fundamental issues which need to be address when adapting the Linda model to a distributed environment: some form of logical integration of tuple spaces as in MTS-Linda, and runtime optimisation of tuple locality and efficient network usage in the case of Rowstron & Wood's system. As we will see in later chapters, Tupleware also addresses these issues, however in a slightly different way than these two systems.

3.5.2 Jada

Jada (Ciancarini & Rossi 1996) is a toolkit for the Java programming language intended to add distributed tuple space functionality to Java for a wide area, Internet scale environment. It is designed for distributed multi-user applications, such as groupware and eCommerce, rather than to provide a high-performance computing resource (Ciancarini & Rossi 1996, p. 2). However, it is worth discussing here due to it being one of the early attempts to add multiple tuple spaces to a Java-based tuple space platform.

As mentioned, one of the motivations behind Jada was to develop a Java platform to Internet applications, in particular for the World Wide Web. It was envisaged that Jada processes could execute as Java applets inside a web browser, and use the Jada components to communicate with other Jada applets (Ciancarini & Rossi

1997, p. 1).

Jada consists of a library which contains the classes needed to allow distributed processes to coordinate their activities via one or more tuple spaces. Each tuple space is represented by an `ObjectServer` object, and processes can access these objects, via the network, using an `ObjectClient` object (Ciancarini & Rossi 1997, p. 8). Tuple spaces are first-class entities in Jada, and can be explicitly created and manipulated by Jada processes. Therefore, Jada is an example of a disjoint multiple tuple space implementation, where each tuple space has no logical relationship with other tuple spaces. Furthermore, each tuple space must be explicitly referenced within the program code when it is to be accessed, and thus the existence of multiple tuple spaces is not transparent in any way to the application programmer. However, at the time it was developed it addressed some limitations of the Java platform in distributed environments, and preceded the development of the Java Spaces platform (discussed in Section 3.5.6).

3.5.3 LIME

LIME (Linda in a Mobile Environment) is a distributed tuple space system designed for use with both mobile software agents and physically mobile devices. It consists of a Java-based middleware which provides a coordination layer between hosts, and is specifically targetted at ad-hoc mobile computing environments (Murphy 2007). Due to the nature of the systems for which LIME is intended, its primary goal is to achieve high availability and fault-tolerance rather than high performance. Nonetheless, it has some notable features which are relevant to the system being presented in this thesis.

LIME is based on the Linda coordination model, and is notable for its use of multiple tuple spaces which are transiently shared between hosts in a LIME system. A host, in this case, may be either a software agent or a physical device. It is assumed that these hosts are mobile in some way; for example, agents which may relocate between physical computers, or mobile devices which will physically move from one location to another. These hosts are assumed to be connected to a network of some description, however, due to the hosts' mobility, network connectivity between hosts is often ad-hoc and not all hosts may be able to communicate at a given point in time.

Each LIME host maintains its own local tuple space, in contrast to Linda's single global tuple space. For hosts which are able to communicate via the network, each of their local spaces are transiently and transparently shared to produce a *federated tuple space*, allowing the group of all connected hosts to share the contents of their individual local spaces (Murphy, A 2006). The local spaces which make up the federated tuple space may change over time as host connectivity alters.

LIME is relevant to the system being presented in this thesis in that it utilises the idea of tuple locality and the sharing of tuple spaces of neighbouring, or connected, agents. However, the intended applications of LIME and Tupleware are quite different.

3.5.4 SwarmLinda

SwarmLinda (Menezes & Tolksdorf 2003) is a distributed tuple space implementation which utilises techniques based on swarm intelligence to increase the scalability of systems based on the Linda model. In particular, its tuple search and retrieval algorithms are based on the collective intelligence displayed by ant colonies and an attempt to make Linda-style system more suited for use in open distributed systems.

Biologically-inspired systems such as SwarmLinda are characterised by agents (in this case, ants) acting individually, but whose individual actions combine to exhibit a collective intelligence. In SwarmLinda, these agents ”act extremely decentralised” and perform their actions ”by making purely local decisions and by taking actions that require only a few computations.” (Charles et al. 2004, p. 3). It is for these reasons that these techniques were identified by the SwarmLinda authors as being an interesting approach to the Linda scalability problem.

A SwarmLinda system consists of a collection of participating nodes connected in a decentralised fashion. Nodes may store tuples, and communicate directly with any other nodes to which they are connected. However, the operation of a SwarmLinda system is not carried out by these nodes, but rather the ”ant” agents which may exist on each node, and also travel between nodes. The activities undertaken by agents include the tuple distribution and tuple retrieval; SwarmLinda systems attempt to dynamically store tuples in the most optimal location on the network in order to increase the efficiency of its later retrieval.

Firstly, SwarmLinda's tuple distribution algorithm is used when a new tuple is created, and needs to be stored on some node in the network. Both the distribution and retrieval of tuples is influenced by the characteristics of other tuples stored on each particular node (Charles et al. 2004, pp. 4-5). An agent takes the tuple which is to be stored, and traverses nodes until either a) it finds a node which stores tuples with similar characteristics to the one it is attempting to store, and decides to store the tuple there, or b) the agent becomes ”tired” (akin to a time-to-live value which guarantees that a tuple will eventually be stored), and decides to store the tuple at the node at which it is currently located. The end result being that there will be a clustering effect for stored tuples, whereby small groups of neighbouring nodes in the network will store tuples which have similar characteristics.

In keeping with the ant swarm metaphor, groupings of similar tuples emit a ”scent” which is used in the tuple retrieval process by acting as a guide for the system’s ant-like agents. When a tuple retrieval operation is invoked, a template-ant agent is dispatched to search for a matching tuple (Charles et al., 2004, p. 6). As the agent arrives at each node, it searches that particular node for a matching tuple. If one is found, the agent returns it to the requesting process. However, if one is not found, then the template-ant determines the scent, or characteristics, of the tuples stored at the current neighbouring nodes, and based on that information, the agent determines a ”fitness value” used to decide which node to visit next to continue the search. As with tuple distribution, there are safeguards to ensure that tuple search does not continue indefinitely, in cases where a matching tuple does not exist.

SwarmLinda presents a novel and original approach to solving some of the inherent problems in designing a distributed tuple space. It uses the idea of tuple locality to improve the efficiency of search, and uses deliberate tuple placement strategies to further aid in this regard. The ideas discussed above have been implemented, and the software is described in Charles et al. (2004b). However, at the time of writing there are no published performance results for the SwarmLinda software, and so no conclusions can be made as to the effectiveness of these techniques.

3.5.5 WCL

WCL (Rowstron, 1998) is a coordination model and runtime system developed for use with geographically distributed software agents in wide area network environments which exhibit high latency, such as the Internet. WCL makes use of the Linda-style tuple space, however it introduces some additional operations that extend the Linda model to allow for different styles of process interaction.

The WCL runtime system is specifically designed to support geographically distributed applications, but is not intended as a high-performance parallel computing platform (Rowstron & Wray, 1999, p. 1). However, WCL is relevant to this thesis due to the fact that it addresses some of the issues encountered when adapting the tuple space to distributed environments, and its runtime system uses techniques to dynamically tune the system for optimal performance without placing any extra burden on the application programmer (Rowstron & Wray, 1999, p. 2). Like Linda, the operations included in WCL are intended to be embedded into an existing programming language, and it has successfully been incorporated into Java and C++ (Rowstron & Wray, 1999, p. 3).

A WCL system consists of tuple space servers, which host one or more tuple spaces, and a number of agents which use these servers to access the tuple spaces. The runtime system, or kernel, maintains control of the entire system, and consists of three distinct parts: a tuple management system, which controls individual tuple space servers, a control system, which oversees the entire collection of a system's tuple space servers, and finally an agent library, which allows individual agents to interact with the other components of the system (Rowstron & Wray, 1999, pp. 5-10).

WCL includes both synchronous and asynchronous version of common Linda operations such as `out()`, `in()` and `rd()`, and bulk retrieval operations `collect()` and `copy-collect()`. The semantics of the synchronous operations are the same as their Linda equivalents, however the asynchronous operations necessitate the inclusion of additional operations to receive the eventual response. As an example, consider the asynchronous `rd()` operation (`rd_sync()`). This operation simply dispatches the request, and returns a value, `reply_id`, which is used to uniquely identify the request. The point of control is now returned to the next instruction in the program. The response to the operation will arrive at the requesting agent at some point in the future, and so the process needs a way in which to check whether or not the response has arrived. This is achieved with one of either `check_async()` or `check_sync()`, both of which take `reply_id` as an argument. These methods will return the previously requested tuple if it is now available; if it is not available, `check_async()` will return null, whereas `check_sync()` will block until it does become available. The choice of whether to block or not is useful in that it gives the application the ability to continue computation if the requested value is not essential, or to block if it is.

A unique feature of WCL is its handling of tuple spaces. As previously mentioned, each tuple space server may maintain multiple individual tuple spaces, and agents communicate with the server to access the tuple spaces that it is hosting. However, due to WCL being targetted at high-latency WAN environments, it is possible for an entire tuple space to migrate between tuple space server in order to increase the efficiency of access to the space. Migrations may be triggered to reduce the latency between a tuple space and the agents that are accessing it, or, if several frequently accessed tuple spaces are hosted on one server, a tuple space may be migrated to another server as a load balancing mechanism (Rowstron & Wray 1999, pp. 10-11).

The results of these techniques shows that they are effective in meeting WCL's goals (Rowstron & Wray 1999, pp 13-17), and highlighted the potential performance gains which can be realised from the use of a distributed rather than centralised tuple space. However, the functionality of WCL does come at the expense of simplicity. As Table 3.1 shows , WCL includes a relatively large number of operations. While this does give the application programmer a high degree of control over the operation of the system, it is relatively low level, and would undoubtedly increase the complexity of the application code. This issue is addressed in Rowstron (1998, p.23), which compares the complexity of WCL to that of systems such as CORBA and Java RMI. While this is a valid comparison, and these systems do have a relatively high level of complexity, none of these come close to matching the elegant simplicity of the original Linda model.

Type	*Operation*
Access Primitive	`out_sync(), out_async(), in_sync(), in_async(), rd_sync(), rd_async(), check_async(), check_sync(), touch_sync(), touch_async(), cancel()`
Bulk & Streaming	`move_sync(), move_async(), comp_sync(), copy_async(), bulk_in_async(), bulk_rd_async(), monitor()`

Table 3.1: Summary of WCL operations.

3.5.6 JavaSpaces

JavaSpaces (Sun Microsystems, 2003) is an implementation of the spaces paradigm from Sun Microsystems. Specifically, it is a service which forms part of the Jini distributed software architecture. It provides a stand-alone object space, called a JavaSpace, which can be utilised by any process in the system. The system of course may have more than one space, however each space is a separate entity and their respective roles in the system are not coordinated. Each application must contain the logic for utilising the available JavaSpaces infrastructure.

The fundamental unit of communication in a JavaSpaces system is an *entry* (Freeman et al. 1999, p. 22), and processes coordinate their activities by reading and writing entries to and from the JavaSpace service. A entry is simply any Java object which implements the **net.jini.core.entry.Entry** interface, shown in the following code listing:

```
public interface Entry extends java.io.Serializable {}
```

Any object which implements the Entry interface must provide a no-arguments constructor and all fields must be declared as public; these conditions are stipulated to allow template matching to be carried out. Given this background, the code listing below (Freeman et al. 1999, p. 77) shows an implementation of a semaphore entry suitable for coordinating access to a named resource in a JavaSpaces application:

```
public class SemaphoreEntry implements Entry {
    public String resource;

    public SemaphoreEntry() {}

    public SemaphoreEntry(String resource) {
        this.resource = resource;
    }
}
```

JavaSpaces provides three fundamental operations (Freeman et al 1999, pp. 28-36) which can be used by processes to interact with the space:

- **write()** places a copy of an entry object into the space. Equivalent to Linda's **out()**.
- **read()** retrieves a copy of an entry from the space. Equivalent to Linda's **rd()**.
- **take()** retrieves an entry, permanently deleting it from the space. Equivalent to Linda's **in()**.

Both of the JavaSpaces' **read()** and **take()** operations make use of associative lookup in the same vein as Linda's **rd()** and **in()**. An Entry object may be used as a template, where some fields may have values defined and some may be left as **null**. This object is then used as a parameter to **read()** and **take()**, and used to associatively match against entries in the space. If a matching entry does not exist in the space, then the operation will block until a matching entry becomes available. The alternative operations **readIfExists()** and **takeIfExists()** are non-blocking equivalents to **read()** and **take()** respectively, and will simply return **null** if no matching tuple exists.

An additional operation included in JavaSpaces is **notify()**, which allows the developer to use an event-based programming style. Using this operation, a process is able to register its interest in a future incoming entry to the space which matches a given template (Sun Microsystems, p. 17). When a matching entry becomes available in the space, the requesting process will be notified and will respond to the event in whatever way is defined in its related `RemoteEventListener` object.

A final feature of interest of JavaSpaces are transactions. Transactions allow a set of JavaSpace operations to be grouped together and executed in an ”all-or-none” manner; ie. either all of the operations will complete successfully, or if one fails, then the state of the space will be rolled back to as it was before the first operation was executed. This allows the space to be kept in a consistent state during complex interactions should an error occur midway through the operations included in a transaction.

Like most derivative implementations of the tuple space model, JavaSpaces is an effective platform for implementing a range of distributed applications and utilising common design patterns, as shown in Freeman et al. (1999). In particular, it has been shown in Atkinson and Malhotra (2003) to be ideally suited to the Master/-Worker style of parallelism, particularly coarse-grained parallel applications. For applications which are more fine-grained or tightly-coupled, JavaSpaces can experience scalability problems due to the increased communication demands inherent in these applications.

3.5.7 Other Java-based Linda implementations

The Java Spaces system has provided a reference framework which has been implemented in some commercial products including GigaSpaces (GigaSpaces 2008) and IBM's TSpaces (Lehman 1999). The operation of each is fundamentally the same as Java Spaces, and so won't be described in depth here, however both are aimed at enterprise environments, and each introduce some new operations to allow for different styles of interaction with tuple space. They also include features such as transactions, object-orientation, leases, and event notification capabilities.

Each of these systems support multiple tuple spaces in some form: TSpaces using specific operations which interact with each available tuple space, and GigaSpaces through the use of specific space-enabled objects. There is no logical relationship between the tuples spaces in the system, yet GigaSpaces does transparently hide their existence behind their own data structures.

3.5.8 Scope

Scope (Merrick & Wood, 2000) is a formal model for the addition of multiple tuple spaces to Linda-like systems. It aims to address the scalability problem of Linda, and also to increase the expressiveness of Linda-like operations so as to enable operations such as transactions, and prevent semantic limitations such as the multiple-read problem.[1] Most relevant to the research presented in this thesis, however, is

[1]The multiple-read problem occurs in tuple space-based systems when one wishes to read, using `rd()`, all tuples in tuple space that match a given template. There is no way to prevent the retrieval of

the generalised way in which Scope handles the issue of multiple tuple spaces, in particular its idea of ”overlapping” tuple spaces.

Multiple spaces have traditionally been added to tuple space systems in one of two ways: by nesting spaces hierarchically, as in MTS-Linda, or by simply adding disjoint spaces which have no logical relationship, as in JavaSpaces (Merrick & Wood, 2000, pp. 1-3). These approaches are illustrated in Figure 3.2.

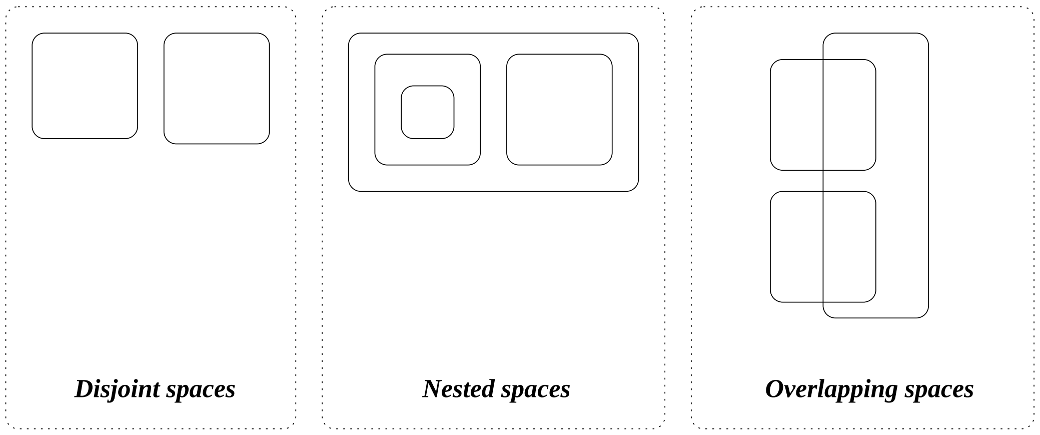


Figure 3.2: Disjoint, nested, and overlapping tuple spaces.

Scope presents a generalised approach to the addition of multiple spaces, introducing the idea of overlapping tuples spaces. This allows some parts of each space to be shared, and other parts to be separate. In concrete terms, tuples are able to belong to more than one space at a time. Essentially, each ”portion” of tuple space is represented by a named scope, and these portions can be combined and arranged based on defined scope operations. These operations are based on the set operations union, complement and intersection, and can be used to define tuple membership to one more more scopes. The expressiveness of Scope allows it to implement hierarchical and disjoint tuple spaces in addition to overlapping spaces.

A scope itself is a set containing sets of names (Merrick & Wood 2000, p. 3), and a scope matching rule is introduced such that scopes match if they share at least one element in common (that is, the intersection of the two sets is not empty). These scopes are incorporated into the primitive Linda operations so that every operation (`out()`, `in()`, `rd()` etc) take a scope parameter, which specifies on which scope the operation should be performed. The scopes provided to the operations also effect the normal Linda template matching rule, in that not only must a template match a tuple, but the scopes being used must also match. For example, if an `out()`

the same tuple multiple times, as `rd()` is a non-destructive operation, and therefore it is impossible to guarantee that all matching tuples have been retrieved.

operation acts on scope *A*, and a subsequent `in()` operation acts on scope *B*, the retrieval operation can only be successful if *A* and *B* also match. In this way, scopes partition the global tuple space, and each process can only ”see”, or have access to, certain partitions.

A concrete implementation of Scope is presented in Merrick (2003), which presents how the formal model described above might be implemented in both a local and distributed manner. However, no performance results are available for any Scope-based implementation, and no subsequent research seems to have been done at the time of writing. Nonetheless, Scope provides a useful framework from which to draw ideas on how multiple tuple spaces may be logically integrated into a singular distributed space. Several of these ideas influence the operation of the Tupleware system, as we will see in later chapters.

3.6 Performance & Scalability Issues

Many tuple space implementations have suffered from scalability issues when used with real-world, non-trivial applications. The catalyst for this research was the author’s experiences with using JavaSpaces for a network-of-workstations style system, which is discussed in (Atkinson 2003) and (Atkinson & Malhotra 2004). This work clearly showed some of the scalability limitations of the JavaSpaces platform, in particular with tightly-coupled applications such as parallel sorting (shearsort was one of the applications used to test the performance of the resulting system). It was concluded that one of the major causes of the limited scalability was the use of a single, centralised tuple space; hence the motivation to investigate how to support multiple spaces or implement a distributed space.

A survey of the scalability of various Linda implementations in Java was performed by Wells, Chalmers & Clayton (2004), who evaluated the scalability of TSpaces, JavaSpaces, GigaSpaces, and also the authors’ systems eLinda1 and eLinda2. They evaluated these systems using a ray tracing application executed on a network of commodity workstations. The results of these experiments were that:

- the speedup of the majority of the systems peaked at 6-7 worker nodes,
- the highest speedup obtained was by GigaSpaces, at just over 3.5 on 7 nodes, and
- of the other commercial offering, JavaSpaces obtained just under a speedup of 3 (6 and 7 nodes) and TSpaces slightly under 3.5 (6 nodes).

All speedups are relative to the time taken by a single worker node, and the application grid size was 200x200.

The authors also noted that at over eight nodes for JavaSpaces "the system became unstable", and that the performance of GigaSpaces was "erratic" due to virtual memory issues. Indeed it should be pointed out that the workstations used in these experiments had only 32MB of memory, barely enough to adequately host a Java runtime. Nonetheless, the correlation of results for the systems profiled suggest a common limitation in terms of their speedup, and compare favourably to the results presented for Tupleware.

Another study on tuple space performance for scientific applications was carried out by Noble & Zlateva (2001), which focussed on the viability of using an unmodified JavaSpaces platform for the execution of two parameterised applications: a Pi digit finder and a particle shielding application. This research is particularly relevant to this thesis as it provides an empirical study of JavaSpaces using applications with similar characteristics to the ones presented here.

In the first instance, the authors observe the difficulties of using a tuple space system for typical parallel algorithms including prime number generation (using the algorithm presented in Carriero & Gelernter (1990) and grid-based numerical applications such as a Laplace equation (and the ocean model presented in this thesis). Using calculations of the respective IO times of the original Linda and JavaSpaces, and the runtime of a sequential C solution, they conclude that tuple space platforms typically provide runtimes of an order of magnitude greater than sequential solutions. They then venture that the true benefits of a tuple space solution will be found in applications with a high computation/communication ratio and where simplicity and loosely-coupled interaction is desired. These conclusions and observations are consistent with the reasoning presented in this thesis, and somewhat validates the approach taken given that the ocean model achieved increased performance.

Secondly, the authors investigate the performance of the Pi digit finder and the particle shielding application; each of these applications have minimal communication requirements, similar to the modified quicksort presented here. These applications both achieve significant speedup, particularly the particle shielding application, which is to be expected given their characteristics. Again, these results are in line with those presented for the modified quicksort application.

These studies would suggest that there is still vast scope for improvement of the scalability of tuple space systems, which is precisely what this research hopes to achieve.

3.7 Summary

This chapter has provided an overview of the related work in the field of coordination models, and in particular the area related to the addition of multiple tuple spaces. The approaches covered are quite diverse, and are influenced by the motivation for using multiple tuple spaces in the system. For example, the focus of LIME is to support mobile applications, and so its concept of federated tuple spaces maps well to these scenarios. JavaSpaces is aimed at an enterprise environment, and as such has mechanisms to ensure data consistency and availability, but does not address high performance or the transparent integration of multiple tuple spaces.

As the systems covered in this chapter also clearly show, despite the fact quite a few models have been suggested, there is very little in the way of concrete implementations or performance data available for these models. The comparison of other Java-based Linda implementations discussed in Section 3.6 showed that there is a general limit to the scalability of systems such as these, and this is one area that this research aims to address.

Chapter 4

Tupleware Implementation

This chapter details the aims, design, implementation, and operation of the Tupleware system.

4.1 Aims

The aims of this research were described in Chapter 1, but are stated here formally as they were used to inform the design decisions which are described in subsequent sections. The four core aims of the Tupleware system are as follows:

- **Scalability:** In designing a tuple space for a distributed computing environment, one must allow for systems where the number of participating processes becomes large. It is imperative that the system can scale to support these cases, and that the addition of extra processes does not negatively effect the overall operation of the system. Furthermore, the system should be able to scale in terms of problem size. An increase in the amount of work which needs to be performed should only result in a relative increase in execution time.

- **Performance:** The system aims to provide scientific and numerical applications with a viable platform for the parallel execution on a computing cluster. Viability is determined by whether applications are able to gain an appreciable increase in performance when deployed using Tupleware.

- **Ease of programmability:** To the greatest extent possible, the operational details of the underlying Tupleware system should be transparent to the application programmer. This includes the existence of multiple tuple spaces and the location of stored tuples. The strengths of Linda-style coordination languages are their relative simplicity, and logical separation of communication and computational program code. We should strive to preserve these strengths in implementing any variants of the Linda coordination model.

- **Logical integration of distributed tuple spaces:** The main component of a Tupleware system is its distributed tuple space, which consists of multiple tuple sub-spaces. It is highly desirable that the spaces are logically related and self-managed by the underlying runtime system so that they appear as a single, unified tuple space to the application programmer.

These core aims guided the subsequent design and implementation of the Tupleware system, and all design decisions will be explained in terms of how they try to achieve these aims.

The following sections describe how Tupleware is designed and implemented. We begin at the lowest level, with the fundamental building blocks of the tuple space, before moving on to the question of how Tupleware distributes tuple space over the nodes in a cluster, and how the runtime system controls the operation of each node, and by extension, the whole system.

4.2 How these aims will be addressed

As previously stated, the aims of the system outlined above form the basis of the design and implementation of the Tupleware system. The overall approach will be to implement a distributed tuple space in such a way that tuples will be stored across the nodes in the cluster, so that we can gain performance gains by balancing the load across a number of nodes. We also wish that this distribution of the tuple space is transparent to the application programmer, and this is achieved by providing an interface through which the programmer can interact with tuple space without the concern of where tuples are physically located.

In the following section we will discuss in more detail the specific approaches which were used in order to address each of the system's aims.

4.2.1 Scalability & Performance

In order for Tupleware to provide a viable platform for distributed computing, it must give an application the opportunity to attain increased performance. Furthermore, the system must be able to scale so that the addition of nodes results in better performance, and also so that the problem size may be increased without unduly effecting the performance of the system.

These requirements imply that the use of Tupleware should aim to minimise any additional overhead due to its distribution over a cluster. Two main approaches have been used to this end: we attempt to store tuples locally (on the portion of the tuple space located on the producer-node), and we implement a probability-based search algorithm which is used to retrieve tuples that are stored remotely. The search algorithm makes use of the success or failure of previous remote requests, and based on this historical data targets its requests to those remote nodes which have the highest chance of being able to fulfil the request. The fact that tuples are stored locally when produced further aids in the accuracy of the requests.

The reasoning behind the search algorithm stems from the observation that, for the class of array-based applications for which Tupleware is designed, application processes which are processing neighbouring parts of an array will tend to communicate frequently, whereas those processes that are processing other parts of the array will tend to communicate rarely, if at all. By targeting our requests to those nodes which successfully fulfilled our previous requests, we minimise the occurrences of network communication, and, hopefully, minimise the overhead associated with the distributed of the tuple space.

4.2.2 Ease of Programmability & Logical Integration of Distributed Tuple Spaces

The other aims of Tupleware relate to its usability: namely, that it is relatively easy for the programmer to develop an application to run on the system. Due to the level of complexity inherent in distributing the tuple space across a cluster, we wish to hide this complexity from the programmer and this distribution to be transparent, and for the distributed tuple space to behave no differently than a single centralised tuple space such as Java Spaces.

In order to achieve this, the application accesses the underlying Tupleware system via a middleware layer, which contains the system logic for interacting with each distributed part of tuple space. The middleware is responsible for local tuple storage, and for retrieving tuples from other remote nodes in the cluster using the search algorithm outlined in the previous section. The programmer does not need to be burdened with the tuple storage locations, the location or number of the remote parts of tuple space. As we will see later in this chapter, the application programming interface is comparatively simple, and retains the same easy-to-understand semantics of the original Linda operations.

4.3 Basic Components

The system components covered in this section provide sufficient functionality for one to construct a non-distributed, multithreaded tuple space system. These components may be integrated with others in order to construct a complete distributed tuple space, as we will see in Section 4.4.

All source code listings that follow will include only the method headers, apart from specific instances when the implementation details are relevant to the discussion. Full code listings can be found in the appendices.

4.3.1 Tuples & Templates

The fundamental data object in a tuple space system is a tuple, which is used to encapsulate one or more data objects. A tuple has one or more fields, each of which contain a value. Fields should not contain any **null** values, and tuples are treated as immutable objects. Tuples are defined by the `Tuple` class, shown in the program listing below.

```
public class Tuple implements Serializable {
    protected Object[] fields;

    public Tuple(Object ... fields) {...}
    public int size() {...}
    public Object field(int i) {...}
    public String toString() {...}
}
```

Templates are used to perform content-based associative lookup on tuples, and are implemented in Tupleware as the `TupleTemplate` class, which inherits from the `Tuple` class.

```
public class TupleTemplate extends Tuple {
    public TupleTemplate(Object ... fields) {...}
    public TupleTemplate(Tuple t) {...}
    public boolean matches(Tuple tuple) {...}
    public String toString() {...}
}
```

`TupleTemplate` is similar to `Tuple` in that it encapsulates a set of data fields. However, unlike `Tuple`, some (or all) of these fields may be assigned **null** values, denoting wildcards which may match against any value during associative lookup.

Associative lookup involves the use of the `matches()` method, which determines whether a `Template` object matches a given `Tuple` object.

Together, the `Tuple` and `TupleTemplate` classes provide the fundamental building blocks of data storage and content-based addressing required in a tuple space system.

4.3.2 Local Tuple Space

A tuple space is implemented by the `TupleSpaceImpl` class, which provides the basic functionality required for tuple storage and lookup. These objects may be shared among threads in a multi-threaded process, allowing threads to communicate and coordinate their actions during their concurrent execution.

Threads interact with a `TupleSpaceImpl` object by invoking its Linda-style methods `out()`, `in()`, `rd()`, `inp()`, and `rdp()`, along with the additional bulk operations `rdAll()` and `inAll()`. These methods are semantically the same as for Linda, with the only difference being that tuple fields are not assigned to local variables for input operations, but rather a complete `Tuple` object is returned by these methods, or **null** if none can be retrieved by a non-blocking operation (refer to code listing below).

```
public class TupleSpaceImpl {
    private Hashtable<String, Vector<Tuple>> tuples;

    public TupleSpaceImpl() {...}
    public void out(Tuple t) {...}
    public void outAll(Collection<Tuple> tpls) {...}
    public Tuple in(TupleTemplate t) {...}
    public Tuple inp(TupleTemplate t) {...}
    public Vector<Tuple> inAll(TupleTemplate t,
                              int expected) {...}
    public Tuple rd(TupleTemplate t) {...}
    public Tuple rdp(TupleTemplate t) {...}
    public Vector<Tuple> rdAll(TupleTemplate t,
                              int expected) {...}
    private String generateKey(Tuple t) {...}
}
```

4.3.2.1 Tuple Storage & Lookup

As stated above, the main role of `TupleSpaceImpl`, apart from providing a Linda-style interface to the programmer, is to store tuples and allow these tuples to be retrieved. Tuples are stored in tuple space when the `out()` method is invoked. In the interests of being able to retrieve stored tuples efficiently, a hashtable is used to index tuples of similar content, with keys being generated from the first fields of the tuple to be stored. Each key is in turn associated with a vector of tuples with matching header information.

As an example, consider an application which processes a 2-D array, $A[0..n][0..m]$, of integers. Each element in the array would be represented as a tuple of the form $<$ ”*A*”, i, j, *val* $>$, where the fields of the tuple are, respectively, the array name, the row and column indices, and the value stored in the array element. Such a tuple would generate a key consisting of the array name, row, and column indices concatenated together. For example, element $A[3][4]$ would generate the string ”A34”, which would then be used as a key and associated with a vector containing the values of this array element for a predetermined number of previous timesteps. This scenario is illustrated in Figure 4.1.

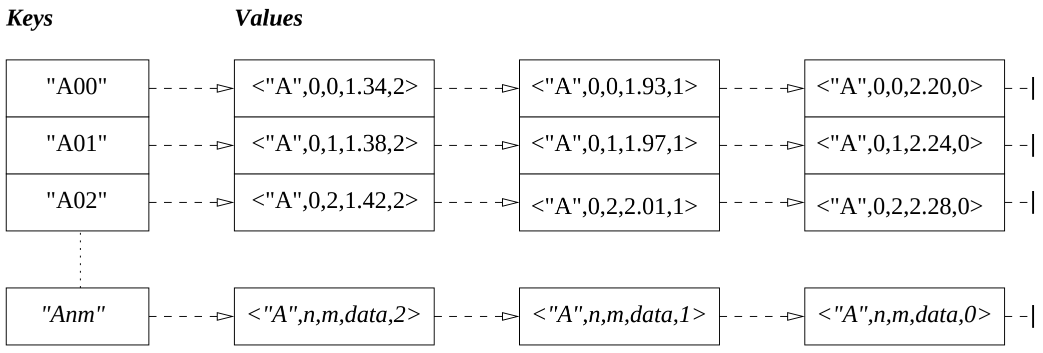


Figure 4.1: Local tuple storage using a hashtable.

