Coalescing Idle Workstations as a Multiprocessor System using JavaSpaces and Java Web Start

by

Alistair Kenneth Atkinson, BComp.

A dissertation submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Bachelor of Computing with Honours

UNIVERSITY
OF TASMANIA

November 2003

Declaration

I, Alistair Kenneth Atkinson, certify that this thesis contains no material which has been accepted for the award of any other degrees or diploma in any tertiary institution, and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person except where due reference is made in the text of this thesis.

Signed: ..... Date: .....

Abstract

Many of an organisation's workstations spend the majority of their time either unused or relatively idle. If the wasted processing capacity of these workstations could be aggregated, it could be used to provide processing for computationally intensive applications. Such a system could potentially provide an inexpensive alternative to costly custom built parallel computers or clusters.

This thesis presents the design, development and evaluation of a system which aims to meet this goal. There are two main parts to the system: a distributed environment to which users can submit jobs for execution, and a development framework which can be used to build distributed applications suitable for deployment on the system.

The development framework greatly eases the development of applications for the system, and also provides general coordination mechanisms to control the execution of reasonably complex applications.

The distributed environment is based on the master/worker distributed system architecture. This allows large, computationally-intensive tasks to be divided up into smaller subtasks and distributed out to worker computers, in this case idle workstations, for processing. This generally results in increased performance for the application.

The system is evaluated using two sample applications: the n-Queens problem, and a parallel sorting (shearsort) application. These applications have relatively contrasting characteristics, highlighting several important properties of the system. The evaluation of the system indicated that it is indeed capable of utilising the wasted capacity of idle workstations. It was found that applications are able to achieve excellent performance gains. However, the contrasting performance of the two sample applications unambiguously showed that if an application is too fine-grained, or includes too much sequential execution, then the performance gain actually realised will be much less.

Acknowledgements

I firstly wish to thank Dr Vishv Malhotra for supervising this research over the course of the year. His assistance, guidance, and helpful suggestions were invaluable, and really helped to improve this dissertation.

Also, I would like to thank everyone at the School of Computing for making the place such an enjoyable place to study. In particular, I would like to acknowledge the work done by Jacky Hartnett and Dr Mike Cameron-Jones in coordinating the course. Their advice and encouragement were a great help. Thankyou also to my fellow Honours students, who have made the year a very enjoyable one.

Finally, I wish to extend all of my gratitude to my parents, and my partner, Bonne, for their constant support and encouragement over the past year, without which this would not have been possible.

Contents

Chapter 1

Introduction

This chapter provides an introduction and overview of the research that has been carried out, including the motivation for the research and also the benefits of the solution that was developed.

1.1 Motivation

The traditional means of supplying high performance computing have been with custom-built parallel computers, or perhaps clusters of computers, among other alternatives. Organisations that required access to this kind of processing power had the choice to either build their own supercomputer or cluster, or lease time on an existing system. Either choice has serious cost implications.

Many of these same organisations have a large number of workstations, which are often under-utilised. These workstations, even when in use, may only require a small fraction of the total processing power of the machine. If all of this wasted processing capability could be aggregated, the resultant computing power could potentially be comparable to the other methods mentioned previously. Furthermore, the cost of doing so would be vastly less than the traditional approaches, as the infrastructure is already in place, and all that is required is appropriate software to put it to use.

1.2 Overview

This thesis presents research into the design and development of a suitable distributed system to allow computers to be easily donated by their users to a pool of other underutilised computers. Computers may be added to and removed from the system dynamically, without affecting its correct operation. The software required for a computer to join the system is automatically downloaded and launched when the user clicks on an appropriate hyperlink on a web page, avoiding the need for the software to be installed and configured manually.

A development framework is also provided to facilitate the implementation of applications for deployment on the system. The framework allows the programmer to specify the structure of an application, including which parts can be parallelised and which should execute sequentially. It is also possible to specify any data dependencies between each part of the application.

An evaluation of the system is performed to determine whether it is successful in meeting its stated goals, namely whether it is capable of effectively putting to use the wasted processing power of underutilised computers on a network.

1.3 Benefits System Delivers

The benefits provided by this system can be summarised as providing potentially large amounts of processing capacity at very low cost. The system takes advantage of infrastructure that is generally already in place, generally workstations connected by a local area network. The systems is able to leverage this (possibly heterogeneous) environment, and aggregate the wasted processing capacity for the processing of computationally intensive tasks.

As the system is able to use an organisation's existing infrastructure, the cost of accessing this potentially large amount of processing capacity would be only a small fraction of the amount of, say, a custom-built parallel machine, or a cluster of computers.

