Remora - Implementing Adaptive Parallelism on a Heterogeneous Cluster of Networked Workstations

THESIS

Submitted in fulfilment of the requirements

for the Degree of

MASTER OF SCIENCE

of Rhodes University

by

Geoffrey Michael Rehmet

September 1994

Abstract

Computers connected to a local area network are often only fully utilized for short periods of time. In fact, most workstations are not used at all for a significant portion of the day. The combined "idle time" of the workstations on a network constitutes a significant computing resource, which is generally wasted. If harnessed properly, such a resource could constitute a cheap alternative to expensive high-performance computers.

Adaptive parallelism refers to the parallel execution of a computation on a dynamically changing set of processors. This thesis investigates the viability of this approach as a vehicle to harness the "idle cycles" available on a heterogeneous cluster of networked computers. A system, called Remora, which implements adaptive parallelism via the Linda programming paradigm, is presented.

Experiments, performed using Remora, show that adaptive parallelism provides an efficient vehicle for using idle processor cycles, without having an adverse effect on the tasks which constitute the normal workload of the computers being used.

"**Remora**, also called SHARKSUCKER, or SUCKERFISH, any of 8 - 10 species of marine fishes of the family Echenidae (order Perciformes) noted for attaching themselves to, and riding about on, sharks, other marine animals, and oceangoing ships. Remoras adhere by means of a flat, oval sucking disk on top of the head. The disk, apparently derived from the spiny portion of the dorsal fin, contains a variable number of paired, crosswise plates.

"Remoras are thin, elongated, rather dark fishes, from 30 to 90 centimetres (1 to 3 feet) long; they live in warmer waters and are found around the world. Remoras feed on the leavings of their hosts' meals or, in some instances, act as cleaners by eating the external parasites of their transporters."†

† The New Encyclopædia Britannica, Micropædia (Ready Reference), 15th Edition, 1986, Vol. 9, p. 1019

Acknowledgements

Throughout this project, Peter Clayton, my supervisor, has provided me with a great deal of advice and support. His enthusiasm for my work, in an area which he calls one of his "pet projects", has been a constant source of encouragement during the development of Remora. He deserves special thanks for reading the draft copies of this thesis.

My fellow postgraduate students, as well as the other staff of the Computer Science Department have provided a lot of input which has helped to shape the progress of my work. They provided a testing ground for a lot of my ideas, and helped to expose flaws in some of my arguments.

I would like to thank my friends and family, who have provided me with encouragement, especially at times when the obstacles in my path seemed insurmountable. My thanks also go to Arne Bier, who read the final draft of this thesis, and made a number of suggestions for its improvement.

I acknowledge the financial support of Gold Fields Mining and Development ltd.

Trademark Information:

Helios is a Trademark of Perihelion Software Limited.
Linda is a trademark of Scientific Computing Associates.
PostScript is a trademark of Adobe Systems Inc.
SPARC is a trademark of SPARC International.
SunOS and Solaris are trademarks of Sun Microsystems Inc.
Transputer is a trademark of Inmos.
UNIX is a trademark of X/OPEN.
Microsoft Windows and Windows-NT are trademarks of Microsoft.

Contents

Figures and Listings **ix**

Chapter 1. Introduction **1**

1.1. Parallel Computing on Networked Clusters of Computers 1
1.2. Adaptive Parallelism 2
1.3. Introducing Linda 3
1.3.1. Tuples 3
1.3.2. The Eval Mechanism 4
1.3.3. Hardware Independence 4
1.3.4. Anatomy of a Linda Program 5
1.4. Goals of this Project 5

Chapter 2. Distributed Computing **7**

2.1. Distributed Operating Systems 8
2.2. Remote Job Execution 9
2.3. Distributed Programming Environments 10

Chapter 3. Using Idle Cycles on Workstations **12**

3.1. Finding Idle Workstations 13
3.2. Migrating Work off Active Nodes 13
3.3. Piranha 14

Chapter 4. Adaptive Parallelism with Linda **16**

4.1. Remora System Structure 16
4.1.1. A Typical Configuration 17
4.1.2. Processes and Tasks 17
4.1.3. Distributing Work 18
4.2. Decoupling 18
4.2.1. Spatial Decoupling 18
4.2.2. Temporal Decoupling 19
4.3. Backing off 19
4.4. Tuple Logging 21
4.5. Potential Problems 22
4.5.1. Interprocess Dependencies 22
4.5.2. Referential Transparency 22

Chapter 5. Heterogeneous Computing Environments **24**

5.1. Approaches to Heterogeneity 24
5.2. Manifestations of Heterogeneity 25

5.2.1. Hardware 25
5.2.2. Operating Systems 26
5.2.3. Communication Media 26
5.2.4. Programming Languages 27
5.3. Advantages of Linda 27
5.4. Design Considerations 28
5.4.1. Data Representation 28
5.4.2. System Layering 29
5.5. Implications for Adaptive Parallelism 29
Chapter 6. System Coordination **31**
6.1. Network Transport 31
6.1.1. TCP Streams 31
6.1.2. Berkeley Sockets 32
6.2. Coordination Overview 33
6.2.1. Coordination Functions 34
6.3. Message Protocols 34
6.4. Running User Programs 35
6.5. The Remora Daemon 36
6.5.1. Behaviour 36
6.5.2. Load Monitoring 36
6.5.3. Backing Off 38
6.5.4. Starting Remora Processes 39
6.5.5. Configurable Options 39
6.6. The Coordinator 40
6.6.1. Information about Nodes 40
6.6.2. Tuple Group Placement 40
6.7. Configuration 41
6.8. System Monitoring 41
Chapter 7. Implementing Adaptive Linda Clients **43**
7.1. Client Program Structure 43
7.2. Client Communication 44
7.2.1. Message Types 44
7.2.2. Message Format 45
7.2.3. Header Fields 46
7.2.4. Data Types 47
7.3. Behaviour 47
7.4. Accessing Tuple Space 49
7.4.1. The Eval Group 49
7.4.2. Contacting a Tuple Space Server 50
7.4.3. Tuple Space Operations 50

7.5. Controlling Client Processes ... 51
7.6. Tasks ... 52
7.7. Replaying Tuple Space Transactions ... 53
7.8. Direct Manipulation of Tuple Logs ... 53

Chapter 8. The Distributed Tuple Space Server 55
8.1. Tuple Space Distribution ... 55
8.2. Server Structure ... 56
8.3. Tuple Storage ... 58
8.3.1. Tuple Groups ... 59
8.3.2. Storage Policies ... 59
8.3.3. Tuple Logs ... 61
8.4. Supporting Adaptive Parallelism ... 62
8.4.1. Creation and Deletion of Tuple Logs ... 62
8.4.2. Appending Tuples to Logs ... 63
8.4.3. Replaying Tuple Messages ... 63
8.4.4. Direct Manipulation of Logs ... 64

Chapter 9. Example Programs and Benchmarks 65
9.1. Tuple Space Interaction ... 65
9.1.1. The InOut Benchmark ... 66
9.1.2. The PingPong Benchmark ... 66
9.1.3. Optimizations ... 66
9.2. Example Application Programs ... 68
9.2.1. Queens ... 68
9.2.2. Finding Prime Numbers ... 72
9.2.3. Optimal Matrix Multiplication Order ... 72
9.3. Idle Time available on a Networked Cluster ... 73

Chapter 10. Future Work 74
10.1. Porting Remora to Other Architectures ... 74
10.2. A Better Preprocessor for Application Programs ... 75
10.3. Network Transport ... 75
10.4. Workload and Tuple Space Distribution ... 76
10.4.1. A More Balanced Distribution of Tuple Space Partitions ... 76
10.4.2. Functional Clustering ... 76
10.4.3. Placement of Tasks on Suitable Computers ... 76
10.5. Tuple Space Storage ... 77
10.6. Fault Tolerance ... 77
10.7. Higher Level Programming Tools ... 77
10.7.1. Algorithmic Skeletons ... 78
10.7.2. The Linda Program Builder ... 78
10.8. Behavioural Model Debugging ... 78

Chapter 11. Conclusion **79**

Appendix A: Remora System Configuration **81**

Appendix B: The Queens Problem **83**

Appendix C: The Sieve of Eratosthenes **87**

Appendix D: Optimal Matrix Multiplication Order **91**

References **95**

Figures and Listings

Figure 1: Transfer rates for TCP and UDP (in kbytes per second) 32
Figure 2: Remora coordination framework 33
Figure 3: Messages transmitted between coordinator and daemon processes 35
Figure 4: Signals used by the daemon to control a client process 39
Figure 5: Tuple message identifiers 45
Figure 6: Tuple Message Format 46
Figure 7: Structure of a tuple space server process 57
Figure 8: Structure of tuple space storage 60
Figure 9: Methods available for storage policies 61
Figure 10: Timings (seconds) of the *InOut* and *PingPong* benchmarks 67
Figure 11: *PingPong* and *InOut* timings (seconds) on different architectures 68
Figure 12: Speedup obtained placing 14 queens 70
Figure 13: Performance of the *Queens* program in an adaptive environment 71
Figure 14: Times (seconds) taken to calculate Optimal Matrix Multiplication Order ... 72
Figure 15: Idle time, measured in a week, on a cluster of workstations 73

Listing 1: Breaking referential transparency 23
Listing 2: Client process main loop 48
Listing 3: *PingPong* algorithm 66
Listing 4: Reliable node configuration 81
Listing 5: Unreliable node configuration 82
Listing 6: The Queens Problem 83
Listing 7: The Sieve of Eratosthenes 87
Listing 8: Optimal Matrix Multiplication Order 91

Chapter 1.

Introduction

High performance computing resources have traditionally only been available to those organizations whose budgets allow the purchase of supercomputers and specialized parallel architectures. Clusters of computers on local area networks (LANs), which now form part of most computing environments, do however provide attractive alternatives to the purchase of expensive parallel computing hardware.

1.1. Parallel Computing on Networked Clusters of Computers

LANs are now common in most computing environments, and the collective computing power of the computers on a LAN is often of a similar magnitude to that of a specialized parallel computer. While communication between computers on a LAN is generally much slower than between processors in a specialized parallel computer, LANs can still be used as effective environments for parallel computing. This is especially so since it is often possible to "get something for nothing" on a LAN, as workstations on a LAN tend to be unused for a very large proportion of the time.

The "idle time" available on a LAN, which is often wasted, can be used effectively as a resource for performing parallel computations. This thesis presents a system, named Remora[1], which provides an environment in which a collection of general purpose computers on a LAN can be coordinated in order to perform parallel computations.

In order to make use of the idle time on a LAN it is necessary to monitor the computers on the LAN, so as to determine when they are idle. The criteria used to determine when workstations are idle must be flexible, so as to accommodate the requirements of the owners of the workstations whose idle time is being exploited. Failure to accommodate these requirements will result in the owners of workstations refusing to allow the idle

[1] The name *Remora* is derived from the fact that the system is "attached" to workstations and uses up the idle time, which is left over on these systems, much as the remora fish feeds on the scraps left over from its host's meals.

time on their computers to be used in distributed computations. Remora caters for this fact by providing a number of different criteria which can be used in order to determine when a workstation is idle. These criteria can be configured to cater for the needs of the owners of individual workstations.

Most networked clusters of computers are heterogeneous collections of machines, of different architectures, and also running different operating systems. A system which is able to accommodate this heterogeneity will be able to harness a far greater pool of processing power than one which requires a homogeneous pool of processors. It is with this in mind that Remora provides facilities for architecturally different computers, which are possibly running different operating systems, to participate together in the same computation.

1.2. Adaptive Parallelism

In an environment where idle time on workstations is being used, a varying number of workstations will be available to participate in a parallel computation. As workstations become idle they join the host pool, and leave it again when they are reclaimed by their normal users.

Performing parallel computations on a pool of processors, which varies in size, is referred to as "*Adaptive Parallelism*" [Carriero *et al* 1993]. Processors may join or leave the pool of available nodes as they wish during a computation. This allows for workstations to be used as participants in a parallel computation *only* when they are not being used for other purposes.

Remora provides facilities which implement adaptive parallelism on a cluster of workstations, all running variants of the UNIX operating system. The implementation makes use of the *Linda* coordination language[2] in order to provide communication between, and coordination of, the workstations participating in a computation. An important reason for the choice of Linda as an implementation vehicle is the fact that it provides facilities for anonymous communication between processes. The environment provided by Linda is one in which there are no direct dependencies between processes. This makes it possible for a process to be stopped, moved to another processor, and then restarted, without any other processes being affected.

[2] A coordination language provides facilities to create computational entities and support communication between them. A coordination language is combined with a computation language to provide a complete programming environment. [Gelernter and Carriero 1992]

Details of the actual varying pool of hosts are hidden from the view of the programmer via the abstractions provided by the Linda tuple space. The abstractions provided by Linda also provide the means to achieve the goal of implementing an environment in which portable parallel programs can be written, and then executed, in a heterogeneous environment.

1.3. Introducing Linda

The Linda coordination language [Gelernter 1988, Carriero and Gelernter 1989] provides primitives which allow anonymous, and temporally and spatially decoupled communication between concurrent processes, via a shared, associative virtual memory which is known as *tuple space*. Tuple space is an unordered bag of data items known as tuples, which may be inserted, removed and read by client processes, using the primitives provided by Linda.

In order to construct a complete parallel programming environment, Linda is combined with a computation language. For instance, *C-Linda* is the result of adding Linda primitives to the C programming language. Other languages which have been combined with Linda include C++, Modula-2, Scheme, Fortran and PostScript [Carriero and Gelernter 1989].

1.3.1. Tuples

A Linda tuple consists of a set of typed fields, the first of which is generally regarded as a label or logical name. Tuples are referenced associatively, and data structures can be constructed via tuples stored in tuple space. Examples of tuples are *("A", 25, TRUE)* and *("counter", k)*.

Tuples are placed into tuple space by using the *out* operation. If any of the fields listed in the *out* operation are variables, their values will be placed into the tuple which is deposited into tuple space. For instance, if the variable *i* has the value *10*, then the operation *out("A", 25, TRUE, i)* will deposit the tuple *("A", 25, TRUE, 10)* into tuple space. A process which performs an *out* operation does not block on depositing a tuple into tuple space, but proceeds to execute the instructions following the *out* immediately.

Once they have been deposited into tuple space, tuples can be retrieved by making use of the *in* and *rd* operations. The *template* given in an *in* or *rd* statement is matched against the tuples present in tuple space, and if possible a matching tuple is retrieved. For example, the operation *in("X", 5)* will remove the tuple *("X", 5)* from tuple space. If

there is no tuple matching the template given in the *in* statement, then the process executing the operation will block until a suitable tuple is deposited into tuple space by another process. The *rd* operation behaves similarly to *in*, except it does not remove a tuple from tuple space. Predicate versions of *in* and *rd*, namely *inp* and *rdp* are also available. These operations do not block if a matching tuple cannot be found, but return *false* instead.

The *in* and *rd* operations may also make use of *formals* (which are prefixed by a question mark) in their tuple templates. For instance, if *c* is an integer variable, *in("Counter", ?c)* will retrieve a tuple which has the string "*Counter*" in its first field, and any integer value in its second field, and then place that value in the variable *c*. When a tuple template is matched against a tuple in tuple space, any values must match exactly by both type and value, while formals must only match by type.

1.3.2. The *Eval* Mechanism

In order to create new computational entities (generally processes), Linda provides the *eval* operation, which is a modified version of *out*. While *out* is used to place passive data items into tuple space, *eval* deposits active or "live" tuples into tuple space. By executing the operation *eval("func1", 25, func1())* an active tuple, in which the field *func1()* is unevaluated, is placed into tuple space. In order to evaluate the field, a process executing *func1()* is started on a suitable processor. When this process terminates, the value resulting from its computation replaces the field *func1()* and the tuple becomes a passive tuple, containing only data.

The *eval* mechanism provided by Remora differs slightly from the normal *eval* mechanism used by Linda. In Remora, an active tuple only has one field, which names the piece of code to be executed. On termination of that piece of code, the active tuple ceases to exist, and is not transformed into a passive tuple as in Linda.[3]

1.3.3. Hardware Independence

A feature of Linda, which is important to this work, is the fact that the programming paradigm adopted by Linda is independent of any particular parallel architecture. Linda presents an abstract programming environment of its own. While it might be argued that this results in a loss of efficiency, the benefit of being able to develop portable parallel programs is probably greater.

[3] The *eval* mechanism used by Remora is discussed in section 4.1.2. (See page 17.)

1.3.4. Anatomy of a Linda Program

When a Linda program is compiled, the steps which are followed differ slightly from those used for the compilation of a program in a normal computation language such as C or Modula-2.

Linda programs, which are written using *ideal Linda syntax* (i.e. programs containing Linda operations embedded in a computation language), must first be translated into *concrete syntax* by a preprocessor. This replaces Linda operations with calls into a run-time library. The resulting concrete syntax can be passed to the normal compiler used to compile programs written in the computation language in which the Linda operations were embedded. Let us consider, for example, the compilation of a C-Linda program, written using ideal Linda syntax. First, the program is preprocessed, and Linda operations are replaced with calls into a run-time library, which implements the Linda operations. The resultant concrete syntax is passed to the C compiler. Finally, the object code produced by the C compiler is linked against the Linda run-time library to produce an executable program.

Remora makes use of a modified version of the Linda preprocessor, written by George Wells at Rhodes University, as well as a tuple space analysis program, written by Fred de-Heer-Menlah [de-Heer-Menlah 1991] in order to perform the preprocessing phase of the compilation of application programs.

1.4. Goals of this Project

Parallel processing has traditionally been the domain of those whose funds allow the purchase of expensive, dedicated hardware. This project seeks to investigate the viability of implementing a parallel programming environment, using networked clusters of general purpose computers, in such a way that the normal usage of the hardware is unaffected by parallel computations which are run as background jobs.

In order to maximize the amount of processing power available to us in a distributed computing environment, we make use of heterogeneous computing resources. This is done by providing mechanisms whereby the differences between computers in a heterogeneous network are abstracted away, and hidden from the view of the programmer.

In order to realize these goals, the facilities for temporal and spatial decoupling, and hardware independence, which are available in Linda are exploited, so as to produce a

complete distributed programming environment, which provides support for adaptive parallelism in a heterogeneous, distributed computing environment. The resultant implementation is a modular system, whose components are linked via a set of well-defined interfaces. The rationale for this design is to allow Remora to serve as a basis for further work on both the Linda paradigm and on adaptive parallelism.

Chapter 2.

Distributed Computing

Until the early 1980's, centralized mainframe computers were used for most data processing. Since then the development of fast microprocessors and the advent of personal computers have brought about a movement of processing power onto the desktop. The processing power of the microprocessors used in personal computers has more than doubled every two years, giving rise to generations of processors whose performance on scalar code exceeds that of the Cray YMP supercomputer [Brooks *et al* 1991].

Aided by the benefits of mass production, coupled with high sales volumes, the prices of personal computers and scientific workstations continue to drop as their performance increases. This means that most organizations are able to acquire a considerable amount of computing power at a very modest price. Low prices also make it possible to upgrade computing equipment at regular intervals, a financial impossibility when considering mainframes and supercomputers. Fierce competition between vendors has had a great effect on the rate of development of the PC and workstation markets, placing downward pressure on prices and speeding up the rate of development of more powerful systems. It thus seems highly desirable to exploit the vast amount of cheap processing power available on micros and workstations, as an alternative to the purchase of highly specialized and expensive supercomputers.

As CPUs have become more powerful, we have also seen the development of local area networks (LANs). Most working environments now possess a collection of powerful personal computers and scientific workstations, which are able to communicate with each other via the LANs to which they are connected. Currently, LANs are mainly used for sharing files, providing remote login facilities, email, file transfers and providing shared access to resources such as printers. This tends to ignore much of the potential utility of a LAN, which can be used as a coordination and communication channel for distributing work amongst the computers on a network.

Apart from their more conventional uses, LANs provide the functionality to harness the collective power of the computers connected to them, so that they can be coordinated in order to perform tasks together. This allows the construction of a distributed computing engine which can be used to perform some of the tasks traditionally handled by supercomputers.

A significant amount of research has been conducted into different methods of exploiting the processing power of networked clusters of computers. (A number of these are listed in [Turcotte 1993].) Some of these systems make use of dedicated clusters of workstations, which are used *only* for the purpose of executing parallel computations, while most are designed to make use of existing LAN-based infrastructures. Success has been attained in solving some "Grand Challenge"[4] problems by using networked clusters of computers [Dongarra *et al* 1993]. It has also been shown that networked clusters of computers can exceed the performance of supercomputers for certain classes of application [Whiteside and Leichter 1988].

2.1. Distributed Operating Systems

Distributed operating systems function in a seamless manner across a group of interconnected computers. The user of such a system is presented with a view of a single "computer", and the tasks run by the user are managed by the operating system in such a way that they are placed on suitable processors without the user needing to do this explicitly. A large number of distributed operating systems have been developed. We examine just three examples of such systems.

The *Amoeba* distributed operating system [Tanenbaum 1992a, Tanenbaum 1992b] is based on a microkernel architecture, with most system facilities being provided by user-level server processes. Interaction between different computers is performed by making *remote procedure calls* (RPC). Under this model a client thread will send a message to a server, and block until a reply is returned. A user who is logged into an Amoeba system will see what looks like a conventional time-sharing computer. Resources such as disk drives are served in such a way that the user is not aware of the physical location of these facilities. Facilities for the writing of parallel applications are also available in Amoeba.