Such a scheme was used due to the nature of the applications that Tupleware is targetted towards; these are generally timestepped array processing applications which require the current and single last previous iterations of values to be stored in tuple space. So, in practice, each vector object will only store a small number of elements, whereas the number of entries in the hashtable will be comparatively large. Therefore tuple lookups will benefit from the $O(1)$ lookup times provided by hash tables, making local tuple searches very efficient, especially so when taking into account the availability of concurrent access.

More detailed discussion on particular applications can be found in Chapter 5.

4.3.2.2 Local Space Synchronisation

In the context of the local tuple storage data structure described above, we should also explain how synchronisation is handled locally, within the partition of the tuple space maintained by each individual node.

One of the core functions utilised both during tuple storage and retrieval is `String generateKey(Tuple t)`, which is used to generate a key value for the hash table used for tuple storage. The full type signature of the structure is `java.util.Hashtable`
`Vector<Tuple»`, whereby each of these keys are associated with a vector of tuples, as explained in the previous section. The key generated by the function is based on the first three fields of array tuples, and only the first fields of all others (typically application-specific meta-tuples or those used in system control).

Now, when threads access this data structure, whether it is to retrieve or store a tuple, the accesses will need to be synchronised in such a way as to allow as high a level of concurrency as possible, without allowing deadlock or data inconsistencies to occur. Consider the following code snippet, which shows the steps involve in performing an `out()` operation:

```
public class TupleSpaceImpl implements TupleSpace {
    private Hashtable<String, Vector<Tuple>> tuples;

    // other variable declarations, constructors omitted.

    public void out(Tuple t) {
        Vector<Tuple> vals = tuples.get(generateKey(t));
        if(vals == null) {
            vals = new Vector<Tuple>();
            tuples.put(generateKey(t), vals);
        }

        synchronized(vals) {
            vals.add(0, t);
            vals.notifyAll();
        }

        synchronized(tuples) {
            tuples.notifyAll();
        }
    }

    // other method definitions...
}
```

The steps involved can be summarised as follows:

1. Attempt to retrieve the vector of tuples associated with the given tuple's key.

2. If there is no existing associated vector, then we create one and place the tuple to be written into it.

3. The accessor thread will now obtain the object lock for the associated vector, add the tuple, and finally notify all other waiting threads before releasing the lock.

4. Finally, we notify all threads which may be waiting to obtain the lock for the `tuples` hash table. This is required for threads which may be blocking during an `in()` or `rd()` operation.

The synchronisation involved in the `out()` method is, for the most part, required so that other threads which have blocked while performing a blocking retrieval operation may be woken. It may be the case that the tuple being written to the space provides a match for a tuple being requested, and so all blocking threads must be notified when the `out()` operation completes.

The `outAll()` operation is for all intents and purposes identical to `out()`, however it performs a `notifyAll()` for each vector "bucket" a tuple is placed into.

Next we cover the tuple retrieval operations, all of which are generalised into two methods: `findTuple()` (for retrieving a single tuple), and `findAllTuples()` (for bulk retrieval).

Firstly, the relevant code segment for `findTuple()` is listed below:

```
private Tuple findTuple(TupleTemplate t,
                        boolean remove,
                        boolean block)
{
    Vector<Tuple> vals = tuples.get(generateKey(t));

    while(vals == null) {
        if(block) {
            synchronized(tuples) {
                try {
                    tuples.wait();
                } catch(InterruptedException e) {
                    return null;
                }
            }
        } else {
            return null;
        }
```

```
            vals = tuples.get(generateKey(t));
        }

        synchronized(vals) {
            Tuple result = null;

            for(;;) {
                for(Tuple i: vals) {
                    if(t.matches(i)) {
                        result = i;

                        if(remove)
                            vals.remove(result);

                        return result;
                    }
                }

                if(block) {
                    try { vals.wait(1000); }
                    catch(InterruptedException e) {
                        return null;
                    }
                } else {
                    return result;
                }
            }
        }
    }
```

The semantics of this method are such that it will return the first tuple found which matches the given template; if the `remove` parameter is true, then the matching tuple will also be removed from the space, otherwise only a copy will be returned. If a matching tuple is not found, then what happens depends on the `block` parameter; if true, then the accessor thread will block until a matching tuple becomes available, otherwise, a `null` value will be returned. If a blocking thread is interrupted at any time, a `null` value will also be returned.

Synchronisation constructs are used in two instances in this method. Firstly, in the case where there is no associated vector for the given template and blocking behaviour has been specified, the accessing thread will synchronise on the `tuples` object and block until notified by another thread performing a tuple production operation. At this point it will attempt again to obtain a reference to the associated vector.

The second use of synchronisation in this method occurs when a reference to an associated "bucket" vector has been obtained, and must be searched for a matching tuple. All searches are synchronised on the vector itself, and once again, if no matching tuples are found then the thread will wait on the vector object until notified by another thread, if blocking has been specified.

The final operations which should be mentioned are those which are used for bulk tuple retrieval. Both the `inAll()` and `rdAll()` methods utilise the `findAllTuples()` method, which performs a search of potentially the entire tuple space, retrieving a specified number of matching tuples. Each of these methods utlilise the `findAllTuples()` method, which searches the local partition of the space and returns between zero and a specified number of tuples (specified by the `expected` method parameter).

In summary, the synchronisation constructs used within each partition of the tuple space are used minimally, due to our desire to achieve the maximum level of concurrent access possible, and it is the access to each vector "bucket" being serialised. However, different "buckets" are able to be accessed concurrently by different threads and/or remote nodes. In practice and in the context of the applications being presented, this means that threads reading from or writing to the same array elements will serialise their accesses. All other array elements may be accessed concurrently. The other main usage of synchronisation is for the notification of blocking threads, which is obviously required.

A final important consideration is how these synchronisation mechanisms affect the application programmer. It is desirable for the synchronisation described above to be as transparent as possible, so that the distribution of the space and the parallelism of the application being implemented is able to be expressed implicitly rather than have to be explicitly specified in the application code. However, as we can see from the description of the local tuple space synchronisation constructs above, access to each individual array element or tuple in the space is strictly serialised. Whilst we wish to avoid serialised access wherever possible, it is important to note that, for the applications for which this system is intended, quite often each array element will also incorporate a timestep value. As such, very rarely will it be the case that an element will need to be modified; the vast majority of the time these elements will be treated as read-only, with subsequently produced new values being tagged with an incremented timestep value. Therefore, we believe the impact on the application programmer to be minimal, in keeping with the goals of this research.

4.4 Distribution of Tuple Space

The Tupleware components discussed in Section 4.3 allow a programmer to construct a multi-threaded program in which threads communicate and interact via a shared local tuple space. The transition to a distributed system, with processes having individual address spaces and being hosted on individual nodes, requires some additional components to handle inter-process communication and algorithms for the retrieval of tuples located on remote nodes. This section covers the components which implement these features, namely the server and client objects which handle communication requests, and the runtime system, a middleware which provides transparent distribution of the tuple space.

4.4.1 Inter-node Communication

The Tupleware system includes the classes `TupleSpaceService` and `TupleSpaceStub` which facilitate communication across a network between processes executing on different nodes on a cluster. Communication between these components is relatively low-level, using TCP sockets, and their main role is to ensure point-to-point communications are efficient and that errors are caught and handled correctly.

4.4.1.1 Tuple Space Service

Firstly, the `TupleSpaceService` object runs as a service on each cluster node, and provides the generic tuple space services to remote nodes. Each service contains a reference to an instance of a local tuple space (discussed in Section 4.3.2) for each individual node, and carries out tuple space operations on the local tuple space on behalf of remote nodes.

The service runs as a separate thread, and is itself also multithreaded, allowing the concurrent servicing of multiple requests from remote nodes. When a connection has been established with a remote node, a new thread, defined by the `RequestHandler` class, is spawned, and this thread maintains the open socket connection and handles all subsequent requests from the remote node to which it is connected.

4.4.1.2 Tuple Space Stubs

Tuple space stubs are defined by the `TupleSpaceStub` class, and these objects act as a proxy for client communication with a tuple space service on a remote node. All communication in Tupleware is initiated by the client node, from the initial establishment of a socket connection through to interaction during application execution.

The stub class presents an interface containing all of the usual tuple space operations, and the invocation of these operations triggers a request being sent to the service running on a remote node (after establishing a socket connection if one does not yet exist). All of the low-level socket code is encapsulated in the stub class, and as such the actual input/output events are transparent to the client (specifically, the Tupleware runtime system, discussed in Section 4.4.2).

4.4.1.3 Communication Protocol: Request & Response Objects

As previously stated, stub and server objects communicate via TCP sockets, using the `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` classes. Each instance of communication involves a single request/response interaction, and the specific details of this communication are contained in objects defined by the `TupleSpaceRequest` and `TupleSpaceResponse` classes. These classes both implement Java's `Serializable` interface to enable them to be sent across the network.

A `TupleSpaceRequest` object simply specifies which type of operations is being requested, and contains any other data that is required in order for the operation to be completed. In the case of an `out()` operation, the tuple or tuples to be stored will be stored in the request object, whereas for a tuple retrieval operation, a tuple template will be included along with an integer specifying how many tuples are expected to be returned (this is required in the case of bulk retrieval operations). The relevant source code for `TupleSpaceRequest` is shown below.

```
public class TupleSpaceRequest implements Serializable {
    public static enum RequestType { OUT, IN, INP, RD, RDP,
        INALL, RDALL, OUTALL };
    private RequestType type;
    private Tuple tuple;
    private Vector<Tuple> tpls;
    private Integer expected;

    public TupleSpaceRequest(RequestType type, Tuple tuple)
        {..}
    public TupleSpaceRequest(RequestType type, Tuple tuple,
        int expected) {..}
    public TupleSpaceRequest(RequestType type, Vector<Tuple
       > tpls) {..}
    public Tuple getTuple() {..}
```

```
        public Vector<Tuple> getTuples() {..}
        public RequestType getRequestType() {..}
        public int getExpected() {..}
    }
```

A `TupleSpaceResponse` object is sent from a tuple space service to a stub in response to a request. The response object contains a status field specifying whether or not the requested action was successful, and may also contain any tuples which were been requested as part of a tuple retrieval operation. The relevant `TupleSpaceResponse` source code is shown below.

```
    public class TupleSpaceResponse implements Serializable {
        public static enum Status { SUCCESS, ERROR, TIMED_OUT
           };
        private Status status;
        private Tuple tuple;
        private Vector<Tuple> tuples;

        public TupleSpaceResponse(Status status) {..}
        public TupleSpaceResponse(Status status, Tuple tuple)
           {..}
        public TupleSpaceResponse(Status status, Vector<Tuple>
           tuples) {..}
        public Status getStatus() {..}
        public Tuple getTuple() {..}
        public Vector<Tuple> getTuples() {..}
    }
```

In summary, the request and response objects implement a simple communication protocol which is used to communicate data (in the form of tuples or tuple templates) between stubs and tuple space services. These objects also contain some additional status fields which indicate whether any errors have occured during the request-response cycle. We discuss how these are used further in the sections that follow.

4.4.1.4 Stub/Service Interaction

Tuple space services and stubs engage in two main forms of communication: initial establishment of a socket connection, and then subsequent request-response communications thereafter. The steps involved in establishing a socket connection between a stub and a service are as follows:

1. The tuple space service, when created, will open a server socket and listen for incoming connections.

2. A tuple space stub, upon invocation of one of its tuple space operations, checks to see whether a connection with its associated tuple space service has previously been established. If it has, then nothing more need be done. If it hasn't, then one must be established.

3. The tuple space stub creates a new socket, and attempts to connect to the relevant tuple space service.

4. The tuple space service responds to the connection request by spawning a new request handler thread (defined in the `RequestHandler` class). This thread is passed a reference to the resultant socket, and will handle all subsequent communications with the corresponding stub.

5. The connection has now been established.

These steps for establishing a connection between stub and service generally need to only be performed once, being the first time that a process needs to communicate with another. These connections are client-server in style, with the stub acting as a client to access the service provided by the remote tuple space service. In order for true two-way communication to take place, each process involved must initiate this connection process, so we end up with each node having a stub object which is connected to the other node's tuple space service. For an illustration of this, see Figure 4.2 on page 60.

Once a connection is created, it is left open until the termination of one of the connected pair of application processes. The only exception would be when a network error occurs which breaks the connection, in which case it would need to be re-established the next time the processes needed to communicate.

The other form of communication between processes, apart from the initial creation of socket connections, is for one process to invoke tuple space operations on the other process' tuple space. The steps involved in tuple space stub/service interaction are summarised as follows:

1. A tuple space operation is requested by invoking the relevant stub method.

2. Stub creates new request object, specifying the type of operation being requested and any additional data that is required.

3. The request object is sent across the network to the service, and the stub waits for a response.

4. The tuple space service (or more specifically, the request handler thread), upon receipt of a request object, determines the type of operation that is being requested, and extracts the included data relevant to that operation.

5. Using this information, the request handler thread performs the requested operation on its local tuple space. This may be a storage operation, such as `out()`, which will result in a new tuple being stored in the process' local tuple space. Or it may be a retrieval operation, such as `rd()`, in which case the local space will be search for a matching tuple, and if one is found then it will be retrieved so it can be sent back to the requesting process.

6. The request handler thread now prepares a response to send back to the requesting process. This response takes the form of a `TupleSpaceResponse` object, which encapsulates any data that is required as part of the requested operation. For example, a retrieval operation will result in the response object being used to store the retrieved tuple (or possibly **null**, if no matching tuple could be found). The response object also contains a **status** field indicating whether the operation encountered any errors.

7. Once the response object has been created and its data fields assigned, it is now sent back to the requesting stub. The request handler's role in this interaction is now complete, and it will now wait for another incoming request.

8. The stub object, on receiving the response from the remote service, inspects the response object's status field to ensure that no errors have occurred during the execution of the operation. If the requested operation was a tuple retrieval operation, the stub then extracts the returned tuple from the response object, and returns it to the application layer.

9. The interaction is now over, and the stub remains idle until the application again invokes another of its methods.

4.4.2 Runtime System

The final component of the distributed tuple space is the Tupleware runtime system. The runtime system acts as a middleware, providing a platform upon which applications can be developed, and integrates and organises the Tupleware components discussed in previous sections in order to implement the core logical functionality of the system. It provides the API used by applications to interact with the underlying system, and also implements the algorithms used for tuple search and retrieval.

The runtime system is the only component of Tupleware that an application need interact with; processes in Tupleware generally act as producers and consumers of tuples, and these interactions are all performed via the runtime system. The runtime system presents an interface to the application layer which contains all of the tuple space operations, and is shown in the code listing below.

```
public class TupleSpaceRuntime {
    private final InetSocketAddress GTS_ADDRESS;
    public TupleSpaceStub gts;
    public TupleSpaceImpl ts;
    private TupleSpaceService service;
    private Vector<TupleSpace> remoteSpaces;
    private boolean isGlobal;

    public TupleSpaceRuntime(int port, boolean isGlobal)
       {..}
    public void start() {..}
    public void stop() {..}
    public void out(Tuple t) {..}
    public void outAll(Vector<Tuple> tuples) {..}
    public void outEach(Tuple t) {..}
    public void outRand(Tuple t) {..}
    public Tuple in(TupleTemplate t) {..}
    public Tuple inp(TupleTemplate t) {..}
    public Tuple rd(TupleTemplate t) {..}
    public Tuple rdp(TupleTemplate t) {..}
    public Vector<Tuple> rdAll(TupleTemplate t, int
       expected) {..}
    public Vector<Tuple> inAll(TupleTemplate t, int
       expected) {..}
    public void register() throws IOException {..}
}
```

Every process in the system has exactly one instance of the Tupleware runtime, which handles that process' local tuple storage, and retrieves requested tuples from other processes. Likewise, each runtime object contains a single instance of the tuple space service, along with a stub object for communicating with each other node in the system. The complete structure of this layered architecture is illustrated in Figure 4.2.

The operations provided by Tupleware give the application developer a high degree of flexibility as to how their application behaves, however this is not at the expense of simplicity. The semantics of each operation are relatively straightforward for anyone familiar with standard Linda-style operations.

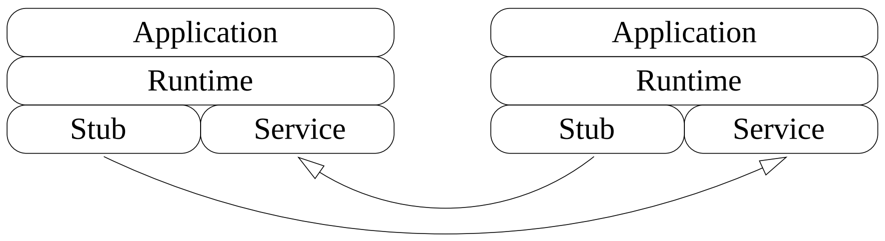


Figure 4.2: Architecture of a Tupleware system.

In addition to providing an interface for application components to communicate, the runtime system contains the logic for searching for, and retrieving, tuples from remote nodes. This function is fundamental to ensuring the efficiency of the system, and minimising the overhead introduced by the distributed execution of an application. Section 4.5 describes in detail how each of the operations are performed, and explains the reasoning behind the approaches which were adopted.

4.4.3 Application Processes: Master & Workers

The Master is the process which submits a task to be executed by the system. Briefly summarised, its role is to submit a task to be executed by the Worker processes, and to reassemble the results of the computation once it has been completed. The role of Worker processes is to take this submitted task, and to execute the task in parallel with other Workers processes in the system.

The definitions of the Master and Worker processes are the only responsibility of the application programmer; the remaining components of the system are generalised in such a way that any suitably implemented Master/Worker process combination may use them without modification. Each of these processes has their own instance of a Tupleware runtime to enable their participation in the system. Specific implementations of these processes in relation to concrete applications are discussed in Chapter 5.

4.4.4 Summary

This section has given an overview of the various Tupleware components which combine to provide a platform for the distributed execution of applications. We have described how the various components are assembled into a layered architecture, with the system logic encapsulated in the runtime system, which in turn controls the operation of the lower-level tuple space service, local tuple space, and tuple space stubs.

The following section describes how the Tupleware runtime system carries out the services required to provide this platform, including the semantics of the operations included in its API, and the algorithms that were implemented to ensure that these services are performed in an efficient and scalable way.

4.5 Runtime System

The main role of Tupleware's runtime system is to enable the communication between distributed application processes, and to do so as efficiently as possible. As we discussed in Section 4.4, the actual low-level network communication is carried out by the tuple space stub and service components. The runtime system utilises these components, and implements the algorithms for searching for and retrieving tuples which are being stored by processes executing on remote cluster nodes.

In this section we will discuss the application programming interface (API) exposed by the runtime system for use by applications, and detail how each of the operations in the API execute in the context of the system components discussed in the previous section.

4.5.1 Runtime System API

The Tupleware API includes all of the original Linda operations (discussed in Chapter 3), with the exception of `eval()`, which was excluded in favour of Java's native thread creation mechanisms. The other standard operations are included: `in()` and `rd()`, their non-blocking variants `inp()` and `rdp()`, as well as `out()` for tuple creation.

In addition to the standard Linda operations, several bulk operations were implemented to allow multiple tuples to be either produced or consumed at a time.

The first of these is `outAll()`, which takes a collection of tuples as its argument, and places them into the local tuple space of the invoking node. The reason for its inclusion was that it was observed that application code often contained multiple invokations of `out()` at a time, often, for example, when a master process was producing task tuple to be executed, or when worker nodes were sharing boundary values. The inclusion of this simple operation gives the benefit of reduced application code complexity, and so was deemed a worthwhile inclusion.

The second bulk tuple production operation is `outEach()`; its semantics are that it takes a single tuple argument, and writes this tuple to the local tuple space of every other node in the system. It was included in the API as a neccesity: a common pattern in tuple space systems is to use a *poison pill* (specially tagged tuple) to signal to worker nodes that they should terminate. In order for the system to shutdown gracefully, it is necessary for worker nodes to terminate at a reasonably uniform time, and `outEach()` facilitates this by allowing the master node to deliver a poison pill to each worker simultaneously.

Finally, the bulk tuple retrieval operations `inAll()` and `rdAll()` were included in the API. They allow for multiple tuples to be retrieved from local and/or remote partitions of tuple space while only carrying out a single search across the cluster. This increases the efficiency of retrieving multiple tuples, as the alternative would be to use multiple invokations of either `in()` or `rd()`, each of which would result in a remote search being carried out. The decision to include these operations was based on the observed characteristics of the applications presented in Chapter 5, in particular the ocean model: a frequent task carried out is to retrieve all boundary values from neighbouring nodes at the end of each iteration. The addition of these bulk retrieval operations provides the benefits of decreasing the network communication involved in carrying out this task, and also decreases the complexity of the application code.

It should be noted that each of the bulk retrieval operations require an *expected* parameter which explicitly specifies how many tuples are expected to be retrieved by the invokation of the operation. The semantics adopted for these operations are that they will block until such time as the required number of tuples can be retrieved. This approach was adopted, once again, due to the characteristics of the applications being used: for applications involving processing of array partitions, it is typically known how many boundary values need to be retrieved, and making this explicit allows for other efficiencies to be realised when actually carrying out the tuple search and retrieval across the cluster.

Some consideration was given to generalising the operations even further, by including comparative operators to tuple templates such as greater-than and less-than operations. As it was eventually implemented, template fields must match exactly with a corresponding tuple field in order to successfully match.

While it could be argued that the addition of such operators may be useful in some instances, this must be balanced by the stated goal of this system, which was to maintain the simplicity of the original Linda model as far as possible. Operators such as >, < etc could arguably be a source of bugs and unneeded complexity for the applications programmer, with questionable usefulness for the class of applications intended for execution with the system. In addition, the extra matching comparisons which would be required if such operators were to be included would reduce the efficiency of local tuple lookup considerably.

Overall, the operations included in the Tupleware API allow an equivalent level of expressivness as Linda or JavaSpaces, along with the additional bulk operations which, as we discussed, can be useful for the class of application targetted by Tupleware.

A complete summary of the Tupleware API can be found in Table 4.1. In the following section, we describe in more detail how the operations in the Tupleware API are implemented by the runtime system and underlying objects.

4.5.2 Tuple Production

Tuples are produced on invocation of the runtime system's `out()`, `outAll()` and `outEach()` methods. The first two methods listed are purely local operations, involving no network communication, whereas the latter involves communication with remote nodes. Due to their relative simplicity, the local operations will be discussed first.

The `out()` method outputs a single tuple into a process' local tuple space (ie. the runtime system's `TupleSpaceImpl` object). Its method signature is `void out(Tuple t)`. Similarly, the `void outAll(Vector<Tuple> tuples)` method takes a collection of tuples as its argument, and outputs these tuples into the local tuple space. As each of these operations are purely local, and only need to interact with the local tuple space object, their implementation is a matter of simply invoking the relevant method provided by `TupleSpaceImpl`, which are the identically named `out()` and `outAll()` respectively.

As mentioned above, the `void outEach(Tuple t)` method is a remote operation which, rather than write a tuple to the process' local tuple space, will instead write the tuple to the spaces of each other remote process in the system. This operation is useful for instances when an application needs to notify other application processes of some event, or it it has produced some data that it going to be required by all other processes.

Each of these tuple production methods take a single argument, being the tuple that is to be written to tuple space, or in the case of `outAll()`, a collection of tuples. The application programmer need not be concerned as to which particular space the tuple or tuples are being written. The tuple production operations are illustrated in Figure 4.3.

Method Name	*Return Type*	*Description*
out	void	Stores a single tuple in process’ local tuple space.
outAll	void	Stores a collection of tuples in process’ local tuple space.
outEach	void	Stores a single tuple in every remote process’ tuple space. Local process’ local tuple space remains unchanged.
in	Tuple	Retrieves a tuple from tuple space (either local or remote), permanently removing the tuple. If matching tuple does not exist, method blocks until one becomes available.
inp	Tuple	Retrieves a tuple from tuple space (either local or remote), permanently removing the tuple. If matching tuple does not exist, method returns null value.
rd	Tuple	Retrieves a copy of a tuple from tuples space (either local or remote). If matching tuple does not exist, method blocks until one becomes available.
rdp	Tuple	Retrieves a copy of a tuple from tuple space (either local or remote).. If matching tuple does not exist, method returns null value.
rdAll	Vector<Tuple>	Retrieves a collection of a specified number of tuples from tuple space (either local and/or remote). A non-destructive operation, only copies of the tuples are retrieved. If the specified number of matching tuples are not available, the method will block until such time as there are.
inAll	Vector<Tuple>	Retrieves a collection of a specified number of tuples from tuple space (either local and/or remote). A destructive operation, matching tuples are permanently removed from their current tuple space. If the specified number of matching tuples are not available, the method will block until such time as there are.

Table 4.1: Summary of the Tupleware API.

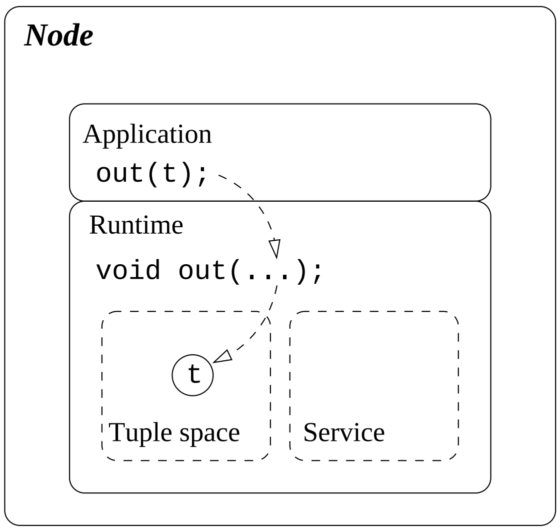


Figure 4.3: Example of a tuple production operation.

4.5.3 Single Tuple Retrieval

Single instances of a tuple object may be retrieved with the `in()`, `inp()`, `rd()`, and `rdp()` operations. The semantics of these operations are similar to the identically named Linda operations discussed in Section 3.3. Briefly summarised, `Tuple in(TupleTemplate t)` and `Tuple inp(TupleTemplate t)` are destructive retrieval operations, which remove a matching tuple from tuple space. If a matching tuple cannot be found in tuple space, the `in()` operation will block until a matching tuple becomes available, whereas the `inp()` operation is non-blocking, and simply returns a **null** value if no matching tuple can be found. The `Tuple rd(TupleTemplate t)` and `Tuple rdp(TupleTemplate t)` operations are used for non-destructive tuple retrieval: only a copy of a matching tuple is returned from tuple space, and original instance remains intact. The operations are blocking and non-blocking respectively, in the same vein as `in()` and `inp()`.

Each of these retrieval operations take a single argument, being the template object used for content-based matching against tuples, and their return value is a single tuple object, or possibly a **null** value in the case of the non-blocking variants. The application programmer does not need to provide any specific details as to where the tuple should be retrieved from, as the process of searching for a tuple is handled by the runtime system, and is discussed in Section 4.5.5. Each of the operations described in this section are illustrated in Figure 4.4.

It is also worth noting that the problem of tuple consistency will not become an issue during the execution of an application, as these is no tuple replication used by Tupleware. The only time a copy of a tuple is created is when one of the non-

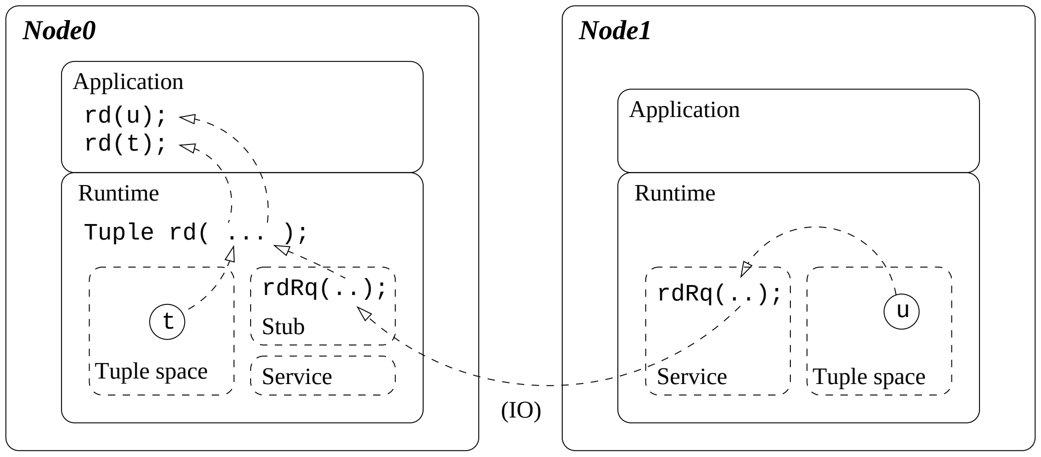


Figure 4.4: Single tuple retrieval from both a local and remote segment of tuple space.

destructive retrieval operations are used, which include `rd()`, `rdp()` and `rdAll()`. However, these operations do not create any copies of a tuple in tuple space, but rather they retrieve a copy of a tuple from tuple space and return it to the application layer. If the application then goes on to reintroduce the tuple to tuple space via the `out()` method, then this tuple is treated as a unique object, despite the fact it may contain the same fields and values as another tuple.

There may be instances, such as in the ocean model application discussed in Chapter 5, where there are more than one version of some particular data represented by a tuple. In cases such as these the application should tag these tuples with a value representing its version so that there is no ambiguity with regards to which is the latest version of the data. Local storage of tuples takes this into account, by arranging their storage in a data structure such that the current version of a particular tuple will be found first when a search of the local tuple space is performed. This arrangement is depicted in Figure 4.1 on page 49.

4.5.4 Bulk Tuple Retrieval

Tupleware includes two bulk tuple retrieval operations, `inAll()` and `rdAll()`, which enable an application to retrieve multiple tuple instances with a single method invocation. These operations can be used to minimise the number of remote communication events required to retrieve multiple tuples, which would otherwise involve multiple invocations of one of the single tuple operations discussed in the Section 4.5.3. Similar to the single tuple retrieval operations discussed previously, `inAll()` is a destructive read operations, whereas `rdAll()` is non-destructive. However, the semantics of these bulk-retrieval operations are slightly more complex than for single-tuple retrieval.

The method signatures of the two bulk-retrieval operations are as follows:

```
Vector<Tuple> inAll(TupleTemplate t, int expected)
Vector<Tuple> rdAll(TupleTemplate t, int expected)
```

The most obvious differences compared to the single tuple retrieval methods are that the return value of each is a collection of tuple objects, and that each methods take an additional integer argument, `expected`, which specifies how many tuples the application requires to be retrieved. Aside from the differences regarding destructive vs. non-destructive retrieval mentioned previously, the semantics of these methods are the same: the runtime system will attempt to retrieve the expected number of tuples from tuple space (either local or remote), and will return a collection of tuples (specifically, a vector of tuple objects) equal in size to the expected number specified. If the specified number of tuples are not available in tuple space, then each method will block until they are. Due to the blocking nature of these methods and the potential for causing deadlock, they should be used carefully by the application programmer to ensure that the application logic prevents this from occurring. The operation of the these batch retrieval operations are illustrated in Figure 4.5, and their specific operation at the runtime level is described in Section 4.5.5.