An additional benefit of the system is that its processing capacity is not fixed; computers can be dynamically added and removed from the system at any time, thereby dynamically changing the overall processing capability. Depending on its scalability, the addition of extra machines to the system will result in a relative increase in processing capacity (ie. doubling the amount of computers will result

in a doubling of processing capacity). This is in contrast to custom-built machines or clusters, which must be upgraded to increase the available processing capacity. This upgrade generally requires a complete system shutdown, during which time no processing will be performed.

Finally, the process a user must follow in order to join their workstation to the system is very simple, due mostly to the client-side software being deployed as a Java Web Start application. Similarly, a user can also easily remove their workstation from the system at any time they wish. It is hoped that this ease of use would encourage users to take part in the system were it ever deployed.

1.4 Thesis Structure

Chapter 2 explores the relevant literature for this research, including an overview of the basic concepts of distributed computing, and also specific distributed computing technologies. We will also detail the various Java based distributed computing frameworks, focusing particularly on those pertinent to this research. Finally, we will discuss some current research in distributed computing, with a specific emphasis on dynamic utilisation of computers, and Java based distributed computing.

In Chapter 3, we present the system that was developed, including an overview of its architecture and details of its operation. We also detail the development framework that can be used to build parallel applications suitable for deployment on the system. Finally, we look at two example applications that were developed for the system using this development framework.

Chapter 4 details some performance results and analysis of some testing conducted using the two example applications that were developed for the system. An analysis of the meaning and consequences of these results is also given.

Chapter 5 discusses the conclusions that can be drawn from the research presented, particularly whether or not the proposed system successfully meets its stated goals, and can effectively make use of the wasted processing capabilities of underutilised computers. This chapter also discusses some worthwhile future directions for research in this area.

All program source code that comprises the system and the tested applications is included in the appendices.

Chapter 2

Literature Review

The potential processing power of distributed systems has resulted in much research effort in the area of distributed computing. Distributed systems can provide an effective alternative to expensive high-performance computers, an option that businesses and organisations are beginning to take advantage of.

This review will explore the fundamental aspect of distributed computing, along with some important technologies in use. The Java platform from Sun Microsystems will be discussed, specifically its distributed computing technologies, as it is widely used in distributed computing. Finally, past and current research in distributed computing will be detailed.

2.1 Distributed Systems

Coulouris et al (1994, p. 1) define a distributed system as "a collection of autonomous computers linked by a network, with software designed to produce an integrated computing facility."

The advances made in microprocessor design and the emergence of high speed networks have seen distributed systems' emerge as a solution to many computationally intensive problems. Massively distributed systems have been made possible by the rise of the Internet, some systems growing as large as hundreds of thousands of computers. Also, many large organisations have realised the benefits of utilising the unused CPU cycles of their idle workstations.

This section will detail some essential properties that all distributed systems aim to achieve, and the motivation for the distribution of systems.

2.1.1 Desired Properties of Distributed Systems

The development of distributed systems is a complex task. The increased complexity involved raises many issues which are not often as critical when developing a traditional, non-distributed system. Nonetheless, if these issues can be dealt with, distributed systems can provide numerous advantages over traditional, standalone systems.

2.1.1.1 Reliability

Applications can benefit from the potentially increased reliability of a distributed system relative to a standalone system. Tel (1994, pp. 3-4) states that this is due to distributed systems' *partial failure* property, also known as *graceful degradation*. This property allows some nodes in the system to fail, and their tasks taken over by the other correctly-operating nodes. In contrast, a standalone application will experience complete failure if the computer, on which it is running, crashes.

The reliability of a distributed system is not only determined by the reliability of its nodes but also the reliability of the network by which it is connected. For systems implemented on a local area network (LAN) this may not be a critical issue, as a LAN typically provides highly reliable communications. However for systems that use the Internet as its underlying network it must be taken into consideration, and also it must handle issues such as higher latency and reduced bandwidth.

2.1.1.2 Scalability

Tanenbaum & Steen (2002, p. 10) state that with widespread connectivity available through the Internet, "scalability is one of the most important design goals for developers of distributed systems". However, they also add that scalable systems often exhibit some performance loss as a consequence. Therefore it is desirable to minimise the overhead associated with achieving scalability.