---

4 Grand Challenges are "fundamental problems in science and engineering with broad economic and scientific impact, whose solutions require the application of high-performance computing". [NSF 1993]

The *Hybrid* version 2 system [Muller 1994] provides microkernel based support for distributed programming. This system has been developed with the specific purpose in mind of running parallel programs on a multicomputer consisting of workstations connected to a LAN. Distributed programs implemented under Hybrid make use of a *supervisor-worker* model of parallel programming. Worker processes actively request tasks from a supervisor process. The results generated by executing these tasks are returned to the supervisor.

The model of parallel programming used by the *Guide* distributed operating system [Balter *et al* 1991] is based on an object-oriented design. Guide applications are structured into objects, whose methods may be invoked either locally or remotely. *Jobs*, which form the basic unit of execution, may "diffuse" from one node to another by invoking the methods of objects which are located on remote nodes. Complex applications may consist of a number of cooperating jobs, which run in parallel. In contrast to Hybrid and Amoeba, Guide is implemented on top of a host operating system, generally a flavour of UNIX. This avoids the additional development work of writing low level kernel services which must interact directly with the hardware of the computers on which the system is running. The use of a host operating system also makes it possible for the computers which are involved in running the distributed operating system to be used for other purposes as well, rather than just being dedicated to running the distributed system. Guide places a great degree of emphasis on high-level features and the reusability of code.

2.2. Remote Job Execution

Distributed operating systems traditionally don't allow for computers which are already being used for other purposes to be used for distributed processing. Remote job execution systems, such as *Condor* [Bricker *et al* 1992], allow for users to spawn CPU intensive jobs to other workstations which are currently not being used. This allows users, who need to run a lot of computationally intensive tasks, to have access to more processing power than can be harnessed on their own workstations alone. The decision to allow for jobs to be run remotely on workstations which are not actively being used is based on the observation that a lot of users either only make use of their workstations for text editing purposes, or leave their workstations idle for large portions of the day. Computers which are idle are placed into a "*processor bank*". Jobs submitted to Condor are then placed on the computers in this pool. While *Condor* does permit some degree of distribution of work across a LAN, it does not provide facilities for running parallel programs.

2.3. Distributed Programming Environments

A number of parallel programming environments for LANs are also available. These are systems which do not require a distributed operating system for their execution environment, but which make use of the networking facilities of an underlying operating system in order to implement a parallel programming environment. Most such systems either provide a library of functions for communication and coordination between processes running on different nodes, or make use of parallel programming extensions to existing programming languages.

Amber [Chase *et al* 1989] provides facilities for parallel programming on a network of DEC Firefly multiprocessors. This system is based on a model of computation in which a collection of mobile objects interact via location-independent invocation. Passive objects contain private data, and public methods (or operations), which may be invoked either locally or remotely. Active entities are *thread objects* which possess processor state and a run-time stack. A typical Amber application consists of a number of parallel threads, concurrently executing object operations. Amber programs are written in a subset of the C++ language, with extensions for thread control and object mobility between nodes. The use of objects hides details of internal state, execution and synchronization. Object references can be transmitted across node boundaries, allowing the dereferencing of objects on any node.

Another object-based parallel system which makes use of extensions to C++ is *Charm++* [Kale and Krishnan 1993]. This system places a great deal of emphasis on portability and has been implemented on both shared memory and message passing multicomputers, as well as LAN-based clusters of workstations. This emphasis on the development of portable parallel systems is significant, given the current architectural diversity of parallel computing systems. Charm++ makes use of a concurrent object model of computation, where messages are passed between objects. This system places a high degree of emphasis on providing tools which make the development of parallel programs easier, mainly through the use of object orientation.

PVM (Parallel Virtual Machine) [Dongarra *et al* 1993, Sunderam *et al* 1993] is a portable, architecture independent implementation of a message passing system for networked clusters of computers. While Amber and Charm++ are programmed using extended versions of languages (C++ in both cases), PVM facilities are accessed via a run-time library of functions, which can be included in either C or FORTRAN programs. The run-time library provides a small kernel of functions which are necessary for message passing, rather than trying to provide an extensive collection of features. Nodes

which participate in a PVM computation are grouped into a dynamically configurable *host pool*, which may have nodes added or removed at any time. A daemon process, which runs on each of the nodes in the host pool, provides the functionality necessary for all of the nodes to cooperate in order to form a virtual message passing architecture.

The *Linda* coordination language [Gelernter 1988, Ahuja *et al* 1986], upon which the work described in this thesis is based, was conceptualized in such a way as to be independent of any parallel architecture, allowing for it to be ported to almost any parallel architecture. The run-time libraries which implement the Linda primitives are hidden from the view of the programmer by the syntax of the Linda operations, which are embedded in a host language. Parallel languages, such as Linda, also benefit from performance improvements achieved by compile-time optimizations. This allows for support of far more sophisticated and high-level programming constructs than those available under distributed operating systems [Gelernter 1988]. Unlike PVM, where messages are passed between named processes, communication under Linda is anonymous. This simplifies the execution of programs in an environment where the number of available processors varies, and where it is not always possible to determine on which processor a given component of a computation is being executed. Apart from Linda implementations on shared memory and message passing architectures [Bjornson *et al* 1988], a number of LAN-based implementations of the *Linda* paradigm exist [Whiteside and Leichter 1988, Wilson 1991, Smith 1993].

Chapter 3.

Using Idle Cycles on Workstations

Workstations connected to a LAN are typically dedicated to serving the needs of one user. Most users do however not fully utilize the processing resources of their workstations. In actual fact, a large amount of the CPU time available on a workstation goes totally unused. This is both due to the fact that most workstations are only used during office hours, and because of the nature of the tasks for which most users use their workstations. Some measurements have shown that idle time percentages of up to 90% are common [Cap and Strumpen 1993]. Thus it is clear that the amount of wasted CPU time is enormous. Programming environments which are capable of making use of idle workstations include Condor [Bricker *et al* 1992], LiPS [Roth and Setz 1992], and Piranha [Kaminksy and Carriero 1991, Carriero *et al* 1993].

The typical work done by most workstation owners consists mainly of activities such as text editing, desktop publishing, CAD design, or program development [Theimer and Lantz 1989]. This kind of activity generally requires fast response times, but does not place a continuous load on the workstation's CPU. There may also be pauses for significant periods of time, during which the workstation is not used at all. Since most modern workstation operating systems support multiprocessing, it is possible for other CPU-intensive tasks to be run concurrently with a workstation's normal workload, when the workstation's load is sufficiently low. If an interactive user starts to make greater demands on his workstation while a process of a distributed application is being run in the background it should be possible to stop or remove the background job so that the interactive user has all of the processing power of the workstation available to him.

The good will of workstation owners is an important factor to be considered when attempting to make use of "idle cycles" on workstations. Workstation owners need to be convinced that they will not be inconvenienced by a system which is running portions of computationally intensive applications on their personal systems. This means that any such system must ensure that a workstation's response time while performing interactive work remains low, and that, if an interactive user should start a computationally intensive task, his task should be given priority. Ideally, the owner of a workstation should not be

able to perceive any performance degradation due to the additional workload generated by the presence of part of such a distributed computing system on his computer.

3.1. Finding Idle Workstations

In order to make use of idle workstations, it is first necessary to determine a policy which defines when a workstation is idle and when it is busy. Such a policy should allow for as much flexibility as possible, so that workstation owners can have a say in how the idle cycles on their computers may be used. Some of the different metrics which have been used to determine whether or not a workstation is idle include:

- allowing use only at specified times of the day [Roth and Setz 1992],
- when less than a specified number of users are logged in, or when logged-in users are idle on input [*ibid*],
- when there is no keyboard or mouse activity on the workstation console [Kaminksy and Carriero 1991, Bricker *et al* 1992],
- when the average system load falls below a specified level [Bjornson *et al* 1991], or
- by making use of a statistical model of typical user behaviour [Theimer and Lantz 1989].

Once a decision on metrics for determining when a workstation is idle has been made, it is necessary to implement a mechanism which allows the monitoring of potentially available workstations, and then making the availability of idle workstations known, so that portions of distributed computations can be sent to them for processing. Normally some sort of daemon process, running on a candidate workstation, will monitor the computer's load, taking samples at regular intervals. It is important that this monitoring should only consume a negligible amount of CPU time. When the monitoring daemon detects that the workstation has become idle it normally contacts a central master node, which will delegate work to it.

3.2. Migrating Work off Active Nodes

It will often happen that the demands placed on a workstation by interactive users increase to the degree that continued participation in a distributed computation would cause a degradation of the interactive performance of the workstation. If this happens, it is necessary to remove the job currently being processed from the workstation, and possibly try to relocate it elsewhere. Some of the strategies used in such a situation include:

- suspending the job which is being processed until the workstation is idle again,
- saving the job's state, and restarting the job on another workstation, or
- killing the job, and restarting it from scratch elsewhere.

The approach of suspending jobs, and waking them up later, is taken by LiPS [Roth and Setz 1992]. This approach has the disadvantage of possibly deadlocking a computation due to the fact that one process is not runnable, since the workstation it was running on is being used by its owner.

Condor [Bricker *et al* 1992] takes the second approach, which requires that the entire state of the job be transported across the LAN to another workstation, where it is restarted. This can be a costly operation, since it may be necessary to transfer a large amount of information. When a Condor process has to be moved to another CPU, a signal is sent to it, so that it can checkpoint itself and then dump core. The core image is then transferred across the network to another workstation, where it can be restarted. This approach has the additional disadvantage that it does not work in heterogeneous environments, as a core image generated under one architecture cannot be restarted on another.

The third approach, of restarting tasks from scratch, is taken by Piranha [Kaminksy and Carriero 1991, Carriero *et al* 1993]. This has the unfortunate cost that some work will be redone. This cost can be reduced by making the tasks relatively short-lived, keeping in mind that communication overheads increase as the granularity of a computation is made finer.

An additional problem may also occur when a task is removed from a node: sometimes it is not possible to immediately restart the task on another node. This can lead to a deadlock situation in an environment where a lot of communication takes place between the processes running on different workstations. This problem can be dealt with either by allowing additional processes to be started on available workstations (which in turn introduces other problems, such as a computation degenerating into a large number of processes running on one node), or making use of a decoupled method of communication, such as that provided by Linda.

3.3. Piranha

The *Piranha* programming model [Kaminksy and Carriero 1991, Carriero *et al* 1993] is based on the master-worker model of parallelism, and makes use of the Linda programming paradigm. Piranha programs make use of a single, general-purpose worker function, which is executed on all available processors on a LAN. This worker function,

which is called a *piranha()*, consumes tasks which are deposited in tuple space by the master. On completing a task, its results are deposited back into tuple space, where they can be retrieved at a later stage by the master. Each *piranha()* will continue to consume tasks until either the end of the program is reached, or the workstation it is running on is claimed by its owner.

One of the important advantages of Piranha programs is that they do not actively create processes, or rely on there being a specific named set of active processes. This is because the model of parallelism used is based on *tasks* (units of computational work) rather than *processes* ("threads" of execution). Processes are created and destroyed as workstations become available to consume tasks, and are destroyed as these workstations are reclaimed by their owners. (A user program will contain no operations which create processes.)

If a workstation becomes unavailable for use by the Piranha system, the task, which is currently being executed by the *piranha()* which is running on the node, will be replaced into tuple space, and will be started again from scratch elsewhere. Thus there is no need to migrate processes between CPUs. This avoids the complexity of saving the state of a process so that it can be reinstated elsewhere. There is the unfortunate effect that some amount of work has to be redone when a task, which has not been completed, is terminated, due to the node on which it is running becoming unavailable. This cost can however be reduced if the granularity of the tasks being performed is adjusted.

A Piranha program consists of three main user-supplied components, namely the functions *feeder()*, *piranha()* and *retreat()*. The *feeder()* function is executed on the *home node*, which must always be available. This function generates the jobs which are consumed by the *piranha()*. When a workstation on which a *piranha()* is running becomes unavailable, the *retreat()* function, whose responsibility it is to place the currently executing task back into tuple space, is called. The *feeder()* also has the responsibility of collecting the results of completed tasks, and may additionally also take part in the consumption of tasks.

Remora overcomes the need for additional programmer intervention, which is required by Piranha, by supporting adaptive parallelism in the underlying implementation of the Linda tuple space, rather than at a level which is immediately visible to the application programmer. This means that steps, such as those performed by the programmer-supplied *retreat()* function in Piranha, are handled by Remora without any assistance being required from the programmer.

Chapter 4.

Adaptive Parallelism with Linda

The Linda coordination language is particularly amenable to use as a basis for implementing adaptive parallelism. In this chapter we examine the features of Linda which are important in this respect, and focus on the mechanisms used in the implementation of Remora. The techniques which are used are to a degree similar to those used in Piranha, while some concepts are derived from fault-tolerant implementations of Linda. Parts of Remora are based on the transputer [Wentworth 1990] and network [Smith 1993] implementations of Rhoda[5].

4.1. Remora System Structure

Remora implements the Linda tuple space and its associated operations on a heterogeneous cluster of networked computers. The computers which are used are classed into two groups, namely *reliable* and *unreliable* nodes. Reliable nodes are required to participate in a Remora computation for its entire duration, while unreliable nodes may withdraw from a computation if they become unavailable (due to being claimed by their owners) for use by the Remora system.

The Linda tuple space is housed on reliable nodes, which are permitted, in addition to running Remora client processes, to host tuple space servers. This restriction on the location of tuple space server processes avoids the complexity of moving the data stored in tuple space to another node when a node becomes unavailable for computation. One of the reliable nodes must be configured as the *master node*. This node is responsible for the overall control of the computation.

Unreliable nodes are only permitted to run Remora client processes (which interact with tuple space, using the Linda operations). When the load of an unreliable node rises above predefined levels, or the owner of that node claims it by logging into the console,

5 Rhoda is the name given to the Linda implementations developed at Rhodes University.

then the client processes which are running on the node must be removed. This is done by a procedure called *back-off*. (An unreliable node may rejoin a computation at a later stage.)

4.1.1. A Typical Configuration

A typical Remora configuration consists of a small number of reliable nodes which host tuple space. One of the these acts as the master node. The minimum number of reliable nodes is one, in which case tuple space will be centralized on that node, which also acts as the master node.

The most common kind of configuration which has been used during the work on this project has been:

- Master node: Sun SparcServer
- Reliable nodes: Sun SparcStations
- Unreliable nodes: other Suns, and a collection of i486 based PC's, running FreeBSD, some of which may be rebooted into DOS at times.

Depending on the programs being run, not all of the reliable nodes are actually used to host the tuple space. Generally all of the nodes are used to run client processes.

4.1.2. Processes and Tasks

Within the context of Remora, we make an important distinction between "processes" and "tasks". A process is regarded as a "stream" or "thread" of execution or control, which has associated with it a processor state and a run-time stack etc. When a node becomes available to participate in a distributed computation a *client process* is started on that node. This means that new processes are only created as new nodes join a computation, while processes are destroyed when nodes become unavailable, and are required to back off from a computation.

A task, on the other hand, is the unit of computation, which is launched by placing an active tuple into tuple space by making use of the *eval* Linda primitive. (This differs from many other Linda implementations, where *eval* is used to create new processes.) A client process will consume active tuples, thus executing tasks. Each active tuple has associated with it a program component which constitutes the task which is executed when the active tuple is processed by a client process. (This program component is specified by the argument passed to *eval*.)

4.1.3. Distributing Work

As with Piranha, there is no explicit creation of processes within a Remora user program. The Linda *eval* operation is used to distribute work to other nodes by placing active tuples into tuple space, and thus launching potentially concurrent program components. A Remora client process, when started on an available node, will begin to consume active tuples from tuple space, and execute the code associated with them. It will continue doing so until either the computation is completed, or it has to back off due to the workstation becoming unavailable for participation in a distributed computation. Thus an active tuple itself does not constitute a process, but rather a unit of computation (or program component) to be executed by a client process.

Provided enough active tuples are placed into tuple space, there will always be as many active tuples being processed as there are client processes running on available computers. When more computers become available to run client processes, active tuples which are still in tuple space will be consumed, while active tuples may be replaced into tuple space if the computers on which they are being processed become unavailable for use by the Remora system. This simplifies the implementation of a system in which the number of available processors varies with time and cannot be determined in advance. There is no predetermined number of processes which must be executing concurrently, instead the system will attempt to consume, concurrently, as many active tuples as is possible.

4.2. Decoupling

Direct communication between two components of a Remora program is not always possible, since it cannot be determined beforehand whether these components will actually execute concurrently. This problem is elegantly solved by the decoupling action of the Linda tuple space. Any communication between different program components is routed through tuple space by the Linda input and output operations, rather than being performed directly between the program components concerned [Gelernter 1988].

4.2.1. Spatial Decoupling

Program components may communicate with each other, without knowing the location of the other party to the communication (i.e. the node on which the other program component is running). This *spatial decoupling* makes it possible for program components to be moved to different nodes, without it being necessary to make any other

program components aware of such a move. It is only necessary for each client process to be able to communicate with tuple space. Program components thus communicate with each other anonymously, since there is no need to open any directed communication channels, as is generally the case in message passing systems.

4.2.2. Temporal Decoupling

The *temporal decoupling* characteristics of Linda make it possible for two program components which are not executing concurrently to communicate with each other. Again this is made possible since program components communicate via tuple space, and not directly with each other. This feature is important, since not all active tuples can always be executed concurrently, as there may not be enough Remora client processes available to consume all of the active tuples in tuple space at the same time. In this way one program component does not even need to know whether a program component with which it wishes to communicate is executing or not.

The delays associated with accessing tuple space are assumed to be non-deterministic. Thus a Linda client cannot make any assumptions about the duration of the delay which is incurred when tuple space is accessed. This allows tuple space the freedom to indulge in any number of time-variant activities without violating the Linda paradigm.[6] In the context of Remora this is particularly useful in that there is no requirement that any given set of processes actually execute concurrently. Also, this allows for delays which may be introduced when a thread of execution is temporarily suspended, and then restarted at a later stage.

4.3. Backing off

One of the most complex parts of an adaptive parallel system is the mechanism used for *backing off*, that is, removing a process from a node which is no longer available for computation. This procedure entails making a record of the state of the process which is to be moved, and then transferring this state information elsewhere, so that the work that was being done can be continued or restarted on another available node. Back-off should be a relatively quick procedure, and should not create a heavy system load. (It should be possible for back-off to be completed during a normal operating system shutdown.)

6 Formal specifications of Linda, and the restrictions placed on timing by these specifications are described in detail in [Sewry 1994].

The state information of an executing process is generally relatively complex, and includes low-level information, such as CPU state (values of registers, the current program counter etc.), the state of the run-time stack and the values of all of the data objects (i.e. variables) being used by the process. This information is architecture dependent, and may require a significant amount of storage to represent (depending on the size of the process). Thus it is not feasible to attempt to restart a process on a computer of a different architecture. The amount of information which would need to be transmitted across a LAN would also pose problems when trying to move state information to another node quickly.

Rather than trying to restart a program component from the point at which it was stopped, it is easier to restart it from a recorded checkpoint. This approach has been taken in the implementation of fault tolerant Linda systems [Kambhatla 1990]. The computation that was done between recording the last checkpoint and the time when the process was stopped must then be redone. If the program component which was stopped is in fact relatively short-lived, then there is not much cost involved in restarting it from scratch (as Piranha does). Remora programs take the same approach as Piranha by default, but it is possible for the programmer to add checkpoints, from which a program component can be restarted. The information needed for checkpoints is stored in tuple space. This information will generally be far more compact than the information describing the entire state of a process. Since checkpointing information is not transmitted at the time when back-off occurs, this makes backing off a much quicker procedure.

Making use of checkpoints, or restarting a task from scratch may result in the execution of a task being resumed from a position other than that at which it was stopped in a back-off procedure. We reason that it is possible to restart a program component from a position other than that at which it is stopped as long as the externally visible communication of that process is not affected. This is since the behaviour of the program component can be described by its communication with other parts of the system [Milner 1989]. It does not matter what internal steps that component performs, as long as the behaviour which is visible to the rest of the system is unaffected by the back-off procedure.

As part of the back-off procedure, the active tuple which is associated with the program component being executed is replaced into tuple space, so that it can be restarted elsewhere. This means that all of the information required for back-off can be stored in tuple space, and thus the entire state of the distributed computation is represented by tuple space.

4.4. Tuple Logging

When a program component is restarted, either from the beginning or from a checkpoint, it will naturally need to perform all of the tuple space interactions which it did before it was stopped when the node it was previously running on backed off. Tuples which were, for example, previously removed from tuple space would however no longer be there, and the behaviour of the program component would be altered, since its interaction with tuple space would be affected. Similarly, extra copies of any tuples which were placed into tuple space previously would be deposited there again while the program component performs the actions necessary to reach the state at which it was previously stopped. This is clearly a problem, since the behaviour of the program as a whole would be altered from its intended behaviour as a result of the communication of that particular component differing from its intended pattern.

This problem is solved by logging all of the communication which a program does via tuple space. All of the tuples which a program component retrieves are stored in a separate *log space* until the program component completes execution.[7] A record of all tuples deposited into tuple space by the program component is also kept. This makes it possible to *replay* the tuple space interactions which occurred previously, before a back-off. Tuples which were previously removed from tuple space can be retrieved from the log-space, while Linda output operations which were previously performed can be replaced by null operations.