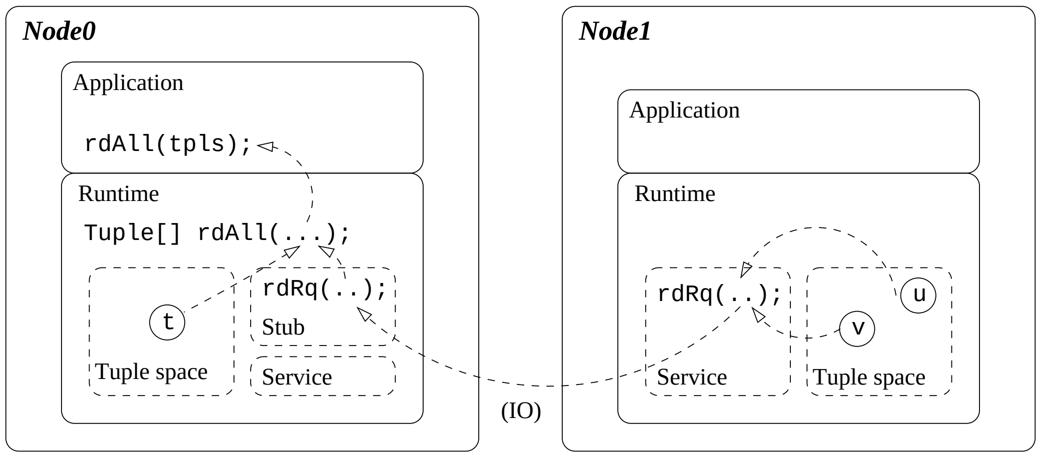


Figure 4.5: Batch retrieval operations.

4.5.5 Tuple Search Algorithm

The immediately previous sections have dealt with the semantics of the operations provided by the Tupleware runtime system as an interface to the application layer. In this section, we describe the lower-level operation of the runtime system, in particular, the algorithms which have been implemented to conduct tuple search operations in an efficient manner.

4.5.5.1 Rationale

The principle behind the search algorithm is to minimise the number of communication instances required to retrieve a tuple by targeting retrieval requests to those nodes which have the highest probability of being able to satisfy the request, based on the success of previous requests. This technique was adopted due to the nature of the applications at which Tupleware is targetted for use, that is, array-based applications in which the array is decomposed into individual regions, and each region is processed in parallel. The characteristics of applications like these are such that any communication between processes are going to tend to occur between those processes which are processing ”neighbouring” regions of the array, whereas nodes processing unrelated regions of an array are going to tend to communicate very rarely, if at all.

Thus, an executing Tupleware system consist of individual nodes, each with its own runtime system, each of which will need to communicate with a subset of all nodes in the system almost exclusively, and rarely if at all with all other nodes. These groupings, or clusters, of nodes will emerge quickly during the execution of an application as each individual runtime system dynamically adapts to the patterns of communication instances it is tasked with carrying out.

4.5.5.2 Request Success Factor

Underpinning the operation of the search algorithm is a *success factor* which is associated with each tuple space stub object maintained by the runtime system. The success factor is a numerical value between zero and one, and is used to denote the likelihood of the tuple space service associated with a given tuple space stub being able to fulfil a request for a tuple. A higher success factor represents that there is a greater chance of success, and vice versa.

At the beginning of an application’s execution, each stub has a success factor of 0.5 as there are no previous requests from which to calculate another value. A value of 0.5 is meant to represent an intermediate chance of success. Due to all stubs starting with an equal success factor, the initial requests made are random, however the success factor will be recalculated based on the success or failure of these requests, and quite quickly a distinct ordering, or ranking, emerges which can be used to prioritise subsequent requests.

The recalculation of the success factor occurs every time a stub is used to perform a request, and is based on the following equation:

$$S = \begin{cases} S + (1 - S) \times A & Success \\ S - S \times A & Failure \end{cases}$$

where:

- S is the success factor, and
- A is the adjustment factor.

The adjustment factor, A, is a value between zero and one used to specify by how much the success factor should be adjusted each time it is recalculated. This value will determine how quickly the success factor moves towards either one or zero, or in other words, by how representative a successful request is in terms of the prioritisation of subsequent requests.

This value should be chosen based on the application and the number of processes in a system. If there is a weak relationship between the data being computed on each process, then each process may ultimately end up needing to communicate with a relatively large number of other processes. In cases such as this a small adjustment factor should be used, as one successful request to a remote tuple space does not imply that there is a much greater probability of success for future requests. However, if there is a tight relationship between the data segments being computed by each process, then it follows that these processes will likely communicate very frequently, and that a successful request should have a higher bearing on the probability of success for subsequent requests.

In practice, for the application used to evaluate the performance of Tupleware (discussed in Chapter 5), an adjustment factor of 0.2 was used as it reflects the characteristics of these particular applications. An adjustment factor of greater than 0.5 would reflect a fairly volatile system with a very weak relationship between processes, where a value of 0.1 or 0.2 would represent a more stable relationship.

The usage of the success factor if described in the following sections, where we will see how it is used to determine the prioritisation of remote requests when tuple retrieval operations are invoked.

4.5.5.3 Searching for a Remote Tuple

The search process is initiated when a node's runtime system receives a request from an application to retrieve a tuple. This can be triggered by the invocation of any of the following `TupleSpaceRuntime` methods: `in()`, `inp()`, `rd()`, `rdp(`, and also bulk operations `rdAll()` and `inAll()`.

Each node's runtime system maintains an array, A, which contains references to N stub objects, where N is the number of all other nodes that are participating in the system. For example, $N = 9$ for a system that has ten participating nodes, and each runtime system has a stub object associated with each of the other nine nodes.

The ordering of these stubs in A reflects the relative probability of the associated remote node being able to successfully satisfy a request for a tuple, determined by its success factor. Whilst the stub order is initially random, once the execution of the system begins and the success factor of each stub is adjusted, the runtime system alters the ordering to reflect the success or failure of any remote requests that it undertakes. Relatively quickly, the ordering the stubs will reflect the node's unique group of neighbouring nodes that it frequently communicates with; stubs associated with these nodes will be stored towards the beginning of A, while those that have not successfully fulfilled previous requests, or have not yet been communicated with at all, will be stored towards the end. Node groupings for a small Tupleware system are illustrated in Figure 4.6. This figure shows the nodes in a cluster, with each of their success factor values in relation to *Node5*. As the diagram shows, this node forms a grouping with the nodes with a high associated success factor, to the exclusion of those nodes with a low success factor.

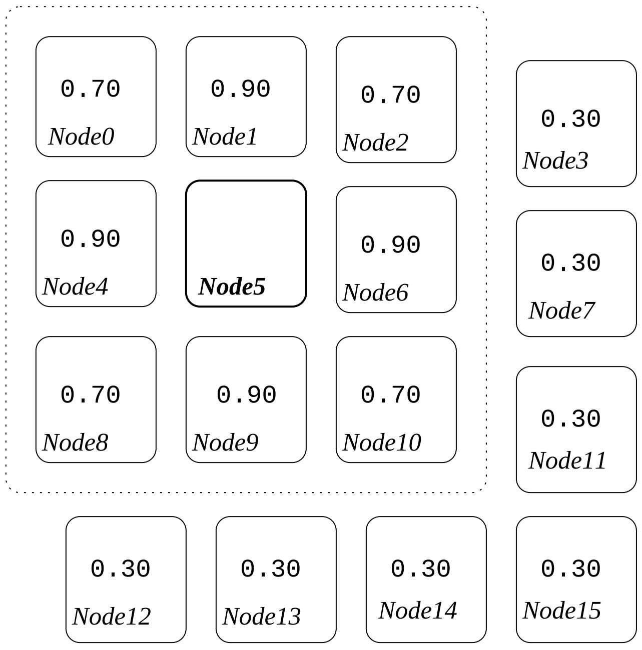


Figure 4.6: Grouping of neighbouring nodes based on success factor.

It important to note that the logic behind maintaining these groupings is completely distributed; each individual runtime system dynamically updates its array of stubs based on the success or failure of its own requests, and no overarching system-wide coordination or creation of these groupings is used at any point. All decisions by the runtime system related to where to dispatch each request are made locally, based on information gained locally. This minimises any overhead that would be introduced from a system-wide logic being maintained, while still providing the means for each node to adapt to its role in the execution of a particular application.

The ordering of the stubs in array A is utilised when the runtime system makes a request, with requests being performed sequentially, based on their ordering. Therefore, the first remote node queried will be the one associated with stub stored at $A[0]$, and if that request is not successful, then the node associated with the stub at $A[1]$ will be queried, and so on, until either a matching tuple has successfully been retrieved, or all nodes have been queried unsuccessfully. At each conclusion of each individual request instance, the success factor of the stub used to execute the request is updated, whether the request was successful or a failure. Assuming that one stub in A executes a successful query, then the successful stub's success factor will increase, and any stubs used to execute an unsuccessful request will have their success factor decreased.

There will be situations encountered when all remote nodes have been queried and have been unable to successfully fulfil the tuple request, and what actions occur subsequently depends on whether the operation being request is blocking or non-blocking. If it is the latter (either `inp()` or `rdp()`), then the runtime system's invoked method will simply return a **null** value to the application layer, in keeping with the semantics of these methods. However, if it is the former (`in()` or `rd()`), then the semantics of these operations dictate that we must wait until a matching tuple becomes available before returning it to the application. To do this, rather than continually polling each remote node until we are successful, parallel blocking requests are dispatched to each node, and the runtime simply waits until one of these node can provide a matching tuple. When this eventually occurs, the other pending requests are aborted, and the stub which executed the successful request has its success factor adjusted accordingly.

At the conclusion of any of the tuple retrieval operations, A is reordered based on the updated success factor values of the stubs which it contains. Thus, when subsequent retrieval operations are invoked, the stubs in A will be ordered correctly so that $A[0]$ will contain the stub most likely to execute a successful request, and $A[N-1]$ will be the stub least likely to be capable of fulfilling the request.

4.5.5.4 Strengths and Weaknesses of the Search Algorithm

The strengths of the search algorithm are its ability to adapt to the dynamically changing communication patterns of cluster nodes, and its ability to gradually approach near-optimal tuple search behaviour over time. This allows nodes to conduct their communication in an efficient way for stable applications, and to adapt to a changing environment if and when the assigned array partitions alter. In addition, it alleviates the burden on the application programmer, who need not specify at compile time which partitions are assigned to specific nodes. This simplifies the development of applications for the system, and in turn the application code.

One possible weakness of the algorithm is the disproportionate effect of corner-cases. That is, the remote location of array elements in the corner of an array partition cannot always be known in general terms, and are often application-specific. To use the example of the ocean model presented in the following chapter, the surface elevation, and the north/south and east/west current velocity variables at each gridpoint are retrieved from diffent partition of the array at the conclusion of each timestep. We could direct the tuple search to the correct remote node given the value that is being requested by the applications, however in order to maintain a generalised approach we retain the search algorithm unchanged from above to direct our search, and update the success factor accordingly.

Put simply, we expect to have a significantly smaller chance of querying the correct remote node for corner cases, certainly much lower than for other (non-corner) boundary values. This issue may be addressed by performing a more thorough analysis of tuple characteristics, which would need to take into account the overall dimensions of each array partition. However the difficulty remains of how to do this in a non-application-specific manner.

Overall, the search algorithm as presented provides a general way in which to target the search for the majority of tuples stored on remote nodes. Its adaptability to the particular communication characteristics of an application deployment on a cluster are a real strength, with a possible weakness noted above in the instance of corner cases. The effectiveness of the algorithm is explored further in the results of two example applications in Chapter 6.

4.5.6 Summary

This section has described the operations supported by Tupleware, including their semantics, and the detail of their operation. The Tupleware API as provided by `TupleSpaceRuntime` is summarised in Table 4.1 on page 64.

Also, we have detailed the algorithm which underpins the tuple search process undertaken by the runtime system, and described how the system dynamically adapts to the communication characteristics of the executing application, and how that in turn allows the system to minimise the number of communication instances that are required to retrieve a tuple from remote segments of the distributed tuple space.

In the following section, we will describe the way in which a representative generic application would execute on the Tupleware system.

4.6 Deployment & Execution

Tupleware applications consist of a number of application processes, each of which has its own Tupleware runtime system, and associated components such as a tuple space service. These processes may each execute on separate nodes of the cluster, or multiple processes may execute on a single node. Additionally, it is possible to run a standalone tuple space service on a node without an associated application process; this feature is useful for when tuple space needs to store data persistently, and not be limited by the lifespan of an application process.

It is assumed that the cluster nodes are connected by a TCP/IP network, and that a suitable version of the Java Runtime Environment (JRE) is installed.[1] Due to the use of Java, Tupleware should, in theory, be able to run on any operating system which supports a JRE. However, all development and testing for this research was carried out on Linux, and no other systems have been tested.

4.6.1 Initialisation

Initialisation of a Tupleware application requires all participating Worker processes to register with the Master process. For this to occur, the IP address of the Master process must be known to all Workers beforehand. To register, a Worker will place a tuple in the Master's local tuple space containing the Worker's IP address and port number on which it is accepting connections.

Once a Worker has provided its network information, it will attempt to retrieve the information provided by all other Workers in the system. Once this has been successfully completed, each Worker has the necessary information required to instantiate a stub object for each of the other Worker processes in the system. However, no socket connections are created between Workers until they are required

[1]Tupleware was developed with the Java SDK version 6, and therefore for best results the same version of the JRE is recommended for use.

for a Worker process pair to communicate. Once a socket connection is created, however, it will remain open for the duration of the application's execution, unless it becomes disconnected due to some other cause, at which point the disconnected Worker will attempt to reestablish to connection.

Once the initialisation phase is complete, all Worker processes are able to communicate with each other, and with the Master process.

4.6.2 Task Submission

After initialisation, Worker processes will wait until the Master submits some task to be executed, and provides any required initial data. Sine the Master and Worker processes are implemented as two logical cooperating parts of the application, the precise format of this task submission may vary widely. Generally, the Master will define how the application is to be parallelised, and assign each Worker a particular role in the application's execution. In the example of the ocean model (see Section 5.1), the Master assigns a panel to each Worker process, and makes available the initial panel data for each Worker to retrieve.

Once the Master has submitted a task to be executed and each Worker has retrieved the initial data, then the system moves into the execution phase, in which the Master process has minimal involvement.

4.6.3 Execution

The execution of an application involves the Worker processes performing whichever processing they may have been assigned, and communicating intermediate data with other Worker processes as required. All communication during the execution phase is carried out in decentralised fashion. That is, Worker processes communicate directly with each other, without requiring any network traffic to pass through the Master process.

All instances of communication during the execution phase involve a Worker making a request for a tuple which is not stored in its own local space. When a Worker requires a tuple, it will first search its own local portion of the tuple space. If it is found, then the process will continue; however, if the tuple is not available locally, then the spaces of other remote Worker processes must be queried. The algorithm used to perform these queries was previously discussed in Section 4.5.5.

In the instances where this sequence of requests does not return the desired tuple, the runtime system will perform one of two options, depending on the semantics of the operation being implemented. If the operation being carried out is a non-blocking operation, such as `rdp()`, then a **null** value will simply be return to signify that the requests tuple is not available. If, however, the operation is one of the blocking operations, such as `rd()`, then in order to conform with the semantics of these operations it is necessary to persist with the request until it can be fulfilled. Therefore, in these cases the runtime will send blocking requests to all remote processes in the system, and wait until a matching tuple becomes available, at which point all other requests are terminated and the retrieved value is return to the application.

4.6.4 Gathering Results

When an application has completed execution, the final data must be returned to the Master process. As the Master and Worker processes are implemented as complementary pieces of the application, each Worker process is able to determine when processing is complete; this will generally be either when all data have been processed, or when the specified number of iterations have been completed. In either case, the result gather phase usually consists, or each Worker process writing its final computed data back to the Master's local tuple space. It is then the Master's responsibility to merge these data back into a meaningful whole.

4.6.5 Fault tolerance & dynamic reconfigurability

Two important considerations for any distributed system are its ability to handle and recover from faults, and its capacity to reconfigure its operation due to changing system characteristics.

Fault tolerance must be built into the system if we are to treat hardware and network failures as inevitable occurrences. This is not an unreasonable assumption, particularly for clusters which consist of commodity PC components. When one of these failures do occur, we wish for the execution of the system to be disrupted as little as possible; ideally, an application will still be able to complete its task, albeit with some possible performance penalty.

Common mechanisms which can be used to avoid some of the negative effects of faults include: *checkpoints*, whereby intermediate application states are saved to allow applications to restart in the event of a fault; *transactions*, which allow sequences of related operations to be grouped such that either all of the operations complete, or if a fault occurs the system is "rolled back" to its state as it was previous the any of the operations; and *replication*, whereby data and/or processing are duplicated on various nodes in the system, with the idea being that if one node fails then the system can fall back to the remaining node.

Tupleware provides some allowance for faults, however it does not utilise the mechanisms mentioned above. As it stands, network errors are handled by the runtime system, and if a node is disconnected for any reason, the system will attempt to re-establish the connection so that an application can continue. However, if a node becomes completely unavailable due to a hardware or operating system fault, the runtime system includes no means to salvage the execution of an application and continue, unless the application itself contains such features. Generally it would be the case that all data stored in a node's local partition of the tuple space would be irretrievably lost, requiring the application to be restarted from the beginning. Obviously this is not ideal should this system be deployed in a non-research setting for real applications, and this issue would need to be addressed if this were to eventuate.

The issue of dynamic reconfigurability is not completely unrelated to the issues of fault tolerance discussed above, in that the latter are a subset of the former. Dynamic reconfigurability is concerned with a broader scope of issues including one or more of the following: How can a system make optimal use of its resources? How can the system adapt when the available resources change? Can the role of each individual node change during execution? And finally, can nodes join or leave the system at runtime without causing the application to halt or crash?

As we have discussed previously in Section 4.5.5, Tupleware does allow a certain level of reconfiguration to occur at runtime due to its tuple search algorithm. This algorithm optimises its searches based on the communication characteristics of the system's nodes and the historical accuracy of previous searches. This allows the underlying runtime system to use the system's resources efficiently, namely network capacity (by minimising the number of search requests).

However, it is also the case that the number of participating processes (though not necessarily the specific details of their location or network address) are known at compile time. Currently it is not possible for nodes to join the system at runtime and participate in the execution of an already-running application, nor is it possible for nodes to leave the system during execution without adversely affecting the application, as discussed above. As it currently stands, the application programmer will decompose an application into a number of parallel tasks, and each task will be assigned to a node at the beginning of an application's execution. It is expected that each task will be executed to completion by the node to which it is assigned, with node alterations to assignments possible. To allow otherwise would require some mechanism by which a snapshot of a task's state can be captured and transferred to another node, something which may be addressed as further work. And given the application programmer's considerations discussed above, to allow additional nodes to join the system during an application's execution may require a way in which the degree of parallelism can be modified "on the fly", something which may be very difficult to achieve in a general way.

Both of these issues are identified as areas of further work, and are discussed further in Chapter 7.

4.7 Application Development

In this section we focus on the requirements for developing an application for the Tupleware system. The example application being presented uses a replicated-worker style of parallelism (Carriero & Gelernter, 1990, p. 18). This involves a single master process which initialises the application and submits it to tuple space so that a pool of worker processes may carry out the computation. Once complete, the final computed data is collected by the master process.

An application such as this is typically suited to implementation on Tupleware, and, despite it being a coarse-grained parallel problem, will neatly illustrate the simplicity of implementing a Tupleware application.

4.7.1 Implementation with Tupleware

For the purposes of our example, the application will apply a function, f, to each element of an array, A. Implemented sequentially, a solution would look something like the following:

```
for(int i = 0; i < A.length; i++)
    A[i] = f(A[i]);
```

In our parallel implementation, the master process will simply add the array elements to tuple space as individual tuples, and the worker processes will retrieve an element, compute its new value, and add the result back to tuple space. Once all elements have been computed, the application is complete.

4.7.1.1 Master Process Implementation

Firstly, we begin with the definition of our master process. In order for it to execute on the Tupleware platform, it must contain an instance of a `TupleSpaceRuntime` object, and this object, upon instantiation, must be given an integer argument specifying on which TCP port the tuple space service will listen for incoming connections. Also, the runtime system object's `start()` method must also be invoked to initialise the underlying Tupleware system. Other than these requirements, the master process will contain only application specific code; in this case, creating array *A* and adding its values to tuple space, and waiting for computed values to become available. A partial code listing for the master process is listed below.

```
public class ArrayMaster {
    public final boolean IS_MASTER = true;
    public final int N = 100;
    public TupleSpaceRuntime space;

    public ArrayMaster(int port) {
        space = new TupleSpaceRuntime(port, IS_MASTER);
        space.start();
    }

    public void init() {
        int[] a = new int[N];
        for(int i = 0; i < N; i++)
            a[i] = (int) Math.round(Math.random()*Integer.
               MAX_VALUE);

        Vector<Tuple> tpls = new Vector<Tuple>();
        for(int i = 0; i < N; i++)
            tpls.add(new Tuple("A", new Integer(i), new Integer
               (a[i]), new Boolean(false)));

        space.outAll(tpls);
    }

    public Vector<Tuple> getResults() {
        TupleTemplate template = new TupleTemplate("A", null,
           null, new Boolean(true));
        Vector<Tuple> results = new Vector<Tuple>();

        for(int i = 0; i < N; i++)
            results.add(space.in(template));

        return results;
    }
}
```

One important decision that needs to be made when developing an application for Tupleware relates to the format of the tuples. More specifically, what fields do the tuples need, and in what order should they be arranged? A consistent approach is needed in order to allow Tupleware processes to communicate successfully, and it should be based on the characteristics of the data that is being shared, and also the application logic relating to the purpose of the communication itself.

In the example application presented here; the data being shared are elements of an array. A typical approach to sharing array elements is to include a name for the array, its index (or indices for multidimensional arrays), and finally the value of the element. This approach is adopted here, however we use an additional field to denote whether or not the element has yet been processed. We use a boolean value for this, with unprocessed elements having a false value in its final field, and processed elements having a true value. Thus, the format of our tuples are as follows: $< String : name, Integer : index, Integer : value, Boolean : processed >$.

4.7.1.2 Worker Process Implementation

The role of the worker processes is simply to retrieve an unprocessed tuple element, apply a specified function to it, and return the result to tuple space. Unprocessed tuples are retrieved from tuple space using a tuple template based on the tuple format described above. The *index* and *value* fields are left as null, and the *processed* field is set to false so that only unprocessed array elements are retrieved. Once the element has been processing, the result is written back to tuple space as a tuple with the same index and with the *processed* field set to true.

Like the master process, before accessing the tuple space, the worker process must first instantiate and start a tuple space runtime system, as shown in the `ArrayWorker` constructor below. The worker process will carry out these steps until all unprocessed elements have been exhausted, at which point it will terminate. A partial code listing for the worker process is shown below, and an illustration of the complete system can be found in Figure 4.7.

```
public class ArrayWorker {
    public final boolean IS_MASTER = false;
    public TupleSpaceRuntime space;

    public ArrayWorker(int port) {
        space = new TupleSpaceRuntime(port, IS_MASTER);
        space.start();
    }

    public void doWork() {
        TupleTemplate template = new TupleTemplate("A", null,
           null, new Boolean(false));

        for(;;) {
                Tuple t = space.inp(template);

                if(t == null)
                    return;

                int result = f(t.field(2));
                space.out(new Tuple("A", t.field(1), new Integer(
                   result), new Boolean(true)));
            }
        }

        public int f(int n) {
            // this is the function to be applied to array element
        }
    }
```

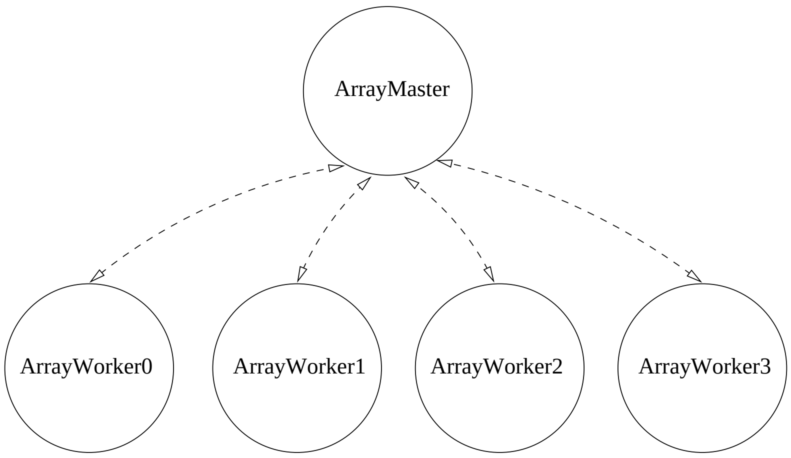


Figure 4.7: Overview of example system with five nodes.

4.7.1.3 Load Balancing & Skewed Access Considerations

An important issue which the application programmer must consider during implementation is that of load balancing. There are two important aspects to load balancing: firstly, that the amount of processing assigned to each node is as equal as possible, and secondly that the number of requests being served by each node in the cluster is roughly equal, as far as practically possible. Tupleware does assist to a certain extent with these issues, particularly the latter, however it is still the responsibility of the programmer to structure their application in such a way as to minimise, and hopefully avoid, any problems related to these issues.

In the case of the former, when parallelising an application we must ensure that it is done in such a way as to decompose the processing that is to be completed into an appropriate number of smaller sub-tasks, and that none of the participating processes experience starvation. For loosely-coupled data parallel applications, this is relatively straightforward, and the main consideration is on the granularity of the application. However, for more tightly-coupled applications where there are data dependencies between sub-tasks, the programmer must ensure that these are managed so as to avoid imposing too high a performance penalty from the resulting synchronisation overhead.

In the case of the latter, we must try to distribute the sub-tasks amongst nodes such that the communication requirements between nodes are spread as equally as possible. We wish to avoid a situation where one (or a small subset) of the nodes are servicing the overwhelming majority of requests. Such a scenario would lessen the time these overloaded nodes can devote to processing their own sub-task, and cause other nodes to block until these nodes "catch up" (if there is synchronisation being carried out during execution). This would obviously have a negative impact on the performance of the application.

Whilst the application programmer should be mindful of avoiding the situation described above, and structure the application's parallelism in such a way that it does not occur, Tupleware also assists in this regard, in the first instance, by employing a peer-to-peer communication style. This helps to avoid any system-induced communication bottlenecks.

The other consideration with regards to skewed access is that of tuple storage location. Tupleware's behaviour is to store tuples in the space partition of the tuple producer (ie. `out()` is a purely local operation). It is possible that the problem of skewed access could be addressed by preemptively migrating tuples nearer to where they will be accessed, so that they will already be stored locally when they are requested. The benefits of such an approach would depend on the application's communication characteristics, and would only be realised if, firstly, the accuracy of tuple propagation could be guaranteed, and secondly, the time spent performing tuple propagation was less than the time taken to retrieve the tuple at the time it was required. It should also be pointed out that, regardless of whether tuples are to be propagated preemtively or only upon being requested, there will be no net reduction in total communication requirements; in fact, extra communication events may be introduced if tuples are propagated unnecessarily.

The major shortcoming of a preemptive tuple propagation approach is that the search algorithm used by the runtime system relies on tuples' locality, and so therefore the location at which a tuple is stored must be deterministic, otherwise the algorithms effectiveness would be severely hampered. However, it may be unfeasible to store tuples completely deterministically, and more likely is that a probabilistic approach would need to be adopted, along the same lines as the one used by the search algorithm.

Ultimately, given incompatible requirements of the two approaches outlined above, it was decided to adopt the retrieval-on-demand model presented in this chapter. It is felt that this approach caters for the widest range of application communication characteristics, and the drawbacks of the alternaive have been outlined above.

4.7.1.4 Implementation Summary

Several aspects of the above application are worth discussing from a development perspective. Firstly, the requirements for a process to participate in a Tupleware system are simple; all it needs is to instantiate and start a `TupleSpaceRuntime` object. Once this has been done, the process will have its own local tuple space, and is able to interact with other processes participating in the system. Secondly, the usage of the tuple space operations is also simple and straightforward; the developer does not need to be concerned with the problem of where a tuple is physically stored. It may exist in the process' local tuple space, or it may be in any one of the other participating processes' tuple space. The practicalities of searching for and retrieving a tuple are handled by the runtime system, as discussed in Section 4.5, and so the application code remains as simple as possible, keeping the focus on the application itself rather than the inner workings of inter-node communication. Also, to this end, the runtime system handles any network communication errors, meaning that the tuple space operations used in an application do not need to be enclosed in endless try/catch blocks, which further assists in keeping the application code as simple as possible.

4.8 Summary

This chapter has given a detailed overview of the Tupleware system. The aims of the system were explicitly stated, giving a foundation upon which to evaluate the system in the following chapters. These aims were mainly concerned with the performance and ease-of-use of the system: two quite different criteria, yet ones that are critical to the usefulness of the system.

The approach which was adopted to meets these aims was described, and the basic components of Tupleware were detailed. These components allow a non-distributed, multi-threaded application to be developed and executed in parallel.

Following this we discussed the way in which Tupleware components were distributed across a cluster, so that remote nodes may access other nodes' hosted portion of the tuple space. The specific components which enabled the inter-node communication were described, including the use of stub objects to interact with remote tuple space services.

Arguably the core component of Tupleware, its runtime system, was described in detail. This component acts as a middleware upon which an application executes, and controls and coordinates the operation of the rest of the system. It also provides an application programming interface, giving the programmer access to the operations which are available for accessing the distributed tuple space. The semantics and low-level operation of these operations was also described, and we highlighted their relative simplicity and similarity to the original Linda operations.

One of the main contributions of this research was the search algorithm implemented for the efficient retrieval of tuples from remote cluster nodes. This algorithm was described in detail, including its concepts of a success factor, and the way the system dynamically adapts to an application's communications characteristics to produce naturally forming ”clusters” of nodes which communicate frequently. The justification for this search algorithm was explained in terms of the applications that Tupleware is targetted towards, these being applications which perform some form of processing on arrays. A characteristic of this category of application is that a subset of the cluster nodes will communicate frequently if they are processing neighbouring segments of an array, and rarely otherwise. Thus, the search algorithm was designed to reflect this fact.

Next we described the general way in which an application executes on the Tupleware system, including the three main phases of initialisation, parallel execution, and result gathering. Finally, we gave a detailed example of how a typical master/-worker application would be developed using the Tupleware API.

Chapter 5

Applications

Previous chapters have described Tupleware, including the architecture of the system and the operation of its runtime system. In this chapter, we will examine the applications which were implemented on top of Tupleware, and describe how each application was tailored in order to fulfil any requirements or restrictions imposed by the Tupleware system.

5.1 Ocean Model

The ocean model is a two-dimensional simulation of an enclosed body of water. The model calculates the current velocity and surface elevation of the water based on a specified wind velocity and bathymetry.

The body of water is represented by the model in the form of a grid, and each cell in the grid represents a single grid point. Grid points each individually store descriptive data, including the depth of the water at that point, along with the surface elevation and current velocity. Wind velocity is assumed to be universal and static across the grid. The variables stored in each grid point are staggered in such a way that the u and v variables are associated, respectively, with the x-axis and y-axis edges of each grid point. The eta variable is simply associated with the centre of each point.

This application was adapted from a sequential Fortran program by Dr. John Hunter, for which a code listing can be found in Appendix B.1.4.

5.1.1 Equations

When executed, the model iterates through a fixed number of time-steps; at each time-step, each grid point's surface elevation and current velocity values are recalculated based on the values stored at neighbouring grid points. This process continues for the specified number of time-steps, at which point the model should be in a steady-state. The equations used to calculate each grid point are given below.

Firstly, updates to the east/west (u) and north/south (v) current velocity are defined by the following equations:

$$\begin{aligned} u(i,j,kuv_{new}) &= u(i,j,kuv_{old}) - facgx \times (eta(i,j,keta_{old}) - eta(i,j-1,keta_{old})) \\ &\quad - \frac{facbf \times u(i,j,kuv_{old}) \times uf}{hu(i,j)} + \frac{facwx}{hu(i,j)} \end{aligned}$$

$$\begin{aligned} v(i,j,kuv_{new}) &= v(i,j,kuv_{old}) - facgy \times (eta(i,j,keta_{old}) - eta(i,j-1,keta_{old})) \\ &\quad - \frac{facbf \times v(i,j,kuv_{old}) \times uf}{hv(i,j)} + \frac{facwy}{hv(i,j)} \end{aligned}$$

where:

- u and v define the velocity of the water current,
- kuv_{new} and kuv_{old} index the current and previous iterations of u and v respectively, and
- hu, hv, uf, $facgx$, $facbf$, $facgy$, $facwy$, and $facwx$ are model-related constants.