The scalability of a system can be measured in terms of size, geography and administration (Tanenbaum & Steen 2002, p. 10). A system is said to be scalable in terms of its size if computers, or nodes, can be easily added to the system, and the benefits of adding these computers is not outweighed by the overhead incurred in doing so, thus defeating the purpose. A geographically scalable system can have nodes separated by long distances. An administratively scalable system is easy to

administer even if the system spans multiple organisations, something especially important in decentralised systems.

2.1.1.3 Security

Security is one of the most difficult properties to instill into a system, because it must be pervasive throughout the entire system (Tanenbaum & Steen 2002, p. 413). A single vulnerability could be used to compromise the entire system. The security requirements for any given distributed system will vary depending on the environment in which it will be deployed. Obviously, a dynamic Internet-based system with thousands of nodes will have different security risks than a system consisting of three computers connected by a LAN. There are, however, some security threats that are common to all networked systems. Stallings (2001, pp. 6-7) identifies *interception*, *interruption*, *modification* and *fabrication* as threats which should be considered.

An interception attack involves some unauthorised party gains access to system resources, compromising the confidentiality of the system. An example of such an attack would be someone, whether a person, program or computer, capturing data packets off the network.

Interruption occurs when a system resource becomes unavailable or is rendered unusable for some reason, affecting the availability of the system. The highly publicised denial of service attacks perpetrated on the Internet are an example of interruption, as are hardware failures and loss of network communications.

Modification involves an unauthorised party tampering with or corrupting a system resource, usually data, thus compromising the integrity of the system. Examples include modification of data files or the contents of network messages. Fabrication attacks are similar to modification attacks, however they involve new objects being inserted into the system instead of existing objects being modified. Such an example would be the transmission of bogus messages on the network.

Various cryptographic tool are available to counter these threats. IPSec can be used to address the lack of security in IP networks, including confidentiality of data and also authentication of users of the system. Kerberos is another system that can provide authentication of users trying to access resources on a system. The effectiveness of any security systems implemented is, however, dependant on the correct identification of the threats to the system.

2.1.1.4 Transparency

A fully transparent distributed system is defined by Tanenbaum & Steen (2002, p. 5) as one that is "able to present itself to users and applications as if it were only a single computer system." They also suggest that there are several aspects of transparency, including *access*, *location*, *migration*, *relocation*, *replication*, *concurrency*, *failure* and *persistence*.

While transparency is certainly an important goal when designing a distributed system, total transparency is not always desired, or in fact possible. The exact nature of a distributed system will dictate the degree of transparency possible for each of the criteria listed above. For example, a distributed file system may be almost fully transparent to the end user, however in the occurrence of a disk crash the file systems may become unusable. Therefore this system could not be classed as being failure transparent.

An important note is made by Tel (1994, p. 2), who suggests that "the implementation of this transparency requires the development of intricate distributed control algorithms", inferring that the inclusion of transparency as a required property of a distributed system is too restrictive. Tanenbaum & Steen (2002, p. 7) state that "there is also a trade-off between a high degree of transparency and the performance of a system." Therefore it seems prudent to stay mindful of these trade-offs when designing a distributed system, along with the benefits that it can provide.

2.1.1.5 Performance & Efficiency

For many applications, a distributed approach can result in significant performance gains. Parallelisation is often a natural consequence of a distributed system, which can be viewed as a collection of interconnected processors each able to perform computation concurrently. Distributed applications can be designed to take advantage of this feature by dividing up large tasks into smaller subtasks and sending each subtask to a processor in the system for computation.

Distributed systems that are highly efficient generally have low overheads as a consequence of their distribution, and hence experience a god scalability as a result.

2.1.2 Motivation for Distributing Systems

There are many possible motivating factors for using a distributed system, the most obvious being performance. As mentioned in Section 2.1.5, distributed systems allow the parallelisation of applications which often results in performance gains. There are many applications that could use this property to harness the processing power of a number of computers to perform some computation, particularly in large organisations where workstations are idle for large periods of time.

Carriero & Gelernter (1990, p. 1) claim that parallelism is the natural way for humans to solve complex problems. They suggest the example of a watch or a steam engine, which are built out of many components acting simultaneously. Therefore, it can be said that many complex problem solving tasks could be aided by a distributed system.

Applications may also wish to exploit the scalability and reliability of a distributed system. These factors were discussed in previous sections.

2.2 Distributed Computing Technologies

There are numerous technologies which can aid in the development of a distributed system, from basic interprocess abstractions such as sockets through to elaborate enterprise-level component systems. Distributed systems can generally be classified, based on their communication mode, as *message passing*, *remote procedure calling*, *remote method invocation*, or *shared memory* systems.