The use of logging hides the fact that a program component's communication is being replayed. This allows consistency to be maintained in tuple space, despite the fact that a program component may redo part of its computation while attaining a previously reached state. Other program components will not be aware of the fact that one component is being restarted (they will in fact not be aware of the fact that a component was ever stopped), while the program component which is replaying Linda operations will not be aware of this fact either. Once a program component has reached the position at which it previously stopped, it will cease making use of logs, and will interact with the tuple space normally again.

---

7 This appraoch is similar to the one taken by [Kambhatla 1990] in the implementation of fault tolerant Linda processes.

4.5. Potential Problems

4.5.1. Interprocess Dependencies

There are, unfortunately, some potential problems with the model of adaptive parallelism described here. If there are less processors available than there are active tuples ready to be processed in tuple space, then not all of these active tuples will be consumed concurrently by the system, since the number of Remora client processes depends on the number of processors available.

A program, which makes assumptions about the number of concurrently executing processes, may deadlock. This can occur if one program component attempts to input a tuple which should have been created by another program component which is still waiting for an available processor on which to run. Such a program is however unsuited to an adaptive model of parallelism, since it fails to properly exploit the dynamically varying pool of processors (and subsequently processes) available under such a model.

The Linda model attempts to present an environment for parallel programming in which the number of processors on which a program runs can be scaled without changing the program itself [Ahuja *et al* 1986] - a feature which a program with a fixed number of processes cannot exploit. Typically, under the Linda model, processes are only added as processors are added, and the number of processes per CPU is limited to one, in order eliminate unnecessary context switching [*ibid*]. If a program explicitly requires that a certain number of processes be loaded and executing at a given time it can be regarded to be a bad Linda program, as it does not take into account the temporally decoupled nature of Linda.

4.5.2. Referential Transparency

The C programming language (which is the host language for Remora) does not support referential transparency. One implication of this is that, if a piece of code is executed again, while a program component is restarting after a back-off, the results generated while executing that piece of code a second time might, under certain circumstances, differ from the results which were produced the first time it was executed.

Let us consider the contrived example in listing 1. We assume that the program component is stopped due to a back-off while the function *something_else()* is being executed. When it is restarted an almost certainly different pseudorandom number will be placed in the variable *y*. The tuple created by the first *out* operation would have been

```
worker()
  {
    long y;

    y = random();
    out("value", y);
    something_else();
    out("value2", 2*y);
  }
```

Listing 1: Breaking referential transparency

placed into tuple space before the back-off occurred. This operation will be converted into a null operation the second time it is executed, since the tuple concerned has already been placed in tuple space. When the second *out* operation is finally executed, a different value of *y* will be used from when the first tuple was placed into tuple space, thus creating an inconsistency and a potential source of error.

It is however improbable that the behaviour described above is likely to cause problems in the coding of real applications. As a result no attempt is made to resolve this problem within Remora, and it is left as the programmer's responsibility to ensure that such problems do not occur.

Programs which make assumptions about the times (relative or absolute) at which events occur are likely to have similar problems to the one described above. Such programs would however already be "bad" Linda programs, since they do not make allowance for the temporally decoupled, and nondeterministic, nature of Linda.

Chapter 5.

Heterogeneous Computing Environments

Most networked environments tend not to be a homogeneous collection of one vendor's hardware, all running the same operating system, but tend rather to be a heterogeneous collection of architecturally different computers, running operating systems from different vendors. In such an environment, while the total number of computers connected to a LAN may often be relatively large, the numbers of each type of computer may be relatively small by comparison.

When developing a distributed programming environment for performing parallel computations on a networked cluster of computers, we would thus like to exploit as great a proportion of the total number of computers available to us as is possible. This means developing a system which can transparently run across different platforms, possibly hiding the differences between the different types of computers on which it is hosted.

The use of heterogeneous computing resources makes it possible to utilize existing resources more effectively, and presents a cost saving when constructing an environment for high-performance computing. Thus powerful computing resources can be placed within the reach of organizations who lack the funds required for the purchase of supercomputers.

5.1. Approaches to Heterogeneity

One approach to implementing a heterogeneous distributed system is to selectively make use of the strengths of each type of computer available on a network. Applications written for such a system would be broken down into components which are suited to being performed on certain types of architectures. Thus one might make use of a supercomputer with a vector processor for one part of a computation, perform another part on a specialized parallel architecture, and display the results on a workstation with a powerful graphics subsystem. This kind of approach is adopted in the implementation of *Schooner* [Homer 1992].

In contrast with this, other heterogeneous systems are developed in such a way as to try and hide the differences between computers, rather than explicitly exploiting these differences. This approach attempts to produce a view (as far as the application programmer is concerned), of a sea of essentially equivalent processors, across which applications can be distributed. This approach, which is the one used by Remora, is also taken by PVM [Dongarra *et al* 1993], and is also generally the approach taken for supporting heterogeneity under Linda [Carriero *et al* 1992, Smith 1993]. Systems which make use of this approach attempt to present a portable, general-purpose programming environment, in which the programmer is shielded from the lower level aspects of the hardware architecture and operating system on which his programs are running.

5.2. Manifestations of Heterogeneity

A number of factors present in both hardware and software need to be taken into account in order to allow transparent interchange of data between computers in a heterogeneous environment. Once these factors have been identified, an external data representation, which can be used for communicating between different computers must be developed.

5.2.1. Hardware

At the lowest, level the differences between different CPU architectures need to be recognized. In the implementation of Remora the following architectures (all present in computers in use at Rhodes University) have been considered:

- Intel i386 and i486 (or compatible)
- Sun SPARC
- Motorola 68020 (or compatible)
- Inmos T800 (transputer)[8]

The most important factors which are considered for different CPU architectures are as follows:

- Byte ordering ("endianness")
- Data alignment - some architectures require that data be aligned on word boundaries (e.g. the SPARC requires that a data item of size *n* bytes be aligned on an *n* byte boundary).

[8] The T800 port of Remora is currently being implemented by Simon Barratt, as part of a graduate project at Rhodes University.

- Word size - this affects the sizes of the basic data types available on an architecture.
- Floating point representations - while most modern architectures adhere to the IEEE standard, architectures such as the VAX do not.

Apart from considering the CPU itself, it may also be necessary to consider other characteristics of an architecture. One such characteristic is the number of CPU's present in one computer of a particular type. Taking this factor into account allows for easier integration of multiprocessors[9] into a distributed system.

5.2.2. Operating Systems

Unfortunately differences between computers are not restricted to hardware, but are also noticeable in their operating systems. While these differences don't normally affect the data representations used, the network interfaces and interprocess communication facilities provided by different operating systems vary, and are sometimes incompatible. For this reason it is safest to restrict oneself to using widely implemented and standardized operating system interfaces.

In the implementation of Remora, only a limited amount of operating system heterogeneity is supported, and the present implementation assumes a UNIX type operating system which supports Berkeley sockets and either BSD or POSIX signals. The initial development of Remora was done using FreeBSD[10] (i486), SunOS 4.1.x (SPARC and Motorola 68020) and Solaris 2.x (SPARC). Remora has been ported to Linux[11] (i386 and i486), and currently work is underway on a port to a cluster of Inmos T800 transputers running Helios.

5.2.3. Communication Media

A factor which is not considered in this project is the possibility that the computers being used in a distributed computation may not all be connected by a homogeneous network. While Remora assumes that all of the nodes in a cluster are connected by a uniform network (such as an ethernet LAN), it is possible that other forms of connectivity may also be present, for instance:

---

9 Multiprocessor architectures presently in use at Rhodes University include a Sun SparcServer and two T800 based MC^2 Maxiclusters

10 Initially development was done on 386BSD

11 The Linux port does not yet include the preprocessor. (While Linux does share a lot of common features with SYSV and BSD UNIX, it is worth noting that the Linux port of Remora was completed in approximately two hours.)

- some nodes may be connected by a high-speed fibre-optic network,
- CPU's in specialized parallel message passing architectures (e.g. a network of transputers) are generally connected by a high-speed bus or network, or
- some nodes may be situated further away, on another LAN, and may only be reachable by low-speed links.

Knowledge of the nature of the communication links between nodes may be used in order to place processes more effectively, so that communication overheads are minimized. This kind of optimization however also requires *a priori* knowledge of the communication behaviour of programs.

5.2.4. Programming Languages

Another aspect which is not explored here is heterogeneity with respect to programming languages. It could potentially be desirable to have an environment in which components written in different high-level languages could interact to perform a computation together [Gelernter and Carriero 1992]. This would be a further generalization of portability, and would promote the reuse of software components, even when these are written in different programming languages.

5.3. Advantages of Linda

The decoupling action of the Linda tuple space facilitates the implementation of heterogeneous systems. Since Linda processes only interact with tuple space, and do not communicate directly with each other, they do not need to know any information about each other [Carriero *et al* 1992]. This characteristic of Linda makes it easy to permit communication between program components executing on different architectures, and even written in different languages. (It would, for example, be possible for a C-Linda program component to communicate with a Modula-2-Linda component.)

The abstraction provided by tuple space is also helpful in providing facilities to convert between different data representations. Data items which are deposited in tuple space are stored in a common representation. Routines in the Remora run-time library convert tuples into this representation as they are deposited into tuple space. Tuples are then converted to the correct host data representation when they are retrieved from tuple space.

Details of the particular network topology of a cluster of workstations being used in a computation are also hidden behind the interface provided by tuple space. The programmer need not concern himself with details about the location of nodes in the

cluster, or routing between nodes in the cluster. All of these functions are handled by tuple space. The fact that tuple space completely hides the underlying network transport mechanisms even makes it possible to accommodate clusters of computers which span a collection of networks on which possibly different networking protocols are used.

5.4. Design Considerations

The differences between computers in a heterogeneous cluster are mainly hardware-related. As a result, most of the design decisions which must be made, in order to support heterogeneity, are of a low-level nature, and tend to involve a fair degree of technical detail. In seeking methods for supporting heterogeneity, system features which might not be supported by some platforms have been avoided.

5.4.1. Data Representation

The formats of the messages used to interchange information between computers participating in a Remora computation have been designed to make use of a simple external data representation, which can easily be implemented on any new architecture to which Remora is ported. The data representation used is similar to Sun Microsystems' External Data Representation protocol (XDR) [SMI 1990], but provides minimalist functionality when compared to XDR. It was decided not to make use of Sun XDR in the implementation of Remora due to the potential difficulties which might be experienced when porting Remora to an operating system which does not support XDR.[12]

In implementing the data representation used for Remora the following criteria were applied:

- all numeric values are represented as 32 bit values,
- all items are aligned on 32 bit boundaries,
- all numeric values are stored in network byte order (most significant byte first),
- character strings are padded to be multiples of 32 bits long, and
- real numbers must be represented in a portable format.

While the use of 32 bit values throughout may seem wasteful for representing small numbers, this waste is small compared with the effort which would be required to unpack messages containing data which is not aligned on word boundaries on an architecture which requires this. (Support for 64 bit architectures has been omitted.)

---

[12] When the development of Remora was started, XDR was not fully supported under 386BSD. Support for XDR is also not available under Helios [Perihelion 1991].

The data representation makes no allowance for the handling of structures, but restricts itself to basic data types, namely integers, strings and double precision floating point numbers. All other atomic data types can be represented making use of these data types, while more complex data structures can be represented by tuples.

5.4.2. System Layering

The design of Remora makes use of a layered approach, in which one of the lower layers is devoted to the handling of external data representations. Apart from this layer, the rest of the system is constructed much as if it were intended for use in a homogeneous environment.

5.5. Implications for Adaptive Parallelism

Operating in a heterogeneous environment complicates the migration of program components from one node to another when a node is required to back off. Since the computers taking part in a computation may be architecturally different, it is not possible to merely transfer a process image from one computer to another, and then restart it. This problem is addressed by storing the state of a computation in tuple space, in an architecture independent form.[13]

Apart from problems related to the architectures of the computers being used, more subtle problems present themselves as well. Since the computers being used are likely to vary a great deal in their performance, the amount of load (created by a distributed computation) which can be tolerated is likely to differ from one computer to another. This means that the methods used to measure system load (and thus determining when to back off) should be flexible enough to allow for these differences. (It would be possible, for instance, to tolerate a higher load on a powerful server than on a personal desktop computer.)

The potential differences in processing power between different computers in a heterogeneous environment can also have an effect on the load balance of a computation. Since all participating computers will accept the first task made available to them[14], a situation could potentially arise in which one of the least powerful nodes is left executing a task while all the remaining nodes have completed their work.

---

[13] See section 4.3. (page 19)

[14] At present Remora makes no attempt to assign a task to the node most suited to performing that task.

The techniques used in order to overcome the problems mentioned here are discussed in the following three chapters, which focus on the implementation of Remora. Chapter 6 first examines the overall coordination of the different components of the system. Chapters 7 and 8 discuss the implementation of Remora client processes and the tuple space server respectively.

Chapter 6.

System Coordination

Remora makes use of a coordination "harness" which manages the hosts which are participating in a distributed computation. Coordination of the nodes taking part in a computation is performed separately from the communication necessary for access to tuple space. Essentially, Linda communication is implemented on top of the coordination harness.

6.1. Network Transport

Network communication is one of the most critical areas of the implementation of a distributed programming environment. This is due to the fact that the overheads introduced by interprocess communication over a network (be this a WAN or LAN) can have a significant effect on the performance of a program. It is for this reason that networking can be regarded as one of the biggest performance bottlenecks which must be overcome.

A lot of parallel systems make use of network topologies which are chosen specifically because they perform well in the application area in which they are being used. This luxury of choice is however not available to the implementor of a distributed system, which makes use of existing hardware. In this kind of scenario it is important to make the best use of the available networking facilities.

6.1.1. TCP Streams

The present implementation of Remora makes use of TCP streams (over ethernet) for all communication between nodes. This choice is based on the widespread use of TCP/IP, which is almost a *de facto* standard for networking in heterogeneous environments. A further benefit of the use of TCP is that streams provide a reliable point to point connection, relieving the implementor of the responsibility of ensuring that all messages which are transmitted are received correctly.

The facilities provided by TCP streams are oriented towards a general-purpose, point to point, byte stream, which can be used for a wide range of applications. This is actually more generic than what is required by Remora. Remora sends messages as blocks, which could generally fit into single IP packets. Given this observation, it might seem logical that UDP (User Datagram Protocol), which is oriented towards the transmission of datagrams (data blocks or packets, with a fixed maximum length) would possibly deliver better performance in the kind of application under consideration. (The differences in the data transfer rates attainable when transmitting fixed length messages using UDP and TCP are shown in figure 1.)

Message Size (bytes)	32	64	128	256	512	1024	2048
TCP Streams	19	42	72	108	169	248	283
IP Datagrams (UDP)	35	62	110	185	327	464	588

Figure 1: Transfer rates for TCP and UDP (in kbytes per second)[15]

Unlike TCP (which is a reliable protocol), UDP datagrams are not guaranteed to reach their destination. Thus it is necessary for the user program (in this case the Remora system) to ensure that messages are transmitted reliably, placing an additional burden on the implementor. This is one of the major reasons why, at present, TCP is being used rather than UDP.

With the porting of Remora to parallel architectures, like transputer arrays, the system will have to make use of the communication channels available - such as transputer links. In order to facilitate such a port, Remora's communication interface is isolated in one layer, which provides services which do not imply the use of any particular protocol suite. This should provide for the implementation of heterogeneous communications, allowing, for instance, a transputer array to cooperate with workstations on a LAN, in order to form a single distributed system.

6.1.2. Berkeley Sockets

The most widely used interface to the TCP/IP networking protocols is that provided by "Berkeley Sockets", which first appeared in version 4.2 of BSD UNIX. This interface has also been adopted in USL's SVR4 UNIX, making sockets a *de facto* networking

[15] Data from [Smith 1993]

standard in UNIX. While the TLI (Transport Layer Interface) developed at Bell Labs, and initially supported in SVR3, is to a large degree superior to the socket interface, TLI is not nearly as widely used. (The socket interface has also been adopted by Microsoft in MS Windows [Davis 1993] and Windows NT.)

6.2. Coordination Overview

Coordination of Remora is centred around the *master node* which provides a central point of contact for all of the nodes in a cluster (see figure 2). All information necessary for coordinating a computation is centralized on the master node. The coordination harness is implemented via a *coordinator* process, which runs on the master node, and *daemon* processes, which run on each of the nodes (including the master node) which are used to perform distributed computations. The coordinator and daemon processes run in the background, and only consume small amounts of CPU time. (While no computations are being performed these processes sleep, and while computations are being performed, their CPU usage is negligible.)

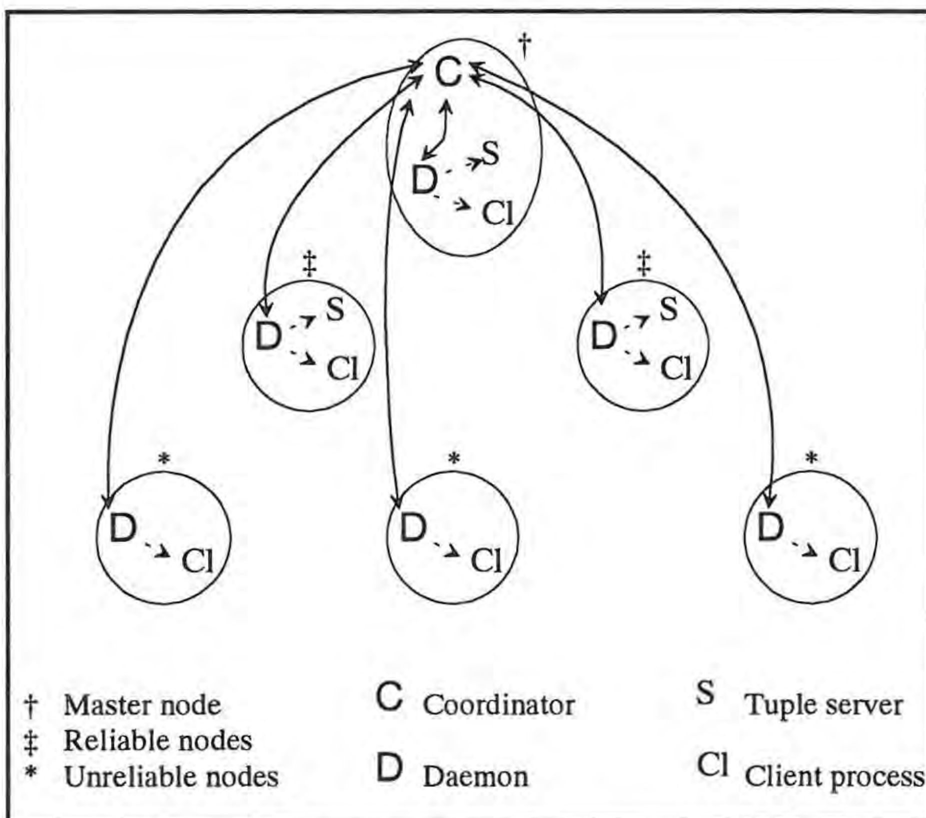

Figure 2: Remora coordination framework

6.2.1. Coordination Functions

The functions which are performed by the coordination harness include:

- providing a mechanism for nodes to enter and leave the group of available hosts,
- providing a service for nodes to locate services such as tuple space servers,
- providing a mechanism for maintaining information on the group of available hosts
- providing facilities for starting and stopping processes on remote nodes and
- performing monitoring of unreliable nodes, in order to determine when these nodes are available to perform computations.

The information which is required to perform these functions is gathered by the daemon processes, which transmit it to the coordinator.

6.3. Message Protocols

The information required for the coordination of nodes forming part of a Remora system is transmitted separately from the messages generated by interaction with tuple space. Information is transmitted between daemon processes and the coordinator in message blocks.

Each message block contains a message identifier (which identifies the type of message being sent), a number of fields which are common to most message types, and an optional variable length set of additional fields which contain information specific to certain message types. All fields are a multiple of 32 bits in length. (Character strings are padded to a multiple of 32 bits.) The structure of the message blocks has been designed in such a way as to be easily extensible, rather than to provide maximum transmission speed. This is because the number of messages transmitted between the daemons and the coordinator is relatively small (especially in comparison to the amount of data transmitted in performing Linda operations), and a small degree of inefficiency will not adversely affect system performance.

Currently six different message types are used for communication between daemon processes and the coordinator. These are described in figure 3.

Message Id.	Direction	Description
D_AVAILABLE	D → C	Daemon announces that node is available
D_LEAVING	D → C	Daemon announces that it is leaving the computation (backing off)
D_DAEMONID	C → D	Coordinator sends *id* to daemon (when the daemon initially contacts the coordinator)
D_CSTART	C → D	Coordinator tells daemon to start client processes (and tuple space server)
D_CSTOP	C → D	Coordinator tells daemon to stop client processes (and tuple space server)
D_DSTOP	C → D	Coordinator tells daemon to stop client processes (and tuple space server), and then terminate execution itself

Figure 3 : Messages transmitted between coordinator and daemon processes

6.4. Running User Programs

The daemon processes running on available nodes cooperate with the coordinator in order to provide the facilities which allow the running of Remora user programs. The facilities which they provide attempt to make it relatively easy for a user to start a distributed computation. This mechanism allows the user to start a program from the command line on *one* node, without having to log into all of the nodes on which the distributed computation is to run.

Before a user program can be executed, it must have been compiled on all of the different architectures represented in the cluster of computers which is being used by Remora. Each node must be able to access an executable copy of the program. In order to start the program, the user will invoke it from the command line while logged on to a node of his choice. As part of a startup procedure, which is linked into the user program, the user program will contact the coordinator (on the master node), and request that the coordinator instruct all available nodes to start the program. The coordinator will then instruct the daemons on all available nodes to start executing the user program (i.e. the Remora client process). Tuple space server processes will also be started on the reliable nodes which have been chosen to provide tuple space service.