The surface elevation at each grid point (eta) is calculated using the following equation:

$$\begin{aligned} eta(i,j,keta_{new}) &= eta(i,j,keta_{old}) \\ &\quad -(facex \times (u(i+1,j,kuv_{old}) \times hu(i+1,j) - u(i,j,kuv_{old}) \times hu(i,j)) \\ &\quad + facey \times (v(i,j+1,kuv_{old}) \times hv(i,j+1) - v(i,j,kuv_{old}) \times hv(i,j))) \end{aligned}$$

- eta contains the surface elevation,
- $keta_{new}$ and $keta_{old}$ index the current and previous iterations of eta respectively,
- $facex$, and $facey$ are model-related constants.

5.1.2 Implementation

The model is parallelised by performing domain decomposition on the grid, which splits the grid into a number of separate *panels*, up to the number of nodes available for processing. Each panel is assigned to a specific node, whose responsibility it is to perform the processing on the panel. As each panel represents only part of the complete grid, at each iteration it is necessary for the boundary values of each panel to be retrieved from neighbouring panels. This process is illustrated by Figure 5.1, which shows a 9x9 grid which has been decomposed into three panels. Each 9x3 panel has a halo region, represented by the shaded cells, whose values are updated after each iteration of the model. The arrows between neighbouring halo region cells represent the communication instances which are involved in each boundary update.

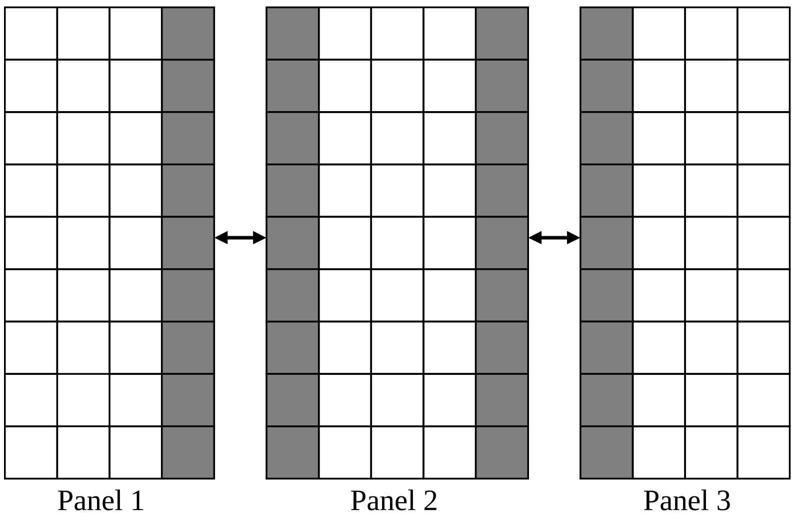


Figure 5.1: Updating panel boundary values.

As Figure 5.1 clearly shows, updating each panel's boundary values introduces significant additional communication. This communication is the main contributor to the overhead of parallelisation of this application, and therefore it is important to perform the update as efficiently as possible. To this end, the runtime system should, after the first iteration of the application, have sufficient information regarding the likely source of a required tuple that the majority of requests will be fulfilled successfully on the first attempt.

Similarly to the modified quicksort application, the granularity of the ocean model can be altered by varying the size of the grid, and also the degree of parallelisation, which determines the relative size of each panel. However, altering either of these variables also has an effect on the total number of boundary points in the application, and this has a proportional effect on the number of communication instances that each process must perform each time a boundary update is required. The end result is a limit to the level of scalability of the application; this will be discussed further when we present the application's performance results in Chapter 6.

5.1.3 Application Components & Execution

The main components of the ocean model are the master and worker processes, and panel objects. These are defined by the `OceanModelMaster`, `OceanModelWorker`, and `Panel` classes respectively.

Firstly, is it the role of the master process to initialise the application, which entails initialising the values stored in the grid, decomposing the grid into an appropriate number of panels, and making these panels available in tuple space. The number of panels created will be equal to the number of worker processes that are participating in the system, and the width of each panel is equal to the grid width divided by the number of processes, or $Width = Grid\,Size/Processes$. The length of each panels remains equal to the length of the grid.

Each panel represents a partition of the grid, and contains its associated partition, along with any other required data, such as model-related constants. Also, each panel has two boolean fields, `SHARE_LEFT` and `SHARE_RIGHT`, which specify which boundaries will be shared by a given panel. As Figure 5.1 illustrates, the leftmost and rightmost panels share only one boundary, whereas all other panels will share both (there is also the unique case of there being only a single panel, which will share none of its boundaries). The master process will set these fields accordingly during initialisation.

Finally, the worker processes in this application are reasonably simple, in that their role is to retrieve an unprocessed panel from tuple space, and process the panel for the specified number of timesteps, performing a boundary update at the end of each. Once the required number of times steps have been completed, the processed panel is returned to tuple space, at which point the master process will retrieve the panel so that it may be reassembled with the application's other panels to reconstruct the grid.

5.1.4 Summary

In this section we have described an ocean modelling application used to compute the current velocity and surface elevation of an enclosed body of water. This simulation was represented numerically as a 2-D grid, with each grid point containing three variables related to the current velocity and surface elevation, along with five constants related to the wind velocity and depth.

The model was parallelised by decomposing the grid into a series of panels, each of which represented a slice of the entire grid. These panels were allocated to individual cluster nodes for processing, with the boundary region of each panel needing to be communicated between nodes working on neighbouring panels at the conclusion of each time step. The model iterates through a fixed number of time steps, at which point the final processed panels are returned to the master process for merging.

The application is time-stepped and fine-grained in nature. Its execution requires a relatively high level of communication, and was chosen due to its efficacy in testing the Tupleware distributed tuple space under a high load.

5.2 Sorting

5.2.1 Overview & Characteristics

Quicksort (Hoare, C 1961) is a well known sorting algorithm with an average case execution time of $O(n\log n)$. It is recognised as an efficient general-purpose sorting algorithm which rarely exhibits its worst-case execution time.

Several characteristics of the quicksort algorithm lend it to being suited for evaluating Tupleware. Firstly, parallelisation of quicksort is reasonably straightforward, and produces tasks which are loosely-coupled and have only moderate dependencies. Secondly, by modifying the quicksort algorithm so that partitioning ends when an array segment length reaches a certain predefined threshold, it is possible to adjust the granularity of the parallelism exhibited by the sorting algorithm. This feature is useful as it allows us to evaluate the performance of the system with various levels of communication frequency.

The algorithm used to evaluate the system in this paper is a modified version of quicksort. As described above, unsorted arrays are partitioned only until their length is greater than a predetermined threshold value. At this point, partitioning ends and the remaining unsorted array segment is sorted using some other sequential algorithm; in this case, insertion sort (Cormen, T. et al, pp. 2-4).

5.2.2 Implementation

Similar to the ocean model, the parallel sorting application consists of a single master process (defined in the QSortMaster class) in addition to one or more worker processes. The role of the master process is to create an array of integers, which is then encapsulated inside a QSort object. These objects are the main data unit used in the application, and, along with an unsorted array partition, also contain the specified threshold value and the required functions for processing the array partition, including partition(), insertionSort() and merge(). The QSort programming interface is shown below:

```
public class QSort implements java.io.Serializable {
    private int[] a;
    private int threshold;

    public QSort(int[] a, int threshold);
    public static int partition(int[] a, int p, int r);
    public static void insertionSort(int[] a);
    public static int[] merge(int[] a, int[] b);
    public static void initData(int[] a, int n);
    public QSort split();
    public boolean readyToSort();
    public int[] getData();
    public int size();
}
```

Given the QSort class described above, the steps completed by the QSortMaster are as follows:

```
// assuming a TupleSpaceRuntime has been initialised and
// values for array LENGTH and THRESHOLD are specified

// initialise array and encapsulate in QSort object
int[] a = new int[LENGTH];
QSort.initData(a, LENGTH);
QSort qs = new QSort(a, THRESHOLD);

// write QSort object to space
space.out(new Tuple("qsort", qs, "unsorted"));

// create required values for retrieval of sorted
// array partitions
Vector<int[]> sortedParts = new Vector<int[]>();
int n = 0;
// template used to match against tuples containing
// the sorted partitions
TupleTemplate template = new TupleTemplate("qsort",
                                           "sorted",
                                           null);

// loop until all sorted partitions obtained
while(n < LENGTH) {
    Tuple t = ts.in(template);
    QSort q = (QSort) t.field(2);
    sortedParts.add(q.getData());
    n += q.size();
}

// place a "poison pill" into tuple space to signal to
// workers that the application is complete
Tuple poisonPill = new Tuple("qsort",
                             new QSort(),
                             "complete");
space.out(poisonPill);

// finally, merge sorted partitions back into complete array
a = reconstructArray(sortedParts);
```

The final components in this application are the worker processes, which are defined in the `QSortWorker` class. These workers simply obtain `QSort` objects, and either partition the array if its length is above the specified threshold, or sort it if it is not. Each partition performed by a worker process produces two unsorted array segments, where all values in one segment will be of a lesser or equal value to the values in the other segment. One of these segments will continue being processed by the worker, either by being partitioned further, or by being sorted if its length is below the specified threshold value. The other segment is placed into the worker's local tuple space, where it can be accessed by other workers who do not have any unsorted segments to process. Thus, this application has natural load balancing due to its similarity with the Master/Worker style of parallelism. The operation of the worker processes is shown in the code listing below:

```
// assuming a TupleSpaceRuntime has been initialised
TupleTemplate template = new TupleTemplate("qsort",
                                           null,
                                           "unsorted");
Tuple t = ts.in(template);
QSort qs = (QSort) t.field(1);
String status = (String) t.field(2);
```

```
// loop until poison pill from master process signals
// that application is complete.
while(!status.equals("complete")) {
    // test if array length is below threshold
    if(qs.readyToSort()) {
        // array length is below threshold, so we sort it
        qs.insertionSort(qs.getData());

       // return sorted partition to tuple space
        ts.out(new Tuple("qsort", "sorted", qs));

       // try to obtain new unsorted partition
        t = ts.in(new TupleTemplate("qsort", null, null));
        qs = (QSort) t.field(1);
        status = (String) t.field(2);
    } else {
        // array length is still above threshold, so we
           partition
        QSort dup = qs.split();

        // write new unsorted partition to tuple space,
        // to be obtained by other worker processes
        ts.out(new Tuple("qsort", dup, "unsorted"));
    }
}
```

Figure 5.2 illustrates the interactions which occur when these components are put together into a complete system. The numbered interactions shown in the dia-

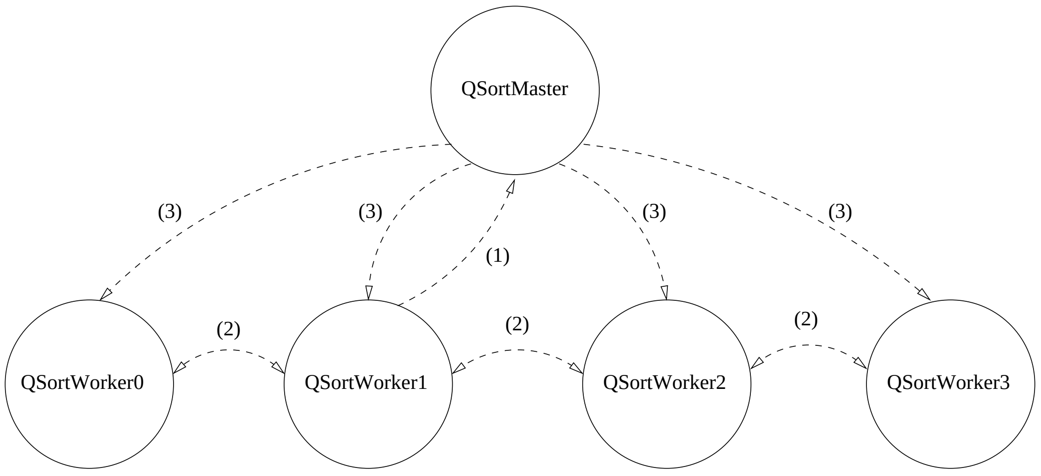


Figure 5.2: A complete parallel sorting application.

gram correspond to the following actions:

1. One worker process obtains the unsorted array segment from the master process. At the beginning of the application there is only one array segment, being the complete array.

2. These interactions involve sharing unsorted array partitions between worker processes. It is important to note that this step involves only worker processes; the master process is idle during this time, until sorted array partitions begin to be produced by the worker processes.

3. Finally, the interactions labelled (3) involve the master process obtain the sorted array segments from each worker process as they become available.

Due to the way the sorting algorithm is parallelised, there is a certain amount of time that elapses before all participating worker processes obtain unsorted array segments to process. Only one worker process can obtain the complete unsorted array in step1, which it must then partition into two array segments, one of which can then be obtained by another worker process, and so on, until there are enough unsorted segments for each worker process to work on. The total number of segments which will be produced by a given array is non-deterministic, and depends on the values in the array and the operation of the `partition()` function.

The flow of array segments through an instance of this application is illustrated in Figure 5.3 on page 94, which features an initial array of twenty elements, and a threshold of five. As this illustration clearly shows, *QSortWorker1* does not begin any processing until the first partitioning has been completed by *QSortWorker0*. However, after this has happened, processing continues in parallel until the final sorted array segments are returned to the *QSortMaster* process to be reconstructed into the sorted array. In the experimental setting discussed in Chapter 6, where large arrays are being sorted with a relatively small threshold value, each worker process quickly obtains an unsorted segment and the processing load is nicely balanced among participating worker processes.

5.2.3 Sorting Summary

This sorting application implements a slightly unconventional algorithm compared to other parallel sorting programs, however it will allow us to test the Tupleware system using different levels of granularity, due to the use of a threshold value. This allows us to adjust the application's behaviour so that it may range from a fine-grained through to a coarse-grained style of system.

5.3 Summary

The two applications discussed in this chapter exhibit a reasonably broad range of characteristics, which is one of the main reasons they were chosen. The ocean model is a tightly-coupled timestepped application, whereas the modified quicksort provides a highly configurable level of granularity by way of the threshold value while still requiring the communication of intermediate results between worker processes during execution. The contrasting properties of these two applications will provide us with a relatively ”complete set” of execution behaviours by which to test the performance of the system.

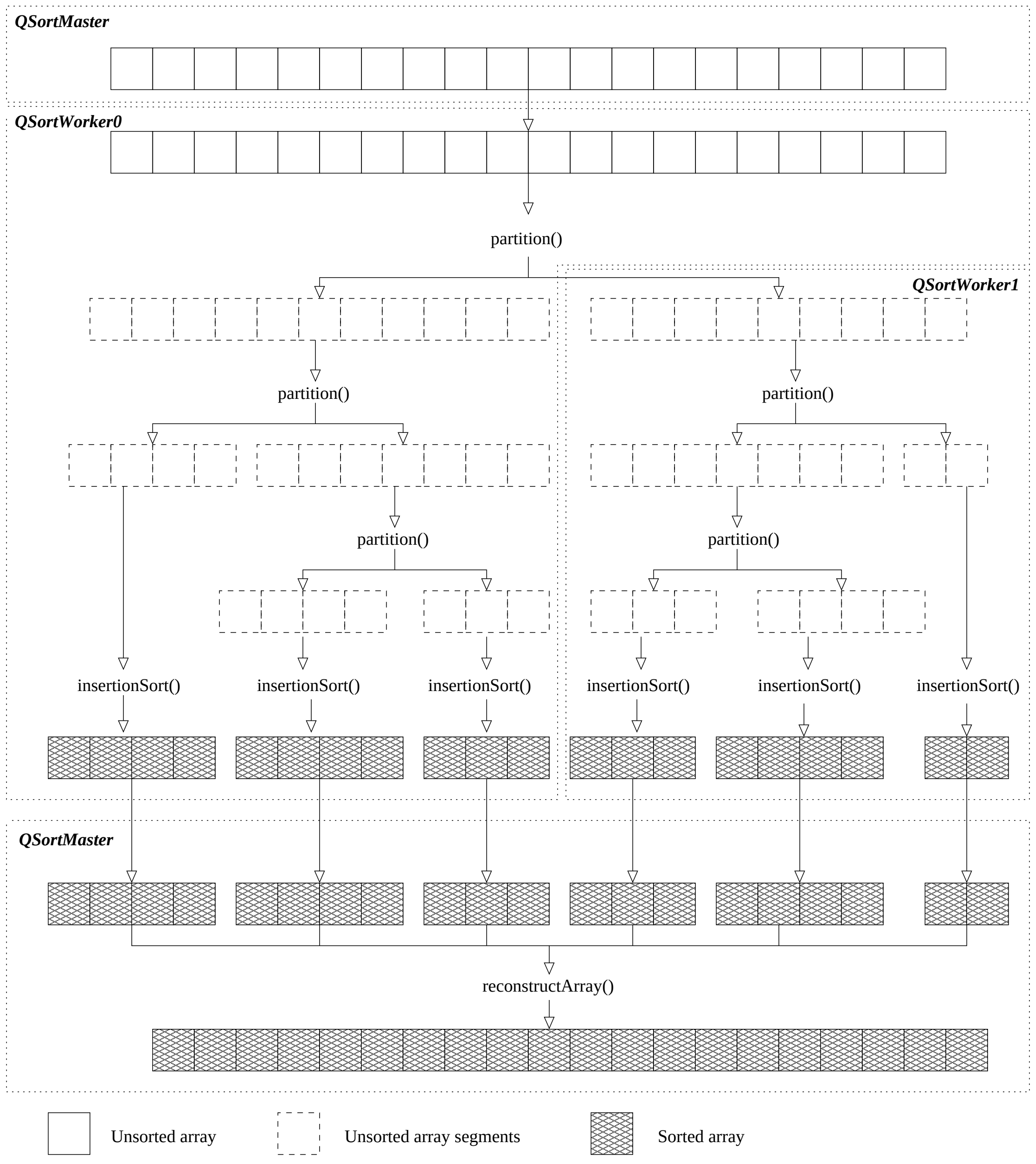


Figure 5.3: Flow of array segments through the modified quicksort application.

Chapter 6

Performance Evaluation

In order to test the success or otherwise of Tupleware in meeting its stated aims, the applications presented in Chapter 5 were executed on a cluster in order to measure their performance characteristics. This chapter introduces the performance metrics which will be used to evaluate the applications, and presents the results and analysis of this evaluation.

6.1 Performance Metrics

This section details the performance metrics which will be used in our evaluation of Tupleware. These metrics are mainly concerned with the speedup, communication efficiency, and scalability of the system, and they provide a way for us to understand the system's behaviour and evaluate its success or otherwise as a platform for the applications discussed in Chapter 5.

6.1.1 Speedup

When evaluating the performance of a parallel system, one of the most important measures is *speedup*, defined by Carriero & Gelernter (1990, p. 74) as ”the ratio of sequential runtime to parallel runtime”. For example, if a program runs twice as fast on two processors as it does on one, then the program's speedup will be *2*. We wish the speedup value to be as high as possible, assuming that the main goal of running a program in parallel is for it to execute faster. Based on Carriero & Gelernter's definition, speedup will be defined by the following formula:

$$S_p = \frac{T_1}{T_p}$$

where:

- p is the number of processors.
- T_1 is the runtime of the program when executed one one processor. Applied to the applications being evaluated in this chapter, this is equivalent to the time taken to execute with one worker node.
- T_p is the runtime of the program when executed on p processors.

An inverse relationship to speedup is *slowdown*, whereby a program's runtime may decrease as the degree of parallelism increases, or, even worse, the parallel runtime may be greater then the sequential runtime. Slowdown is obviously an unwanted occurrence, however it can be encountered when the overhead of parallelism begins to outweigh the the performance benefits gained.

For the applications being used to test Tupleware, we would expect them both to achieve a significant level of speedup. Due to the differing level of granularity of each, we would expect the sorting application to achieve a greater level of speedup than the ocean model, due to it being more coarse grained in nature, and hence require less network communication.

6.1.2 Efficiency

In a distributed system, any system overhead is usually in the form of network communication. Communication operations are relatively expensive in distributed environments, and so we wish to keep the frequency of these operations to a minimum. Furthermore, we wish to maximise the time each processor spends on doing actual computation. The ratio between computation and communication will be called the *efficiency* of the system for the purposes of discussion, and we wish this value to be as high as possible. For the purposes of our performance evaluation, we will define efficiency as the following:

$$Efficiency = \frac{T_{Computation}}{T_{Communication}}$$

where:

- $T_{Computation}$ is the time a process spends performing computation tasks, and
- $T_{Communication}$ is the time a given process spends performing network communication.

In practice, we would expect to find that as an application's workload is distributed amongst an increasing number of nodes, that the efficiency of an application would naturally decrease. This of course depends on the communication characteristics of the application in question: those whose communication time increases along with the number of nodes will experience dramatic decline in efficiency; those whose communication time remains constant will experience a gradual decline; and those whose communication times decrease with the number of nodes may be able to maintain a high level of efficiency.

6.1.3 Amdahl's Law

Even an application with very high efficiency will experience a limit on the level of speedup it will be able to achieve. This is due to the fact that all programs have an inherent limit to the degree that they can be parallelised. Stated another way, all programs contain parts that may be parallelised, and parts which cannot, and must be executed sequentially. It is this sequential section which imposes the limit on speedup. Amdahl's Law (Amdahl 1967) explains this scenario, and gives us a model with which to calculate the maximum possible speedup of a parallel program. It can be stated as:

$$S = \frac{1}{(1-P)+\frac{P}{N}}$$

where:

- S is the speedup of the program,
- P is the portion of a program which can be parallelised,
- $(1-P)$ is the portion of a program which cannot be parallelised (executes sequentially), and
- N is the number of processors.

What this formula basically states is that as N grows large (ie. the degree of parallelism increases) then the speedup experienced tends towards $1/(1-P)$. As an example, consider a program in which 95% of its execution can be parallelised, but the remaining 5% must be executed sequentially. A program such as this will not be able to gain more than a speedup of *20* (1/0.05), no matter how many processors are used.

6.1.4 Gustafson's Law

Amdahl's Law makes no allowance for the sequential part of a program changing in size as the number of processors and/or the problem size increases or decreases. Gustafson's Law (Gustafson 1988) makes this allowance by removing the idea of a fixed sequential portion of the program per processor, and instead introduces the idea of a fixed time for sequential execution followed by completely parallel execution on the available number of processors. Thus, given a large enough parallel processing to be completed, the sequential part of a program becomes insignificant. This distinction will be useful when it comes to analysing the performance of the applications being evaluated in this chapter. Gustafson's Law defines the speedup of a program as:

$$S(P) = P - \alpha \,\boldsymbol{.}\, (P - 1)$$

where:

- S is the speedup of the program,
- P is the number of processors, and
- α is the sequential part of the program.

The relevance of both Amdahl's Law and Gustafson's Law to the performance evaluation of Tupleware is that the common pattern of application execution is for the master process to be involved in the initialisation and final results merging phases of execution. These phases can be classified as sequential execution, as they both rely on the participation of the master process, and the time they take to complete is largely dependent on the capacity of the master process. The middle stage of a Tupleware application consists of worker processes executing in parallel, and so we can directly relate the execution of the applications being used for testing in this chapter to each of these formulae.

6.1.5 Scalability

Closely related to the concept of speedup is that of *scalability*. There are two aspects of scalability which will be outlined for the purposes of the remainder of this chapter: scalability in terms of the number of processors, and scalability in terms of the problem size. The former is directly related to speedup and Amdahl's Law; if a parallel program tends towards a speedup of N when executed on N processes, then it is said to be scalable. The latter aspect, scalability in terms of problem size, is concerned with how effectively the problem can be split amongst the available processors. That is, if a parallel system can execute a problem of size S in a time of T, then if the size of S doubles we wish the execution time to be no greater than $2.T$. If doubling the problem size results in significantly more than doubling the total runtime, the system would not be scalable in terms of problem size.

In a distributed system, an increase in problem size may result in an increase in the frequency of network communication, or in the amount of data that needs to be communicated. Situations such as this may cause the interconnection network to force a limit to the system's scalability, whether it is due to the limit of the network's throughput capacity, or due to the cumulative effects of latency experienced by network operations. The addition of extra nodes to the system may also cause these problems, by introducing additional load on the network.

6.1.6 Summary

The performance metrics discussed in this section will be used in the evaluation of the applications which follows. The profiling performed on the applications aimed to measure the relevant timings to allow us to discuss the Tupleware system in these terms.

6.2 Evaluation Methodology

The performance evaluation presented in the remainder of the chapter is intended to determine whether Tupleware is able to provide performance benefits in terms of scalability of problem size and number of participating nodes.

To this end, each of the applications described in Chapter 5 were executed a number of times, and the execution times of various components were measured, particularly the sequential and parallel parts of each application. This allowed us to form an overall picture of the behaviour of the system. More detailed descriptions of the experiments conducted are included in following sections.

The results gained from this evaluation are then compared with other published results of similar systems and applications.

6.3 Ocean Model

The ocean model discussed in Chapter 5 is an example of a tightly-coupled timestepped application. It exhibits frequent communication between processes, as boundary updates must occur after every timestep. However, as we will see, an increase in the problem size reduces the frequency of these communications, albeit with an increase in the amount of data that must be transferred. This makes for some interesting behaviour as the problem size is altered.

6.3.1 Execution Environment

The ocean model was executed on a sixteen-node cluster, with each node consisting of a Pentium 4 (3GHz) processor with 1GB of memory, and running Ubuntu Linux 8.04 (kernel 2.6) along with Java 6 update 10. Nodes were connected by a 100Mbps Ethernet network. Performance profiling was carried out using the *Clarkware Profiler* (Clark, 2008), which is able to measure the total and per-iteration runtimes (wall-clock time) between specified points in a program. Each process was executed with the following java command-line options: `-Xms512M` to set an initial heaps size of 512MB, and `-Xmx2048M` for a maximum heap size of 2GB.

6.3.2 Experimental Details

The ocean model was tested on a varying number of nodes, from one through to sixteen. When discussing the number of nodes taking part in the system, we are specifying the number of worker nodes. In this case, these are the nodes which are executing an `OceanModelWorker` process. In every instance there is also one master node (an `OceanModelMaster` process), however it is not involved in the parallel execution phase of the application, and so when evaluating the performance of the system we are concerned mainly with the number of worker nodes.

The size of the grid was also varied for experimental purposes, ranging from 1200x1200 through to 2400x2400 in increments of 200x200. This gives a substantial range of grid sizes, keeping in mind that the total number of grid points increases exponentially as the grid grows larger; this is illustrated in Table 6.1, which also details the amount of raw data stored in each sized grid.[1] A 2400x2400 grid was the largest possible for execution before some nodes, particularly the master node, began to use virtual memory, which would artificially effect the behaviour of the system.

The number of timesteps completed by the model remained constant at fifty; this gave the system sufficient parallel execution in order for a rigorous performance evaluation to be performed.

[1]Each grid point stores eight 64-bit double-precision floating point values. The size of the data in Table 6.1 should viewed as a conservative estimate, as the encapsulation of the data inside of objects would also increase memory usage.

Grid Size	*Grid Points (million)*	*Data (MB)*
1200x1200	1.44	87.9
1400x1400	1.96	119.6
1600x1600	2.56	156.3
1800x1800	3.24	197.8
2000x2000	4.00	244.1
2200x2200	4.84	295.4
2400x2400	5.76	351.6

Table 6.1: Total grid points and data size.

Finally, the Clarkware Profiler (Clark, 2008) was used to measure the execution times of specific sections of each worker process. The values measured are as follows:

Sequential Runtime This is the sequential part of the application's execution, and includes the initial setup phase where worker nodes are obtaining initial data and task details from the master node, and the final merge phase where each processed panel is returned by each worker node to the master node.

Communication This measure is of the time each worker process spends performing communication operations with other worker nodes during the parallel phase of the application's execution. It does not include the communication performed during the sequential phase.

Computation This measure is of the time each worker node spends performing actual computation of the grid points in its assigned panel.

Parallel Runtime This value represents the total execution time a worker node spends performing the parallel phase of the application's execution. The total runtime should be approximately equal to the time spent on computation plus time spent on communication, however there are some small parts of the code which are not included in these measures, so in practice the total runtime will be slightly greater than this.

Total Runtime This final value represents the total runtime of the system, and should be approximately equal to the sequential runtime plus the parallel runtime.

The complete experimental data for the ocean model can be found in Table 6.2 on the following page.

		Grid Size						
Nodes	*Profiled Task*	*1200*	*1400*	*1600*	*1800*	*2000*	*2200*	*2400*
1	Sequential	51.5	75.0	92.9	135.9	176.8	187.0	273.8
	I/O	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Processing	61.3	104.6	145.3	209.1	265.9	366.2	482.7
	Parallel Runtime	61.3	104.6	145.3	209.1	265.9	366.2	482.7
	Total Runtime	112.8	179.6	238.2	345.0	442.7	553.3	756.5
2	Sequential	28.5	39.7	49.1	62.0	77.0	113.9	148.8
	I/O	19.2	24.6	27.9	31.5	35.7	38.9	40.3
	Processing	14.4	24.3	35.7	54.7	74.1	109.8	135.2
	Parallel Runtime	33.7	49.0	63.5	85.0	109.8	146.8	174.5
	Total Runtime	62.2	88.6	112.6	147.0	186.8	260.7	323.3
4	Sequential	16.2	23.7	29.3	38.7	48.3	78.9	113.2
	I/O	61.4	65.5	73.0	75.2	81.5	90.6	100.2
	Processing	6.3	9.1	12.2	16.3	20.9	26.3	34.2
	Parallel Runtime	68.4	74.3	85.3	93.6	100.5	117.3	139.5
	Total Runtime	84.5	98.0	114.5	132.4	148.8	196.1	252.7
6	Sequential	15.5	22.7	29.7	33.5	44.7	56.3	85.7
	I/O	62.9	72.0	74.4	83.4	89.2	100.1	120.2
	Processing	4.2	5.9	8.2	10.2	12.9	15.4	19.2
	Parallel Runtime	67.2	77.9	82.7	93.6	102.1	115.6	153.2
	Total Runtime	82.6	100.6	112.4	127.1	146.8	171.9	238.9
8	Sequential	15.0	21.5	27.9	34.1	43.2	62.7	79.6
	I/O	75.8	78.0	89.0	90.1	90.3	97.3	128.6
	Processing	3.6	5.4	7.5	9.8	11.0	13.4	15.8
	Parallel Runtime	79.4	83.7	97.6	101.0	101.5	111.0	145.3
	Total Runtime	94.3	105.1	125.4	135.1	144.7	173.7	224.8
10	Sequential	13.4	18.4	24.0	34.6	46.2	77.3	70.0
	I/O	91.1	92.4	104.5	96.7	89.0	94.2	131.7
	Processing	2.4	3.4	4.6	6.5	9.1	10.0	12.8
	Parallel Runtime	93.6	99.7	109.0	103.7	98.2	103.8	145.0
	Total Runtime	107.0	118.1	133.0	138.4	144.4	181.1	215.0
12	Sequential	13.8	17.2	25.1	35.9	43.8	63.4	70.1
	I/O	99.4	98.7	109.7	107.2	107.9	113.7	144.0
	Processing	2.0	2.8	3.8	5.3	7.4	8.5	10.1
	Parallel Runtime	101.6	101.9	114.0	112.9	115.6	122.3	154.1
	Total Runtime	115.4	119.2	139.1	148.8	159.3	185.8	224.3
14	Sequential	14.9	20.6	27.3	36.3	43.1	59.7	68.9
	I/O	116.3	117.8	119.2	122.3	124.7	139.1	162.5
	Processing	1.9	2.5	3.3	5.0	6.9	7.2	9.0
	Parallel Runtime	119.1	120.9	112.8	127.3	132.4	146.8	172.3
	Total Runtime	134.0	141.5	140.2	163.6	175.5	206.5	241.2
16	Sequential	16.8	23.1	28.8	36.2	42.9	56.7	67.6
	I/O	142.7	147.5	149.2	164.3	185.3	172.4	176.9
	Processing	1.6	2.2	2.9	4.1	5.0	6.1	7.2
	Parallel Runtime	144.4	149.7	152.3	168.4	190.4	178.7	199.5
	Total Runtime	161.2	172.8	181.0	204.6	233.3	235.4	267.0

Table 6.2: Complete runtimes of ocean model.