This section details some notable distributed computing technologies, significant because of either their widespread use or historical significance.

2.2.1 Remote Procedure Call

All distributed computing technologies are based, at their core, on passing messages over a network (Tanenbaum & Steen 2002, p. 57). However this is a complex and error-prone process if done manually, and it does not provide any transparency at all to the programmer as every piece of communication had to be done explicitly. Remote Procedure Call (RPC) provides this missing transparency.

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

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

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

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

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

RPC was an innovative technology that simplified the complex task of de-

veloping a distributed system. It introduced many concepts that are still in use in many current distributed technologies, including remote method invocation systems such as Java RMI.

2.2.2 CORBA

The Common Object Request Broker Architecture is a distributed systems specification developed by the Object Management Group (OMG), a group of more than 800 members, many from within the commercial computing field. CORBA was developed to address the issue of interoperability between networked systems. It is an open standard, and as a consequence vendors are free to make their own changes and extensions if they wish.

The central component of any CORBA system is the Object Request Broker (ORB). Brose et al (2001, p. 31) state that

> It acts as a message bus between objects that may be located on any machine in a network, implemented in any programming language, and executed on any hardware or operating system platform. The caller needs only an Object Reference and well-formed arguments in the language mapping of choice to invoke an operation as if it were a local function and receive results.

Another core piece of the CORBA specification is the Interface Definition Language (IDL). IDL is a language-independent way to produce well defined interfaces between object in the system. Brose et al (2001, p. 34) IDL is used to "inform clients of an object offering an interface exactly what operations an object supports, the types of their parameters, and what return types to expect." Compiled IDL produces stubs, similar to those seen in RPC, that allow the programmer to invoke methods in another object in a very transparent way.

CORBA is a very detailed specification, offering various services, communication models, naming services, and interoperability capabilities that will not be covered here. It is important to note, however, that CORBA is an industry standard that can be used many client/server systems, and is especially suited for systems that require interoperability between platforms.

2.2.3 DCOM

The Distributed Component Object Model is an object based distributed system from the Microsoft Corporation. DCOM is an extension of COM, which is used extensively throughout the Microsoft Windows 9x and NT operating systems. DCOM supports communication among objects located on different computers, similar in many ways to CORBA.

DCOM is an example of a *component architecture*, whereby individual components are specified that can interact and be dynamically activated (Tanenbaum & Steen 2002, p. 526). Components can take the form of a dynamically linkable library or an executable program.

The remote-object model common to most object-based distributed systems can also be found in DCOM. DCOM is programming language independent, due to the use of the Microsoft Interface Definition Language (MIDL). This is very similar to the IDL found in CORBA, except DCOM uses binary interfaces which eliminate the requirement for language-specific bindings (Tanenbaum & Steen 2002, p. 527).

One major difference between DCOM and CORBA is the fact that it is a proprietary technology, whereas CORBA is an open standard designed by committee. As a result, DCOM has little support for interoperability between different platforms. The popularity of the Windows operating system has placed DCOM as one of the most widespread distributed computing technologies in use.

2.2.4 Linda

The Linda coordination language is an example of a space-based distributed system. This style of system was first proposed by David Gelernter (1985) at Yale University. Linda differs from all previous distributed computing technologies in that it is distributed not only in space but also in time (Gelernter 1985, p. 81). Its notable features included a persistent shared memory which could be used to store tuples, along with operations to read from and write to the memory. Another characteristic of Linda is the loose coupling of processes; instead of communicating directly via message passing, processes communicate through the persistent tuple space. This feature was referred to by Gelernter (1985, p. 80) as *generative communication*, and it formed the basis of the Linda system.

The goal of Linda was not the development of a platform as such; in fact,

Linda was implemented in many languages, including C, Prolog and Fortran. Rather, it was a model that could be used to implement parallel algorithms and develop distributed data structures in a very simple and elegant way. The model also resulted in very robust and scalable distributed systems.

The development of Linda introduced an innovative new paradigm for distributed computing. Space-based simplify the implementation of otherwise complex parallel algorithms. Recent systems such as JavaSpaces and TSpaces are based heavily on Linda.

2.3 Java Distributed Computing

Sun Microsystems' Java platform was designed to be an "Internet programming language", and as a result it incorporates strong support for networking, security and multi-threading (Farley 1998, p. 6). These important features, along with Java's inherent architecture neutrality, make it a platform ideally suited for distributed computing.