The copy of the user program which is invoked from the command line starts executing from the main entry point into the program, while the copies running on all other nodes run in a loop retrieving active tuples from tuple space, and processing them. (In a typical master-worker type computation, the program invoked from the command line will act as the boss, placing work into tuple space, and then retrieving the results.)

All of the client processes executing on remote nodes are instructed to terminate when the program invoked by the user terminates. The same mechanism is used when the user aborts execution of the program by entering a `<control>-C` sequence from the keyboard. Once all client processes have terminated, tuple space server processes are also shut down.

6.5. The Remora Daemon

The Remora daemon is run as a background process on all nodes that may participate in a Remora computation. It does not require any special privileges, and can either be started automatically on system bootup, or interactively by a normal user. Once started, the daemon controls the node's participation in Remora computations. The daemon will only terminate on system shutdown, or on receipt of a termination message from the coordinator process.

6.5.1. Behaviour

When started, the daemon first determines whether the node it is running on is "idle". Once the node has become idle, the daemon proceeds to contact the coordinator process (which is running on the master node), and announce that it is willing to join the pool of nodes which are available for use. The coordinator returns a "handle" or identifier to the daemon. This identifier is used by the daemon in all further messages it sends to the coordinator.

The daemon will then monitor the load of the node (if it is an unreliable node), and will initiate a backoff sequence if the load exceeds a predefined level. When the load drops below this level again, the daemon will contact the coordinator again, announcing that it is once again available.

When a node is available to take part in a computation (reliable nodes are always available) the daemon will accept messages from the coordinator, which direct it to start a Remora client process and (in the case of reliable nodes) a tuple space server process.

6.5.2. Load Monitoring

On unreliable nodes, it is the daemon's responsibility to monitor system load, in order to determine whether the node can be allowed to take part in a Remora computation. This ensures that a node is only used when it can be said to be "idle". It is important that the load monitoring procedure should be configured in such a way that the normal users of a

computer are not adversely affected by Remora's use of the computer. If this is not the case, then the owners of workstations and other computers will not be prepared to allow their computers to be used in distributed computations.

In the load monitoring subsystem of the Remora daemon, we make a clear distinction between mechanism and policy. This allows various different load measures (policies) to be used, so as to suit a specific environment. (If necessary, it is also possible to add additional policies, where these are not already available.)

Mechanism

A dual threshold is used in order to determine system load. If system load is below the first threshold, then a node is determined to be available. When system load exceeds the first threshold, a node is determined to be unavailable for use. Between the first and second threshold a node will accept no new work, but will complete the work it is currently doing. When the system load of a node which is participating in a computation exceeds the high threshold, all work being done on that node will be stopped immediately.

The use of a dual threshold is based on the observation that sometimes system load rises gradually. It would thus be desirable to attempt to detect the rising load, and finish all work that is currently being done, and not accept any work until the system load drops below the low threshold. This attempts to avoid the situation (which occurs when the high load threshold is exceeded) of having to stop a Remora task and restart it elsewhere, resulting in an additional amount of work being done.

Load samples are taken periodically. The frequency of load samples can be customised to local requirements. Samples should not be taken too frequently, since this will generate additional system load. Typically samples might be taken every minute, although even the additional load created by sampling every 10 seconds is negligible.

Policy

Each node may choose to monitor load in a different way, and may apply one or more different policies, which are determined via a configuration file when the daemon starts running. Each policy makes use of a different metric for measuring system load. When system load is determined, the current load threshold is determined for each policy which is in use. If, when measuring against any one policy which is in use, system load is determined to have exceeded one of the preset thresholds, then the node is required to back off.

The metrics which are currently used for load monitoring are derived from those discussed in [Kaminksy and Carriero 1991], [Bricker *et al* 1992], [Bjornson *et al* 1991] and [Roth and Setz 1992].[16] Currently the following metrics are used to determine system load:

- UNIX system load (number of currently runnable processes)[17],
- the number of users currently logged in,
- the number of active users, and
- console activity.

It has been found that the load metric which results in the least adverse affect to normal users of nodes is a combination of console activity and the number of active users. Since the user logged into the console of a workstation (who is often making use of a graphical user interface) is the most affected by tasks being run in the background, the decision is generally made to terminate the currently executing task as soon as activity on the console is sensed. Tasks are only accepted again after a period of console inactivity (irrespective of whether a user is logged in on the console or not).

6.5.3. Backing Off

When, after taking a system load sample, a node is determined to have become unavailable for participation in a computation, the daemon initiates a back-off procedure. This involves instructing the currently executing client process to terminate its work, and then informing the coordinator that the node has backed off.

The daemon communicates with Remora client processes via signals. This allows the daemon to control the activity of a client process, and instruct it either to back off, or to continue execution. The signals sent by the daemon to client processes are shown in figure 4. When a client process exits, the daemon receives a SIGCHLD signal, and is able to record the fact that the client has terminated execution.

Once the daemon has sent signals to any client processes running on the node, it sends a message to the coordinator, informing it that the node is no longer available to accept tasks. When the node's load once again drops below the first threshold, the daemon will contact the coordinator, and inform it that the node is available again.

---

16 See section 3.1. (page 13)

17 This measure has been found to be unreliable on some operating systems (especially SVR4).

Signal	Usage
SIGUSR1	Tell the client to finish its current task, and then exit. (This is done when the first load threshold is exceeded.)
SIGUSR2	Tell a client who has been sent a SIGUSR1 to once again accept new work. (This is done when the node's load once again drops below the first threshold.)
SIGTERM	Tell the client to stop its current task immediately, return the task to tuple space, and exit. (This is done when the high load threshold is exceeded.)
SIGINT	Tell the client to terminate execution immediately in the case of abnormal program termination. (No data is returned to tuple space.)

Figure 4: Signals used by the daemon to control a client process

6.5.4. Starting Remora Processes

One of the daemon's responsibilities is to start Remora client processes. Either when a computation is started, or when a node joins a running computation, the coordinator passes the name of the client program to be executed to the daemon, which then forks the client process. It is possible for the daemon to start client processes at a lowered priority, ensuring that processes run by the normal users of a node enjoy preference for CPU service.

On a subset of the reliable nodes, it is the daemon's responsibility to spawn a tuple space server process. A tuple space server process is forked by the daemon when a message is received from the coordinator, informing the daemon that a computation is starting. Once the computation has ended, it is also the daemon's responsibility to terminate the tuple space server.

6.5.5. Configurable Options

Various options which determine the behaviour of the daemon can be configured via a configuration file, which is read by the daemon at startup. These options include:

- load monitoring policies,
- the frequency of load samples,
- the number of client processes to be started,
- the priority at which client processes should run and
- networking options (e.g. specifying the name of the master node).

6.6. The Coordinator

The coordinator maintains overall control of all the nodes participating in a Remora computation. The master node, on which the coordinator is run, forms the central point of contact for all other nodes wishing to participate in a computation.

All information about the currently available nodes is maintained by the coordinator. Any node wishing to obtain information about other nodes in the pool must do so via the coordinator, as it is the only source of such information. The decision to centralize all information about the pool of nodes on a single coordinator process was made so that each other node need not be aware of the dynamically changing pool of nodes around it. The alternative, of maintaining information about other available nodes on each node would make the implementation of a dynamically changing host pool more difficult, and would also require that more configuration information be recorded for each node.

In a larger network, consisting of clusters of computers connected by low-speed links (e.g. a collection of LANs, connected by long-haul networks), it may be necessary to implement a distributed coordinator in order to minimize the delays experienced by clients when making queries. Thus, instead of having to query a coordinator on a remote LAN, for most queries, all necessary information would be located on a local coordinator process.

6.6.1. Information about Nodes

Each Remora daemon process supplies information about its node when it contacts the coordinator. This information includes:

- whether the node is a reliable or unreliable node,
- what operating system the node is running, and
- whether the node is available to run a tuple space server.

This information is used by the coordinator in order to control computations, and also to respond to lookup queries from clients wishing to connect to a tuple space server.

6.6.2. Tuple Group Placement

One of the responsibilities of the coordinator is to manage tuple space placement, by placing partitions of tuple space onto different nodes, and then providing lookup facilities for client processes wishing to communicate with tuple space. In order to make tuple

space lookups, each client process opens a network socket to the coordinator during its initialization procedure.

Remora makes use of the same kind of tuple space partitioning as is used in the transputer implementation of Rhoda [Wentworth 1990]. Unlike the Transputer implementation of Rhoda, Remora makes use of a distributed tuple space, where different tuple space partitions can be placed on servers running on different nodes.

When a tuple space partition is first accessed, the coordinator will decide which node that partition should be placed on, and will then redirect client processes to that node when they request the location of that partition. At present a round-robin algorithm is used for placing tuple space partitions. A better algorithm, which takes into account the usage patterns of different tuple space partitions would however be more desirable.

6.7. Configuration

Configuration information for each node participating in a Remora computation is stored in a file which is read by each Remora component (the coordinator, daemons, client processes and tuple space server processes) on startup. The configuration file contains details relating to the local node, as well as the name of the master node. Any information which a node requires to know about another node is retrieved via the coordinator. This means that when a new node is added to the pool of hosts, only the configuration file for that machine needs to be edited.

Portions of the configuration file are only relevant to certain system components. For instance, settings for load monitoring policies are only of interest to the daemon, and are ignored by other processes.

6.8. System Monitoring

In addition to the functions already discussed, the coordination harness also provides facilities to monitor the activity of a Remora computation. This facility can be accessed by the user by telnetting to a TCP port on the master node, and then entering monitoring commands. This facility has proved useful as a debugging aid in the development of Remora, as well as being a useful tool for examining the progress of a distributed computation.

At present the following information can be retrieved from the monitoring facility:

- a list of the currently available nodes (including, for each node, whether it is reliable or unreliable, and for a reliable node, whether it is available to run a tuple space server process),
- a count of the number of active tuples processed so far,
- a count of the elapsed wall-clock time for a computation,
- a count of the elapsed CPU time for a computation, and
- status messages, printed as nodes contact the coordinator or back off.

Information on the amount of CPU time used is collected by the daemons as client processes exit, and then forwarded to the coordinator for processing. Client processes pass information about the number of active tuples they have processed to the coordinator when they exit. (Client processes make use of the same network socket used to make tuple space lookups in order to return this information.)

Chapter 7.

Implementing Adaptive Linda Clients

A Remora user program consists of a group of cooperating client processes, which interact with each other via tuple space. The mechanisms described in chapter 6 are used to coordinate client processes, enabling them to communicate with tuple space.

Client programs are written using the "ideal" Linda syntax. At compile time they are first passed through a preprocessor, which converts Linda operations into calls to a run-time support library, and then compiled using an ANSI C compiler.

7.1. Client Program Structure

Each user program consists of a number of sections, one of which is the main entry point at which execution starts when the program is invoked by the user from the command line. The remaining sections are the program components which are invoked via the *eval* operation.

Unlike the transputer version of Rhoda, in which user programs are segmented into separate executable images for each evalled section in a program, Remora user programs are generated as a single monolithic executable image, containing all of the code for the entire application.

The decision to place all of the code into one executable image is based on two observations:

- Firstly, if it were necessary to spawn another executable each time another active tuple is accepted, then the overhead of performing an *eval* operation would become excessive due to the overhead associated with starting a new UNIX process.
- Secondly, since most modern operating systems use demand-paged virtual memory systems, only the pages which are actually needed are faulted into the physical address space when an executable is loaded. Thus we are not penalized due to any overhead of loading large executables.

7.2. Client Communication

Almost all of the communication performed by client processes is done directly with tuple space servers[18]. Client processes also communicate with the coordinator, which provides information on the location of the servers serving individual tuple space partitions.

Communication with the coordinator is done using the same message protocol as is used for communication between daemon processes and the coordinator. All communication with tuple space servers makes use of a message format which is optimized specifically for transmitting Linda tuples. (Tuple space control messages are also sent as tuple messages.)

7.2.1. Message Types

Messages sent between client processes and tuple space servers are divided into 3 main categories, namely:

- Linda operations,
- Tuple space control messages, and
- Log space control messages.

The bulk of all messages sent falls into the first category (see figure 5). It is as a result of this observation that all messages sent to tuple space servers are sent as tuples, allowing for the message format used to be optimized for sending tuples.[19]

Tuple space control messages are used in the opening and closing of tuple space partitions. A tuple space partition is only opened once by a client, and is held open until the client process terminates. This avoids the overhead of opening tuple space partitions for each new task.

Log space control messages are provided to allow client processes to perform direct manipulations of the tuple logs stored on a tuple space server.

[18] The implementation of the distributed tuple space server is described in chapter 8.

[19] This optimization is derived from [Smith 1993].

Message ID	Class	Description
TS_In	Tuple message	Linda *in* operation
TS_Inp	Tuple message	Linda *inp* operation
TS_Read	Tuple message	Linda *rd* operation
TS_Readp	Tuple message	Linda *rdp* operation
TS_Out	Tuple message	Linda *out* operation
TS_Eval	Tuple message	Linda *eval* operation
TS_ReEval	Tuple message	Replace an active tuple into tuple space when backing off
TS_Ping	Tuple message	Echo request (debugging)
TS_Init	Control	Initialize tuple space server
TS_Open	Control	Open a tuple group
TS_Close	Control	Close a tuple group (does not delete)
TS_Delete	Control	Delete a tuple group
TS_Flush	Control	Flush all tuples in a tuple group
TS_StartLog	Log Control	Start logging of tuple space transactions
TS_StopLog	Log Control	Suspend logging of tuple space transactions
TS_LPurgeTo	Log Control	Purge log entries up to a given *tuple id*
TS_LPurgeFrom	Log Control	Purge log entries from a given *tuple id* onwards

Figure 5: Tuple message identifiers

7.2.2. Message Format

The tuple message format (see figure 6) is based on that described in [Smith 1993]. It makes use of a header block which contains information common to all tuple messages, as well as a data block of fixed length items (known as the "*fixed array*") and a block of variable length data items (known as the "*flexi array*"). These two data blocks are used to hold the fields of a tuple. All of the entries in the fixed array are 32 bit items (mostly numeric values). Data items whose length cannot be determined in advance (such as character strings) are stored in the flexi array.

Header fields present in tuple messages include:

- a *message identifier,* which determines the type of message being sent,
- the *task id* field, which identifies the task which generated the message,
- the *tuple group id* of the tuple group which is being accessed, and
- a unique message serial number (*tuple id*).

The values of actual fields are stored in the fixed and flexi arrays. No space is reserved in the message for formal fields. In the case of control messages, the data associated with the message is stored in the fixed and flexi arrays. Items in the flexi array are all padded to multiples of 32 bits.

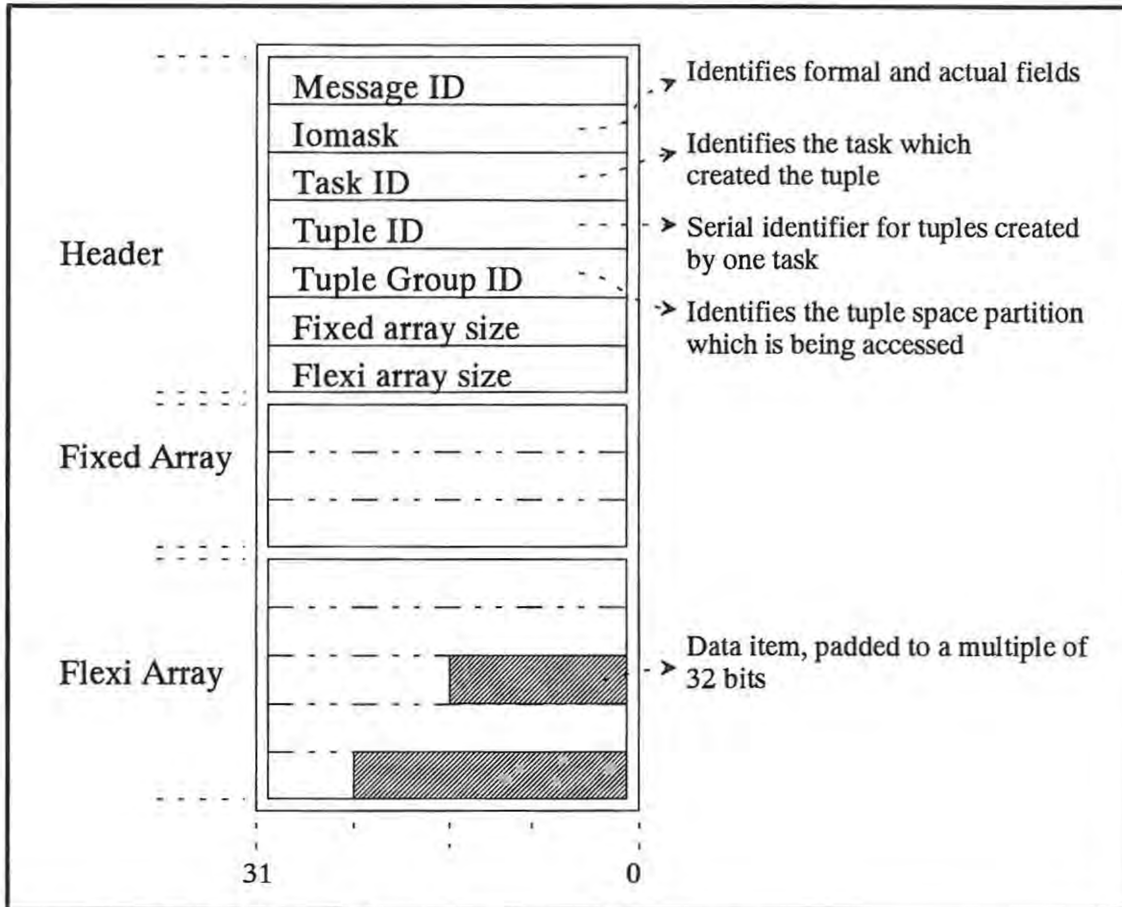

Figure 6: Tuple Message Format

7.2.3. Header Fields

A tuple is uniquely identified by the *task id* and *tuple id* header fields of the tuple message. This unique identification of all messages is required in order to perform logging of tuples retrieved by client processes. The *tuple group id* field identifies the tuple space partition which is being addressed. The *fixed size* and *flexi size* fields specify the size (in 32 bit words) of the fixed and flexi arrays. If a size of zero is given for either array, then the array is considered to be absent.

The *iomask* field is used to distinguish between *formal* and *actual* fields in a tuple.[20] This field is a 32-bit polarity mask, which represents the "direction" in which field values are being sent. (A formal field is regarded as "input" while an actual field is regarded as

[20] The *iopattern* field is derived from [Smith 1993 p. 22] and the Transputer implementation of Rhoda

"output".) Each bit in the *iomask* represents one field in the tuple (up to a maximum of 32 fields). A bit which is set represents an actual field and an unset bit a formal field.

It is not necessary to transmit information about the type and number of fields in each tuple message. This information is transmitted once when a tuple group is opened. This is possible since tuple space is partitioned into groups of tuples with the same arity[21] and field types (thus all transactions with a tuple group will operate on tuples with the same arity and field types).

7.2.4. Data Types

Four basic data types are supported in tuple messages, namely:

- 32 bit integers,
- double precision floating point,
- character strings and
- byte blocks.

Any other data types must be converted into these four data types before being packed into a tuple message. The byte block type is used for transmitting arrays of integers or floating point numbers. (The Remora preprocessor and the client run-time library handle the conversion of array data types.)

Complex data types are not supported by the tuple message format, since tuples themselves constitute complex data types, and can be used to build complex data structures. (It would however be possible to implement support for structures in the Remora preprocessor.)

7.3. Behaviour

Client processes first open a network socket[22] to the coordinator process on the master node when they begin execution. This socket is used to query the coordinator for the location of tuple groups, and is also used to return information which is used by Remora's monitoring subsystem.

All client processes automatically open the "*eval tuple group*", in which active tuples are stored. The instance of the user program which was invoked from the command line

---

21 The number of fields in a tuple.

22 While we refer to sockets here, the Remora implementation makes no direct reference to sockets, except in the low-level transport interface. This makes it possible for networking, or communication, interfaces other than sockets to be supported at a later stage.

(which we shall call the *"primary process"*) will then begin executing user code at the main entry point (generally this will constitute the "*boss*" of the Linda computation), while all other client processes (which are started by the daemons running on available nodes) will enter a loop and consume active tuples. It is in these processes that support for adaptive parallelism is implemented.

A client process will consume active tuples until it retrieves a "poison pill" tuple (deposited into tuple space by the run-time support code in the primary process), or it is told, by the daemon (via UNIX signals), to stop executing.

```
/* loop, retrieving active   */
/* tuples                    */
while(status == LIVE &&
        worker_id = POLLP())
  {
   switch(worker_id)
     {
      /* process active tuple */
     }
  }

/* we have retrieved a       */
/* poison tuple              */
if(status == live )
  {
   /* spread poison */
   EVAL(0);
  }
/* terminate execution       */
```

Listing 2 : Client process main loop[23]

Listing 2 shows a simplified version of the loop in the client which processes active tuples. A status flag, whose value is changed by signal handling routines, is interrogated on each iteration of the loop. As long as the node is available, the client's status remains "*LIVE*". If this status value changes due to a signal being sent to the client when the node's load exceeds the low threshold, then the client will exit when it begins the next iteration of the loop. When the high load threshold is exceeded, the signal handler immediately terminates execution of the client process. Normal termination of the loop occurs when the client retrieves a *poison* tuple, and then terminates execution. The poison tuple is deposited into tuple space by the primary process when it stops executing.