6.3.3 Total Runtimes

Firstly we will look at the total runtimes of the ocean model to gain an understanding of its overall performance. Runtimes for the full range of grid sizes and nodes are given in Table 6.3, and these figures are illustrated in Figure 6.1.

	Grid Size						
Nodes	1200	1400	1600	1800	2000	2200	2400
1	112.8	179.6	238.2	345.0	442.7	553.3	756.5
2	62.2	88.6	112.6	147.0	186.8	260.7	323.3
4	84.5	98.0	114.5	132.4	148.8	196.1	252.7
6	82.6	100.6	112.4	127.1	146.8	171.9	238.9
8	94.3	105.1	125.4	135.1	144.7	173.7	224.8
10	107.0	118.1	133.0	138.4	144.4	181.1	215.0
12	115.4	119.2	139.1	148.8	159.3	185.8	224.3
14	134.0	141.5	140.2	163.6	175.5	206.5	241.2
16	161.2	172.8	181.0	204.6	233.3	235.4	267.0

Table 6.3: Total Runtimes (secs) of ocean model.

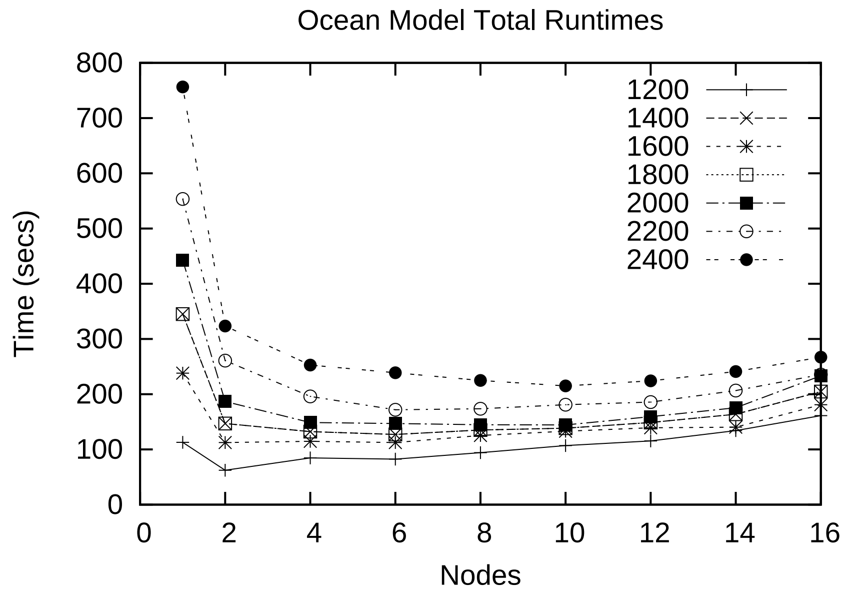


Figure 6.1: Total runtimes (secs) for the ocean model.

The speedup of the application given the same variables are shown in Table 6.4 and in Figure 6.2.

As these figures show, the speedup of the application in terms of total runtime is limited. However, the analysis of the behaviour of the application in greater detail in the sections that follow will describe why this may be the case and explore the relevance of these figures.

	Grid Size						
Nodes	1200	1400	1600	1800	2000	2200	2400
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.81	2.03	2.12	2.35	2.37	2.12	2.34
4	1.33	1.83	2.08	2.61	2.98	2.82	2.99
6	1.36	1.79	2.12	2.71	3.02	3.22	3.17
8	1.20	1.71	1.90	2.55	3.06	3.19	3.36
10	1.05	1.52	1.79	2.49	3.07	3.05	3.52
12	0.98	1.51	1.71	2.32	2.78	2.98	3.37
14	0.84	1.27	1.70	2.11	2.52	2.68	3.14
16	0.70	1.04	1.32	1.69	1.90	2.35	2.83

Table 6.4: Overall speedup of ocean model.

6.3.4 Sequential Runtime

The execution of the ocean model can be divided into three distinct stages: an initial stage where the master node generates the required data and the data are obtained by participating worker nodes, a second parallel stage where the worker nodes perform the processing, and a third and final stage where the final computed results are sent back to the master node. The first and third stages depend mainly on the master node, and can therefore be classed as the sequential parts of the application's execution. It is only the second stage which is completely parallel.

Table 6.5 lists the combined runtimes for these sequential stages of execution, and Figure 6.3 on page 106 illustrates how the sequential runtime relates primarily to the number of participating nodes in the system.

	Grid Size						
Nodes	1200	1400	1600	1800	2000	2200	2400
1	51.5	75.0	92.9	135.9	176.8	187.0	273.8
2	28.5	39.7	49.1	62.0	77.0	113.9	148.8
4	16.2	23.7	29.3	38.7	48.3	78.9	113.2
6	15.5	22.7	29.7	33.5	44.7	56.3	85.7
8	15.0	21.5	27.9	34.1	43.2	62.7	79.6
10	13.4	18.4	24.0	34.6	46.2	77.3	70.0
12	13.8	17.2	25.1	35.9	43.8	63.4	70.1
14	14.9	20.6	27.3	36.3	43.1	59.7	68.9
16	16.8	23.1	28.8	36.2	42.9	56.7	67.6

Table 6.5: Runtime (secs) of sequential stages of ocean model.

As we can see, an increase in the number of nodes substantially decreases the time taken for the initial application data to be delivered to each worker node, and

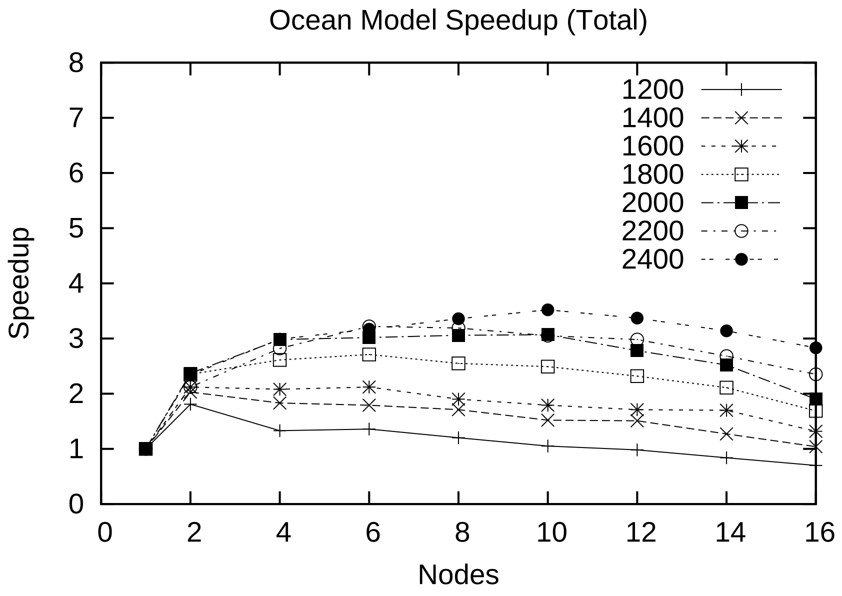


Figure 6.2: Overall ocean model speedup.

for the final processed panels to be returned to the master node. This would most likely be due to the use of a network switch, which would allow data to be sent simultaneously to each worker node. Increasing the number of worker nodes also reduces the size of data being transferred, as the width of each panel would be smaller. For example, in the case of a 1200x1200 grid, than a panel of size 1200x1200 would need to be sent to a worker node in a single node system, whereas a 1200x600 panel would be transferred in a two-node system.

These figures also show that in the cases of fourteen and sixteen nodes, the decrease in sequential runtime becomes negligible, or in some cases increases. This is likely due to the reliance on the master node, which is responsible for either transmitting or receiving all of the data. If we extrapolate these results to a larger number of nodes, then it is likely that these times would continue to slightly increase. However, quite plainly there are significant speedup benefits attained by adding extra nodes to the system in terms of these sequential runtimes.

It is also clear from these sequential runtimes that increasing the grid size results in an increase in time taken to transfer the data. This is due to the fact that larger grids consist of more data, which therefore takes longer to transfer. Figure 6.4 illustrates this in terms of the total number of grid points, and as we can see the increase in runtime is roughly proportional, which is what we would expect.

In summary, the initial setup and final result gathering phases of the ocean model are both necessary parts of the application, and occur before and after the parallel phase of execution. The time taken for these phases to complete depends on firstly

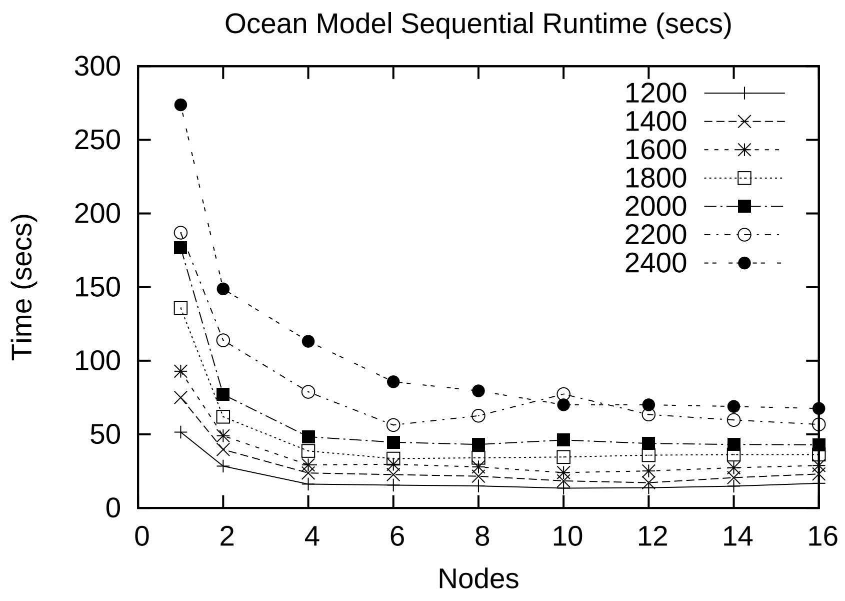


Figure 6.3: Sequential runtimes (secs) for varying number of nodes in the ocean model.

the size of the panels that are being processed, with an increase in size resulting in a longer time taken. Secondly, an increase in the number of worker nodes significantly decreases the time taken for these sequential phases, due to the characteristics of the network and the fact that smaller panels are being transferred.

6.3.5 Parallel Runtime

Now we have described the sequential runtime of the application, we can turn our attention to the behaviour during parallel execution. During this phase of execution, the worker nodes behave in a completely decentralised way, and communicate directly in order to share boundary values at each timestep. In addition to the total runtime of this phase, the other important measurements are time spent performing computation and the time spent performing network communications.

The times measured for network communications include time spent waiting for the requested data to become available. This is noted due to the fact that this is a timestepped application, and a worker node cannot continue on to the next iteration of processing until its neighbouring nodes (those processing a panel with a shared boundary region) have also completed the current iteration. This limitation has a significant impact on the efficiency of this application, as each node must spend a substantial amount of time after each timestep waiting for required data to become available in tuple space. Nonetheless, there are still some benefits realised from the parallelisation of the processing of this application, as we will see.

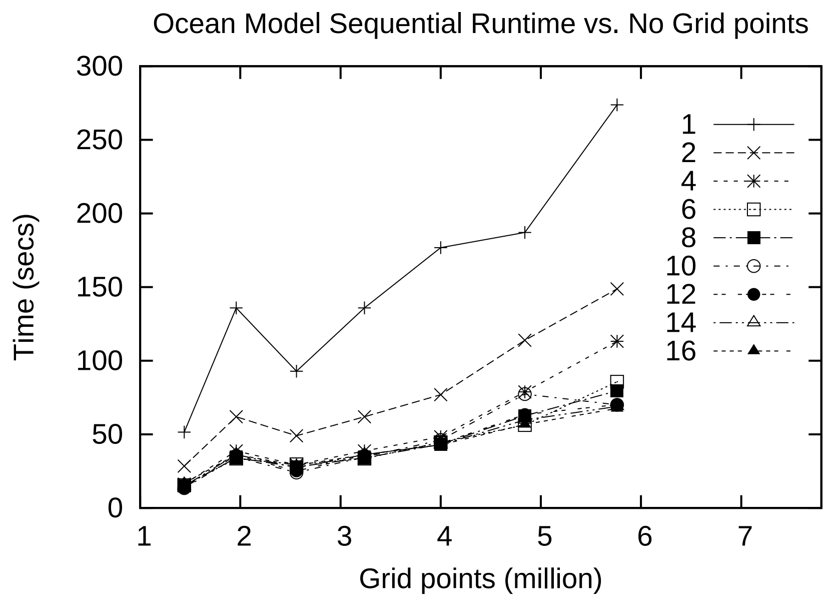


Figure 6.4: Sequential runtimes (secs) in terms of the total number of grid points.

The parallel runtimes for the ocean model for varying grid sizes and numbers of nodes are given in Table 6.6 and Figure 6.5. The related speedups can be found in Table 6.7 and Figure 6.6, which illustrates the speedup in terms of the number of nodes.

	Grid Size						
Nodes	1200	1400	1600	1800	2000	2200	2400
1	61.3	104.6	145.3	209.1	265.9	366.2	482.7
2	33.7	49.0	63.5	85.0	109.8	146.8	174.5
4	68.4	74.3	85.3	93.6	100.5	117.3	139.5
6	67.2	77.9	82.7	93.6	102.1	115.6	153.2
8	79.4	83.7	97.6	101.0	101.5	111.0	145.3
10	93.6	99.7	109.0	103.7	98.2	103.8	145.0
12	101.6	101.9	114.0	112.9	115.6	122.3	154.1
14	119.1	120.9	112.8	127.3	132.4	146.8	172.3
16	144.4	149.7	152.3	168.4	190.4	178.7	199.5

Table 6.6: Ocean Model parallel runtimes (secs).

The two activities undertaken by worker nodes during this parallel phase of execution are computation of the panel's grid points, and communication of boundary regions with other worker nodes. In order to obtain the best possible performance for the application, we wish to minimise the time spent performing communication in relation to the time spent performing computation. In other words, we wish to maximise the efficiency of the system. The relevant measurements of the efficiency

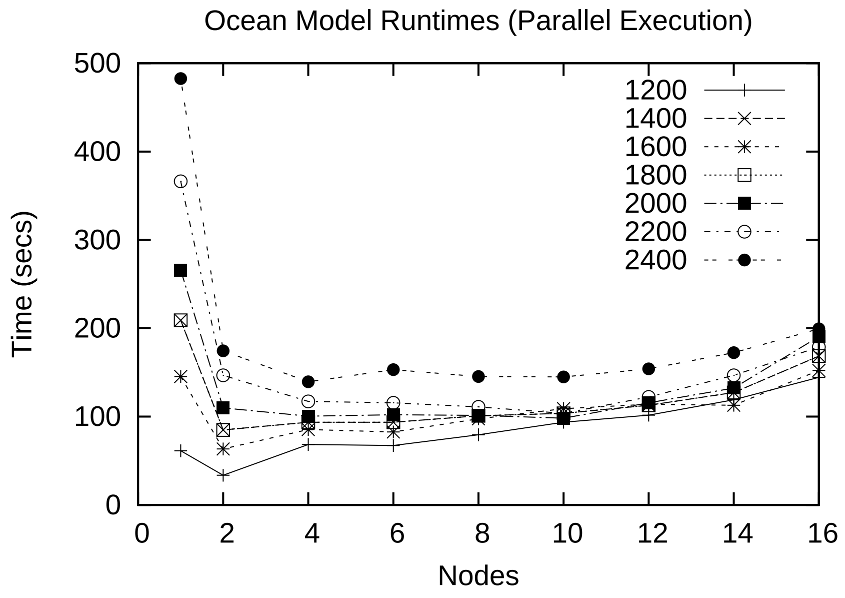


Figure 6.5: Parallel execution runtimes of the ocean model.

	Grid Size						
Nodes	1200	1400	1600	1800	2000	2200	2400
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.82	2.14	2.29	2.46	2.42	2.50	2.77
4	0.90	1.41	1.70	2.23	2.65	3.12	3.46
6	0.91	1.34	1.76	2.23	2.60	3.17	3.15
8	0.77	1.25	1.49	2.07	2.62	3.30	3.32
10	0.65	1.05	1.33	2.02	2.71	3.53	3.33
12	0.60	1.03	1.28	1.85	2.30	2.99	3.13
14	0.51	0.87	1.29	1.64	2.01	2.49	2.80
16	0.42	0.70	0.95	1.24	1.40	2.05	2.42

Table 6.7: Speedup of the ocean model's parallel phase.

of the application are provided in Table 6.8, using the definition of efficiency defined in Section 6.1.2.

What we can see from this table is that, while the efficiency of a two node system is satisfactory, a system with any greater number of nodes experiences a dramatic decrease. This is not due to the amount of data being transferred, as each panel has equal size boundaries (with the exception of the two edge panels, which only have one boundary). Rather, it is due to two factors related to the number of nodes in the system: the number of grid points relative to the size of the panel's boundary region, and the delayed availability of neighbouring panel's boundary values for the current timestep.

Firstly, increasing the number of nodes decreases the width of each panel, re-

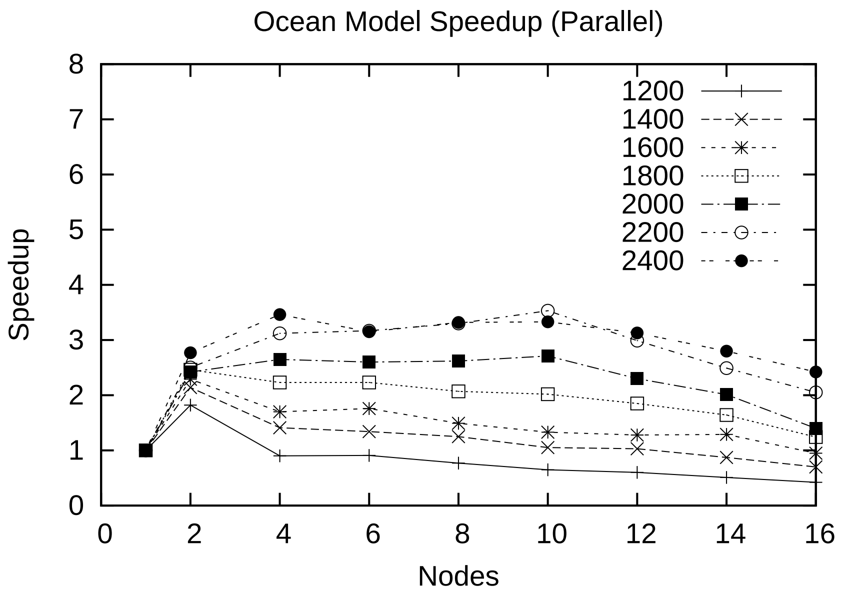


Figure 6.6: Speedup of the ocean model with varying number of nodes.

sulting in fewer grid points which need to be computed, and hence less time spent performing processing. However, with the size of each panel's boundary region remaining the same, the ratio between computation and communication inevitably becomes smaller.

Secondly, an increase in the number of nodes also increases the likelihood that some required boundary values will not be available between timesteps, resulting in additional time a node must spend searching for or waiting for the values to become available. These factors combine to produce a disappointing level of efficiency during the parallel phase of execution.

However, it can also be seen that an increase in grid size generally results in increased efficiency, something particularly apparent for systems with six or less nodes. This is due to an increased grid size resulting in a linear increase in boundary size along with an exponential increase in the number of grid points being computed. Whilst it would have been an interesting exercise to experiment with grid sizes greater than 2400x2400, it was at this point that the cluster nodes began to need to use virtual memory, which artificially effected the results.

6.3.6 Analysis & Summary

The discussion of the ocean model's performance in the preceding sections has shown the following to be true:

	Grid Size						
Nodes	*1200*	*1400*	*1600*	*1800*	*2000*	*2200*	*2400*
1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2	0.75	0.99	1.28	1.74	2.08	2.82	3.36
4	0.10	0.14	0.17	0.22	0.26	0.29	0.34
6	0.07	0.08	0.11	0.12	0.14	0.15	0.16
8	0.05	0.07	0.08	0.11	0.12	0.14	0.12
10	0.03	0.04	0.04	0.07	0.10	0.11	0.10
12	0.02	0.03	0.04	0.05	0.07	0.07	0.07
14	0.02	0.02	0.03	0.04	0.06	0.05	0.06
16	0.01	0.02	0.02	0.02	0.03	0.04	0.04

Table 6.8: Efficiency of the ocean model.

- The Tupleware system does provide the application with an overall speedup gain by distributing the application and processing it in parallel. However, the level of speedup is limited, with the best result being experienced with the largest grid size used.

- A significant part of this speedup gain is due to the decrease in time taken to perform the beginning and end sequential stages of the application's execution. This is due to the increased efficiency of network data transfer and, as the number of nodes is increased, smaller total panel sizes. Thus, the runtime of this sequential phase is not fixed, despite it being reliant on the single worker node.

- The parallel phase of execution also provides a limited level of speedup, and this is due to smaller panels requiring less time to compute. However, the increased time spent on network communications as the number of nodes grows cancels out these benefits.

The overall performance of the ocean model is to be expected given the application's characteristics, in particular its tightly-coupled nature and the fact that each node's execution is synchronised to a high degree with the nodes that are processing neighbouring panels.

Some encouragement can be taken from the fact that scalability tends to increase along with the problem size. Therefore we can conclude that the scalability would likely continue to improve if the grid size were increased to greater than 2400. However, the increase would need to be very significant, as the efficiency of this application clearly showed that the communications time significantly dominated the processing time of the application.

6.4 Sorting

The parallel sorting algorithm presented in Chapter 5 can be classed as a medium-grained application. It required communication between worker nodes in order to share partially sorted array segments, and as such it is not an ”embarrassingly parallel” application in which each node operates entirely independently without any communication being required during parallel execution. However, unlike the ocean model application, it is not timestepped, and so each worker node can execute relatively independently while they have data to process, and will only initiate communication with other nodes in order to obtain unprocessed array segments.

6.4.1 Execution Environment

The sorting application's performance was tested on the same cluster as for the ocean model, which was described in Section 6.3.1.

6.4.2 Experimental Details

Much like the ocean model, the sorting application was tested on a varying number of worker nodes, from one through to sixteen. A complete sorting application also consists of one master process, defined in the `QSortMaster` class, in addition to the worker nodes, which are defined in the `QSortWorker` class.

Profiling of the sorting application was undertaken using the Clarkware Profiler, and the measurements taken are much the same as for the ocean model. However, we will describe them again in terms of this particular application:

Initialisation This is the sequential part of the application's execution, involving the master node creating an appropriately sized array of random values, and for a worker node to obtain the array ready for partitioning. Note that, unlike the ocean model's sequential runtime, this measurement does not include the time spent returning processed data to the master node at the end of the application's execution. For the sorting application, sorted array segments are returned to the master node during the parallel phase of execution. Also unlike the ocean model, this application does not divide the unprocessed data before transferring it to worker nodes; rather, a single worker node obtains the entire unsorted array, and begins partitioning it (see Figure 5.3 on page 94). Other worker nodes cannot begin processing until they have received on of these unsorted partitions. Therefore, the sequential runtime is the average time it takes for a worker node to obtain an initial unsorted array segment.

Communication This has the same meaning as for the ocean model.

Computation This measure is of the time each worker node spends processing the array; this can involve either partitioning or sorting array segments as required.

Parallel Runtime This has the same meaning as for the ocean model, and is equal to the time spent performing communication plus the time spent performing computation.

Total Runtime This has the same meaning as for the ocean model, and is equal to the sequential runtime plus the parallel runtime.

In order to test the performance of this application with varying levels of granularity, three different threshold values were used to alter the amount of computation needing to be performed by each node during each execution. The values chosen were thirty-five thousand, fifty thousand and sixty-five thousand. Keeping in mind that the sorting algorithm being used, insertion sort, has an expected runtime of $O(n^2)$, this provides us with a significantly varied amount of processing for participating worker nodes. In all cases an initial array of five million random integers is used; this gives a sufficient amount of work for all worker nodes to complete and a sufficient number of array partitions for us to explore the behaviour of the Tupleware system.

6.4.3 Total Runtime

The total runtimes of the sorting application are presented in Table 6.9 and Figure 6.7.

	Threshold		
Nodes	35 000	50 000	65 000
1	94.3	131.1	193.7
2	49.0	67.6	86.4
4	35.6	36.5	48.8
6	20.4	28.3	36.7
8	14.8	17.5	23.0
10	12.9	16.6	20.9
12	9.5	14.3	16.7
14	10.6	12.6	14.3
16	11.2	9.0	12.4

Table 6.9: Total runtimes (secs) of the sorting application.

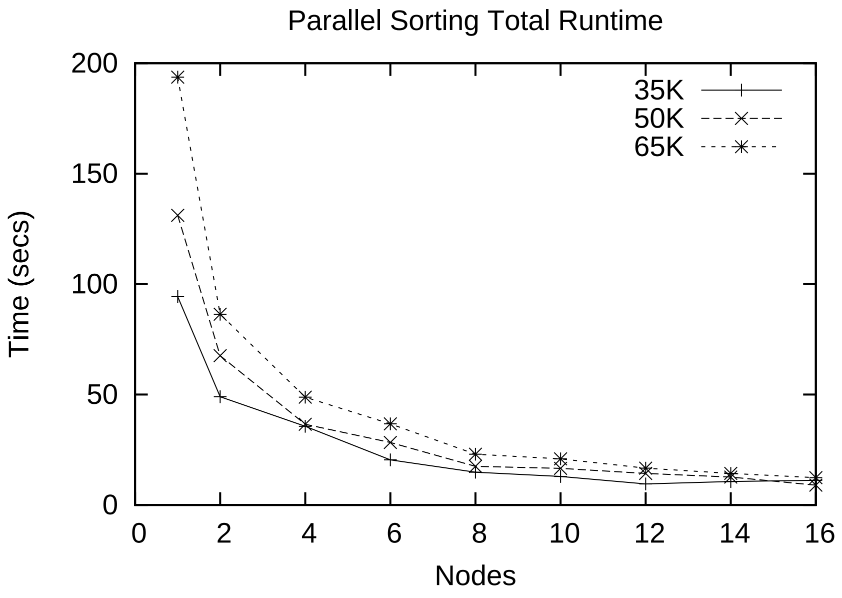


Figure 6.7: Total runtimes of the parallel sorting application.

As can be seen from the runtime for a single node system, sorting the array with the algorithm being used requires a significant amount of processing. Comparing this against a sixteen node system, we can see that the distribution and parallelisation of the application results in a substantial decrease in the overall runtime.

As a whole, the speedup experienced by the application is very pleasing. These speedup values can be found in Table 6.10, and are illustrated in Figure 6.8.

	Threshold		
Nodes	35 000	50 000	65 000
1	1.00	1.00	1.00
2	1.93	1.94	2.24
4	2.65	3.59	3.97
6	4.62	4.63	5.28
8	6.38	7.48	8.42
10	7.33	7.89	9.26
12	9.95	9.19	11.62
14	8.90	10.44	13.54
16	8.42	14.50	15.59

Table 6.10: Total speedup of the sorting application.

In the instances of a thirty-five thousand threshold being used, the speedup peaks at 9.95 on twelve nodes before decreasing on fourteen and sixteen node systems. Nonetheless, this still provides a reduction in total runtime from 94.3 seconds to 9.5 seconds, a total decrease of 84.8 seconds. Considering that this threshold size is the

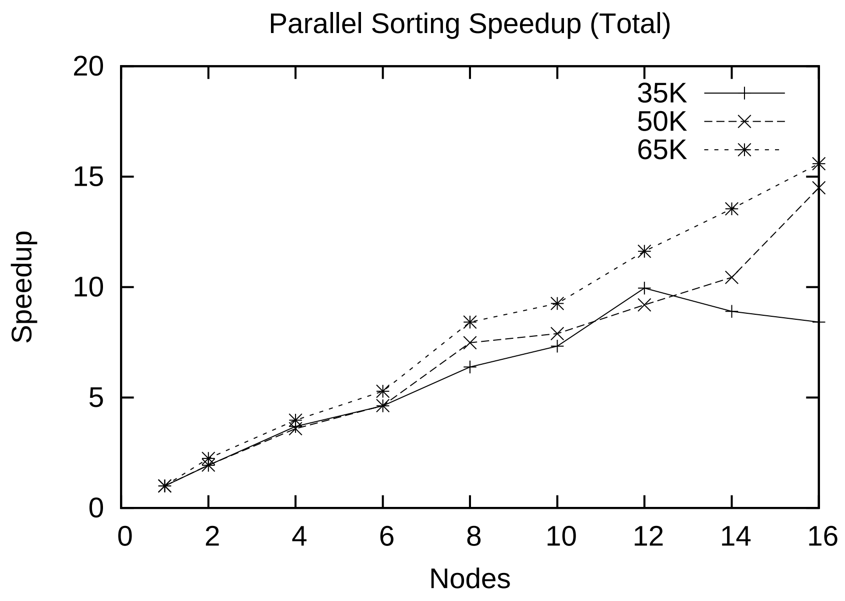


Figure 6.8: Total speedup of the sorting application.

smallest used for testing, and entails the greatest amount of network communication relative to other thresholds used, this is a pleasing result.

The two other threshold values used for testing gave a constant speedup up to sixteen nodes. In particular, for a threshold of sixty-five thousand, the speedup is near to optimal, and provides a total reduction in runtime of 181.3 seconds from 193.7 seconds on a single node system to 12.4 when sixteen nodes are used.

6.4.4 Initialisation

The initialisation phase encompasses the time taken for each worker node to obtain an unsorted array segment and begin processing it. As discussed previously, there will be some variance in how long each node must wait, as it is the responsibility of the worker node which initially obtains the full unsorted array to begin the partitioning process and supply unsorted segments to the other worker nodes. The initialisation times are shown in Table 6.11.

The runtimes being presented for this phase of execution denote the average time that worker nodes wait to obtain an unsorted array segment. In reality there will clearly be a delay between when the first and last worker nodes obtain an unsorted array segment, however by using the average time we can gain a clear overview of the application's behaviour during this phase.

As these figures show, the initialisation times are relatively constant, ranging between three and four seconds, which a few exceptions. The times tend to remain

	Threshold		
Nodes	35 000	50 000	65 000
1	1.7	1.7	1.7
2	3.5	3.5	3.5
4	3.2	3.4	3.8
6	3.3	3.2	3.0
8	3.2	1.0	1.8
10	3.0	2.7	3.0
12	1.1	3.2	1.8
14	3.5	3.0	2.2
16	5.5	1.8	0.6

Table 6.11: Sorting application's initialisation times (secs).

consistent with regards to the number of nodes, suggesting that this phase of the application's execution can be regarded as a fixed time.

6.4.5 Parallel Runtime

Parallel runtime is the sum total of the time spent performing network communications and processing once an initial unsorted array segment has been obtained. Network communications consist of obtaining additional unsorted segments once a worker's own storage in local tuple space has been exhausted, and also transferring sorted segments back to the master process. This will be affected by the threshold size: a smaller threshold requires more frequent communication with the master process, whereas a larger threshold requires less frequent.

Processing time is made up of the time spent performing two activities: array partitioning and sorting. In practice, the time spent partitioning an array is relatively tiny (maximum encountered was a time of 715 milliseconds on a one-node, thirty-five thousand threshold system). The partitioning time correlates with the threshold size, with a smaller threshold resulting in more instances of the partition function than for a larger threshold size. Time spent sorting, however, will increase along with the threshold size, as it takes exponentially longer to sort a larger array segment than it does to sort a short one. The performance data for the parallel phase of the application is shown in Table 6.12 on the following page.

The average total runtimes for this execution phase are illustrated in Figure 6.9 on page 117.

From this data we can firstly calculate the speedup in terms of the parallel phase of execution. Speedup is shown in Table 6.13 and illustrated in Figure 6.10.