The running of a client program is effectively controlled by the primary process. Any input or output to the terminal or to files should be done via this process. It is only once the primary process has contacted the coordinator, that client processes are started on other available nodes. They are stopped again either when abnormal termination of the

23 This loop is implemented as part of the Remora run-time support system.

primary process is detected by the coordinator, or on retrieving the poison tuple which is deposited into tuple space by the primary process on normal termination.

7.4. Accessing Tuple Space

Tuple space is partitioned into tuple groups, each of which contains tuples of the same arity and type signature. Client processes which interact with tuple space must address all transactions to the correct tuple groups, where each tuple group may be physically located on a different server. (The partitioning of tuple space is dealt with in greater detail in the discussion about the distributed tuple space server in chapter 8.)

7.4.1. The Eval Group

All active tuples are stored in a special tuple group, which is known as the "*eval group*". All clients access this group, either to retrieve active tuples, or to place active tuples into tuple space, via the eval operation.

Since the eval operation in Remora uses the semantics used in the other Rhoda implementations, active tuples have no user data fields, and will thus all have the same arity and type signature. This allows all active tuples to be placed in the same partition of tuple space. The only data field carried on an active tuple is a numeric identifier which refers to the program component which must be executed in order to process the tuple. (It is not possible to pass the address in memory of the code to be executed, since the routines to be executed will reside at different locations in binaries generated for different architectures.)

The existence of this tuple group is hidden from the user by the Remora run-time support library, and code generated by the Remora preprocessor. User code, which is written in the ideal syntax, is only permitted to make use of *eval* to place tuples into the *eval group*. Tuples are removed from the *eval group*, via a special operation called *pollp*, which is a modified version of the Linda *in* operation.

The *reeval* operation is provided to allow partially evaluated active tuples to be replaced into tuple space. The *reeval* operation differs from the normal *eval* operation in that it adds information required for tuple logging to the active tuple. This information is required in order to begin replay of Linda operations when processing of an active tuple is restarted.

7.4.2. Contacting a Tuple Space Server

Each client process carries a table of tuple groups which it currently holds open, as well as a table of servers to which it currently has open sockets. The tuple groups used by one particular program component are opened before that component is executed the first time, but are only closed when the client terminates. Opening of tuple space partitions on demand prevents the opening of an excessive number of tuple groups by a client. Leaving tuple groups open after use avoids the additional overhead of reopening a tuple group when a program component is executed another time.

When a client process wishes to open a tuple group, it first queries the coordinator for the location of the tuple group. The query contains the textual name of the tuple group, to which the coordinator responds with the name of the server to contact and numeric identifiers for the tuple group and server. If there is not already a socket open to the server in question, then the client will open a socket to the server before opening the tuple group.

7.4.3. Tuple Space Operations

The Remora run-time support library provides interface functions which implement the basic tuple space operations. Linda operations, which are written using the ideal syntax in user programs are converted into library calls to the Remora run-time system by the Remora preprocessor.

All tuple space operations are implemented via a common back-end routine, which communicates with the tuple space server. The basic steps required to perform each tuple space operation are as follows:

1. Based on the type signature of the tuple group, build the fields provided into a tuple message, making all data conversions necessary to support communication in a heterogeneous environment.
2. Create a message header, and tag the tuple with task and tuple identifiers
3. Transmit the message to the tuple space server
4. If the transaction is an input operation:
    a. Wait for the response from the server, and, when the reply is received,
    b. load any formal fields back into the user's argument list.

Since it is possible for a user program to be interrupted by a signal when commanded by the daemon to back off, it is important to ensure that all tuple space operations are performed atomically, otherwise the integrity of tuple space cannot be guaranteed.

Since it is possible for a client process to block while waiting for a response from the server, it may be necessary to interrupt an input operation in order to terminate the client process when the node backs off. This means that it is necessary to implement input operations in such a way that, while the client is waiting for a response from the server, the operation can be interrupted. This is done by breaking down input operations into two sections which are each executed atomically with respect to signals.

If the client is interrupted while waiting for a response from the server (before executing the second atomic section of an input operation), then the client must cancel the request it has sent to the tuple space server. This has the effect of creating the appearance that the request was never made.

It is worth noting that the correct execution of the user program and the integrity of tuple space are not affected if the server responds simultaneously with the client's cancellation of its input request. This would merely have the effect of creating the impression on the server that the tuple in question has already been retrieved, and would place the tuple into the log space, from where it would be retrieved when the task which requested the tuple is restarted again.

7.5. Controlling Client Processes

Except in the case of the primary process, the execution of a client process on an available node is controlled by the Remora daemon running on that node. Client processes are spawned by the daemon, which will terminate them again, either when the user program terminates, or when the node becomes unavailable and the Remora client process is required to back off.

In order to control a client process, the daemon communicates with it via signals, which are sent to the client process to indicate that certain events have occurred. Signals which are sent to a client act as commands, telling the client to perform certain actions. Figure 4 (see page 39) shows the signals used to control client processes. Communication between a client process and the Remora daemon occurs only in one direction, from the daemon to the client process.

In addition to reacting to signals sent by the Remora daemon, a client process will also react to the SIGTERM signal sent to all running processes by *init(8)* during an orderly system shutdown.

7.6. Tasks

When an active tuple is deposited into tuple space by a user program (via the *eval* operation) this brings into existence a *task*. A task is executed when the active tuple associated with the task is retrieved by an available node.

Each task is uniquely identified by the task identifier which is stored in the active tuple, when it is deposited in tuple space. Unique identification of tasks is necessary in order to track the tuple space transactions made by tasks so that tuple logs can be built by the tuple space server on a per task basis.

Once a client process starts executing a task, it takes on the task *id* of that task, and all tuples generated by the client process are tagged with that task *id*. When the task has completed correctly, the client process invalidates its task id, so as to flag that no task is currently being executed. The client will then reload a valid task identifier when it starts executing a new task.

Making the current task *id* of a client invalid once a task has been completed allows the client process to determine, when it is sent a signal to back off, whether the last task it accepted has been completed or not. If the current task identifier is invalid, then we know that the last task accepted was completed, and no action needs to be taken.

If the task identifier is found to be valid, then we know that a task has been interrupted by a signal telling the client to back off. In this case it is necessary to replace the active tuple into tuple space before the client process can terminate. The active tuple is replaced by using the *reeval* operation, which replaces the tuple into tuple space. The *reeval* operation additionally stores information about the tuple identifier of the last tuple message sent by the client. This value is used when the task is restarted, and it begins replaying tuple space transactions, retrieving tuples from the log space.

When a client process starts a new task, it does not send the tuple space servers it is connected to any kind of message telling them that a new task is being executed. Doing so would be extremely costly, since it would be necessary to send a message to each server that the client process has connected to. Instead, it is left up to the servers to detect that a client has started executing a new task. This is done by recognizing that the client has started tagging its tuple messages with a different task *id*.

7.7. Replaying Tuple Space Transactions

When a task which was previously stopped (due to the node it was running on backing off) is restarted, it must replay any tuple space interaction which must be performed in order to attain the state the task reached previously, before being stopped.

Each time a task is started, a client process loads the tuple *id* of the last tuple message generated by the task before it backed off. (In the case of a task being started for the first time, this value will be zero.) For each tuple message generated, the client process decides whether or not that message was sent before. This is done by comparing the current tuple identifier with the identifier of the last message generated previously.

For output operations (*out* and *eval*), if the message has already been sent before, then the message is discarded by the client, as the tuple generated by the operation will already have been deposited into tuple space. This saves the client from actually sending any messages to tuple space.

In the case of input operations (*in*, *rd*, *inp* and *rdp*) it is still necessary for the client to retrieve a tuple from a tuple space server. The client will thus still contact tuple space when replaying input operations which were performed before. The tuple space server which services the requests will detect that the input operations in question are being replayed, and will satisfy them from the log space.

7.8. Direct Manipulation of Tuple Logs

It is possible for a task to directly manipulate the state of the logs maintained for it by the tuple space servers it is connected to. These facilities can be used by user programs in which the programmer manually implements checkpoints from which a task can be restarted at a later time. This allows a programmer to write programs in which it is possible for a task to restart from a point other than the beginning, when it is restarted after having been backed off.

A problematic aspect of the facilities provided for direct manipulation of tuple logs is that, while these facilities provide a mechanism for making Remora programs more efficient, they provide a fertile breeding ground for errors in user programs. It is possibly best to use these facilities to enhance programs which already work correctly, rather than using them from the beginning of the development of a program.

Currently the following log space manipulation primitives are implemented:

- enable logging,
- disable logging,
- retrieve the current tuple message identifier (acts as a pointer into the log space),
- purge log space up to a given tuple *id*, and
- purge log space from a given tuple *id* onwards.

In order to implement a checkpoint, the programmer would typically store information which would be used in order to restart the program in tuple space, and then purge the tuple logs up to the point where the checkpoint was made. When restarting, a task would then have to retrieve the checkpoint information from tuple space, and reinstate its state, based on that information. The task would then continue execution at the checkpoint. If the task was previously stopped at a point past the checkpoint, then it uses the tuple space logs to replay the tuple space transactions that were previously made after the checkpoint was reached, but before the task was stopped.

Chapter 8.

The Distributed Tuple Space Server

The Remora tuple space is implemented using a distributed server, which is the largest and most complex part of the entire system. The distributed tuple space service consists of a group of collaborating tuple space server processes, running on a subset of the nodes available to the Remora system. Each server process carries a number of tuple space partitions, and manages network connections from client processes which are interacting with tuple space.

When a computation is started, the coordinator sends a command to the Remora daemons on those reliable nodes which have been designated as tuple space servers, instructing them to start running tuple space server processes. (Not all reliable nodes are required to provide tuple space service.[24]) Access by client processes to server processes is controlled by the coordinator process, which provides the facilities necessary for clients to locate tuple space servers. The coordinator is also responsible for providing the services required in order to distribute tuple space partitions to different servers.

8.1. Tuple Space Distribution

In order to distribute tuple space over a number of nodes, it is necessary to provide a mechanism whereby tuples can be deposited into tuple space, and then located at a later time for retrieval. Optimal methods of distributing tuple space across a number of nodes, which make use of techniques such as tuple replication and hashed searches, have been the focus of much research, and are discussed in [Bjornson *et al* 1989, Zenith 1990, de-Heer-Menlah 1991]. The approach to tuple distribution taken in the implementation of Remora is essentially the same as that used in the Transputer implementation of Rhoda[25] and in [Smith 1993].

---

24 A reliable node can be designated as a tuple space server by including the "server" option in its node configuration file. (See listing 4, on page 81.)

25 Tuple space partitioning is discussed in [de-Heer-Menlah 1991]

Tuple Groups

At compile time, tuple space is partitioned into a set of disjoint *tuple groups*, each of which can be placed on a separate server node. Each tuple group contains only tuples of the same arity and type signature. This greatly simplifies the searching of tuple space for a tuple, since only one tuple group (and thus only one server) need be queried when a tuple is retrieved from tuple space.

The placement of tuple groups is determined by the coordinator process, which also handles queries, for the locations of tuple groups, from client processes. When a client requests the location of a tuple group which has not yet been created, the coordinator will nominate a server node on which that tuple group will be placed, and redirect the client to that node. The choice of which node will serve a newly created tuple group is currently based on a round-robin algorithm, which is most probably suboptimal.[26]

The decision to maintain central control over the placement of tuple groups avoids a potential race condition when two or more clients attempt to open the same tuple group in close succession. If the placement of tuple groups were not centrally controlled by one agent, it would be possible, given the race condition mentioned, for a tuple group to be placed on more than one node, which would clearly be undesirable.

8.2. Server Structure

The tuple space server process has been decomposed into a number of separate functional entities. This structure is built on top of the framework provided by the "Generic Client-Server Interface" module (known as *GCSI*), which is implemented on top of the facilities provided by Remora's network transport interface. The *GCSI* layer encapsulates a number of operating system dependent features which are necessary for implementing the server (such as monitoring multiple client connections). These functions are isolated in this interface in order to provide greater support for heterogeneity when adding support for other operating systems and transport mechanisms (such as transputer links or shared memory).

---

[26] The partitioning of tuple space into tuple groups is performed by the preprocessor at compile-time, whereas the actual placement of these groups is controlled by the coordinator at run-time.

The facilities provided by the main components of the tuple space server are:

- client connection handling,
- request decoding and dispatching,
- transaction processing,
- tuple group management,
- tuple storage,
- tuple matching,
- tuple logging, and
- signal handling.

The relationships between these components are shown in figure 7. (Arrows represent the direction of transfer of control.)

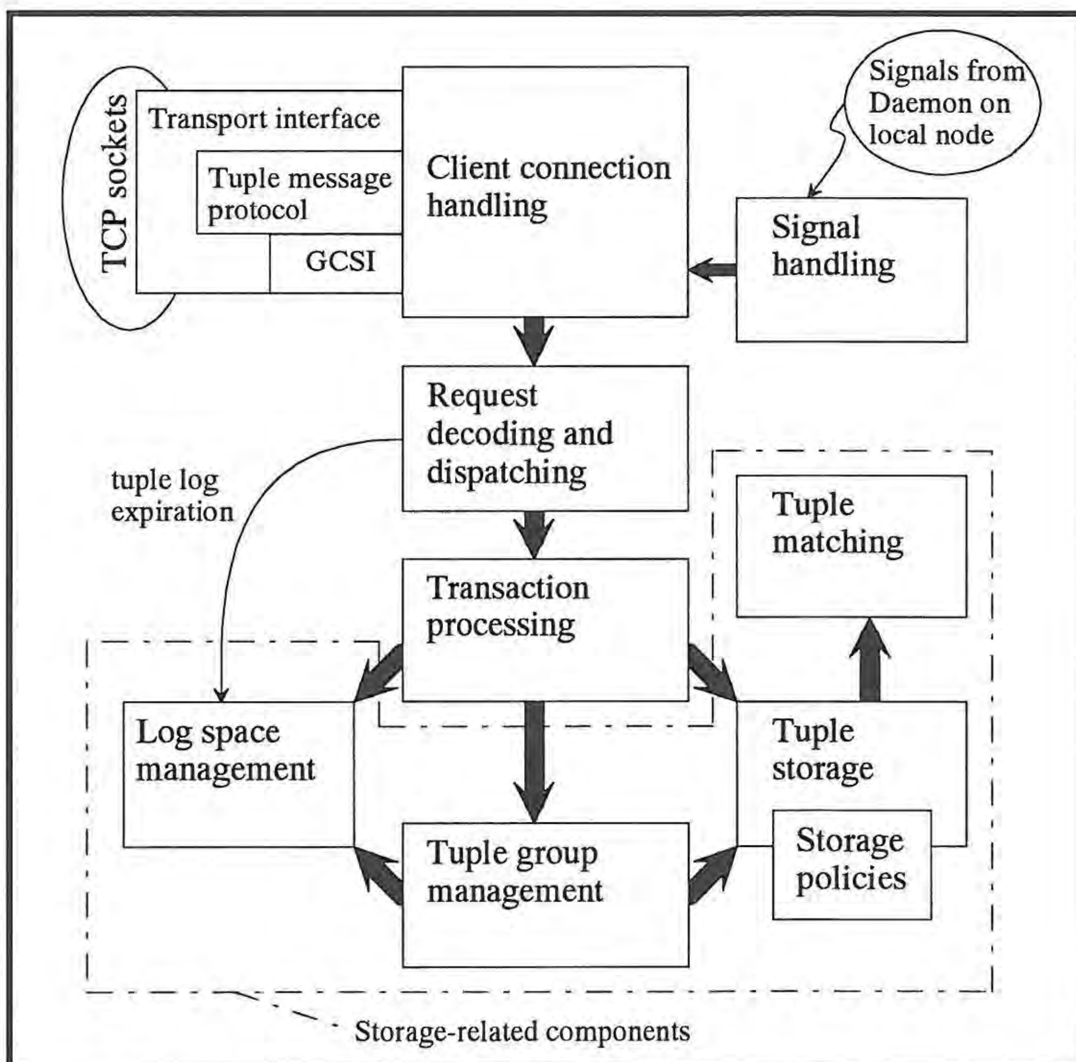

Figure 7: Structure of a tuple space server process

Handling External Input

Incoming messages are received by the server from the transport layer routines. It is first necessary to determine what kind of message has been received, after which the transaction associated with the message is processed. The processing of most messages results in the manipulation of some of the stored tuple space data.

The server listens for signals which are sent by the daemon process on the same node. Signals are used to tell the server to shut down when a computation is completed, or when the entire Remora system is shut down.

The server design is event driven, making use of a similar mechanisms to those used by applications written for windowing systems such as MS-Windows [Petzold 1992]. A main message handling loop (provided by *GCSI*) listens for messages from clients, and dispatches messages from clients to *callback routines* (also known as "*thunks*"). Each time a new client connects to a server, a callback routine is installed for that client. The callback routines process messages received from clients (returning responses if necessary) and then return control to the main server loop.

While processing a request from a client, a callback routine is not permitted to perform any operations which may block pending some external event, such as the receipt of a message from another client. Thus, for instance, if a tuple requested by a client is not available, the request is placed in a pending list to be processed later when a suitable tuple is deposited into tuple space. (This prevents any possible deadlock in the server.)

8.3. Tuple Storage

Each server maintains a set of data structures for each tuple group which it carries. These data structures collectively represent the stored image of tuple space. Three kinds of data item are placed in tuple storage:

Tuples: Tuples placed into tuple space by client processes (via the *out* and *eval* operations) are stored, pending their later retrieval.

Unsatisfied requests: If a request from a client cannot be satisfied from the tuples stored in tuple space, then it is stored, together with details of the client which made the request. When a tuple matching the client's request later becomes available, the tuple is returned to the client.

Logged responses: All tuple messages returned to unreliable clients are logged for possible reuse when tasks are restarted after having been backed off. Messages

are logged on a per-task basis. Logged messages belonging to one particular task are discarded when that task is completed.

8.3.1. Tuple Groups

A server process maintains a list of tuple groups which it is serving. New tuple groups are created as a result of clients opening tuple groups which do not yet exist. Each tuple group contains repositories of the three types of item (tuples, unsatisfied requests and logged messages) as mentioned above. (The structure of tuple space storage is shown in figure 8.)

8.3.2. Storage Policies

Tuples and unsatisfied requests are stored using a *storage policy* mechanism, which makes it possible for a different type of data structure to be used for storing each tuple group. This means that if it is possible to determine that the usage of a particular tuple group makes it more suited to being stored using a particular data structure, then a storage policy which provides that kind of data structure can be used. This allows individual tuple groups to be stored in such a way that tuples can be added and retrieved quickly. (For instance, a hash table might be used to store a tuple group which contains a large collection of records, allowing quick searching of the tuple group.) Suitable storage policies are chosen for tuple groups by the Remora preprocessor when an application program is compiled.

The storage policy mechanism, as implemented in Remora, provides a very flexible method whereby additional policies can be added without a great deal of effort being involved. Each storage policy maintains separate stores for tuples and unsatisfied requests. A policy is defined as a set of data structures and methods (see figure 9) which are used to manipulate them.[27]

[27] The policy mechanism is actually more suited to being coded in C++ rather than C, since it makes use of an object-oriented design. C was however used in order to simplify the reuse of code components from the older Rhoda implementations, and due to the possible lack of a C++ compiler on potential target architectures.