As this shows, the speedup of the parallel phase of execution closely correlates

		Threshold		
Nodes	*Profiler Task*	35 000	50 000	65 000
1	Communication	7.54	6.29	4.21
	Processing	92.63	129.41	192.02
	Partitioning	0.72	0.68	0.61
	Sorting	91.91	128.73	191.40
	Total	100.17	135.70	196.23
2	Communication	6.16	4.78	4.81
	Processing	45.48	64.12	82.88
	Partitioning	0.38	0.37	0.33
	Sorting	45.10	63.75	82.55
	Total	51.64	68.90	87.69
4	Communication	6.08	4.88	2.78
	Processing	22.34	33.04	44.99
	Partitioning	0.21	0.19	0.19
	Sorting	22.13	32.85	44.79
	Total	16.26	28.16	42.20
6	Communication	6.23	4.36	4.21
	Processing	17.16	25.18	33.73
	Partitioning	0.17	0.16	0.16
	Sorting	16.98	25.03	33.57
	Total	23.38	29.54	37.94
8	Communication	6.51	4.07	4.66
	Processing	11.61	16.56	21.16
	Partitioning	0.13	0.13	0.09
	Sorting	11.48	16.43	21.07
	Total	18.12	20.63	25.82
10	Communication	6.49	6.12	5.64
	Processing	9.89	13.87	17.91
	Partitioning	0.10	0.09	0.06
	Sorting	9.79	13.78	17.85
	Total	16.38	20.00	23.54
12	Communication	6.02	8.67	9.50
	Processing	8.34	11.11	14.88
	Partitioning	0.05	0.06	0.02
	Sorting	8.29	11.05	14.86
	Total	14.36	19.78	24.38
14	Communication	7.54	8.90	8.96
	Processing	7.13	9.54	12.08
	Partitioning	0.05	0.05	0.03
	Sorting	7.09	9.50	12.04
	Total	14.68	18.45	21.03
16	Communication	9.67	9.23	9.81
	Processing	5.72	7.27	11.82
	Partitioning	0.04	0.03	0.07
	Sorting	5.68	7.25	11.75
	Total	15.38	16.50	21.63

Table 6.12: Parallel runtime data (secs) of the sorting application.

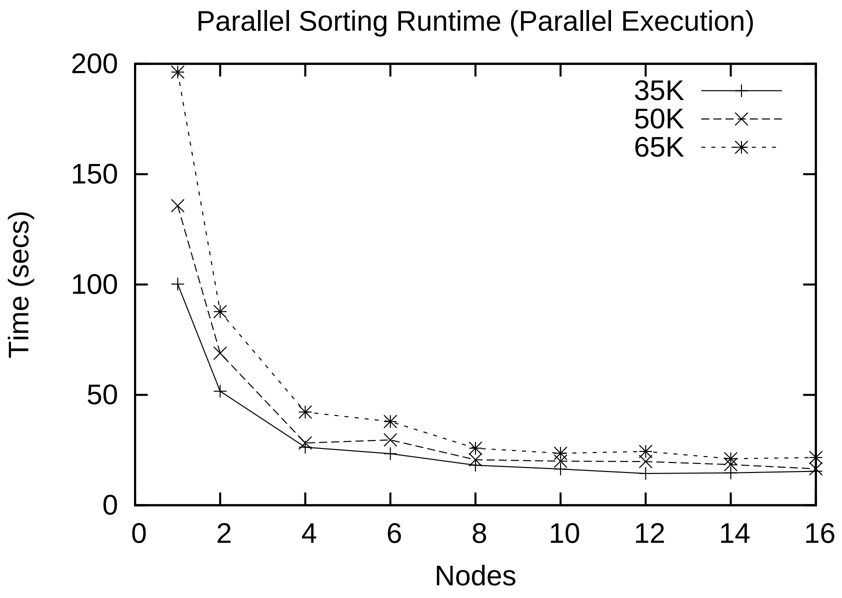


Figure 6.9: Sorting application's parallel execution runtimes.

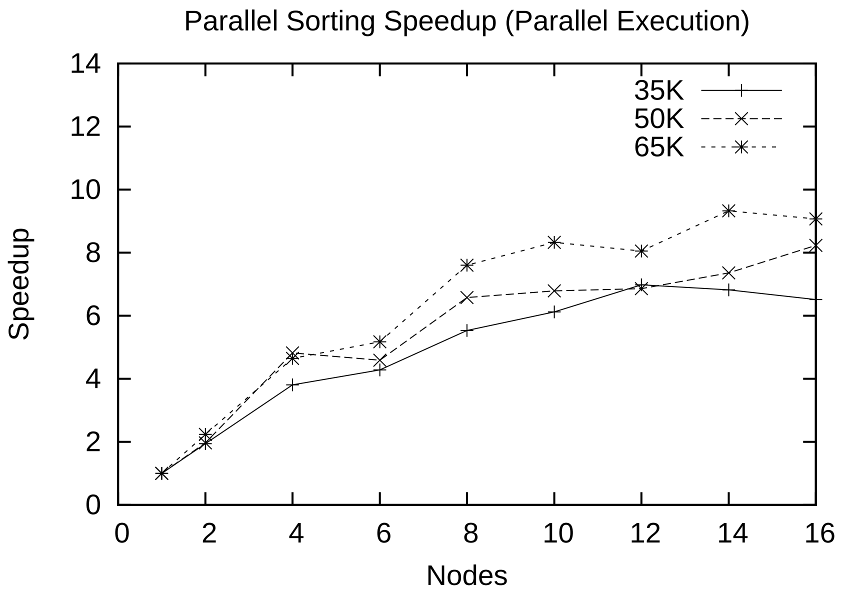


Figure 6.10: Speedup of parallel phase of sorting application execution.

	Threshold		
Nodes	35 000	50 000	65 000
1	1.00	1.00	1.00
2	1.94	1.97	2.24
4	3.81	4.82	4.65
6	4.28	4.59	5.17
8	5.53	6.58	7.60
10	6.12	6.79	8.33
12	6.98	6.86	8.05
14	6.82	7.36	9.33
16	6.51	8.23	9.07

Table 6.13: Parallel speedup of the sorting application.

to the speedup in terms of total runtime. For the smaller threshold value of thirty-five thousand, the speedup peaks at twelve nodes, however the speedup for the fifty thousand threshold continues through to sixteen nodes.

6.4.5.1 Efficiency

We can also analyse the parallel phase of execution in terms of its efficiency, which can be found in Table 6.14 and illustrated in Figure 6.11.

	Threshold		
Nodes	35 000	50 000	65 000
1	12.28	20.56	45.62
2	7.38	13.41	17.23
4	5.32	6.78	16.16
6	2.76	5.77	8.01
8	1.78	4.07	4.54
10	1.52	2.27	3.18
12	1.39	1.28	1.57
14	0.95	1.07	1.35
16	0.59	0.79	1.21

Table 6.14: Efficiency of the sorting application.

As these figures clearly show, the efficiency of the application decreases as the number of nodes increases. In order to find the reason for this, the profiler timings for both the communication and computation parts of the parallel execution phase, as shown in Table 6.9 on page 112, are presented graphically in Figure 6.12.

What is firstly clear from the graph of computation time is that the average time each node spends on processing decreases as the number of nodes becomes larger. The reason for this is obvious: as the number of participating worker nodes

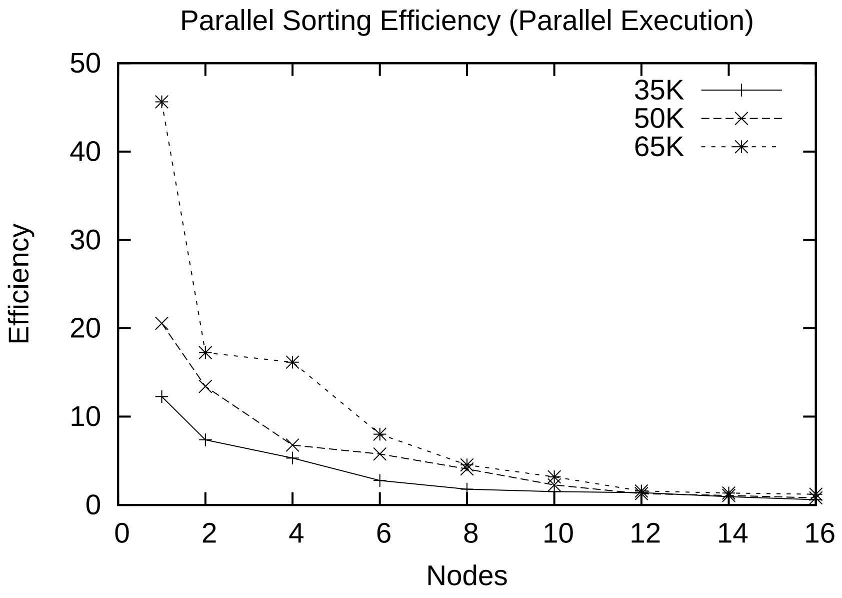
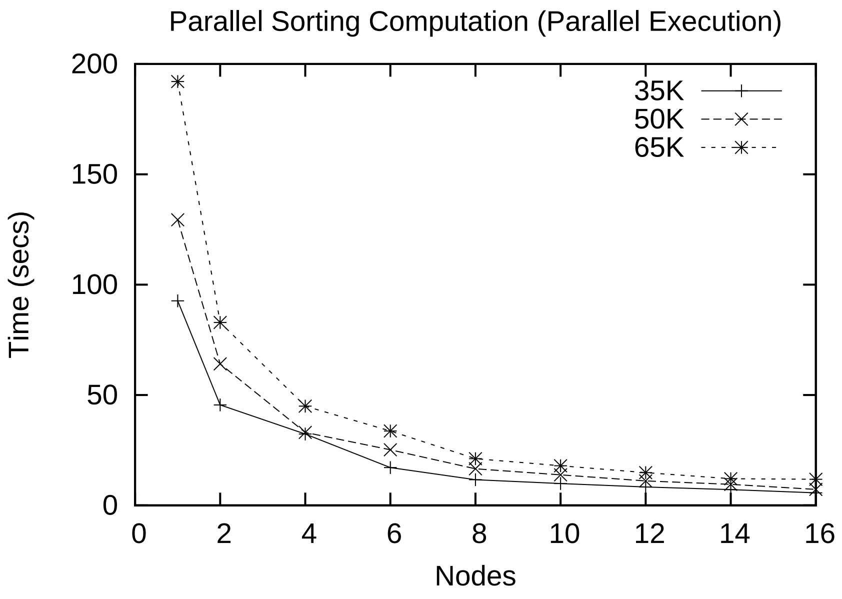


Figure 6.11: Efficiency of the sorting application on a varying number of nodes.

increases, the average portion of the array being sorted by each node will become smaller. This is the desired outcome, as the main goal of distributing a program is to allow the workload to be spread over the participating nodes.

The second observation which can be made, this time from the graph of communication times, is that increasing the number of nodes does not have a pronounced effect on the average time each node spends on communication. The times are reasonably constant, with perhaps a slight trend upwards are we pass ten nodes; however the increase is certainly not dramatic. This makes sense when we consider the behaviour of the application: once each node has obtained a relatively large segment of the unsorted array, it can continue processing mostly independently until this array segment is exhausted, at which point a new one will be obtained from another worker node. Also, because this application is naturally load-balanced, each worker node will spend an approximately equal amount of time sending sorted array segments back to the master process.

Therefore, the conclusion from the efficiency of this application is that it behaves in the way we would expect it to, given the fact that communication time is relatively constant and that the total amount of computation that is to be completed is effectively spread across participating nodes in the system.



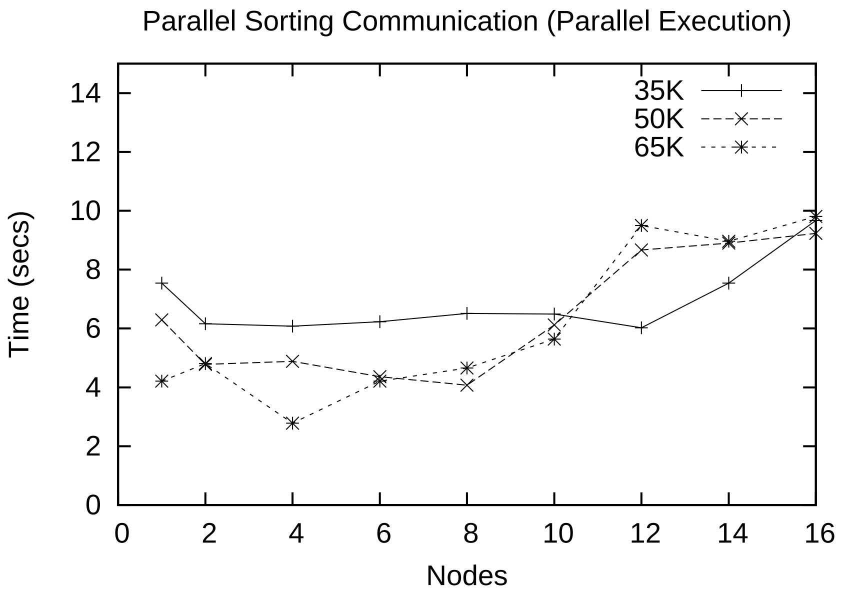


Figure 6.12: Communication and computation times (secs) of the parallel phase of the sorting application.

6.4.6 Analysis & Summary

The results of the performance testing of the sorting application in the previous section has shown the following facts to be true:

- The Tupleware system provides the application with significant performance gains and speedup. The speedup is most pronounced when a larger threshold is used. This is to be expected as increasing the threshold increases the granularity of the applications.
- The efficiency of the systems is exactly as we would expect. As the workload is distributed across the nodes in the cluster, the amount of work needing to be performed by each node naturally decreases. Meanwhile, the time spent on communication remains relatively constant, causing the drop in efficiency.
- Due to the communication time remain relatively static, the performance gains in this instance all occur during the parallel phase of execution.

These results are very pleasing, and demonstrate that the system is able to to provide speedup and performance gains for medium-grained applications. Based on Gustafson's Law, with the average communication time for each process remaining relatively constant as the number of nodes increases, while the total workload becomes larger, we would expect the application to continue to provide a high level of speedup as the number of nodes increases past sixteen. This prediction is further strengthened when we consider that increasing the problem size (via an increased threshold) in fact greatly increases the efficiency of the system.

6.5 Comparison with other systems

As we discussed in Section 3.6, scalability is an issue which faces many different implementations of tuple spaces, whether distributed or centralised. The studies presented in this section provide confirmation, to a point, of some of the observations and assertions made in this chapter.

The research by Wells, Chalmers & Clayton (2004) outlined the limits to the scalability of the centralised commercial systems, whereas the work carried out by Noble & Zlateva (2001) further explored the reasons behind this, and pointed to areas where gains could be made despite the inherent limitations of the tuple space model. The performance of Tupleware, as presented in this chapter, certainly compares favourably to some of the results presented in these cited studies, and addresses some of the issues involved in implementing a distributed tuple space in a scalable way.

6.6 Conclusions

This chapter has presented the performance results of two applications: an ocean model and a parallel sorting application. Each of these applications have different characteristics, which is partly why they were chosen to test Tupleware in terms of its aims of performance and scalability stated in Section 4.1.

The findings of this performance evaluation were pleasing in terms of the sorting application, which displayed a high level of speedup on up to the maximum number of sixteen nodes, and was effective in evenly distributing the processing workload amongst all participating nodes in the system. The performance of this application also clearly illustrated the effect of varying the granularity of each processing task, with the larger threshold size exhibiting a higher degree of speedup than the smaller threshold sizes. This was due to the time each process spent on network communications remaining relatively constant, while the processing performed per process decreased as more nodes were added to the system. This is a result typical of an application such as this, and we can conclude that the Tupleware system has met its aim in this case of providing a scalable platform upon which to develop this style of medium-grained application.

In terms of the ocean model, the results show that the overall speedup gain was limited, and that as the number of nodes increased, the time each node spent performing network communication placed a limiting factor to the continued scalability of the application. However, we also found that an increase in problem size, in this case the size of the grid, did not place a disproportionate load on any processes, and so there remains scope for the grid size to be increased further on a cluster with nodes with more than 1GB of main memory. The level of memory usage was the limiting factor with regards to the grid size on the cluster used for the testing detailed in this chapter, as the largest grid used (2400x2400) began to access virtual memory during execution, which dramatically slows performance.

Overall, we have shown that Tupleware can provide performance gains for distributed parallel applications, and that it can scale in terms of the number of nodes and also in terms of the problem size. While the performance of the tightly-coupled ocean model is not optimal, this is a common problem with tuple space-based systems, and the performance of Tupleware in this instance is comparatively good.

Chapter 7

Conclusions & Further Work

This chapter summarises the research presented in this thesis, and discusses its success in achieving its stated aims. Finally, some directions for possible future work are outlined.

7.1 Conclusions

This thesis has presented Tupleware, a distributed tuple space aimed at array-based parallel applications. We have detailed the design and implementation of the system with the two broad aims of producing a viable platform for the execution of distributed applications, and of doing this in such a way that the level of complexity remains low from the perspective of the application programmer.

The approach to evaluating these aims was to implement two applications on the system: an ocean model and a parallel sorting application. The contrasting characteristics of these applications allowed Tupleware's behaviour to be thoroughly analysed.

7.1.1 Scalability & Performance

The viability of the system relates to its ability to exhibit scalability in terms of the size of the system and in terms of the application problem size, and its ability to provide the application with increased performance. These aims were stated in Section 4.1, and evaluated in Chapter 6. The outcome of the performance testing conducted is summarised below.

For medium- and coarse-grained applications, embodied by the sorting application, Tupleware displayed a high level of scalability and performance gains. We found that the granularity of the application had a significant effect on the level of speedup achieved, with the finer-grained instance of the application achieving less speedup than the two more coarse-grained instances. However, to achieve consistent speedup on up to the maximum number of sixteen nodes demonstrates the viability of the system for this type of application.

For the tightly-coupled ocean modelling application, a the system delivered a performance gain by distributing the application over nodes in the cluster, however the level of speedup achieved was lower than for the sorting application. This was due to two main reasons: the frequency of communication was much higher for the ocean model, and the time spent performing this communication was greatly inflated by the time-stepped nature of the application's execution. The fact that each node spent significant time waiting for required values to become available from nodes processing neighbouring panels greatly diminished the performance gains experienced.

Overall, the performance results were pleasing. Tupleware provided obvious and significant performance gains and scalability to the sorting application. The class of applications which include the ocean model include much higher communication requirements, making it difficult to achieve the same level of speedup in a distributed computing environment, due to the high cost of latency. Given these factors, we believe that achieving even a modest level of speedup is a pleasing result.

7.1.2 Ease of programmability

As discussed in Chapter 4, the application programming interface was designed to preserve the simplicity and semantics of the operations found in Linda. Tupleware contains equivalent operations to Linda with the exceptions of `eval()`, as Tupleware does not support spawning of new processes at runtime, and also of the additional bulk retrieval operations `rdAll()` and `inAll()`. The semantics of Tupleware's operations are very similar to those found in Linda and JavaSpaces.

The other factor related to the ease of programmability of the system relates to the way additional tuple spaces are integrated into the system. Tupleware is implemented in such a way that the distribution of tuple space is completely transparent to the programmer, who does not need to worry about the issues of data distribution or locality, or the existence and location of remote instance of tuple space. It is left to the underlying runtime system to handle these issues, which it does by implementing an efficient algorithm for the retrieval of tuples from remote nodes. This algorithm uses the success or failure of previous requests in order to calculate the probability of a given instance of the tuple space being able to successfully fulfil future requests, and allows us to minimise the amount of network communication carried out by the system.

7.1.3 Contribution of the research

The contribution of this research is that the system produced successfully manages to transparently integrate multiple tuple spaces into a distributed tuple space without any added complexity for the programmer. Despite the distribution of the space, it is able to provide a scalable platform for distributed parallel array-based applications with a broad range of characteristics, and provide a significant level of performance gain.

The search algorithm is a large part of this contribution, as it allows a mechanism for the tuple space to dynamically adapt to the communications characteristics of the application at runtime, allowing naturally forming groups of nodes to optimise their communications. With the exception of SwarmLinda (discussed in Chapter 3), this is not an approach that has been widely researched in the field.

A further contribution of this research is that a complete, concrete implementation of the proposed techniques has been implemented and shown to be effective. This has allowed us to test the system in a real-world environment, which in turn helps to identify further enhancements which may be worthwhile pursuing in the future.

7.2 Further Work

Several possible areas of future research were identified based on the research presented in this thesis. These areas relate to ways that Tupleware might be able to be made more flexible, and to further increase its performance. These areas of further work can be summarised as follows:

- **Dynamic reconfigurability:** Currently, the number of nodes participating in a Tupleware system must be determined at compile time, to allow the tasks to be appropriately parallelised. It would be useful for many applications to be able to dynamically add nodes to the system at runtime, and for the system to reconfigure itself to make use of these nodes. Achieving this in a general way without adding complexity to the application may be difficult, but it is an area worth investigating.

- **Fault tolerance:** Related to the reconfigurability issue is that of fault tolerance. Currently, if a node becomes unavailable for any reason during an application's execution, the execution terminates abruptly. While this is not a huge problem in an experimental setting, it would be worth investigating ways in which the system might recover from faults when they occur in order to allow the application to continue to execute. This may involve some form of application checkpoints or data and task replication.

- **Search algorithm refinement:** The search algorithm used goes a long way towards boosting the efficiency of the system. Currently it utilises the success of historical searches to prioritise future search requests, however it may be worth investigating if the structure and contents of the tuple itself could be used to further refine the probability calculation. For example, in the ocean model we could use the actual indices of the panel being processed to determine exactly which neighbouring panel can provide the values we require. However, to achieve this is a generic way would be challenging, but if possible it could further increase the accuracy of tuple requests.

- **Security:** As is often the case with experimental systems such as this, Tupleware contains no provisions for any kind of security. While this does not pose a problem in a closed research setting, any system which is going to be used for applications which require confidentiality and integrity to be maintained should include mechanisms to do so.

Bibliography

Amdahl, G 1967, ’Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities’, *AFIPS Conference Proceedings*, (30), pp. 483-485.

Atkinson, A 2003 ’Coalescing Idle Workstations as a Multiprocessor System using JavaSpaces and Java Web Start’, Honours thesis, University of Tasmania.

Atkinson, A. and Malhotra, V 2004, ’Coalescing idle workstations as a multiprocessor system using Javaspaces and Java Web Start’, *In: Eighth IASTED Intl. Conference on Internet and Multimedia Systems and Applications*, August 16-18, 2004, Kauai, Hawaii, USA.

Birrel, A. & Nelson, B 1984, ’Implementing Remote Procedure Calls’, *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39-59.

Bittorrent.org 2008, The Bittorrent Protocol Specification, viewed 15 December 2008, <http://www.bittorrent.org/beps/bep_0003.html>

Boincstats 2008, Boincstats, viewed 15 December 2008, <http://boincstats.com/>

Carriero, N & Gelernter, D 1990, *How to Write Parallel Programs*, MIT Press, London.

CERN 2008, Overview, viewed 15 December 2008, <http://lcg.web.cern.ch/LCG/public/overview.htm>

Chapman, B. et al. 2008, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, Cambridge, Massachusetts.

Charles, A et al. 2004, ‘On the implementation of SwarmLinda’. *In ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pp. 297-298, New York, NY, USA.

Ciancarini, P., and D. Rossi. 1996. *Jada: a Coordination Toolkit for Java*. Technical Report, University of Bologna.

Ciancarini, Paolo, and Davide Rossi. 1997. ’Jada: Coordination and communication for Java agents’. In *Mobile Object Systems Towards the Programmable Internet*, pp. 213-226.

Clark, M., *Profiler API*, viewed June 12, 2008, <http://clarkware.com/software/Profiler.html>

Comer, DE 2000, *Internetworking with TCP/IP Vol 1: Principles, Protocols, and Architecture*, 4th edn, Prentice Hall, New Jersey.

Cormen, T. et al. 1999, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts.

Deering, S. & Hinden, R 1998, *RFC 2460: Internet Protocol, version 6 (IPv6) Specification*, Retrieved from ftp://ftp.rfc-editor.org/in-notes/rfc2460.txt.

Dijkstra, E 1968, ’Cooperating Sequential Processes’, in Genuys, F, *Programming Languages*, Academic Press, New York, pp. 43-112.

Distributed.net 2008, Distributed.net: Node Zero, viewed 15th December 2008, <http://distributed.net>

Foster, I & Kesselman, C 2003, *The Grid 2: Blueprint for a New Computing Infrastructure* (The Morgan Kaufmann Series in Computer Architecture and Design). Morgan Kaufmann.

Freeman, E., et al. 1999, *JavaSpaces Principles, Patterns and Practice*, Addison-Wesley, Massachusetts.

Geist, A et al. 1994, *PVM Parallel Virtual Machine, A User’s Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass.

Gelernter, D 1985, ’Generative Communication in Linda’, *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80-112.

GigaSpaces 2008, GigaSpaces, viewed 15 December 2008, <http://www.gigaspaces.com/>

Globus 2008, The Globus Alliance, viewed 15 December 2008, <http://www.globus.org/>

Gorlatch, S 2004, ‘Send-receive considered harmful: Myths and realities of message passing’. *ACM Transactions on Programming Languages and Systems*, vol 26, no 1, pp. 47-56.

Gustafson, J 1988, ‘Reevaluating Amdahl’s law’. *Communications of the ACM*, vol 31, no 5, pp. 532-533.

Hoare, CAR 1961, ‘Algorithm 64: Quicksort’, *Communications of the ACM*, vol 4, no 7.

JavaSpaces^{TM}Service Specification, 2003, Sun Microsystems, California.

Jini Technology Core Platform Specification, 2003, Sun Microsystems, California.

Lea, D., 2000, *Concurrent Programming in Java Second Edition: Design Principles and Patterns*, Addison Wesley Longman Inc., USA

Lehman, TJ et al. 1999, ‘T Spaces: The Next Wave’. In *HICSS*.

Leopold, C 2001, *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches*, John Wiley & Sons, Inc, New York.

Menezes, R and Tolksdorf, R 2003, ’A new approach to scalable Linda-systems based on swarms’, *Proceedings of the 18th Symposium on Applied Computing (SAC’03)*, Melbourne, Florida, USA.

Merrick, I. and A. Wood 2000, ’Coordination with scopes’, *In SAC ’00: Proceedings of the 2000 ACM symposium on Applied computing*, New York, NY, USA, pp. 210-217.

Merrick, I 2003, ’Scope-based coordination for open systems’, PhD thesis, University of York.

Murphy, A., 2007, LIME: Linda in a Mobile Environment, site viewed 25/5/2008, URL -http://lime.sourceforge.net/Lime/index.html

Murphy, A., Picco, G., Roman, G., 2006, *Lime: A Coordination Middleware Supporting Mobility of Hosts and Agents*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 15, no. 3, pp. 279-328.

Nielsen B & Slrensen T. 1994, ’Distributed Programming with Multiple Tuple Space Linda’, Masters Thesis, Aalborg University, Denmark.

Noble, M. S. and Zlateva, S. 2001, *Scientific Computation with Javaspaces*. Lecture Notes in Computer Science 2110, 657-667

Object Management Group 2008, Welcome to the OMG's CORBA Website, viewed 15 December 2008, <http://www.corba.org/>

Peterson, L. & Davie, B 2000, *Computer Networks: A Systems Approach*, Morgan Kaufman Publishers, San Francisco.

Pfister, G 1998, *In Search of Clusters*, Prentice Hall, New Jersey.

Rowstron, Antony I. T., and Alan Wood. 1996. 'An Efficient Distributed Tuple Space Implementation for Networks of Workstations'. *In Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume 1*, pp. 510-513.

Rowstron, A 1998, WCL: A Coordination Language to Geographically Distributed Agents, World Wide Web Journal, Volume 1, Issue 3, pp. 167-179.

Rowstron, A and Wray, S 1999, 'Run-Time System for WCL', *Internet Programming Languages* eds. H. Bal, B. Belkhouche and L. Cardelli, pages 78-96, Springer-Verlag, LNCS 1968.

Snir, M et al. 1996, *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA.

Sun Microsystems 2008, Remote Method Invocation Home, viewed 15 December 2008,
<http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>

Tanenbaum, A. & Steen, M 2002, *Distributed Systems Principles and Paradigms*, Prentice Hall, New Jersey.

University of California 2008, About SETI@Home, viewed 15 December 2008, <http://setiathome.berkeley.edu/sah_about.php>

Wells, GC et al. 2004, 'Linda implementations in Java for concurrent systems: Research Articles', *Concurrent Computing: Practice and Experience*, vol 16, no 10, pp. 1005-1022.

Wernstein, P, Pethick, M & Huang, Z 2003, 'A performance comparison of DSM, PVM and MPI', *Proceedings of the 2003 International conference on Parallel and Distributed Computing, Applications and Techniques*, Chengdu, China.

Appendix A

Tupleware Source Code

The source code listed below is for the components which make up the core Tupleware system.

A.1 Tuple

```
/**
 * Tuple.java
 *
 * Author:   Alistair Atkinson (alatkins@utas.edu.au)
 */

package space;

import java.io.Serializable;
import scope.*;

public class Tuple implements Serializable {
    protected Object[] fields;
    protected Scope scope;

    public Tuple(Object ... fields) {
        this.fields = (Object[]) fields;
        this.scope = new Scope(Scope.EMPTY_SCOPE);
    }

    public Tuple(Scope scope, Object ... fields) {
        this.fields = (Object[]) fields;  // Eclipse likes this better
        //this.fields = fields;  // Java 5 acceptable version
        this.scope = scope;
    }

    public int size() {
        return fields.length;
    }

    public Scope getScope() {
        return scope;
    }

    public void setScope(Scope scope) {
        this.scope = scope;
    }

    public boolean scoped() {
        return (scope == null);
    }

    public Object field(int i) {
        return (i < size() && i >=0)? fields[i]: null;
    }

    /* Overrides Object.toString() */
    public String toString() {
        StringBuffer s = new StringBuffer();

        s.append("<");
        for(int i = 0; i < fields.length; i++) {
            s.append(fields[i].toString() + ", ");
        }

        s.replace(s.length()-2, s.length(), ">");

        return s.toString();
    }
}
```

A.2 TupleTemplate

```
/*
 * TupleTemplate.java
 */

package space;

import scope.*;

public class TupleTemplate extends Tuple {
    public TupleTemplate(Object ... fields) {
        super((Object[]) fields);
    }

    public TupleTemplate(Tuple t) {
        this.fields = new Object[t.fields.length];
        for(int i = 0; i < this.fields.length; i++) {
            this.fields[i] = t.fields[i];
        }
    }

    /**
     * Implements associative matching rule. Wildcard fields are denoted by
     * null values. Fields are compared based on their string representation.
     */
    public boolean matches(Tuple tuple) {
        // cannot match if different sizes
        if(size() != tuple.size())
            return false;

        // test if fields match
        for(int i = 0; i < size(); i++) {
            if((this.field(i) != null)
                && (!this.field(i).toString().equals(tuple.field(i).toString()
                    )))
                return false;
        }

        // tuple must match if we get to this point
        return true;
    }

    public String toString() {
        Object[] fieldsCopy = new Object[fields.length];

        for(int i = 0; i < fields.length; i++) {
            if(fields[i] == null)
                fieldsCopy[i] = "null";
            else
                fieldsCopy[i] = fields[i];
        }

        return new Tuple(scope, fieldsCopy).toString();
    }
}
```

A.3 TupleSpace

```
package space;

import java.io.IOException;
import java.util.Vector;

public interface TupleSpace extends Comparable {
    public void out(Tuple t) throws IOException;
    public Tuple in(TupleTemplate t) throws IOException;
    public Tuple rd(TupleTemplate t) throws IOException;
    public Tuple inp(TupleTemplate t) throws IOException;
    public Tuple rdp(TupleTemplate t) throws IOException;
    public Vector<Tuple> inAll(TupleTemplate t, int expected) throws
        IOException;
    public Vector<Tuple> rdAll(TupleTemplate t, int expected) throws
        IOException;
    public int compareTo(Object o);
}
```

A.4 TupleSpaceImpl

```
/**
 * TupleSpaceImpl.java
 *
 * Author:   Alistair Atkinson (alatkins@utas.edu.au)
 */

package space;

import java.util.*;

public class TupleSpaceImpl implements TupleSpace {
    private Hashtable<String, Vector<Tuple>> tuples;

    public TupleSpaceImpl() {
        tuples = new Hashtable<String, Vector<Tuple>>();
    }

    public void out(Tuple t) {
        Vector<Tuple> vals = tuples.get(generateKey(t));
        if(vals == null) {
            vals = new Vector<Tuple>();
            tuples.put(generateKey(t), vals);
        }

        synchronized(vals) {
            vals.add(0, t);
            vals.notifyAll();
        }

        synchronized(tuples) {
            tuples.notifyAll();
        }
    }

    public void outAll(Collection<Tuple> tpls) {
        for(Tuple t : tpls) {
            Vector<Tuple> vals = tuples.get(generateKey(t));
            if(vals == null) {
                vals = new Vector<Tuple>();
                tuples.put(generateKey(t), vals);
            }

            synchronized(vals) {
                vals.add(0, t);
                vals.notifyAll();
            }
        }

        synchronized(tuples) {
            tuples.notifyAll();
        }
    }

    public Tuple in(TupleTemplate t) {
        return findTuple(t, true, true);
    }

    public Tuple inp(TupleTemplate t) {
        return findTuple(t, true, false);
```

```
}

public Vector<Tuple> inAll(TupleTemplate t, int expected) {
    return findAllTuples(t, true, expected);
}

public Tuple rd(TupleTemplate t) {
    return findTuple(t, false, true);
}

public Tuple rdp(TupleTemplate t) {
    return findTuple(t, false, false);
}

public Vector<Tuple> rdAll(TupleTemplate t, int expected) {
    return findAllTuples(t, false, expected);
}

/*
 * Returns first tuple found which matches given template. Otherwise,
 * returns null.
 */
private Tuple findTuple(TupleTemplate t, boolean remove, boolean block) {
    Vector<Tuple> vals = tuples.get(generateKey(t));

    while(vals == null) {
        if(block) {
            synchronized(tuples) {
                try {
                    tuples.wait();
                } catch(InterruptedException e) {
                    return null;
                }
            }
        } else {
            return null;
        }

        vals = tuples.get(generateKey(t));
    }

    synchronized(vals) {
        Tuple result = null;

        for(;;) {
            for(Tuple i: vals) {
                if(t.matches(i)) {
                    result = i;

                    if(remove)
                        vals.remove(result);

                    return result;
                }
            }

            if(block) {
                try { vals.wait(1000); }
                catch(InterruptedException e) {
                    return null;
                }
            } else {
                return result;
            }
        }
    }
}

private Vector<Tuple> findAllTuples(TupleTemplate t,
                                    boolean remove,
                                    int expected)
{
    Vector<Tuple> allTuples = new Vector<Tuple>();
    Vector<Tuple> matchingTuples = new Vector<Tuple>();

    synchronized(tuples) {
            Collection<Vector<Tuple>> c = tuples.values();

            for(Vector<Tuple> v : c) {
                if(v!=null) // work out why null values can be found here
                    allTuples.addAll(v);
            }

            for(Tuple tpl : allTuples)
                System.out.println(tpl.toString());*/

            for(Tuple tuple : allTuples) {
                if(t.matches(tuple)) {
                    matchingTuples.addElement(tuple);

                    if(remove) {
                        tuples.remove(tuple);
                    }
                }

                if(matchingTuples.size() >= expected) {
                    tuples.notifyAll();

                    return matchingTuples;
                }
            }

        return matchingTuples;
    }
}

private String generateKey(Tuple t) {
    System.out.println(t.field(0));

    /* SPECIAL CASES */
    if(t.field(0).equals("task"))
        return new String("task");

    if(t.field(0).equals("node"))
```

```
            return new String("node");

        if(t.field(0).equals("neighbourhood"))
            return new String("neighbourhood");

        if(t.field(0).equals("panel"))
            return new String("panel");

        if(t.field(0).equals("panel_"))
            return new String("panel_");

        if(t.field(0).equals("intermediate"))
            return new String("intermediate");

        if(t.field(0).equals("eta_inter"))
            return new String("eta_inter");

        if(t.field(0).equals("u_inter"))
            return new String("u_inter");

        if(t.field(0).equals("qtask"))
            return new String("qtask");

        if(t.field(0).equals("qresult"))
            return new String("qresult");

        if(t.field(0).equals("qsort"))
            return new String("qsort");

        if(t.field(0).equals("laplace"))
            return new String("laplace");

        if((t.field(0) instanceof String) && (((String)t.field(0)).contains("
            mb")))
            return new String("mandelbrot");

        StringBuffer buf = new StringBuffer();
        int it = (t.size() > 2)? 3: 1;

        for(int i = 0; i < it; i++) {
            buf.append(t.field(i).toString());
        }

        return new String(buf);
    }
}
```

A.5 TupleSpaceRequest

```
package space;

import java.io.Serializable;
import java.util.*;

public class TupleSpaceRequest implements Serializable {
    public static enum RequestType { OUT, IN, INP, RD, RDP, INALL, RDALL,
        OUTALL };

    private RequestType type;
    private Tuple tuple;
    private Vector<Tuple> tpls;
    private Integer expected;

    public TupleSpaceRequest(RequestType type, Tuple tuple) {
        this.type = type;
        this.tuple = tuple;
    }

    public TupleSpaceRequest(RequestType type, Tuple tuple, int expected) {
        this.type = type;
        this.tuple = tuple;
        this.expected = new Integer(expected);
    }

    public TupleSpaceRequest(RequestType type, Vector<Tuple> tpls) {
        this.type = type;
        this.tpls = tpls;
    }

    public Tuple getTuple() { return tuple; }
    public Vector<Tuple> getTuples() { return tpls; }
    public RequestType getRequestType() { return type; }
    public int getExpected() {
        return (expected == null)? 1: (int) expected;
    }
}
```

A.6 TupleSpaceResponse

```
package space;

import java.io.Serializable;
import java.util.Vector;

public class TupleSpaceResponse implements Serializable {
    public static enum Status { SUCCESS, ERROR, TIMED_OUT };
    private Status status;
    private Tuple tuple;
    private Vector<Tuple> tuples;

    public TupleSpaceResponse(Status status) {
        this.status = status;
    }

    public TupleSpaceResponse(Status status, Tuple tuple) {
        this.status = status;
        this.tuple = tuple;
    }

    public TupleSpaceResponse(Status status, Vector<Tuple> tuples) {
        this.status = status;
        this.tuples = tuples;
    }

    public Status getStatus() { return status; }
    public Tuple getTuple() { return tuple; }
    public Vector<Tuple> getTuples() { return tuples; }
}
```

A.7 TupleSpaceStub

```
/**
 * TupleSpaceStub.java
 *
 * Author:   Alistair Atkinson (alatkins@utas.edu.au)
 */

package space;

import static space.TupleSpaceRequest.*;
import static space.TupleSpaceResponse.*;
import scope.Scope;
import java.io.*;
import java.net.*;
import java.util.*;

public class TupleSpaceStub implements TupleSpace, Comparable {
    public final double ADJUST_FACTOR = 0.2;

    private InetSocketAddress sockAddress;
    private Socket socket;
    private ObjectOutputStream out;
    private ObjectInputStream in;
    private Scope associatedScope;
    public int id;
    public double successFactor;

    public TupleSpaceStub(InetSocketAddress sockAddress) {
        this.sockAddress = sockAddress;
        socket = new Socket();
        in = null;
        out = null;
        successFactor = 0.5;
    }

    public InetSocketAddress getSockAddress() {
        return sockAddress;
    }

    public boolean equals(TupleSpaceStub stub) {
        if(sockAddress.equals(stub.getSockAddress()))
            return true;

        return false;
    }

    /**
 * Implementation of TupleSpace interface
 */
public void out(Tuple t) throws IOException {
    doRequest(t, RequestType.OUT, 0);
}

public void outAll(Vector<Tuple> tpls) throws IOException {
    doRequest(new TupleSpaceRequest(RequestType.OUTALL, tpls));
}

public Tuple in(TupleTemplate t) throws IOException {
    return doSingleRequest(t, RequestType.IN);
}

public Tuple rd(TupleTemplate t) throws IOException {
    return doSingleRequest(t, RequestType.RD);
}

public Tuple inp(TupleTemplate t) throws IOException {
    return doSingleRequest(t, RequestType.INP);
}

public Tuple rdp(TupleTemplate t) throws IOException {
    return doSingleRequest(t, RequestType.RDP);
}

public Vector<Tuple> inAll(TupleTemplate t, int expected) throws
    IOException {
    return doBatchRequest(t, RequestType.INALL, expected);
}

public Vector<Tuple> rdAll(TupleTemplate t, int expected) throws
    IOException {
    return doBatchRequest(t, RequestType.RDALL, expected);
}

public Vector<Tuple> doBatchRequest(TupleTemplate t, RequestType type, int
     expected) throws IOException {
    return doRequest(t, type, expected).getTuples();
}

public Tuple doSingleRequest(TupleTemplate t, RequestType type) throws
    IOException {
    return doRequest(t, type, 1).getTuple();
}

/**
 * The doRequest() methods performs the actual tuplespace request using
 * the given parameters.
 */
private TupleSpaceResponse doRequest(Tuple tuple, RequestType type, int
    expected) throws IOException {
    TupleSpaceRequest request = new TupleSpaceRequest(type, tuple,
        expected);

    if((!socket.isConnected()) || (out == null) || (in == null)) {
        try {
                socket = new Socket(sockAddress.getAddress(), sockAddress.
                    getPort());
                out = new ObjectOutputStream(socket.getOutputStream());
                out.flush();
                in = new ObjectInputStream(socket.getInputStream());
            } catch (UnknownHostException e1) {
                // tuplespace unavailable, need to discover another
                System.out.println("UnknownHostException␣->␣"+sockAddress.
                    getAddress());
                return null;
            } catch(Exception e) {
                System.out.println("Failed␣to␣connect␣to:␣"+sockAddress.
                    getAddress());
                //e.printStackTrace();
                return null;
            }
        }

        out.writeObject(request);
        out.flush();

        TupleSpaceResponse response = null;
        try {

            response = (TupleSpaceResponse) in.readObject();

        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            return null;
        }

        if(response == null)
            return null;

        if(response.getStatus() == Status.ERROR)
            throw new IOException();

        return response; // this may still return null if performing an OUT
    }

    private TupleSpaceResponse doRequest(TupleSpaceRequest request) throws
        IOException {
        if((!socket.isConnected()) || (out == null) || (in == null)) {
            try {
                socket = new Socket(sockAddress.getAddress(), sockAddress.
                    getPort());
                out = new ObjectOutputStream(socket.getOutputStream());
                out.flush();
                in = new ObjectInputStream(socket.getInputStream());
            } catch (UnknownHostException e1) {
                // tuplespace unavailable, need to discover another
                System.out.println("UnknownHostException␣->␣"+sockAddress.
                    getAddress());
                return null;
            } catch(Exception e) {
                System.out.println("Failed␣to␣connect␣to:␣"+sockAddress.
                    getAddress());
                return null;
            }
        }

        out.writeObject(request);
        out.flush();

        TupleSpaceResponse response = null;
        try {
            response = (TupleSpaceResponse) in.readObject();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            return null;
        }

        if(response == null)
            return null;

        if(response.getStatus() == Status.ERROR)
            throw new IOException();

        return response; // this may still return null if performing an OUT
    }

    public void signalFailure() {
        if(successFactor == 1.0) return;

        successFactor = successFactor+(1.0 - successFactor)*ADJUST_FACTOR;
    }

    public void signalSuccess() {
        if(successFactor == 0.0) return;

        successFactor = successFactor-successFactor*ADJUST_FACTOR;
    }

    /* Implementation of the Comparable interface */
    public int compareTo(Object o) {
        if(!(o instanceof TupleSpaceStub))
            return 0;

        if(successFactor < ((TupleSpaceStub) o).successFactor)
            return -1;

        if(successFactor > ((TupleSpaceStub) o).successFactor)
            return 1;

        if(successFactor == ((TupleSpaceStub) o).successFactor)
            return 0;

        return 0;
    }
}
```

A.8 TupleSpaceService

```
/**
 *  TupleSpaceService.java
 *
 *  Author:     Alistair Atkinson (alatkins@utas.edu.au)
 */

package space;

import static space.TupleSpaceRequest.*;
import static space.TupleSpaceResponse.*;
import java.io.*;
import java.net.*;
import java.util.*;
import runtime.*;

public class TupleSpaceService extends Thread {
    public static final int DEFAULT_BULK_TIMEOUT = 2000;

    private TupleSpaceImpl space;
    private int port;
    private ServerSocket srvSocket;

    public TupleSpaceService(int port) {
        this.port = port;
        space = new TupleSpaceImpl();
    }

    public TupleSpaceService(int port, TupleSpaceImpl space) {
        this.port = port;
        this.space = space;
    }

    public int getPort() {
        return port;
    }

    /*
     *  Required Thread.run() method implementation.
     */
    public void run() {
        service();
    }

    public void shutdown() {
        try {
            if(srvSocket != null)
                srvSocket.close();
        } catch (IOException e) {}
    }

    protected void service() {
        try {
            srvSocket = new ServerSocket(port, 256);
            System.out.println("Socket␣opened␣at␣" + srvSocket);

            while(true) {
                Socket request = srvSocket.accept();
                new Thread(new RequestHandler(request , space)).start();
            }
        } catch (IOException ioe) {
            System.out.println("TupleSpaceService␣exiting.");
        }
    }

    public static void main(String[] args) {
        if(args.length < 1) {
            System.out.println("Usage:␣java␣space.TupleSpaceService␣<port>");
            System.exit(1);
        }

        int port = 0;
        try {
            port = Integer.parseInt(args[0]);
        } catch(Exception e) {
            System.out.println("Supplied␣port␣number␣must␣be␣a␣valid␣integer."
                );
            System.exit(1);
        }

        // start new tuplespace service
        System.out.println("TupleSpaceService␣started␣on␣port␣" + port);
        TupleSpaceService service = new TupleSpaceService(port);
        service.service();
    }
}

/**
 * A Runnable class that will handle tuplespace requests.
 */
class RequestHandler implements Runnable {
    private Socket socket;
    private TupleSpaceImpl space;

    public RequestHandler(Socket socket, TupleSpaceImpl space) {
        this.socket = socket;
        this.space = space;

        if(!socket.isConnected()) {
           System.out.println("GARRR!");
           return;
        }
    }

    public void run() {
        boolean commsErrorOccurred = false;
        ObjectInputStream in = null;
        ObjectOutputStream out = null;

        try {
            out = new ObjectOutputStream(socket.getOutputStream());
            out.flush();
            in = new ObjectInputStream(socket.getInputStream());
        } catch(IOException ioe) {
            ioe.printStackTrace();
        return;
    } catch(Exception e) {
        e.printStackTrace();
        return;
    }

    for(;;) {
        TupleSpaceResponse response = null;
        Tuple result = null;
        Vector<Tuple> batchResult = null;
        TupleSpaceRequest reqObj = null;

        try {
            reqObj = (TupleSpaceRequest) in.readObject();
        } catch(IOException ioe) {
            ioe.printStackTrace();
            response = new TupleSpaceResponse(Status.ERROR);
            commsErrorOccurred = true;
        } catch(Exception e) {
            e.printStackTrace();
            response = new TupleSpaceResponse(Status.ERROR);
            commsErrorOccurred = true;
        }

        switch(reqObj.getRequestType()) {
        case OUT:
          space.out(reqObj.getTuple());
          break;
        case OUTALL:
          space.outAll(reqObj.getTuples());
          break;
        case IN:
          result = space.in((TupleTemplate) reqObj.getTuple());
          break;
        case INP:
          result = space.inp((TupleTemplate) reqObj.getTuple());
          if(result == null) {
              try { synchronized(space) {space.wait(250);} }
              catch(InterruptedException e) { continue; }
              result = space.inp((TupleTemplate) reqObj.getTuple());
          }
          break;
        case INALL:
          batchResult = space.inAll((TupleTemplate) reqObj.getTuple(),
                                     reqObj.getExpected());
          break;
        case RD:
          result = space.rd((TupleTemplate) reqObj.getTuple());
          break;
        case RDP:
          result = space.rdp((TupleTemplate) reqObj.getTuple());
          if(result == null) {
              try { synchronized(space) {space.wait(250);} }
              catch(InterruptedException e) { continue; }
              result = space.rdp((TupleTemplate) reqObj.getTuple());
          }
          break;
```

```
            case RDALL:
              batchResult = space.rdAll((TupleTemplate) reqObj.getTuple(),
                                          reqObj.getExpected());
              break;
            default:
              result = null;
            }

            if(!commsErrorOccurred) {
                if(result != null)
                    response = new TupleSpaceResponse(Status.SUCCESS, result);
                else
                    response = new TupleSpaceResponse(Status.SUCCESS,
                        batchResult);
            }

            /* need to catch error immediately when writing response */
            try {
                if(out != null) {
                    out.writeObject(response);
                    out.flush();
                }
            } catch (IOException e1) {
                e1.printStackTrace();

                /*
                 * If an error occurs while writing response after a
                     destructive operation (IN, INP)
                 * then we need to replace the tuple that was remove from the
                     tuplespace.
                 */
                if(reqObj != null)
                    if((reqObj.getRequestType() == RequestType.IN) || (reqObj.
                        getRequestType() == RequestType.INP))
                        space.out(reqObj.getTuple());

                commsErrorOccurred = true;
            }

            if(commsErrorOccurred) {
                try {
                    in.close();
                    out.close();
                    socket.close();
                } catch(Exception e) {
                    return;
                }

                return;
            }
        }
    }
}
```

A.9 TupleSpaceRuntime

```
/**
 *  TupleSpaceRuntime.java
 *
 *  Author:   Alistair Atkinson (alatkins@utas.edu.au)
 */

package runtime;

import scope.*;
import space.*;
import java.io.*;
import java.util.*;
import java.net.*;
import com.clarkware.profiler.*;

public class TupleSpaceRuntime {
    public final int NODES = 1;

    private final InetSocketAddress GTS_ADDRESS =
                                    //new InetSocketAddress("144.6.40.143",
                                        6001); // staff-143
                                    //new InetSocketAddress("144.6.40.116",
                                        6001); // cluster-nhm-01
                                    //new InetSocketAddress("144.6.40.115",
                                        6001); // cluster-nhm-16
                                    //new InetSocketAddress("127.0.0.1", 6001)
                                        ;    // localhost
                                    new InetSocketAddress("10.10.10.13", 6001)
                                        ;    // software lab

    public TupleSpaceStub gts;
    public TupleSpaceImpl ts;
    private TupleSpaceService service;
    private Vector<TupleSpace> remoteSpaces;
    private RequestLogger log;
    private boolean isGlobal;
    public int requestCount, totalRequests;

    public TupleSpaceRuntime(int port, boolean isGlobal) {
        ts = new TupleSpaceImpl();
        remoteSpaces = new Vector<TupleSpace>(1, 1);
        this.isGlobal = isGlobal;
        log = new RequestLogger(3);
        requestCount = 0;
        totalRequests = 0;

        service = new TupleSpaceService(port, ts);
    }

    public void setScope(Scope s) {
        thisScope = s;
    }

    public void setDefaultScope() {
        thisScope = DEFAULT_SCOPE;
    }
```

```
public void start() {
    service.start(); // starts TupleSpaceService thread

    try {
        this.register();
    } catch (IOException e) {
        e.printStackTrace();
        return;
    }
}

public void stop() {
    service.shutdown(); // stops TupleSpaceService thread
    //Profiler.print();
    System.out.println(log.toString());
    log.reset();
}

/* Tuple space operations */
public void out(Tuple t) {
    Profiler.begin("out()");
    ts.out(t);
    Profiler.end("out()");
}

public void outAll(Vector<Tuple> tuples) {
    Profiler.begin("outAll()");
    ts.outAll(tuples);
    Profiler.end("outAll()");
}

/* Output given tuple to every connected remote node */
public void outEach(Tuple t) {
    Profiler.begin("outEach()");

    for(TupleSpace service : remoteSpaces) {
        try {
            service.out(t);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    Profiler.end("outEach()");
}

/* Output given tuple to randomly selected remote node */
public void outRand(Tuple t) {
    Profiler.begin("outRand()");

    int i = (int) Math.round(Math.random()*(remoteSpaces.size()-1));
    TupleSpace service = remoteSpaces.elementAt(i);
    try {
        service.out(t);
    } catch (IOException e) {
        e.printStackTrace();
    }
    Profiler.end("outRand()");
```

```
}

public Tuple in(TupleTemplate t) {
    sortStubs();
    requestCount++;

    Profiler.begin("in()");

    Tuple tpl = ts.inp(t);
    TupleSpace successful = null;

    if(tpl == null) {
        log.signalFail();
        for(TupleSpace service : remoteSpaces) {
            totalRequests++;
            try {
                tpl = service.inp(t);

                if(tpl != null) {
                    successful = service;
                    break;
                } else {
                    ((TupleSpaceStub) service).signalFailure();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    // move last successful stub to front of list
    if(successful != null) {
        remoteSpaces.remove(successful);
        remoteSpaces.add(0, successful);
        //((TupleSpaceStub) successful).signalSuccess();
    }

    // If we still haven't retrieved required tuple, send out requests
    // and wait for reply.
    if(tpl == null) {
        log.signalFail();
        Vector<Tuple> results = new Vector<Tuple>();
        dispatchConcurrentRequests(remoteSpaces,
                                   results,
                                   t,
                                   ConcurrentRequestThread.IN_OP);

        if(results.size() > 0) {
            tpl = results.elementAt(0);
        }

        if(results.size() > 1) {
            for(int i = 1; i < results.size(); i++)
                ts.out(results.elementAt(i));
        }
    }
```

```
        log.signalSuccess();
        Profiler.end("in()");

        return tpl;
    }

    public Tuple inp(TupleTemplate t) {
        sortStubs();

        Profiler.begin("inp()");

        Tuple tpl = ts.inp(t);
        Profiler.end("inp()");
        return tpl;
    }

    public Tuple rd(TupleTemplate t) {
        sortStubs();
        requestCount++;

        Profiler.begin("rd()");

        Tuple tpl = ts.rdp(t);
        TupleSpace successful = null;

        if(tpl == null) {
            log.signalFail();
            for(TupleSpace service : remoteSpaces) {
                totalRequests++;
                try {
                    tpl = service.rdp(t);

                    if(tpl != null) {
                        successful = service;
                        break;
                    } else {
                        ((TupleSpaceStub) service).signalFailure();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        // move last successful stub to front of list
        if(successful != null) {
            remoteSpaces.remove(successful);
            remoteSpaces.add(0, successful);
            //((TupleSpaceStub) successful).signalSuccess();
        }

        // If we still haven't retrieved required tuple, send out requests
        // and wait for reply.
        if(tpl == null) {
            log.signalFail();
            Vector<Tuple> results = new Vector<Tuple>();
            dispatchConcurrentRequests(remoteSpaces,
                                        results,
                                        t,
                                        ConcurrentRequestThread.RD_OP);

            if(results.size() > 0) {
                tpl = results.elementAt(0);
            }
        }

        Profiler.end("rd()");
        log.signalSuccess();
        return tpl;
    }

    public Tuple rdp(TupleTemplate t) {
        sortStubs();

        Profiler.begin("rdp()");

        Tuple tpl = ts.rdp(t);
        Profiler.end("rdp()");
        return tpl;
    }

    public Vector<Tuple> rdAll(TupleTemplate t, int expected) {
        sortStubs();

        Profiler.begin("rdAll()");

        Vector<Tuple> tpls = new Vector<Tuple>();
        Vector<Tuple> v = ts.rdAll(t, expected);
        if(v!=null)
            tpls.addAll(v);

        if(tpls.size() >= expected)
            return tpls;

        try{
            int i = 0;
            while((tpls.size() < expected) && (i < remoteSpaces.size())) {
                v = remoteSpaces.elementAt(i).rdAll(t, expected - tpls.size())
                    ;
                if(v!=null)
                    tpls.addAll(v);
                i++;
            }

        } catch(IOException e) {
            if(tpls.size() == expected)
                return tpls;
            else
                return null;
        }

        Profiler.end("rdAll()");

        return tpls;
```

```
}

public Vector<Tuple> inAll(TupleTemplate t, int expected) {
    sortStubs();

    Profiler.begin("inAll()");

    Vector<Tuple> tpls = new Vector<Tuple>();
    Vector<Tuple> v = ts.inAll(t, expected);
    if(v!=null) {
        tpls.addAll(v);
    }

    if(tpls.size() >= expected) {
        return tpls;
    }

    try {
        int i = 0;
        while((tpls.size() < expected) && (i < remoteSpaces.size())) {
            v = remoteSpaces.elementAt(i).inAll(t, expected - tpls.size())
                ;
            if(v!=null) {
                tpls.addAll(v);
                System.out.println(v.size());
            }
            i++;
        }

    } catch(IOException e) {
        if(tpls.size() == expected)
            return tpls;
        else
            return null;
    }

    Profiler.end("inAll()");

    return tpls;
}

private void batchReorder(Vector<Tuple> batchedTuples, final int field) {
    // Ensure all tuples are the same size. If not, then they are not
    // mutually comparable and so method returns
    int size = batchedTuples.elementAt(0).size();
    for(Tuple t : batchedTuples) {
        if(t.size() != size)
            return;
    }

    // Convert batchedTuples vector into array
    Tuple[] tpls = batchedTuples.toArray(new Tuple[batchedTuples.size()]);

    // Define our Comparator
    Comparator<Tuple> c = new Comparator<Tuple>() {
        public int compare(Tuple t1, Tuple t2) {
            return ((Comparable) t1.field(field)).compareTo((Comparable)
                t2.field(field));
        }
    };

    Arrays.sort(tpls, c); // Sort tuples
    batchedTuples = new Vector<Tuple>(batchedTuples.size());
    batchedTuples.copyInto(tpls); // copy array back into vector
}

public void register() throws IOException{
    if(isGlobal) {
        for(int i = 0; i < NODES; i++) {
            Tuple peerInfo = ts.rd(new TupleTemplate("node", null, (
                Integer) 6002+i));

            remoteSpaces.addElement(new TupleSpaceStub(new
                InetSocketAddress((String) peerInfo.field(1), (Integer)
                peerInfo.field(2))));
        }

        remoteSpaces.addElement(ts);

        System.out.println("Master␣successfully␣registered.␣" +
            remoteSpaces.size() +"␣services␣registered.");

        return;
    }

    gts = new TupleSpaceStub(new InetSocketAddress(GTS_ADDRESS.getHostName
        (),
                                                    GTS_ADDRESS.getPort()))
                                                        ;

    Tuple localInfo = new Tuple("node",
                                getLocalIPAddress(),
                                new Integer(service.getPort()));

    gts.out(localInfo);

    for(int i = 0; i < NODES; i++) {
        if(service.getPort() == 6002+i)
            continue;

        Tuple peerInfo = gts.rd(new TupleTemplate("node", null, (Integer)
            6002+i));

        remoteSpaces.addElement(new TupleSpaceStub(new InetSocketAddress((
            String) peerInfo.field(1), (Integer) peerInfo.field(2))));
    }

    System.out.println("we␣have␣all␣other␣node's␣info.␣" + remoteSpaces.
        size() +"␣services␣registered.");
}

private void dispatchConcurrentRequests(Vector<TupleSpace> stubs,
                                        Vector<Tuple> results,
                                        TupleTemplate template,
                                        boolean opType)
{
    Thread[] requestThreads = new Thread[remoteSpaces.size()];

    for(int i = 0; i < remoteSpaces.size(); i++) {
        requestThreads[i] = new ConcurrentRequestThread(remoteSpaces.
            elementAt(i),
                                                        results,
                                                        template,
                                                        opType);
    }

    for(Thread t : requestThreads) {
        t.start();
    }

    while(results.size() == 0) {
        synchronized(results) {
            try {
                results.wait();
            } catch(InterruptedException ie) {}
        }
    }

    for(int i = 0; i < requestThreads.length; i++) {
        requestThreads[i].interrupt();
    }

    for(int i = 0; i < requestThreads.length; i++) {
        try {
            requestThreads[i].join(500); // wait for threads to return?
        } catch(InterruptedException ie) {}
    }
}

public static String getLocalIPAddress() throws SocketException {
    Enumeration<InetAddress> interfaceAddresses =
        //NetworkInterface.getByName("eth0").getInetAddresses();
        NetworkInterface.getByName("eth1").getInetAddresses();

    interfaceAddresses.nextElement(); // pop first address (seems to
        generally be the IPv6 address)

    return interfaceAddresses.nextElement().getHostAddress();
}

private boolean isArrayElement(Tuple t) {
    if(t.size() < 4)
        return false;

    // Deal with special case for "task" tuples
    if((t.field(0) != null) &&
       (t.field(0) instanceof String) &&
       (((String)t.field(0)).equals("task")))
    {
        return false;
        }

        return ((t.field(1) != null) &&
                (t.field(2) != null) &&
                (t.field(1) instanceof Integer) &&
                (t.field(2) instanceof Integer));
    }

    private void sortStubs() {
        java.util.Collections.sort(remoteSpaces);
    }
}

class ConcurrentRequestThread extends Thread {
    public static final boolean RD_OP = false;
    public static final boolean IN_OP = true;

    private TupleSpace stub;
    private TupleTemplate template;
    private Vector<Tuple> result;
    private boolean opType;

    public ConcurrentRequestThread(TupleSpace stub,
                                   Vector<Tuple> result,
                                   TupleTemplate template,
                                   boolean opType)
    {
        this.stub = stub;
        this.template = template;
        this.result = result;
        this.opType = opType;
    }

    public void run() {
        while((result.size() == 0) && (!interrupted())) {
            Tuple res;

            try {
                if(opType == RD_OP) {
                    res = stub.rdp(template);
                } else {
                    res = stub.inp(template);
                }
            } catch(IOException ioe) {
                ioe.printStackTrace();
                return;
            }

            if(res != null) {
                synchronized(result) {
                    result.add(res);
                    result.notifyAll();
                    return;
                }
            }

            if(interrupted()) {
                return;
            }

            synchronized(this) {
                if(result.size() == 0) {
                    try {
                        this.wait(250);
                    } catch(InterruptedException ie) {
                        return;
                    }
                }
            }
        }
    }
}
```

Appendix B

Application Source Code

The source code listed in this section of the appendix is for the applications used to evaluate the Tupleware system.