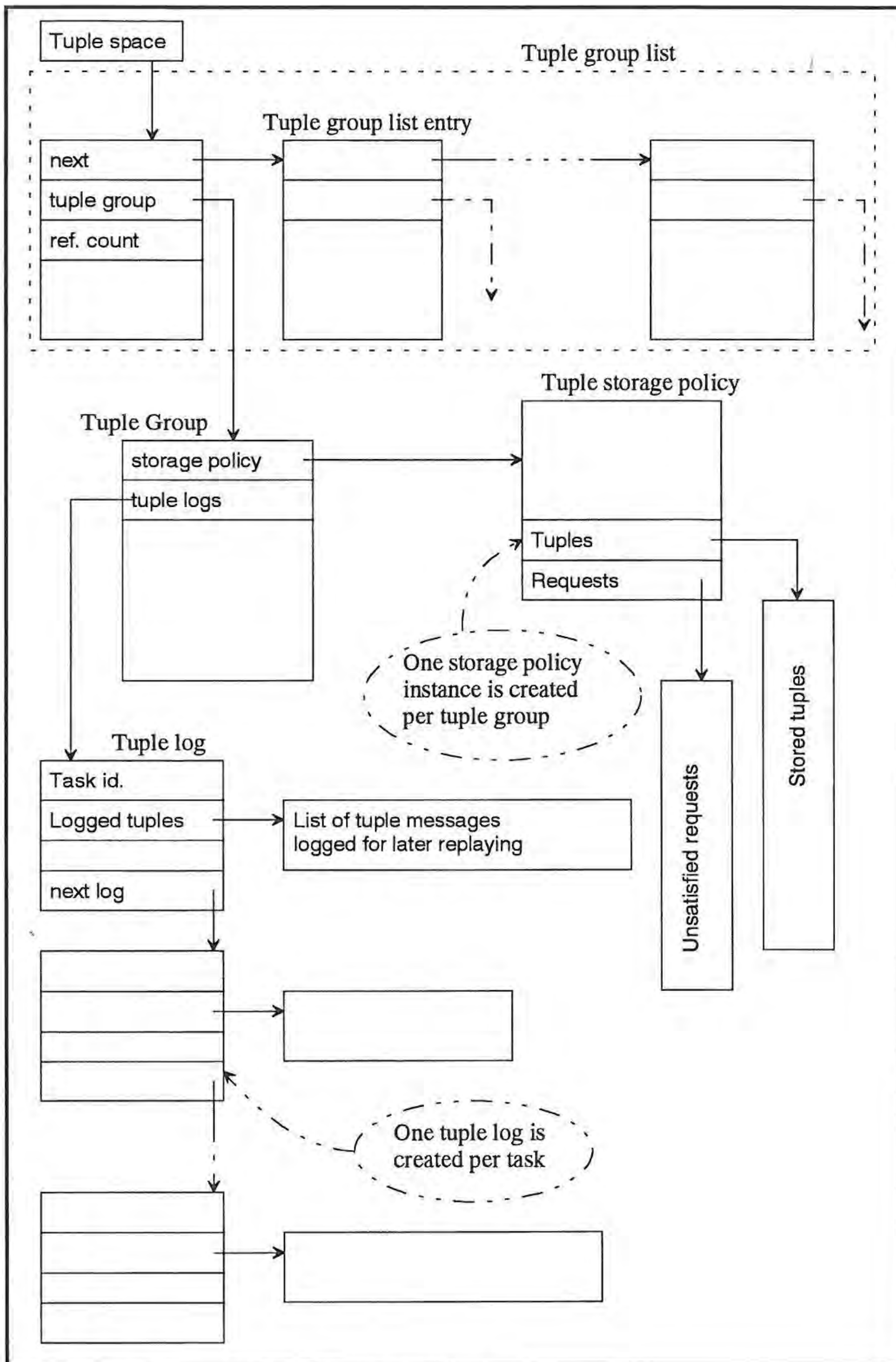

Figure 8: Structure of tuple space storage

At present, only two storage policies have been implemented, namely the *default* policy, which stores tuples in a linear list, and the *eval* policy which is a modified version of the default policy. The eval policy differs from the default policy, in that it has been specially tailored to handle active tuples. This allows the storage policy mechanism to take on special functions, like the allocation of unique task identifiers to active tuples as they are deposited in tuple space. The eval policy also makes special allowance for the replacement of active tuples into tuple space. This feature is used when a task is terminated without running to completion, due to a client backing off.

Method	Description
new_policy	Create a new policy instance (constructor).
destroy	Delete a policy instance (destructor).
flush	Delete all tuples and unsatisfied requests
add_tuple	Add a tuple to the policy instance
read_tuple	Retrieve a tuple without deleting it. This is used for performing *rd* and *rdp* operations
remove_tuple	Retrieve a tuple, deleting it from the policy instance. This method is used for performing *in* and *inp* operations.
add_pending	Add a tuple to the list of unsatisfied requests.
get_pending	Search the list of unsatisfied requests for a request matching a provided tuple
del_pending	Delete a request with a specific tuple *id* from the list of unsatisfied requests. This is used to delete any pending requests which a client may have when it backs off.

Figure 9: Methods available for storage policies

One of the reasons for implementing the storage policy interface was due to the decision not to spend much time on implementing optimal data structures for tuple storage, as this is not the focus of this project. Instead, it is possible to investigate the optimal storage of tuples at a later stage.

8.3.3. Tuple Logs

Tuple logs are stored as linear lists which are sorted in ascending order of tuple identifier (which is the order in which the messages are sent by clients). Attached to each tuple group is a list of tuple logs, where each of these logs contains only tuples generated by one particular task.

Each time a tuple is sent to a client process in response to the client performing an *in*, *inp*, *rd*, or *rdp* operation, the tuple is appended to the log belonging to the task currently being executed by that client. (Note that messages returned to clients, not messages received from clients are logged.)

It is potentially possible for tuple logs to consume large amounts of storage in the tuple space server. It was however found that, in practice, tuple logs did not become excessively large, since most tasks are relatively short-lived, and their tuple logs are destroyed when they terminate. The storage overheads of tuple logs are also kept to a minimum by moving, rather than copying, tuples from tuple space to the log space wherever possible.

8.4. Supporting Adaptive Parallelism

Most of the support for adaptive parallelism in the tuple space server is related to the implementation of tuple logs. There are five main areas of functionality which are required for the maintenance of the log space, namely:

- opening new tuple logs for new tasks,
- deleting the tuple logs of tasks which have completed execution,
- appending tuple messages to logs as transactions are performed,
- replaying messages to tasks which are being restarted, and
- performing direct log manipulations in response to client requests.

8.4.1. Creation and Deletion of Tuple Logs

A new tuple log must be created every time a new task is started, while logs must be deleted when tasks complete execution (as the data they contain has no further use). These operations are dealt with together as it is generally the case that a client will complete one task and start a new one directly afterwards.

As was explained in section 7.6. (see page 52), client processes do not explicitly inform tuple space servers that they are starting to run a new task, as this would require sending a large number of messages to all of the servers to which the client is connected. Instead, it is the server's responsibility to detect that a task is terminating, or that a new task is starting.

In order to detect when a client process starts running a new task, the server keeps a record of the task *id* of the task that each client is executing. Each time a message is received from a client, the server compares the saved task *id* with that in the message. If

they differ, then one task has completed, and a new task is starting. The server must take appropriate action by deleting the old task's tuple log and creating a new log for the new task. These actions must be performed before the transaction requested by the client is processed, as the correct tuple logs must be in existence before any tuple space transactions can be performed.

8.4.2. Appending Tuples to Logs

During the execution of a task, any tuples that are returned to a client process as a result of it performing a tuple space input operation (i.e. *in*, *inp*, *rd* and *rdp*) are placed in the tuple log belonging to that task. Both the tuple which is returned to the client, and the logged copy of that tuple are tagged with the message *id* of the request received from the client. If a task needs to retrieve logged tuples at a later stage, then the message *id*'s of the task's requests are compared against the message *id* fields of the logged tuples, in order to determine which tuple should be returned.

8.4.3. Replaying Tuple Messages

When a task is started, a marker which points to the current position in its tuple log is created. The initial value of this marker is set to the tuple *id* of the last tuple message previously sent by that task. In the case of a task which was not previously backed off, this marker will always be set to zero, since the tuple log in question will be empty.

As the task begins sending requests to the server, the tuple *id*'s of the requests are compared to the marker in order to determine whether the requests should be satisfied from tuple space, or from the log space. If the tuple *id* of a request is less than the marker, then the client is replaying previously processed requests. In this case, the tuples stored in the log space are used to satisfy the client's requests. As soon as the tuple *id*'s generated by the client become larger than the marker, replay is terminated, and requests are once again satisfied from tuple space.

This mechanism assumes that the behaviour of a task, with respect to its interaction with tuple space, is independent of factors such as the time when the task is executed. (i.e. The behaviour of a task, when it is being restarted, should be the same as its behaviour before it was previously stopped.) This is not considered a problem, as programs which violate this assumption are already problematic under Linda (due to its temporally uncoupled nature[28]).

[28] See section 4.2.2. (page 19) and section 4.5.2. (page 22).

8.4.4. Direct Manipulation of Logs

Client processes are also permitted to perform operations which directly affect the contents of tuple logs, allowing a task to be fine-tuned so as to redo less work when it is restarted after being backed off. The tuple space server assumes that any client making use of these facilities will ensure that no operations are performed which could result in the contents of tuple space being affected in such a way that the application being executed does not complete correctly.

Clients can issue the following log-manipulation commands to tuple space servers:

- enable logging (logging is enabled by default),
- disable logging,
- purge a tuple log from a given tuple id. onwards, and
- purge a tuple log up to a given tuple id.

The potential uses of these operations by clients are described in section 7.8. (see page 53).

Chapter 9.

Example Programs and Benchmarks

During the development of Remora three main areas were investigated in order to assess the system and its performance, namely:

- speed of tuple space interaction,
- performance of example application programs, and
- measurements of the amount of idle time available on a networked cluster of workstations.

For the first two categories above, example programs were used for testing the system. In the last two areas, the monitoring facilities built into Remora were used in order to monitor the activity of nodes in a networked cluster.

During the process of benchmarking the system, some enhancements were added, mainly in the area of network transport. These enhancements have in some cases enhanced performance to a considerable degree. There is however still a significant amount of room for further optimizations to be made to Remora.

9.1. Tuple Space Interaction

The rate at which client processes can interact with tuple space has an important effect on the efficiency of client programs. The latencies of the networks over which messages are transmitted are considerable, and it is therefore desirable to minimize the delays which are incurred due to network communication, so as to maximize the rate at which tuples can be deposited in and retrieved from tuple space.

Two different benchmarks, which we will call "*InOut*" and "*PingPong*" [Thomas 1991] were used to measure the rate at which clients can exchange tuples with tuple space. These two simple programs interact with tuple space in ways which attempt to characterize different extremes of the normal tuple space interaction of Linda applications.

9.1.1. The *InOut* Benchmark

The program *InOut* first deposits a large number of tuples (generally 10000) into tuple space, and then retrieves them again. The wall clock times required to deposit the tuples and then retrieve them are measured. While this kind of behaviour is probably not typical of any real Linda program, it is a useful method of testing speed of communication.

The results obtained from the *InOut* benchmark are shown in figure 10.

9.1.2. The *PingPong* Benchmark

The *PingPong* program bounces a tuple back and forth between two client processes. (See listing 3.) While running this test, the boss, worker and tuple space were each placed on different computers, and the time taken to perform 10000 iterations was measured.

The results obtained from timing the *PingPong* benchmark were considerably slower than those obtained by [Thomas 1991]. This is probably a result of Remora using TCP and not UDP for network transport. It might thus be possible to improve the performance of the system by rewriting the transport interface to use UDP transport.

```
%%boss
boss()
  {
   int i;

   eval(worker);
   for (i=1; i<1000; i++)
     {
       out("ping");
       in("pong");
     }
  }

%%worker
worker()
  {
   int i;

   for(i=0; i<1000; i++)
     {
      in("ping");
      out("pong");
     }
  }
```

Listing 3: *PingPong* algorithm

9.1.3. Optimizations

With the aid of a network monitoring tool, the ethernet traffic generated by the *InOut* and *PingPong* benchmarks was examined. In an attempt to speed up interaction with tuple space, the transport level interface was modified in a number of different ways, and the results were observed by timing the benchmarks mentioned above for each different technique, and by monitoring network traffic. The following techniques were applied:

- a set of routines was written to perform buffering of messages in the Remora network transport interface,
- the socket options controlling TCP level buffering were modified, and
- an adaptive algorithm which selectively modifies TCP level buffering in Remora clients was developed.

It was found that the addition of buffering routines to the Remora network transport interface resulted in an excessive CPU overhead and that these routines were not as efficient as the buffering provided by the kernel level TCP routines (which is manipulated via the TCP_NODELAY socket option [Leffler *et al* 1990]). In certain cases it was found that disabling all forms of buffering produced an improvement in performance, as messages were then transmitted without any delay.

	InOut		*PingPong*
	Input	Output	
TCP buffering enabled (default)	29	10	500[29]
TCP buffering disabled	27	15	53
Buffering in transport interface	40	11	86
Adaptive manipulation of TCP buffering	28	10	53

Figure 10: Timings (seconds) of the *InOut* and *PingPong* benchmarks[30]

The adaptive algorithm for manipulating TCP level buffering toggles the TCP_NODELAY option in Remora client processes, depending on the characteristic of tuple space interaction. When a client performs long sequences of *out* operations it is more suitable to enable buffering. On the other hand, when the proportion of Linda input operations is high, buffering of messages is undesirable. This is due to the fact that Linda input operations involve a *query-response* sequence of messages being exchanged between the client and the server. If buffering is enabled, the query is not transmitted immediately, as a time-out period must first elapse before a packet is transmitted over the network. This causes the total elapsed time for an input operation to be increased. Thus disabling TCP level buffering introduces a performance improvement when the mix of Linda operations being performed by a client contains a high proportion of input

[29] Estimated (1000 iterations of *PingPong* were performed, and the resulting times mulipled by 10.)

[30] These results were measured using a Sun SparcServer 10-512 as the tuple space server, and running client processes on Sun SparcStation Classics, all running Solaris version 2.2. 10000 iterations of *InOut* and 10000 iterations of *PingPong* were performed. Tests were performed on an unloaded network.

operations. The performance of the different buffering techniques used can be seen in figure 10.

A significant difference in performance of the *InOut* and *PingPong* programs, when executed under the different architectures and operating systems on which Remora has been implemented, was also noted. These differences are shown in figure 11. (These tests were done using the adaptive algorithm for manipulating TCP buffering.)

	InOut		*PingPong*
	Input	Output	
Sun Classic (Solaris 2.2)	28	10	53
Sun IPC (SunOS 4.1.1)	22	5	41
i486 PC[31] (FreeBSD 1.0)	18	4	33

Figure 11: *PingPong* and *InOut* timings (seconds) on different architectures

9.2. Example Application Programs

Three application programs were used in the testing of Remora. Each program shows different characteristics in its interaction with tuple space, and the amount of communication which is performed.

9.2.1. Queens

A problem often used as an example Linda program asks "how many ways can eight queens be placed on a chess board in such a way that no one queen can take any other queen?" In the tests done on Remora, we have "scaled up" the problem and, instead of using an 8×8 chessboard, we use a "chessboard" with $n \times n$ squares, and find all the different ways in which n queens can be placed on this board. This problem is typical of the "*master-worker*" class of application, in which a "*master*" decomposes the problem into smaller subtasks, which are given to "*workers*" to perform.

The algorithm used attempts to place one queen at a time, starting with an empty board and placing a queen in the first row, and then progressively placing a queen in a "safe" position in each successive row of the board. When attempting to place a queen in a given row, a number of potentially safe positions may be found. Each of these needs to

31 For this test a 486DX33 based PC with a 16bit SMC ethernet card was used.

be considered as part of a possible solution, resulting in a search tree which rapidly explodes in size as more queens are placed on the board.

The master starts by placing the first few queens on the board, and then divides the rest of the problem amongst a group of workers by placing partial solutions into tuple space as tasks. The workers then explore the section of the solution space associated with each partial solution, and return completed solutions to tuple space. The master finally retrieves all of the completed solutions from tuple space. Source code for an implementation of this algorithm can be found in Appendix B.

The grain of the computation can be varied by changing the number of rows of the chessboard which are filled in by the master before it starts placing tasks into tuple space. As the number of rows filled in by the master is increased, the number of tasks placed into tuple space increases, and the grain of the computation becomes finer.

In the tests performed with the *Queens* program two main aspects of Remora were examined, namely:

- to investigate how well the performance of the program scales, as more workers are added to a homogeneous cluster[32], where Remora's facilities for adaptive parallelism are not enabled, and
- to find out what effects Remora's facilities for adaptive parallelism have on the program, when running on a heterogeneous cluster of workstations. (This allows us to quantify the overheads introduced by adaptive parallelism.)

Scalability on a Homogeneous Cluster

In order to investigate system performance as the number of worker nodes is scaled up a cluster of i486 based PC's, with a Sun SparcServer 10 acting as tuple space server, was used.

Since the *Queens* problem is an ideal example of the master-worker type of parallel program, it scales well as the number of workers is increased. However, as the number of workers is increased, the amount communication with tuple space increases, and the network connecting the workstations becomes increasingly saturated, limiting any further speedup. The rate at which the master can retrieve completed solutions from tuple space

[32] Tests of scalability need to be performed in a homogeneous environment, since the widely differing processing power and networking performance of different workstations in a heterogeneous cluster would make results meaningless.

also tends to limit the speedup which can be achieved.[33] The speedup obtained when increasing the number of worker nodes used for placing queens on a 14×14 chessboard is shown in figure 12.

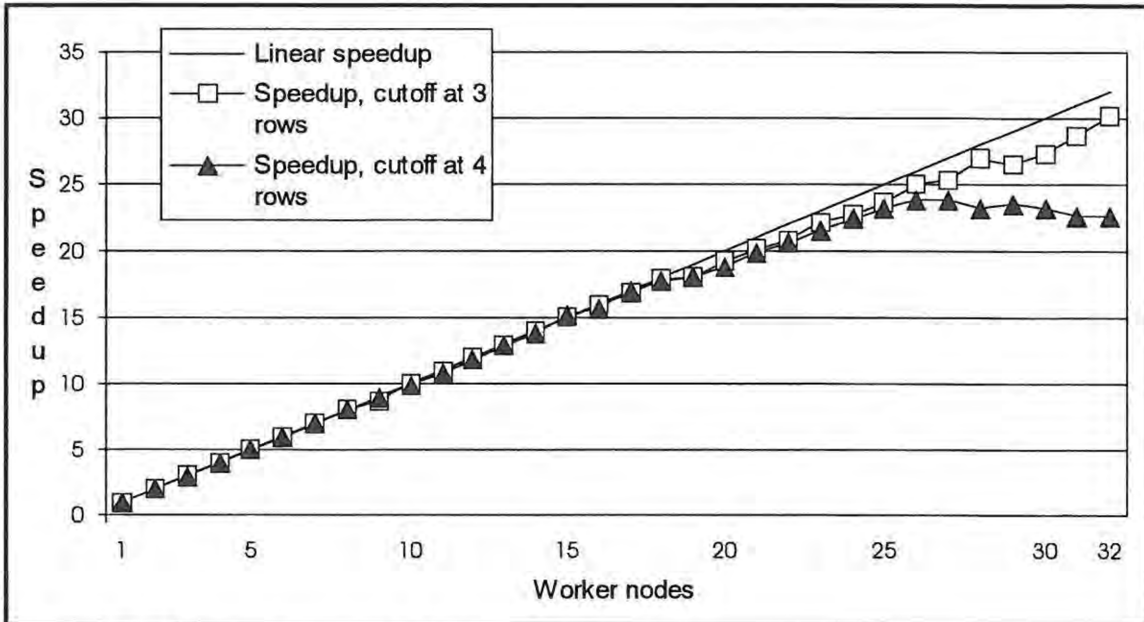

Figure 12: Speedup obtained placing 14 queens

In order to observe the effect of the grain of the computation on system performance, the test was run, first with the master computing three rows of the board before shedding tasks, and then with the master computing four rows. Under finer grained conditions, the amount of network traffic resulting from tuple space interaction eventually saturated the network, making any more speedup impossible. Using 29 worker nodes, the number of collisions observed on the tuple space server's ethernet interface increased threefold when the cutoff at which the master began shedding tasks was changed from three rows of the board to four.

Overheads Introduced by Adaptive Parallelism

It is reasonable to expect that an application run in an adaptive parallel environment would not perform as well as when it is run using the same hardware, but in an

[33] It is possible for a situation to arise, in which the master is still busy retrieving solutions from tuple space after all of the tasks have been completed by the workers. This problem was partially addressed by modifying the *Queens* program to batch large numbers of solutions into a single tuple, thus reducing the number of tuples (and thus packets) which must be transmitted over the network.

environment where that hardware is used only for parallel computation. In order to obtain an indication of how much slower an application might run in an adaptive environment, comparisons were made of the performance of the *Queens* program in adaptive and dedicated execution environments.

For these tests a heterogeneous cluster of ten workstations[34] was used. Tests were first performed during off-peak hours, using the cluster as a dedicated parallel computing environment, and then during office hours using the cluster as an adaptive computing environment. In these tests, 16 queens were placed (using a 16×16 "chessboard"), with the master placing the first four queens on the board before shedding tasks into tuple space. A total of 19688 tasks were *evalled* in this computation.

During these tests the total runtime (wall clock time) of the test as well as the amount of time for which each workstation was available for computation during the run were measured. While the total runtime in the adaptive environment was approximately 20% longer (this can vary significantly depending on other system utilization), the aggregate time (the sum of the times for which each of the nodes were actively processing tasks) was very similar in both the adaptive and dedicated environments.[35] A comparison of execution times in the dedicated and adaptive environments is shown in figure 13.

	Workers	Runtime (hours)	Aggregate Time (hours)
Adaptive Environment	10	1.68	14.19
Dedicated Environment	10	1.47	14.73

Figure 13: Performance of the *Queens* program in an adaptive environment

The slightly lower aggregate time observed in the adaptive environment can probably be attributed to the fact that when there was any other load on a workstation, the client process running on it was stopped, whereas in the dedicated environment it was not always possible to ensure that workstations were totally unloaded. (Programs such as regular daily runs, which are run in the early morning have an effect on the performance of the program when running in the dedicated environment.)

34 The cluster used was composed as follows: 1 Sun SparcServer 10-512 (tuple space server, and master node), 2 SparcStation IPC's, 2 Sparcstation ELC's, 1 SparcStation IPX, 2 SparcStation Classics and 2 i486DX33 based PC clones. During adaptive tests, only the SparcServer was used as a reliable node. All nodes were used to run worker processes.

35 Data given in [Carriero *et al* 1993] shows similar trends in experiments with Piranha.

9.2.2. Finding Prime Numbers

A problem which is discussed in many programming texts is that of finding prime numbers. This is also a problem which attracts interest in areas such as cryptography. Since finding large prime numbers is a very time-consuming task, it is not surprising that parallel algorithms have been developed to do this.

A parallel algorithm for finding prime numbers, based on the "Sieve of Eratosthenes", and written as a Linda program is presented in [Carriero and Gelernter 1990, p. 93]. This algorithm also makes use of the master-worker model of parallelism, in which workers are given lists of numbers, which they scan for prime numbers. Source code for an implementation of this algorithm can be found in Appendix C.