B.1 Ocean Model

B.1.1 OceanModelMaster

```
/**
 *  OceanModelMasterv5.java
 *
 *  Author:    Alistair Atkinson (alatkins@utas.edu.au)
 *             Adapted from a program by John Hunter (John.Hunter@utas.edu.au)
 */

package apps.oceanModel;

import java.util.*;
import space.*;
import runtime.*;
import java.io.*;
import java.text.DecimalFormat;

public final class OceanModelMasterv5 {
    public static final int DEFAULT_PORT = 6600;
    public static final int NUM_PROCS = 16;
    private final int GRID_SIZE;
    private final int SLICE_WIDTH;

    // Model-related constants
    private final int im        = 10;
    private final int jm        = 10;
    private final int dx        = 300;
    private final int dy        = 300;
    private final int dt        = 25;
    private final int tout      = 500000;
    private final int tend      = 1225; //2200; //1000000;
private final double g      = 9.8;
private final int rho       = 1025;
private final double rho_a  = 1.2;
private final int uf        = 2;
protected final double cd   = 0.000025;
protected final double cd_a = 0.0013;
protected final double wx   = 5.0;
protected final double wy   = 5.0;

protected final double facgx = g*dt/((double) dx);
protected final double facgy = g*dt/((double) dy);
protected final double facbf = cd * ((double) dt);
protected final double facwx = wx*Math.sqrt(Math.pow(wx,2.0)+Math.pow(wy
    ,2.0))*cd_a*rho_a*dt/((double) rho);
protected final double facwy = wy*Math.sqrt(Math.pow(wx,2.0)+Math.pow(wy
    ,2.0))*cd_a*rho_a*dt/((double) rho);
protected final double facex = ((double) dt)/((double) dx);
protected final double facey = ((double) dt)/((double) dy);
protected final double hmax  = 0.8/(g*Math.pow(dt,2.0)*(1.0/Math.pow(dx
    ,2.0)+1.0/Math.pow(dy,2.0)));
protected final int itmax    = (int) Math.round(tend/dt)+2;

private Panel[] panels;

protected TupleSpaceRuntime ts;

public OceanModelMasterv5(int port, int gridSize) {
    GRID_SIZE = gridSize;
    SLICE_WIDTH = GRID_SIZE / NUM_PROCS;
    ts = new TupleSpaceRuntime(port, true);
}

public void createPanels() {
    panels = new Panel[NUM_PROCS];

    for(int i = 0; i < NUM_PROCS; i++) {
        panels[i] = new Panel(SLICE_WIDTH, GRID_SIZE, i);
        panels[i].numPanels = NUM_PROCS;
        panels[i].itmax = itmax;
        panels[i].facgx = facgx;
        panels[i].facgy = facgy;
        panels[i].facbf = facbf;
        panels[i].facwx = facwx;
        panels[i].facwy = facwy;
        panels[i].facex = facex;
        panels[i].facey = facey;
        panels[i].hmax  = hmax;
        panels[i].itmax = itmax;

        panels[i].SHARE_LEFT = true;
        panels[i].SHARE_RIGHT = true;

        if(i == 0) {
            panels[i].SHARE_LEFT = false;
        }

        if(i == NUM_PROCS-1) {
                panels[i].SHARE_RIGHT = false;
            }
        }
    }

    protected void start() {
        ts.start();

        this.createPanels();

        for(int i = 0; i < NUM_PROCS; i++) {
            ts.out(new Tuple("panel", panels[i]));
        }

        // Collect final results
        panels = new Panel[NUM_PROCS];
        System.gc();

        int timestep = itmax;

        for(int i = 0; i < NUM_PROCS; i++) {
            panels[0] = (Panel) ts.ts.in(new TupleTemplate("panel_", null)).
                field(1);
        }
        System.out.println("Master:␣All␣results␣received.");

        ts.stop();
    }

    public static void main(String[] args) {
        if(args.length != 2) {
            System.out.println("Usage:␣java␣OceanModelMasterv5␣<PORT>␣<
                GRID_SIZE>");
            System.exit(1);
        }

        int port = Integer.parseInt(args[0]);
        int grid = Integer.parseInt(args[1]);

        new OceanModelMasterv5(port, grid).start();
    }
}
```

B.1.2 OceanModelWorker

```
/**
 * OceanModelWorkerv5.java
 *
 * Author:   Alistair Atkinson (alatkins@utas.edu.au)
 *           Adapted from a program by John Hunter (John.Hunter@utas.edu.au)
 */

package apps.oceanModel;

import java.io.IOException;
import runtime.TupleSpaceRuntime;
import space.*;
import scope.Scope;
import com.clarkware.profiler.*;
import java.util.Vector;

public final class OceanModelWorkerv5 implements Runnable {
    private int port;
    private TupleSpaceRuntime ts;

    public OceanModelWorkerv5(int port) {
        this.port = port;
        ts = new TupleSpaceRuntime(port, false);
    }

    /* Runnable implementation */
    public void run() {
        System.out.println("Worker␣thread␣" + toString() + "␣started.");
        try {
            Profiler.begin("Total␣time␣doWork()");

            this.doWork();

            Profiler.end("Total␣time␣doWork()");
            Profiler.print();
        } catch(IOException e) {
            System.err.println("Worker␣thread␣" + toString() + "␣has␣died␣a␣
                horrible␣death.␣RIP.");
            e.printStackTrace();
            return;
        }
        System.out.println("Worker␣thread␣" + toString() + "␣finished.");
        System.out.flush();
        System.exit(1);
    }

    private void doWork() throws IOException {
        Profiler.begin("Sequential");
        ts.start();

        Panel task = (Panel) ts.gts.in(new TupleTemplate("panel", null)).field
            (1);
        Profiler.end("Sequential");

        while(task.kuv < task.itmax) {
            Profiler.begin("Iteration");
            Profiler.begin("processing");
            task.process();
            Profiler.end("processing");

            if(OceanModelMasterv5.NUM_PROCS == 1) {
                Profiler.end("Iteration");
                continue;
            }

            Profiler.begin("IO");

            Vector<Tuple> bvals = task.getBoundaryValues();
            for(Tuple t : bvals)
                ts.out(t);

            Vector<TupleTemplate> templates = task.getBoundaryTemplates();
            Vector<Tuple> intermediate = new Vector<Tuple>(templates.size()-1)
                ;

            for(TupleTemplate template : templates) {
                Tuple t = ts.in(template);
                intermediate.addAll((Vector<Tuple>) t.field(3));
            }

            Profiler.end("IO");

            task.updateBoundaries(intermediate);
            Profiler.end("Iteration");

            intermediate = null;
        }

        task.cleanUp();

        //return final result
        Profiler.begin("Sequential");
        ts.gts.out(new Tuple("panel_", new Panel()));
        Profiler.end("Sequential");

        System.out.println("Requests:␣" + ts.requestCount +"␣Instances:␣" + ts
            .totalRequests);

        ts.stop();
    }

    public static void main(String[] args) {
        if(args.length != 1) {
            System.out.println("Usage:␣java␣apps.oceanModel.OceanModelWorkerv5
                ␣<PORT>");
            System.exit(1);
        }

        int port = Integer.parseInt(args[0]);

        new OceanModelWorkerv5(port).run();
    }
}
```

B.1.3 Panel

```
/**
 * Panel.java
 *
 * Author:   Alistair Atkinson {alatkins@utas.edu.au}
 */

package apps.oceanModel;
```

```
import java.io.Serializable;
import java.util.*;
import space.*;
import scope.*;

public class Panel implements Serializable {
    public boolean SHARE_LEFT;
    public boolean SHARE_RIGHT;

    public double[][][] u, v, eta;
    public double[][]   h, hu, hv;

    public int kuv, keta, kuv_old, kuv_new, keta_old, keta_new;
    public int itmax, uf;
    public double facgx, facgy, facbf, facwx, facwy, facex, facey, hmax;

    public int width, length, id, numPanels;

    private Scope localScope;

    public Panel() {}

    public Panel(int width, int length, int id) {
        this.width = width;
        this.length = length;
        this.id = id;

        u   = new double[width+1][length][2];
        v   = new double[width+1][length][2];
        eta = new double[width+1][length][2];
        h   = new double[width+1][length];
        hu  = new double[width+1][length];
        hv  = new double[width+1][length];

        this.init();

        kuv = 1;
        keta = 1;
        kuv_old  = 0;
        kuv_new  = 1;
        keta_old = 0;
        keta_new = 1;
    }

    /* Return all boundary data points encapsulated in Tuple objects */
    public Vector<Tuple> getBoundaryValues() {
        Tuple l = null;
        Tuple r = null;

        Vector<Tuple> vals = new Vector<Tuple>();

        if(SHARE_LEFT) {
            for(int j = 0; j < length-1; j++) {
                vals.addElement(new Tuple("u", 1, j,
                                        new Double(u[1][j][kuv_new]),
                                        new Integer(kuv), new Integer(id)));
            }
        l = new Tuple("u_inter", id, kuv, vals);
    }

    vals = new Vector<Tuple>();

    if(SHARE_RIGHT) {
        for(int j = 0; j < length-1; j++) {
            vals.addElement(new Tuple("eta", width, j,
                                new Double(eta[width-1][j][keta_new]),
                                new Integer(keta), new Integer(id)));
        }

        r = new Tuple("eta_inter", id, keta, vals);
    }

    vals = new Vector<Tuple>();

    if(l != null)
        vals.add(l);

    if(r != null)
        vals.add(r);

    if(vals.size() == 0) return null;

    return vals;
}

/* Return Template objects suitable for retrieving boundary
    data points */
public Vector<TupleTemplate> getBoundaryTemplates() {
    Vector<TupleTemplate> templates = new Vector<TupleTemplate>();

    if(SHARE_LEFT) {
        templates.addElement(new TupleTemplate("eta_inter", id-1, keta,
            null));
    }

    if(SHARE_RIGHT) {
        templates.addElement(new TupleTemplate("u_inter", id+1, kuv, null)
            );
    }

    return templates;
}

/* Update boundary data points uding given tuples */
public void updateBoundaries(Vector<Tuple> vals) {
    for(Tuple val : vals) {
        int j = (Integer) val.field(2);
        String array = (String) val.field(0);

        if(array.equals("u"))
            u[width][j][kuv_new] = (Double) val.field(3);
        else if(array.equals("eta"))
            eta[0][j][keta_new] = (Double) val.field(3);
        }
    }

    /* Process next iteration of this panel */
    public void process() {
        // Step u-velocity:
        for(int j = 1; j < u[0].length-1; j++) {
            for (int i = 2; i < u.length-1; i++){
                u[i][j][kuv_new] = u[i][j][kuv_old]-facgx*(eta[i][j][keta_old
                    ]-eta[i-1][j][keta_old])-facbf*u[i][j][kuv_old]*uf/hu[i][j
                    ]+facwx/hu[i][j];
            }
        }

        // Step v-velocity:
        for(int j = 2; j < v[0].length-1; j++) {
            for (int i = 1; i < v.length-1; i++){
                v[i][j][kuv_new] = v[i][j][kuv_old]-facgy*(eta[i][j][keta_old
                    ]-eta[i][j-1][keta_old])-facbf*v[i][j][kuv_old]*uf/hv[i][j
                    ]+facwy/hv[i][j];
            }
        }

        /* INCREMENT KUV TIMESTEP */
        kuv++;

        /* REVERSE TIME INDICES */
        kuv_old = Math.abs(kuv_old-1);
        kuv_new = Math.abs(kuv_new-1);

        /* CALCULATE STEP ELEVATION */
        for(int j = 1; j < eta[0].length-1; j++) {
            for(int i = 1; i < eta.length-1; i++) {
                eta[i][j][keta_new] = eta[i][j][keta_old]-(facex*(u[i+1][j][
                    kuv_old]*hu[i+1][j]-u[i][j][kuv_old]*hu[i][j])+facey*(v[i
                    ][j+1][kuv_old]*hv[i][j+1]-v[i][j][kuv_old]*hv[i][j]));
            }
        }

        /* INCREMENT KETA TIMESTEP */
        keta++;

        /* REVERSE TIME INDICES */
        keta_old = Math.abs(keta_old-1);
        keta_new = Math.abs(keta_new-1);
    }

    public void init() {
        for(int i = 0; i < width; i++) {
            for(int j = 0; j < length; j++) {
                u[i][j][0]   = 0.0;
                u[i][j][1]   = 0.0;
                v[i][j][0]   = 0.0;
                v[i][j][1]   = 0.0;
                eta[i][j][0] = 0.0;
                eta[i][j][1] = 0.0;
                h[i][j]  = Math.random();
                hu[i][j] = Math.random();
                hv[i][j] = Math.random();
            }
        }
    }

    public void cleanUp() {
        h = null;
        hu = null;
        hv = null;
        System.gc();

        double[][][] utemp = new double[u.length][u[0].length][2];
        for(int i = 0; i < u.length; i++)
            for(int j = 0; j < u[0].length; j++)
                utemp[i][j][kuv_new] = u[i][j][kuv_new];
        u = utemp;
        utemp = null;

        double[][][] vtemp = new double[v.length][v[0].length][2];
        for(int i = 0; i < v.length; i++)
            for(int j = 0; j < v[0].length; j++)
                vtemp[i][j][kuv_new] = v[i][j][kuv_new];
        v = vtemp;
        vtemp = null;

        double[][][] etatemp = new double[eta.length][eta[0].length][2];
        for(int i = 0; i < eta.length; i++)
            for(int j = 0; j < eta[0].length; j++)
                etatemp[i][j][keta_new] = eta[i][j][keta_new];
        eta = etatemp;
        etatemp = null;
    }
}
```

B.1.4 Original FORTRAN version of Ocean Model

The following source code is for the original version of the ocean model application from which the version detailed in Chapter 5 was developed. This code was written by Dr. John Hunter of the Antarctic CRC in Hobart, Tasmania.

```
C ****************************************************************************
C *                                                                          *
C *                           FORTRAN SOURCE CODE                            *
C *                                                                          *
C *   PROGRAM SET :  MODELLING                        REF:JRH:01:11:2004     *
C *                                                                          *
C *   REVISION    :  -------------                       JRH:--:--:2004      *
C *                                                                          *
C *   SOURCE      :  m2d1605.f                                               *
C *   ROUTINE NAME:  m2d1605                                                 *
C *   TYPE        :  MAIN                                                    *
C *                                                                          *
C *   FUNCTION    :  Simple 2-D linear H-N model to show wind-driven         *
C *                  topographic gyres.                                      *
C *                                                                          *
C *                  Version of M2D1598 with two time levels for each variable.*
C *                                                                          *
C ****************************************************************************
C
      implicit none
```

```
C
      integer*4 imax,jmax
C
      parameter(imax=100,          ! Maximum number of grid cells in x-direction
     $          jmax=100)          ! Maximum number of grid cells in y-direction
C
      real*8 u(imax,jmax,2),v(imax,jmax,2),eta(imax,jmax,2),h(imax,jmax)
      real*8 hu(imax,jmax),hv(imax,jmax)
      real*8 sfx(imax,jmax),sfy(imax,jmax)
C
      real*8 dx,dy,dt,tout,tend,g,rho,rho_a,uf,cd,cd_a,wx,wy
      real*8 hmax,t
      real*8 facgx,facgy,facbf,facwx,facwy,facex,facey
C
      integer*4 i,j,im,jm,it,itmax,udum,vdum
      integer*4 kuv_old,kuv_new,keta_old,keta_new
      integer*4 ios
      integer*4 offset, middle
C
      character*32 fmt
C
C     Read in a record of input parameters:
C
C      im ..... number of grid cells in x-direction
C       jm ..... number of grid cells in y-direction
C       dx ..... grid size in x-direction
C       dy ..... grid size in y-direction
C       dt ..... time step
C       tout ... output interval
C       tend ... run duration
C       g ...... acceleration due to gravity
C       rho .... water density
C       rho_a .. air density
C       uf ..... background velocity (in bottom friction law)
C       cd ..... bottom friction coefficient
C       cd_a ... surface friction coefficient (for wind stress)
C       wx ..... wind velocity in x-direction
C       wy ..... wind velocity in y-direction
C
C      read(5,*,iostat=ios) im,jm,dx,dy,dt,tout,tend,g,
C     $                     rho,rho_a,uf,cd,cd_a,wx,wy
C      call error_handler(ios,001)                              ! Error point 001
C
C     Set these variables manually
      im=10
      jm=10
      dx=300
      dy=300
      dt=25
      tout=500000
      tend=100
      g=9.8d0
      rho=1025
      rho_a=1.2d0
      uf=2
      cd=0.000025
      cd_a=0.0013
      wx=5.0d0
      wy=5.0d0
C
C     Maximum depth for CFL criterion to be satisfied:
C
      hmax=0.8d0/(g*(dt**2)*(1.d0/(dx**2)+1.d0/(dy**2)))
C
C     Read in bathymetry (in same order as a printed map):
C    (i=1, i=im, j=1 and j=jm are assumed to be land)
C
      do j=jm,1,-1
C        read(5,*,iostat=ios) (h(i,j),i=1,im)
C        call error_handler(ios,002)                            ! Error point 002
C
C     Manually set bathymetry
        middle=im/2
        do i=1,im
          if (i.eq.1) then
            h(i,j)=0
          elseif (i.eq.im) then
```

```
            h(i,j)=0
          else
            offset=abs(middle-i)
            h(i,j)=10.0*(middle-offset)/middle
          endif
        end do
C
C     Reduce depth so that CFL criterion satisfied
C
        do i=1,im
          h(i,j)=min(h(i,j),hmax)
        end do
C
      end do
C
C     Initialise:
C
      do j=1,jm
        do i=1,im
          u(i,j,1)=0.d0
          u(i,j,2)=0.d0
          v(i,j,1)=0.d0
          v(i,j,2)=0.d0
          eta(i,j,1)=0.d0
          eta(i,j,2)=0.d0
          hu(i,j)=0.d0
          hv(i,j)=0.d0
          sfx(i,j)=0.d0
          sfy(i,j)=0.d0
        end do
      end do
C
C     Set initial time indices:
C
      kuv_old=1
      kuv_new=2
      keta_old=1
      keta_new=2
C
C     Interpolate depth to u and v points:
C
      do j=2,jm-1
        do i=3,im-1
          hu(i,j)=(h(i-1,j)+h(i,j))/2.d0
        end do
      end do
C
      do j=3,jm-1
        do i=2,im-1
          hv(i,j)=(h(i,j-1)+h(i,j))/2.d0
        end do
      end do
C
      t=0.d0
C
C     Calculate constants:
C
      facgx=g*dt/dx
      facgy=g*dt/dy
      facbf=cd*dt
      facwx=wx*sqrt(wx**2+wy**2)*cd_a*rho_a*dt/rho
      facwy=wy*sqrt(wx**2+wy**2)*cd_a*rho_a*dt/rho
      facex=dt/dx
      facey=dt/dy
C
C     Format statement for output:
C
      write(fmt,1) im
    1 format('(a3,''j'',i3.3,''it'',i5.5,',i3,'e12.4)')
C
C     Output depth:
C
      do j=jm,1,-1
        write(6,fmt) 'h  ',j,0,(h(i,j),i=1,im)
      end do
C
C     Step model forward:
```

```
C
      itmax=idnint(tend/dt)
C
      do it=1,itmax
C
C     Step u-velocity:
C
        do j=2,jm-1
          do i=3,im-1
C
            vdum=(v(i-1,j,kuv_old)  +v(i,j,kuv_old)
     $           +v(i-1,j+1,kuv_old)+v(i,j+1,kuv_old))/4.d0
            u(i,j,kuv_new)=u(i,j,kuv_old)
     $               -facgx*(eta(i,j,keta_old)-eta(i-1,j,keta_old))
     $               -facbf*u(i,j,kuv_old)
     $                 *uf/hu(i,j)
     $               +facwx/hu(i,j)
C
          end do
        end do
C
C     Step v-velocity:
C
        do j=3,jm-1
          do i=2,im-1
C
            udum=(u(i,j-1,kuv_old) +u(i,j,kuv_old)
     $           +u(i+1,j-1,kuv_old)+u(i+1,j,kuv_old))/4.d0
            v(i,j,kuv_new)=v(i,j,kuv_old)
     $               -facgy*(eta(i,j,keta_old)-eta(i,j-1,keta_old))
     $               -facbf*v(i,j,kuv_old)
     $                 *uf/hv(i,j)
     $               +facwy/hv(i,j)
C
          end do
        end do
C
C     Reverse time indices:
C
        kuv_old=3-kuv_old
        kuv_new=3-kuv_new
C
C     Step elevation:
C
        do j=2,jm-1
          do i=2,im-1
C
            eta(i,j,keta_new)=eta(i,j,keta_old)
     $                 -(facex*(u(i+1,j,kuv_old)*hu(i+1,j)
     $                         -u(i,j,kuv_old)*hu(i,j))
     $                  +facey*(v(i,j+1,kuv_old)*hv(i,j+1)
     $                         -v(i,j,kuv_old)*hv(i,j)))
C
          end do
        end do
C
C     Reverse time indices:
C         keta_old=3-keta_old
        keta_new=3-keta_new
C
C     Step time and test for output:
C
        t=dt*dble(it)
C
        if(dabs(dnint(t/tout)*tout-t).le.dt/2.d0) then
C
C     Calculate transport streamfunction both ways:
C     (positive around anticlockwise features)
C
          do j=3,jm
            do i=3,im-1
C
              sfx(i,j)=sfx(i,j-1)
     $                   +u(i,j-1,kuv_old)*hu(i,j-1)*dy
C
            end do
          end do
```

```
C
          do j=3,jm-1
            do i=3,im
C
              sfy(i,j)=sfy(i-1,j)
     $                  -v(i-1,j,kuv_old)*hv(i-1,j)*dx
C
            end do
          end do
C
!           do j=jm,1,-1
!             write(6,fmt) 'u   ',j,it,(u(i,j,kuv_old),i=1,im)
!             write(6,fmt) 'v   ',j,it,(v(i,j,kuv_old),i=1,im)
!             write(6,fmt) 'eta',j,it,(eta(i,j,keta_old),i=1,im)
!             write(6,fmt) 'sfx',j,it,(sfx(i,j),i=1,im)
!             write(6,fmt) 'sfy',j,it,(sfy(i,j),i=1,im)
!           end do
C
        endif
C
        do j=jm,1,-1
          write(6,fmt) 'u   ',j,it,(u(i,j,kuv_old),i=1,im)
          write(6,fmt) 'v   ',j,it,(v(i,j,kuv_old),i=1,im)
          write(6,fmt) 'eta',j,it,(eta(i,j,keta_old),i=1,im)
        end do
C
      end do
C
      stop
C
      end
C
      subroutine error_handler(ifail,error_point)
C ******************************************************************************
C *                                                                            *
C *                            FORTRAN SOURCE CODE                             *
C *                                                                            *
C *   PROGRAM SET :  MODELLING                         REF:JRH:06:01:1995      *
C *                                                                            *
C *   REVISION    :  -------------                         JRH:--:--:1995      *
C *                                                                            *
C *   SOURCE      :  forlib.f                                                  *
C *   ROUTINE NAME:  error_handler                                             *
C *   TYPE        :  SUBROUTINE                                                *
C *                                                                            *
C *   FUNCTION    :  Handles error (ifail.ne.0)                                *
C *                                                                            *
C ******************************************************************************
     integer*4 ifail,error_point
C
      if(ifail.ne.0) then
        write(6,1) ifail,error_point
    1   format(/' ifail returned as ',i5,' at error point ',i5,
     $           ' ..... program terminated'/)
        stop
      endif
      return
      end
```

B.2 Sorting

B.2.1 QSort

```
/**
 *   QSort.java
 *
 *   Author:    Alistair Atkinson (alatkins@utas.edu.au)
 */

package apps.sorting;

import space.*;
import runtime.*;
import java.util.*;
import com.clarkware.profiler.*;

public class QSort implements java.io.Serializable {
    private int[] a;
    private int threshold;

    // empty constructor - needed to enable poison pill
    public QSort() {}

    public QSort(int[] a, int threshold) {
        this.a = a;
        this.threshold = threshold;
    }

    public static void quicksort(int[] a, int p, int r) {
        if(p < r) {
            int q = partition(a, p, r);
            quicksort(a, p, q);
            quicksort(a, q+1, r);
        }
    }

    public static int partition(int[] a, int p, int r) {
        int x = a[p];
        int i = p - 1;
        int j = r + 1;

        while(true) {
            do { --j; } while (a[j] > x);
            do { ++i; } while (a[i] < x);
            if(i < j) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            } else {
                return j;
            }
        }
```

```
}

public static void insertionSort(int[] a) {
    for(int i = 1; i < a.length; i++) {
        int value = a[i];
        int j = i-1;

        while((j >= 0) && (a[j] > value)) {
            a[j+1] = a[j];
            --j;
        }
        a[j+1] = value;
    }
}

public static int[] merge(int[] a, int[] b) {
    int[] c = new int[a.length + b.length];
    int aIndex = 0;
    int bIndex = 0;
    int cIndex = 0;

    while((aIndex < a.length) && (bIndex < b.length)) {
        if(a[aIndex] < b[bIndex]) {
            c[cIndex] = a[aIndex];
            cIndex++;
            aIndex++;
        } else {
            c[cIndex] = b[bIndex];
            cIndex++;
            bIndex++;
        }
    }

    if(aIndex < a.length-1) {
        for(int i = aIndex; i < a.length; i++) {
            c[cIndex] = a[i];
            cIndex++;
        }
    }

    if(bIndex < b.length-1) {
        for(int i = bIndex; i < b.length; i++) {
            c[cIndex] = b[i];
            cIndex++;
        }
        }

        return c;
    }

    public static void initData(int[] a, int n) {
        if((a == null) || (a.length < n))
            a = new int[n];

        Random rand = new Random(System.currentTimeMillis());

        for(int i = 0; i < a.length; i++)
            a[i] = rand.nextInt();
    }

    public QSort split() {
        int q = partition(a, 0, a.length-1);
        int[] b = Arrays.copyOfRange(a, 0, q+1);
        int[] c = Arrays.copyOfRange(a, q+1, a.length);

        QSort dup;
        if(b.length > c.length) {
            a = b;
            dup = new QSort(c, threshold);
        } else {
            a = c;
            dup = new QSort(b, threshold);
        }

        return dup;
    }

    public boolean readyToSort() {
        return (a.length <= threshold);
    }

    public int[] getData() {
        return a;
    }

    public int size() { return a.length; }
}
```

B.2.2 QSortMaster

```
/**
 *  QSortMaster.java
 *
 *  Author:   Alistair Atkinson (alatkins@utas.edu.au)
 */

package apps.sorting;

import space.*;
import runtime.*;
import java.util.*;
import com.clarkware.profiler.*;

public class QSortMaster {
    private TupleSpaceRuntime ts;
    private int port;
    private int values;
    private int threshold;

    public QSortMaster(int port, int values, int threshold) {
        this.port = port;
        ts = new TupleSpaceRuntime(port, true);

        this.values = values;
        this.threshold = threshold;
    }

    public void start() {
        Profiler.begin("QSortMaster::TotalRuntime");

        System.out.println("Master␣process␣started.");

        // construct new quicksort object
        System.out.print("Initialising␣values...");
        Profiler.begin("Data␣Init");
        int[] a = new int[values];
        QSort.initData(a, values);
        QSort qs = new QSort(a, threshold);
        Profiler.end("Data␣Init");
        System.out.println("Done.");

        ts.start();
        // write to tuple space
        System.out.print("Adding␣data␣to␣tuplespace...");
        ts.outRand(new Tuple("qsort", qs, "unsorted"));
        System.out.println("Done.");
        a = null;
        qs = null;

        // collect sorted sections of array
        int n = 0;
        Vector<int[]> sortedPartitions = new Vector<int[]>();
        System.out.print("Waiting␣for␣data␣to␣be␣sorted...");
        while(n < values) {
            Tuple t = ts.ts.in(new TupleTemplate("qsort", "
               sorted", null));
            QSort q = (QSort) t.field(2);
            sortedPartitions.addElement(q.getData());
            n += q.getData().length;
            System.out.println("collected␣" + n + "␣elements.")
               ;
        }
        System.out.println("Done.␣" + sortedPartitions.size() +
            "␣partitions␣collected.");

        // distribute poison pill
        ts.outRand(new Tuple("qsort", new QSort(),"complete"));

        // reconstruct array
        a = reconstructArray(sortedPartitions);

        ts.stop();
        Profiler.end("QSortMaster::TotalRuntime");

        Profiler.print();
        //System.exit(0);
    }

    private int[] reconstructArray(Vector<int[]> parts) {
        Profiler.begin("QSortMaster::reconstructArray()");
        int[] res = new int[values];
        int[] nextPart;
        int n = 0;
        do {
            nextPart = nextPartition(parts);
            for(int i = 0; i < nextPart.length; i++) {
                res[n] = nextPart[i];
                n++;
            }
        } while((nextPart != null) && (n < values));
        Profiler.end("QSortMaster::reconstructArray()");
        return res;
    }

    private int[] nextPartition(Vector<int[]> parts) {
        if(parts.isEmpty())
            return null;

        if(parts.size() == 1)
            return parts.elementAt(0);

        int low = 0;
        for(int i = 1; i < parts.size(); i++) {
            if(parts.elementAt(i)[0] < parts.elementAt(low)[0])
                low = i;
        }

        return parts.remove(low);
    }

    public static void main(String[] args) {
        if(args.length != 3) {
            System.out.println("Usage:␣java␣apps.sorting.
               QSortMaster␣<PORT>␣<VALUES>␣<THRESHOLD>");
            System.exit(1);
        }

        int port        = Integer.parseInt(args[0]);
        int values      = Integer.parseInt(args[1]);
        int threshold   = Integer.parseInt(args[2]);

        new QSortMaster(port, values, threshold).start();
    }
}
```

B.2.3 QSortWorker

```
/**
 *  QSortWorker.java
 *
 *  Author:   Alistair Atkinson (alatkins@utas.edu.au)
 */

package apps.sorting;
```

```
import space.*;
import runtime.*;
import java.io.*;
import com.clarkware.profiler.*;

public class QSortWorker {
    private TupleSpaceRuntime ts;
    private int port;

    public QSortWorker(int port) {
        this.port = port;
        ts = new TupleSpaceRuntime(port, false);
    }

    public void start() {
        Profiler.begin("QSortWorker::TotalRuntime");
        ts.start();

        System.out.println("Worker␣process␣started.");

        // get qsort tuple
        System.out.print("Attempting␣to␣fetch␣initial␣data...")
           ;
        //Profiler.begin("QSortWorker::IO");
        Profiler.begin("QSortWorker::Init");
        Tuple t = ts.in(new TupleTemplate("qsort", null, "
           unsorted"));
        Profiler.end("QSortWorker::Init");
        //Profiler.end("QSortWorker::IO");
        System.out.println("Done.");

        QSort qs = (QSort) t.field(1);
        String status = (String) t.field(2);

        while(!status.equals("complete")) {
            // sort OR split
            if(qs.readyToSort()) {
                System.out.println("Sorting␣" +qs.getData().
                   length + "␣items.");
                Profiler.begin("QSortWorker::Sorting");
                Profiler.begin("QSortWorker::Processing");
                //qs.quicksort(qs.getData(), 0, qs.getData().
                   length - 1);
                qs.insertionSort(qs.getData());
        Profiler.end("QSortWorker::Processing");
        Profiler.end("QSortWorker::Sorting");

        try {
            Profiler.begin("QSortWorker::IO");
            ts.gts.out(new Tuple("qsort", "sorted", qs)
               );
            Profiler.end("QSortWorker::IO");
        } catch(IOException e) {
            System.out.println("Error␣returning␣sorted␣
               partition␣to␣master␣process");
        }

        System.out.print("Successfully␣sorted␣partition
           .\nAttempting␣to␣fetch␣more␣data...");

        // try to find unsorted partition to work on
        Profiler.begin("QSortWorker::IO");
        t = ts.in(new TupleTemplate("qsort", null, null
           ));
        Profiler.end("QSortWorker::IO");

        System.out.println("Done.");

        qs = (QSort) t.field(1);
        status = (String) t.field(2);
    } else {
        Profiler.begin("QSortWorker::Partitioning");
        Profiler.begin("QSortWorker::Processing");
        System.out.print("Splitting␣array␣of␣size␣" +qs
           .size());
        QSort dup = qs.split();
        System.out.println("␣into␣" + qs.size() + "/" +
            dup.size());
        Profiler.end("QSortWorker::Processing");
        Profiler.end("QSortWorker::Partitioning");

        System.out.println("Split␣partition..continuing
           ...");

        ts.out(new Tuple("qsort", dup, "unsorted"));
    }
    System.out.print(".");
}
System.out.println();
```

```
        System.out.print("Worker␣process␣finished...Shutting␣
            down...");
        // redistribute poison pill to get all processes to
            terminate
        ts.out(new Tuple("qsort", new QSort(), "complete"));

        ts.stop();

        System.out.println("Done.");

        Profiler.end("QSortWorker::TotalRuntime");
        Profiler.print();
        //System.exit(0);
    }

    public static void main(String[] args) {
        if(args.length != 1) {
            System.out.println("Usage:␣java␣apps.sorting.
                QSortWorker␣<PORT>");
            System.exit(1);
        }

        int port = Integer.parseInt(args[0]);

        new QSortWorker(port).start();
    }
}
```