9.2.3. Optimal Matrix Multiplication Order

A third program used as a test for Remora is one which calculates the optimal order in which a series of matrices should be multiplied, so that the number of multiplication operations is minimized [Wilson and George 1991].[36]

This algorithm is designed in such a way that it makes very heavy demands on the communication facilities available in the environment in which it is being tested. Thus, as a result of the high latencies (relative to specialized parallel architectures) involved in transmitting packets between workstations on an ethernet, this program does not exhibit good performance when run in the Remora environment (see figure 14), and is effectively communication-bound.

Workers	10 Matrices	15 Matrices
1	4.01	16.29
2	3.0	10.79
3	4.07	11.31
4	4.41	14.30

Figure 14: Times (seconds) taken to calculate Optimal Matrix Multiplication Order

The algorithm given in [Wilson and George 1991] requires that a fixed number of processes be concurrently active during the execution of the program. This unfortunately

36 Source code can be found in Appendix D.

makes it unsuited to being run in an environment where the number of available processors (and hence processes) can vary during the execution of a program. In an adaptive parallel environment, this could mean that a deadlock might result if one task is backed off when a node becomes unavailable.

9.3. Idle Time available on a Networked Cluster

An important consideration when attempting to make use of idle time on a network of workstations is that there should be sufficient idle time for such an approach to be viable. With this in mind measurements were taken, in order to ascertain what amount of time is actually available on a cluster of workstations.[37] A breakdown of the idle time measured over a week is shown in figure 15.

Workstation	Workstation description	Idle time	% Idle time
alpha	Sun IPX (SunOS 4.1.1)	4 days, 15:23, 36s	66.30%
beta	Sun ELC (SunOS 4.1.1)	6 days, 8:49, 11s	90.96%
gamma	Sun IPC (SunOS 4.1.1)	6 days, 2:29, 27s	87.19%
delta	Sun ELC (Solaris 2.2)	6 days, 17:59, 9s	96.41%
epsilon	Sun SparcClassic (Solaris 2.2)	6 days, 3:33, 55s	87.83%
zeta	Sun SparcClassic (Solaris 2.2)	6 days, 20:42, 49s	98.04%
clayton	Sun IPC (SunOS 4.1.1)	6 days, 12:22, 13s	93.07%
braae	i486 PC (FreeBSD 1.1.5.1)	6 days, 16:02, 10s	95.25%
csmsc1	i486 PC (FreeBSD 2.0.0)	6 days, 4:28, 45s	88.37%
aggregate		56 days, 5:51, 15s	89.27%

Figure 15: Idle time, measured in a week, on a cluster of workstations

The criteria used to determine when workstations could be regarded as idle were adjusted according to the performance of each workstation. Workstations were determined to be idle if there had been no keyboard or mouse activity for five minutes, and the system load was below certain limits. During working hours (between 8am and 5pm) idle time was found to be approximately 80%, while after hours the amount of idle time was generally over 90%.[38]

[37] The cluster used for these measurements consisted of 2 Sun SparcStation IPC's, 2 SparcStation ELC's, 1 SparcStation IPX, 2 SparcStation Classics and 2 i486 based PC's. A SparcServer 10-512 (which was not included in the measurements) was used as the master node for the cluster. The workstations used in these measurements are most heavily used between 8am and midnight, with users occasionally running jobs overnight.

[38] A typical Remora configuration file for an unreliable node is shown in listing 5 on page 82.

Chapter 10.

Future Work

One of the goals of this project has been to provide a framework upon which further work can be based. An effort has been made to develop a system which is modular in design, so that existing parts can be rewritten, or new components added. Several areas of the system have only been dealt with simplistically, and can still be improved.

Some extensions to the system have already been commenced as graduate projects at Rhodes University. Work is currently underway in the area of porting Remora to other architectures and in developing a better preprocessor for Remora application programs.

10.1. Porting Remora to Other Architectures

The current implementation of Remora concentrates on 32 bit architectures running either BSD or SYSV flavours of UNIX. While a fair degree of architectural and operating system heterogeneity is handled in this environment, the systems which have been catered for so far share many common features.

It would be desirable to cater for systems, including specialized parallel architectures, which differ significantly from those currently supported. This would involve catering for the different communication facilities (such as transputer links) as well as other architectural features which may be encountered. Currently, a port of Remora to a cluster of T800 transputers (running Helios) is underway.[39] This will allow the integration of the transputer cluster into a larger networked cluster of computers which would all be used to perform Remora computations.

Apart from being based on typical UNIX workstations, the current implementation of Remora is oriented towards computers with a 32 bit wordsize. Computers with a larger wordsize (typically 64 bits) are becoming more common, and should be catered for

[39] The Helios port of Remora is being undertaken by Simon Barratt as a graduate project at Rhodes University.

directly via the addition of 64 bit numeric data types to the set of basic data types currently supported by Remora.

10.2. A Better Preprocessor for Application Programs

Remora application programs which are written in *ideal syntax* must first be passed through a preprocessor which converts all Linda operations into library calls to the Remora runtime library. While the present preprocessor performs tasks like tuple space partitioning, it performs very little optimization of the Linda operations contained in a program. A new preprocessor might provide more information to the run-time environment in order to determine the type of storage mechanism which might be most suitable for each tuple space partition. Possibly estimates of the relative numbers of references made to each tuple space partition could be provided by the preprocessor in order to allow, for instance, for heavily used partitions to be placed on separate tuple space servers.

The current preprocessor provides support for Remora programs where Linda operations are embedded in the C programming language. Preprocessors, which provide for embedding Linda primitives in a different programming language, might be developed. A C++ based preprocessor for Remora is presently being developed.[40]

10.3. Network Transport

At present, only a simple TCP stream based network transport layer is available in the system. While a TCP based transport is easy to implement, indications are that it may not provide the best performance for a cluster of computers attached to a LAN. Performance figures from [Thomas 1991] indicate that better performance may be achieved if communication across a LAN were performed via the UDP datagram protocol.

Other transport facilities might also be used when these are available in other architectures, especially on multiprocessor systems. Transport mechanisms which might be used include shared memory (which could be used on a multiprocessor SparcServer) or transputer links.

---

[40] The C++ preprocessor for Remora is being developed by Gwynneth Graham, as a graduate project at Rhodes University.

10.4. Workload and Tuple Space Distribution

At present Remora does not attempt to distribute tuple space partitions and tasks optimally between the different nodes participating in a computation. This is an area of work which may yield many improvements to the current system. Efficient tuple space distribution is especially important in environments where clusters of computers on different networks are used to perform a computation together. Since the cost of communication between computers in different clusters is likely to be high, it would be advantageous to distribute tuple space in such a way that communication is mainly localized within the individual clusters.

10.4.1. A More Balanced Distribution of Tuple Space Partitions

The coordinator, whose task it is to place tuple space on different servers, presently does so by placing partitions with different servers on a round-robin basis, as partitions are opened for the first time. This ignores a program's tuple space usage pattern, and makes no attempt to distribute the load of performing tuple space service in a balanced manner amongst the available servers. It might be possible to use both historical data and information about tuple space usage, extracted during the preprocessing stage, in order to distribute tuple space partitions more effectively.

10.4.2. Functional Clustering

When using groups of computers on different networks (possibly on LANs interconnected by long-haul networks), it would be desirable to partition tuple space in such a way that functionally related data, and the tasks which access that data, are placed on one network, so that the communication costs involved in accessing tuple space are minimized. Effectively this means that parts of a computation would be distributed onto different groups of computers, and the communication between these groups (which might be connected via slow networks) would be minimized. This would probably also require that active tuples be placed into separate partitions. (At present all active tuples are placed in one tuple space partition.)

10.4.3. Placement of Tasks on Suitable Computers

Since Remora operates in a heterogeneous environment, it would be desirable to distribute active tuples in such a way that special features of individual computers may be used more effectively where possible. This might allow, for instance, for tasks which

require floating point arithmetic to be preferentially placed on computers which have powerful floating point units. Similarly it might be possible to place certain tasks on a vector processor, if one is available. In order to make such preferential placement of tasks possible, it may be necessary for the programmer to provide hints, which indicate the nature of a task, to the system.

10.5. Tuple Space Storage

The *storage policy* mechanism used by Remora is designed so that different types of data structures can be used for the storage of tuples on a tuple space server. The current implementation only makes use of a linear linked list for storage of tuples. This might result in inefficient access to tuple space when tuples are not retrieved in the same order in which they were deposited. Examples of other types of data structure which may be used for storage of tuples include hash tables and even simple semaphores (which might be used in cases where only the number of stored tuples in a partition need be recorded). With the use of information collected at the preprocessing stage, it would be possible to choose data structures which suit both the type of data stored in a partition and the access patterns of that partition better.

10.6. Fault Tolerance

While Remora draws on methods used for implementing fault tolerance under the Linda paradigm, the present implementation does not in any way attempt to implement fault tolerance. The facilities, such as tuple logging, which are already in place, could be extended using techniques such as those discussed in [Kambhatla 1990] to make it possible for failures of unreliable nodes to be tolerated. Dealing with failures of nodes which are active as tuple space servers will require considerably more work, as this would require replication of the data in tuple space.

10.7. Higher Level Programming Tools

A common problem encountered by parallel programmers is that the writing of parallel programs is often more complex than the writing of sequential programs. In order to address this problem, various tools have been developed which assist the programmer in developing parallel programs. Such aids could be used to assist the programmer in writing programs which perform better in the adaptive parallel environment which is provided by Remora, for instance by making use of the primitives which Remora

provides for direct manipulation of tuple logs. Two main approaches in this area are *program builders* and *algorithmic skeletons.*

10.7.1. Algorithmic Skeletons

Algorithmic skeletons [Cole 1989, Watkins 1992] provide a framework for writing parallel programs, in which the components which deal with parallelism (especially communication and coordination) have been presupplied. When developing a program, the programmer selects a skeleton which provides an appropriate model of coordination for his application, and proceeds to "fill in" only the higher level details of the program. All details of the lower aspects of coordination are hidden from the programmer's view.

10.7.2. The Linda Program Builder

The Linda Program Builder (LPB) [Ahmed and Gelernter 1991, Ahmed *et al* 1993] is a graphical environment which assists the programmer in writing Linda programs by providing templates for different classes of application. In order to write a program, the programmer selects a suitable template, and then adds the higher-level parts which are required in order to build the required application, while issues such as communication and coordination are dealt with by the template. The program builder steers the programmer towards concentrating on the higher level issues of the program. Unlike the algorithmic skeleton approach, the LPB provides a *transparent* interface to program development, and allows the programmer to modify the lower-level aspects of the template if necessary.

10.8. Behavioural Model Debugging

Debugging of parallel programs tends to be problematic, to a large degree due to the "probe effect" which is introduced by debugging tools. A technique for debugging Linda programs with the use of behavioural models is presented in [Sewry 1994]. This technique effectively does away with the problems resulting from the "probe effect" by making use of the decoupling and non-determinism inherent in Linda. In order to debug a Linda program, the debugger compares the actual run-time behaviour of a Linda program with a formal model of its tuple space interaction, and alerts the programmer to any deviations from this model. A project, currently underway at Rhodes University, aims to investigate the implementation of this debugging technique for Remora.

Chapter 11.

Conclusion

An environment has been developed, which provides facilities to perform parallel computations making use of the existing networks and computers available within an organization. This makes it possible for institutions, whose budgets would not allow for the purchase of specialized parallel computing hardware to perform many demanding computations, which are normally only attempted using expensive high-performance supercomputers or dedicated parallel architectures. This has special significance in third world countries, such as South Africa, where resources for specialized computing hardware are generally not available.

Support for architectural heterogeneity is another feature which makes a system like Remora attractive in situations where limited financial resources preclude the purchase of specialized hardware for parallel computation. The ability to use existing heterogeneous resources makes it possible to harness a far larger pool of processing power than might otherwise be possible, given the heterogeneous nature of most LAN-based clusters of computers.

It was found that adaptive parallelism can be implemented efficiently. Despite the fact that applications which are executed in an adaptive environment will take longer to run than in a dedicated environment, the total amount of time for which workstations are actively computing does not differ significantly. Best results were achieved with algorithms in which one or more master processes distribute tasks into tuple space, and then collect and collate the results. Such problems are well suited to the task-based (rather than process-based) model of parallelism which has been adopted in order to implement adaptive parallelism. This does not mean, however, that other problems which do not immediately fit this model are excluded from this environment. Many algorithms can be made more efficient by transforming them into a master-worker format. [Carriero and Gelernter 1990]

The amount of "idle time" which is available on the workstations on a LAN was found to be significant. This is borne out by the fact that programs run in the adaptive

environment only executed about 20% slower than in a dedicated environment. Part of the success of Remora in using spare CPU time nonintrusively is exemplified in the fact that other users of the workstations on which final testing was done were unaware of the fact that these computers had been used intensively for performing parallel computations for a period of roughly two weeks.

The independence of Remora (and Linda in general) from any particular hardware platform makes it easier for portable parallel programs to be developed. This is not the case in many parallel computing environments, which tend to be oriented towards making hardware features of parallel environments directly visible to the programmer. The development of portable parallel programming environments is an important step towards greater acceptance (especially outside academia) of parallel computing. The development of parallel computing environments which are independent of any particular parallel architecture could possibly be compared to the development of high-level languages for sequential computing - they allow the same program to be run on different computers, without modification.

A secondary goal of the development of Remora has been to provide a framework, which can serve as a base for further work on the Linda model of parallelism. With this in mind, a layered, modular approach was taken during Remora's development. The system consists of a set of "building blocks", which interact via a set of well defined interfaces. This makes it possible for components of the system to be easily rewritten (possibly for porting to different operating systems and architectures), enhanced, or even totally redesigned. Currently three projects, which are building on the base provided by this project, are underway at Rhodes University.

Appendix A:

Remora System Configuration

A cluster of workstations on which Remora is running is configured via a configuration file, which must be present on each node. The configuration file contains information which is specific to the local node as well as information about the network ports used to contact the coordinator. Two example configuration files are given here, the first one for a dual processor Sun SparcServer 10-512 (see listing 4), which is being used as a tuple space server and as the master node for a cluster of computers, and the second for a Sun SparcClassic workstation (see listing 5), which is being used as an unreliable node. In order to make better use of both processors on the SparcServer, two client processes are started on it.

```
# Remora configuration
# G.M. Rehmet, Rhodes University, 1994
# $Id: omega.cf,v 1.1 1994/08/10 09:45:22 g89r4222 Exp $

nodetype:reliable;                  # node may not back off
filesystem : omega2;

# base directory to search for application programs
workdir : "/home/g89r4222/als";

# Type of host (operating system/architecture)
hosttype : solaris2_2;

# Details of network ports, and master node
master : "omega.ru.ac.za";
masterport : 3523;                  # daemon <-> coordinator
clientport: 11000;                  # client <-> coordinator
serverport: 13000;                  # client <-> TS server

# Timeout for controlling the server (seconds)
timeout: 120;

# Additional Remora components to run on this node
# (NB: only reliable nodes may run a tuple space server)
server;                             # Tuple space server
clients: 2;                         # Run 2 client processes
```

Listing 4: Reliable node configuration

```
# Remora configuration
# G.M. Rehmet, Rhodes University, 1994
# $Id: unrel.cf,v 1.2 1994/08/10 09:45:24 g89r4222 Exp $

nodetype:       unreliable;              # may back off
filesystem:     omega2;

# base directory to search for application programs
workdir:        "/home/g89r4222/als";

# Type of host (operating system/architecture)
hosttype:       solaris2_2;

# Details of network ports, and master node
master:         "omega.ru.ac.za";        # name of master
masterport:     3523;                    # daemon <-> coordinator
clientport:     11000;                   # client <-> coordinator
serverport:     13000;                   # client <-> TS server

# Load monitoring
timeout:        30;                      # timeout between load samples
loadlimit:      low 0.85, high 1.05;     # low & high load thresholds
loadmetric:     1;                       # monitor 1-minute load

# Monitor the following devices for activity
# In this case, we are checking for console activity
devmonitor:     timeout 5 name "/dev/console";
devmonitor:     timeout 5 name "/devices/pseudo/conskbd:kbd";
devmonitor:     timeout 5 name "/devices/pseudo/consms:mouse";

userlimit:      low 4, high 7;           # limit on number of users
idletime:       3;                       # users are idle after 3
                                         # minutes of inactivity
activeusers;                             # only count active users

# Specify nice value for running clients
# Client processes on unreliable nodes are run at a lower priority
clientprio:     10;
```

Listing 5: Unreliable node configuration

Appendix B:

The Queens Problem

Listing 6 (below) contains the source code of the *Queens* program, described in Chapter 9 (see page 68).

```
/*
 * queens2.r
 *
 * Place n Queens on a nxn "chessboard", in such a way,
 * that no queen can take any other queen.
 *
 * Original version by          E.P. Wentworth, 1990
 * Optimized by                 G.L. Smith, 1992
 * Rewritten in ideal syntax by G.M. Rehmet, 1993, 1994
 *
 * $Id: queens2.r,v 4.4 1994/08/10 20:55:29 g89r4222 Exp $
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef unsigned char byte;

#define boardMax        300             /* max rows / columns */
#define BBUFMAX         300             /* max boards in buffer */


static time_t  endTime, startTime;
static int boardSize, bossLimit;
static byte board[boardMax];            /* the board */
static int curbuf = 0;                  /* no of solutions in buffer */
static byte boardbuf[boardMax*BBUFMAX]; /* buffer for solutions */
static int evals= 0;                    /* no. of tasks evalled */
```

continued ...

```
/*
 * Print out a completed solution
 * (Don't print every solution, otherwise our tty becomes the
 * bottleneck!)
 */
void printboard(int count, byte board[])
{
                int     i;
        static  int     gap = 1;

        if (count % gap == 0) {
                if (gap < 150) gap *= 2;
                printf("%7d: ",count);
                endTime = time(NULL);
                printf(" Time = %6usecs, ", (endTime-startTime));
                for (i = 0; i < boardSize; i++)
                        printf("%2i ",board[i]);
                printf("\r\n"); fflush(stdout);
        }
}



/*
 * Check if a position is safe
 *
 * Is q safe with respect to column col of the board?  Q is dist away
 * from the place where q is about to be placed.  If it is safe,
 * try column col-1.
 */
int issafe(int q, int col, int dist)
{
        int     bc;

        if (col < 0) return(1);
        bc = board[col];
        if ((q == bc) ||                        /* on same row    */
                        (q-dist == bc) ||       /* on NW diagonal */
                        (q+dist == board[col])) /* on SW diagonal */
                return(0);
        else
                return(issafe(q,col-1,dist+1));
}



/*
 * BOSS
 *
 * Calculate the first few rows of the board, and then place
 * tasks into tuple space
 *
 */
%% boss
```

continued ...

```
/*
 * Place the first few queens
 */
int placeQueens(int num_placed)
{
	int	q, pos, bsize;

	for (q=0; q < boardSize; q++)
		if (issafe(q,num_placed-1,1)) {
			pos = num_placed;
			board[pos++] = q;
			if (pos < bossLimit)
				/* boss calculates another row */
				placeQueens(pos);
			else {
				/* shed work into tuple space */
				bsize = pos*sizeof(byte);
				out("partials", boardSize, board:bsize);
				eval(worker);
				evals++;
			}
		}
}

/*
 * Main entry point
 */
int boss(int argc, char * argv[])
{
	int	q, count, i;
	int	jobdone;		/* has worker completed task? */
	int	tasks_completed=0;	/* no. of tasks completed */

	printf("Boardsize? "); fflush(stdout);
	scanf(" %d",&boardSize);
	printf("At what depth should the boss spawn tasks");
	printf("( up to %d should be okay)?", boardSize/3);
	fflush(stdout);
	scanf(" %d",&bossLimit);
	startTime = time(NULL);
	placeQueens(0);

	/* retrieve solutions from tuple space */
	count = 0;
	do {
		q = sizeof(boardbuf);
		in("completed", ?jobdone, ?boardbuf:q);
		if(jobdone)
			tasks_completed++;
		for(i=0; i< q/(boardSize*sizeof(byte));i++)
			printboard(++count,&boardbuf[i*boardSize]);
	} while (tasks_completed != evals);

	endTime = time(NULL);
	printf("\nTotal number of solutions = %d ", count);
	printf("\nTotal time = %6usecs\r\n", (endTime-startTime));
	printf("Evals performed: %d\n", evals);
	return(0);
}
```

continued ..

```
/*
 * WORKER
 *
 * Pick up a partially completed chessboard from tuple space, and
 * continue working out all possible completed boards, given the
 * partially completed board, supplied by the boss.
 */
%% worker

/*
 * Continue placing queens on a partially filled board
 */
void placeQueensW(int num_placed)
{
        int     q, pos;
        int     bsize;

        for (q=0; q < boardSize; q++)
                if (issafe(q,num_placed-1,1)) {
                        pos = num_placed;
                        board[pos++] = q;
                        if (pos < boardSize)
                                placeQueensW(pos);
                        else {
                                /* Is solution buffer full? */
                                if (curbuf == BBUFMAX) {
                                        bsize = pos*sizeof(byte)*curbuf;
                                        out("completed", 0,
                                                boardbuf:bsize);
                                        curbuf = 0;
                                }

                                /* put solution in buffer */
                                memcpy(&boardbuf[curbuf++*boardSize],
                                        board, boardSize*sizeof(byte));
                        }
                }
}

/*
 * Pick up a partially completed board, and complete it
 */
int worker(void)
{
        int num_placed, j;

        curbuf = 0;
        j = sizeof(board);
        in("partials", ?boardSize, ?board:j);

        num_placed = j / sizeof(byte);
        placeQueensW(num_placed);

        /* There will always be something left in the buffer */
        out("completed", 1, boardbuf:boardSize*sizeof(byte)*curbuf);
        return(0);
}
```

Listing 6: The Queens Problem

Appendix C:

The Sieve of Eratosthenes

In chapter 9 (see page 72) a program used to find prime numbers is discussed. The source code of this program is shown in listing 7. This particular version of the algorithm requires that the table of primes be seeded with a few prime numbers, which are used in order to calculate further prime numbers. (These values could be calculated instead of having them hard-coded into the program.)

The algorithm in [Carriero and Gelernter 1990] has been modified to do two iterations of searching for prime numbers. After the first iteration, all the prime numbers which have been found are collected, and used to reseed the table of prime numbers for the second iteration of the algorithm.

```
/*
 * primes4.r
 *
 * G.M. Rehmet, 1994
 *
 * Based on algorithm given in
 *      "How to Write Parallel Programs, A First Course",
 *      N. Carriero and D. Gelernter, MIT Press,
 *      Cambridge Massachusetts, 1990
 *
 * $Id: primes4.r,v 4.4 1994/08/11 12:32:02 g89r4222 Exp $
 */

#include <stdio.h>
#include <stdlib.h>

#define PTABMAX         10000000        /* Size of primes table */
#define INITPRIMES      26              /* Initial no. of primes */
#define BLOCK           10000
#define GRAIN           1000

unsigned        * primetab = NULL;      /* Initial primes table */
unsigned        results[PTABMAX/10];    /* Results of sieving */
```

continued ...

```
/*
 * Seed the Primes table with prime numbers to around 100
 */
void initprimes(void)
{
        primetab = malloc(PTABMAX*sizeof(int));
        primetab[ 0] =   2;     primetab[ 1] =   3;
        primetab[ 2] =   5;     primetab[ 3] =   7;
        primetab[ 4] =  11;     primetab[ 5] =  13;
        primetab[ 6] =  17;     primetab[ 7] =  19;
        primetab[ 8] =  23;     primetab[ 9] =  29;
        primetab[10] =  31;     primetab[11] =  37;
        primetab[12] =  41;     primetab[13] =  43;
        primetab[14] =  47;     primetab[15] =  53;
        primetab[16] =  59;     primetab[17] =  61;
        primetab[18] =  67;     primetab[19] =  71;
        primetab[20] =  73;     primetab[21] =  79;
        primetab[22] =  83;     primetab[23] =  89;
        primetab[24] =  97;     primetab[25] = 101;
}


/*
 * Worker: sieves a range of numbers for primes
 */
%% worker

/*
 * Do sieving
 */
int sieve_and_collect(int start, int end, int n_primes)
{
                int             i, j, k, pos;
        static  unsigned        sieve[PTABMAX/10];

        /* First prepare the sieve */
        for(i = 0 ; i < end - start+1; i++)
                sieve[i] = 1;

        /* Now sieve against all primes */
        for(i = 0; i< n_primes; i++) {
                j = start % primetab[i];
                if(j)
                        j = primetab[i]-j;
                for(; j < end - start+1; j += primetab[i])
                        sieve[j] = 0;
        }

        /* Now collect the remaining primes */
        for(i = j = 0; i < end - start+1; i++)
                if(sieve[i])
                        results[j++] = start+i;
        return(j);
}
```

continued ...

```
/*
 * Worker:
 * Get the primes db out of tuple space,
 * then sieve a block of numbers
 */
void worker(void)
{
        int     blk_end, n_primes;
        int     t_start, t_end;
        int     n_soln;

        if(primetab == NULL)
                primetab = malloc(100000*sizeof(unsigned));

        /* Get the primes db */
        rd("primetab",?blk_end ,?n_primes , ?primetab);

        /* Get the task to be performed */
        in("task", ?t_start, ?t_end);

        n_soln = sieve_and_collect(t_start, t_end, n_primes);

        /* Put results into TS */
        out("results", n_soln, results:n_soln*sizeof(int));
}


/*
 * Boss: places tasks into TS, and collects returned prime numbers
 */
%%boss
int boss(int argc, char * argv[])
{
        int     max_prime, blk_end, grain, tasks;
        int     n_result, n_primes, limit, sz;
        int     i, j;
        int     iter;
        int     p, f;

        initprimes();
        max_prime = primetab[INITPRIMES-1];
        n_primes = INITPRIMES;
        blk_end = BLOCK;
        grain = GRAIN;

        limit = max_prime*max_prime;
        for(iter = 0; iter < 2; iter++) {
                /* Place the primes table in TS */
                sz = n_primes*sizeof(int);
                out("primetab", limit, n_primes, primetab:sz);

                /* Place work into TS */
                for(i = max_prime+1, tasks = 0; i < limit-grain;
                                i+= grain, tasks++) {
                        out("task", i, i+grain-1);
                        eval(worker);
                }
                printf("tasks %d\n", tasks);
```

continued ...

```
                /* Now collect the results */
                for(i = 0; i < tasks; i++) {
                        in("results", ?n_result, ?results);
                        for(j = 0; j < n_result; j++) {
                                if(results[j] > max_prime)
                                        max_prime = results[j];
                                primetab[n_primes++] = results[j];
                        }
                }

                /*
                 * All workers have returned results -
                 * remove the primes db - it will be
                 * replaced on the next iteration
                 */
                in("primetab", ?i, ?j, ?results);

                /* Set new blocksize and grain */
                limit = max_prime * max_prime;
                grain= grain * grain;
        }

        /* Now print the results */
#ifdef PRINT_ALL_PRIMES
        /* This creates a lot of output! */
        for(i = 0; i < n_primes;)
                printf("%5d %s", primetab[i], (++i%10)?"":"\n");
        printf("\n");
#endif

        printf("Largest prime found %d\n", max_prime);
        printf("Count %d\n", n_primes);

        return(1);
}
```

Listing 7: The Sieve of Eratosthenes

Appendix D:

Optimal Matrix Multiplication Order

The source code given in listing 8 (below) is a Linda implementation of the algorithm described in [Wilson and George 1991]. Only subtle modifications were made to the original code, written by Greg Wilson, in order to run the program under Remora. These changes have been marked in bold italic type in listing 8. This program is discussed in Chapter 9 (see page 72).

```
/*=============================================================*/
/* File: parallel.r             From File: parallel.cl         */
/* Greg Wilson, 25/11/91.                                      */
/* Modified for Remora, Geoff Rehmet, 1993                     */
/*                                                             */
/* Does OMMO calculations in parallel using Linda.             */
/*=============================================================*/

#include <stdio.h>

/*-------------------------------------------------------------*/
/* definitions                                                 */
/*-------------------------------------------------------------*/

#define TRUE 1
#define FALSE 0
#define NERR 0
#define ERR 1
#define MAXINT 0x7FFFFFFF

/*-------------------------------------------------------------*/
/* function types                                              */
/*-------------------------------------------------------------*/

int calc();
int mine();

/*-------------------------------------------------------------*/
/* implementation                                              */
/*-------------------------------------------------------------*/
```

continued ...

```
/*
 @ real_main() : where Linda evaluation starts
 */
%%boss
boss(int argc, char * argv[])
{
    int argd = 2;                    /* argument index */
    int num_calc = 0;                /* number of calculators */
    int num_mat = 0;                 /* number of matrices */
    int * size = NULL;               /* matrix sizes */
    int width;                       /* synchronisation loop control */
    int i, j;                        /* generic index */
    int zero=0;                      /* to fool rpp          */

    /* get (and check) parameters */
    if (sscanf(argv[1], "%d", &num_calc) != 1){
        fprintf(stderr, "parallel: bad num_calc %d\n", num_calc);
        exit(ERR);
    }
    num_mat = (argc - 2) - 1;
    if (num_mat < 1){
        fprintf(stderr, "parallel: bad num_mat %d\n", num_mat);
        exit(ERR);
    }
    if ((size = (int *)calloc(num_mat+1, sizeof(int))) == NULL){
        fprintf(stderr, "parallel: alloc(size) failed\n");
        exit(ERR);
    }
    i = 0;
    while (argd < argc){
        if (sscanf(argv[argd], "%d", &(size[i])) != 1){
            fprintf(stderr,
                    "parallel: failed to get size[%d] from <%s>\n",
                    i, argv[argd]);
            exit(ERR);
        }
        i += 1;
        argd += 1;
    }

    /* put sizes and initial cost values in TS, and start calculators */
    out("size", size:(num_mat+1)*sizeof(int));
    for (i=0; i<num_mat; i+=1){
        out("cost", i, i, 0);
        for (j=i+1; j<num_mat; j+=1){
            out("cost", i, j, MAXINT);
        }
    }

    for (i=0; i<num_calc; i+=1)
      {
      /* eval("calc", calc(num_mat, num_calc, i)); */
      eval(calc);
      out("calc_in", num_mat, num_calc, i);
      }
```

continued ...

```
    /* do synch */
    for (width=1; width<num_mat; width+=1){
        for (i=0; i<num_calc; i+=1){
            in("synch", width);

        }

        out("continue", width+1);

    }

    /* report */
    for (i=0; i<num_calc; i+=1){
        in("calc", ?j);
    }

    in("cost", zero, num_mat-1, ?i);
    fprintf(stderr, "minimal cost => %d\n", i);

    return(NERR);
}



/*
 @ calc() : do calculations
 */
/* worker */
%%calc
int calc(void)
{
    int num_mat;                            /* number of matrices */
    int num_calc;                           /* number of calculators */
    int calc_id;                            /* own id */
    int * size = NULL;                      /* matrix sizes */
    int width, start, end, mid;             /* loop controls */
    int val_new, val_row, val_col;          /* for doing calculations */
    int local_min;                          /* own least value */
    int val_old;                            /* existing value */

    /* get stuff normally passed in params to eval              */
    in("calc_in", ?num_mat, ?num_calc, ?calc_id);

    /* allocate storage and get sizes */
    if ((size = (int *)calloc(num_mat+1, sizeof(int))) == NULL){
        fprintf(stderr, "calc %d: alloc(size) failed\n", calc_id);
        exit(ERR);
    }

    rd("size", ?size);
```

continued ...

```
    /* do calculations */
    for (width=1; width<num_mat; width+=1){
        for (start=0; start<(num_mat-width); start +=1){
            end = start + width;
            local_min = MAXINT;
            for (mid=start; mid<end; mid+=1){
                if (mine(num_calc, calc_id, start)){

                    val_new = size[start] * size[mid+1] * size[end+1];
                    rd("cost", start, mid, ?val_row);
                    rd("cost", mid+1, end, ?val_col);
                    val_new += val_row + val_col;
                    if (val_new < local_min){
                        local_min = val_new;
                    }
                }
            }
            if (local_min < MAXINT){

                in("cost", start, end, ?val_old);

                if (local_min < val_old){
                    out("cost", start, end, local_min);

                } else {
                    out("cost", start, end, val_old);

                }
            }

        }
        /* do synchronisation */
        out("synch", width);

        rd("continue", width+1);

    }

    /* return(NERR); */
    out("calc", NERR);
}

/*
 @ mine() : identifies which elements to work on
 */

static int mine(num_calc, calc_id, start)
    int num_calc;                       /* number of calculators */
    int calc_id;                        /* own id */
    int start;                          /* where we're starting */
{
    return((start % num_calc) == calc_id);
}
```

Listing 8: Optimal Matrix Multiplication Order

References

[Ahmed and Gelernter 1991]
Ahmed S., Gelernter D., *A Higher-Level Environment for Parallel Programming*, Department of Computer Science, Yale University, New Haven, Connecticut, USA, YALEU/DCS/RR-877, 12 November 1991

[Ahmed *et al* 1993]
Ahmed S., Carriero N., Gelernter D., *A Program Building Tool for Parallel Applications*, Department of Computer Science, Yale University, New Haven, Connecticut, USA, 1 December 1993

[Ahuja *et al* 1986]
Ahuja S., Carriero N., Gelernter D., *Domesticating Parallelism - Linda and Friends*, IEEE Computer, 19(8), August 1986, p26(9)

[Balter *et al* 1991]
Balter R., Bernadat J., Decouchant D., Duda A., Freyssinet A., Krakowiak S., Meysembourge M., Le Dot P., Van Nguyen H., Paire E., Riveill M., Roisin C., Rousset de Pina X., Scioville R., Vandôme G., *Architecture and Implementation of Guide, an Object-Oriented Distributed System*, Unite Mixte Bull-IMAG, Gières, France, 1991

[Bjornson *et al* 1988]
Bjornson R., Carriero N., Gelernter D., Leichter J., *Linda, the Portable Parallel*, Department of Computer Science, Yale University, New Haven, Connecticut, USA, Research Report YALE/DCS/RR-520, January 1988

[Bjornson *et al* 1989]
Bjornson R., Carriero N., Gelernter D., *The Implementation and Performance of Hypercube Linda*, Department of Computer Science, Yale University, New Haven, Connecticut, USA, YALEU/DCS/RR-690, March 1989

[Bjornson *et al* 1991]
Bjornson R., Carriero N., Gelernter D., Mattson T., Kaminsky D., Sherman A., *Experience with Linda*, Department of Computer Science, Yale University, New Haven, Connecticut, USA, YALEU/DCS/TR-866, August 1991

[Bricker *et al* 1992]

Bricker A., Litzkow M., Livny M., *Condor Technical Summary*, Computer Sciences Department, University of Wisconsin - Madison, USA, 1992

[Brooks *et al* 1991]

Eugene D. Brooks et al, *The 1991 MPCI Yearly Report: The Attack of the Killer Micros*, Massively Parallel Computing Initiative, Lawrence Livermore National Laboratory, Livermore, California, USA, March 1991

[Cap and Strumpen 1993]

Cap C.H., Strumpen V., *Efficient Parallel Computing in distributed workstation environments*, Parallel Computing, 19(11), November 1993, p1221(14)

[Carriero and Gelernter 1989]

Carriero N., Gelernter D., *Linda in Context*, Communications of the ACM, 32(4), April 1989, p444(15)

[Carriero and Gelernter 1990]

Carriero N., Gelernter D., *How to Write Parallel Programs (a First Course)*, MIT Press, Cambridge Massachusetts, USA, 1990

[Carriero *et al* 1992]

Carriero N., Gelernter D., Mattson T.G., *Linda in Heterogeneous Computing Environments*, Proceedings of the IEEE Workshop on Heterogeneous Processing, Beverly Hills, California, USA, 23 March 1992

[Carriero *et al* 1993]

Carriero N., Gelernter D., Kaminsky D., Westbrook J., *Adaptive Parallelism with Piranha*, Department of Computer Science, Yale University, New Haven, Connecticut, USA, YALEU/DCS/RR-954, February 1993

[Chase *et al* 1989]

Chase J.S., Amador F.G., Lazowska E.D., Levy H.M., Littlefield R.J, *The Amber System: Parallel Programming on a Network of Multiprocessors*, Proceedings of the 12th ACM symposium on Operating System Principles, 1989, p 147(12)

[Clayton *et al* 1990]

Clayton P.G., Wentworth E.P., Wells G.C., de-Heer-Menlah F.K., *An Implementation of Linda Tuple Space under the Helios Operating System*, Department of Computer Science, Rhodes University, Grahamstown, South Africa, Technical Document PPG 90/8, October 1990

[Cole 1989]

Cole M., *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, Cambridge, Massachusetts, USA, 1989

[Davis 1993]
Davis R., *Windows Network Programming*, Addison-Wesley, Reading, Massachusetts, USA, 1993

[de-Heer-Menlah 1991]
de-Heer-Menlah F.K., *Analyzing Communication Flow and Process Placement in Linda on Transputers*, Department of Computer Science, Rhodes University, South Africa, TD91/17, October 1991

[Dongarra *et al* 1993]
Dongarra J.J., Geist G.A., Manchek R., Sunderam V.S., *Integrated PVM Framework Supports Heterogeneous Network Computing*, Computers in Physics, 7(2), March-April 1993, p166(9)

[Gelernter 1988]
Gelernter D., *Getting the Job Done*, Byte Magazine, 13(12), November 1988, p301(8)

[Gelernter and Carriero 1992]
Gelernter D., Carriero N., *Coordination Languages and their significance. (integration or separation)*, Communications of the ACM, 35(2), p96(12), February 1992

[Homer 1992]
Homer P.T., Schlichting R.D., *A Software Platform for Constructing Scientific Applications from Heterogeneous Resources*, Department of Computer Science, University of Arizona, Tucson, Arizona, USA, TR 92-30, 17 November 1992

[Kale and Krishnan 1993]
Kale L.V., Krishnan S., *CHARM++: A Portable Concurrent Object Oriented System Based On C++*, Department of Computer Science, University of Illinois, Urbana-Champaign, USA, 1993

[Kambhatla 1990]
Kambhatla S., *Recovery with Limited Replay: Fault-tolerant Processes in Linda*, Department of Computer Science, Oregon Graduate Institute of Science and Technology, Oregon, USA, CSE 90-019, 1990

[Kaminksy and Carriero 1991]
Kaminsky D., Carriero N., *Experiments with Piranha Parallelism*, in Proceedings : Research Directions in High-level Parallel Programming Languages, Mont Saint-Michel, France, June 17-19 1991, p45(20)

[Leffler *et al* 1990]

Leffler S.J., McKusick M.K., Karels M.J., Quarterman J.S., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, Massachusetts, USA, 1990

[Milner 1989]

Milner R., *Communication and Concurrency*, Prentice Hall (UK), Hemel Hempstead, Hertfordshire, UK, 1989

[Muller 1994]

Muller P.J., *An Environment for Distributed Programming on a Multicomputer*, Department of Computer Science, University of Stellenbosch, South Africa, February 1994

[NSF 1993]

1993 GCAG Press Release - NSF Welcomes 'Grand Challenges of Science', Science and Technology Information System, National Science Foundation (USA), stis@nsf.gov

[Perihelion 1991]

Perihelion Software Limited, Davies I. (editor), *The Helios Parallel Operating System*, Prentice Hall (UK), Hemel Hempstead, Hertfordshire, UK, 1991

[Petzold 1992]

Petzold C., *Programming Windows 3.1*, Microsoft Press, Redmond Washington, USA, 1992

[Roth and Setz 1992]

Roth R., Setz T., *LiPS: a System for Distributed Processing on Workstations*, FB-14 Informatik, Universität des Saarlandes, Saarbrücken, Germany, 24 November 1992

[Sewry 1994]

Sewry D.A., *Behavioural Model Debugging in Linda*, Department of Computer Science, Rhodes University, Grahamstown, South Africa, TD94/09, January 1994

[SMI 1990]

Sun Microsystems , *Network Programming Guide*, Sun Microsystems Inc., March 1990

[Smith 1993]

Smith G.L., *A Distributed Linda Server on a Network of Heterogeneous Processors*, Department of Computer Science, Rhodes University, Grahamstown, South Africa, TD93/06, January 1993

[Sunderam *et al* 1993]
Sunderam V.S., Geist G.A., Dongarra J.J., Manchek R., *The PVM Concurrent Computing System: Evolution, Experiences, and Trends*, Department of Mathematics and Computer Science, Emory University, Atlanta, Georgia, USA, 1993

[Tanenbaum 1992a]
Tanenbaum A.S., *The Amoeba Distributed Operating System*, Vrije Universiteit, Amsterdam, The Netherlands, 1992

[Tanenbaum 1992b]
Tanenbaum A.S., *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, New Jersey, USA, 1992

[Theimer and Lantz 1989]
Theimer M.M., Lantz K.A., *Finding Idle Machines in a Workstation-Based Distributed System*, IEEE Transactions on Software Engineering, 15(11), November 1989

[Thomas 1991]
Thomas O., *A Linda Kernel for Unix Networks*, Linda-Like Systems and Their Implementation, Edinburgh Parallel Computing Centre, University of Edinburgh, Edinburgh, UK, TR 91-13, edited by Greg Wilson, June 24, 1991

[Turcotte 1993]
Turcotte L.H., *A Survey of Software Environments for Exploiting Networked Computing Environments*, Engineering Research Centre for Computational Field Simulation, P.O. Box 6176, Mississippi State, MS 39762, USA, 11 June 1993

[Watkins 1992]
Watkins R.C., *Algorithmic Skeletons as a Method of Parallel Programming*, Department of Computer Science, Rhodes University, Grahamstown, South Africa, October 1992, TD92/15

[Wentworth 1990]
Wentworth E.P., *Parallelism via Linda - A Transputer Implementation (Status Report June 1990)*, Department of Computer Science, Rhodes University, June 1990, PPG 90/5

[Whiteside and Leichter 1988]
Whiteside R.A., Leichter J.S., *Using Linda for Supercomputing on a Local Area Network*, Department of Computer Science, Yale University, New Haven, Connecticut, USA, Technical Report YALEU/DCS/TR-638, June 1988

[Wilson 1991]

Wilson G.V. (editor), *Linda-like Systems and Their Implementation*, Edinburgh Parallel Computing Centre, University of Edinburgh, Edinburgh, UK, Technical Report 91-13, June 1991

[Wilson and George 1991]

Wilson G.V., George F.A.W., *Using Dynamic Programming to Benchmark Communications on Parallel Computers*, Edinburgh Parallel Computing Centre, University of Edinburgh, Edinburgh, UK,17 October 1991

[Zenith 1990]

Zenith S.E., *Linda Coordination Language; Subsystem Kernel Architecture (on Transputers)*, Department of Computer Science, Yale University, New Haven, Connecticut, USA, YALEU/DCS/RR-794, May 1990