Distributed systems that are built with the Java platform are somewhat different than non-Java systems. Instead of a collection of cooperating computers, processes or processors, we instead have a collection of cooperating Java Virtual Machines (JVMs). This gives a distributed system the important property of homogeneity; all nodes are JVMs, and all objects in the system are pure Java objects. This removes many of the problems that found in heterogeneous systems, where interoperation between nodes can be much more difficult.

Another advantage of Java's architecture neutrality is the powerful notion of code and object mobility. Once a class has been compiled into byte code, it can migrate and be executed on any host JVM (Farley 1998, p. 8). Furthermore, actual instantiated objects can be dynamically loaded into a running program as long as the object's byte code is available to the recipient JVM. This is made possible by Java's powerful object serialisation capabilities (Sun Microsystems 2001), of which all Java distributed computing technologies can take advantage. The dynamic loading of objects allows the function of a program to be changed or extended while it is running, by executing the dynamically loaded object. This feature was first used to execute Applets in web browsers, and it is also extremely useful in distributed systems.

The possibility for untrusted code to be executed on a host JVM raises many

security concerns. Originally, Java handled this problem by using a *sandbox* security model, whereby untrusted code executed with no privileges on the host machine. However, Java 2 (versions later than 1.2) has a fine-grained security model which allows specific rights to be granted or revoked, along with code signing (McGraw & Felten 1999, pp. 22-25) which should prevent the execution of malicious code and ease user concerns. Java also has APIs that provide various encryption and hashing algorithms.

The Java platform offers several distributed computing technologies, each one roughly corresponding to one of the broad categories of distributed systems outlined in Section 2. They are discussed in more detail below.

2.3.1 RMI

Java's RMI (Remote Method Invocation) technology provides a simple and transparent way to invoke the methods of remote objects, even if the objects involved in this interaction may be in different JVMs and potentially on different hosts (Öberg 2001, p. 32). It is conceptually very similar to RPC, however it has been adapted for use in distributed object systems (Sun Microsystems 2002, p.1), and it takes advantage of the capabilities that a homogeneous Java-based system offers, such as object serialisation, fine-grained security and dynamic class loading.

RMI has been a part of the core Java API since version 1.1. Before its inclusion, the only distributed computing capabilities that Java had were sockets. Sockets are relatively low-level, and require the programmer to implement custom application-level protocols for communication. Wollrath et al (JavaSoft 1996), the designers of RMI, identified this as a cumbersome and error-prone process, and it was the goal of RMI to simplify matters and make the development of reliable distributed systems easier.

The procedures that must be followed to invoke methods of remote objects is not entirely transparent to the programmer. This is intentional, as the designers of RMI identified the need to deal with distributed objects in different ways than local objects for reasons such as latency, memory access, concurrency and partial failure (Waldo et al 1994).

Objects in an RMI system that wish to make their methods available to accept remote calls, known as remote objects, must do so explicitly by implementing the

`java.rmi.Remote` interface. Remote objects are treated differently than non-remote objects in an RMI system. Non-remote objects are passed by copy between JVMs (as long as they can be serialised) when included as method arguments or return values, whereas remote objects have a stub object transferred in their place (Sun Microsystems 2002, pp. 10-11).

A stub object acts as a proxy for a remote object. They serve a similar purpose as the RPC client/server stubs discussed in Section 2.1. Once the stub object has been transferred to the recipient JVM, it can be used to invoke the methods of its corresponding remote object. The stub object is treated just like any normal local object, and therefore calls to remote objects are practically transparent to the programmer. The stub object takes care of all of the details of the actually communication, including initiating the connection and marshalling and unmarshalling of parameters and return values (Sun Microsystems 2002, pp. 15-16). One factor that differentiates stubs from ordinary local objects is that their methods throw a `java.rmi.RemoteException` if an error occurs when communicating with its corresponding remote object.

The final major requirement for the RMI system is some mechanism to obtain stubs objects. Öberg (2001, p. 69) suggests that this can be achieved in numerous different ways due to the convenience of Java serialisation. An effective naming service is included with the Java Software Development Kit, called `rmiregistry`. This program is used to bind stubs to names, and then lookup stubs using the known names (Öberg 2001, p. 69). A web server can be used to deliver the actual class code.

An overall view of a simple RMI system is shown in Figure 2.1.

2.3.2 Jini

Jini allows individual services, whether they are hardware devices or software, to be grouped into a network, or *federation,* and have their services offered to other members of the federation. Jim Waldo (1999, p. 77), the lead Jini architect, states that

> Jini allows anything with a processor, some memory, and a network connection to offer services to other entities on the network or to use the services that are so offered. This class of devices includes all the

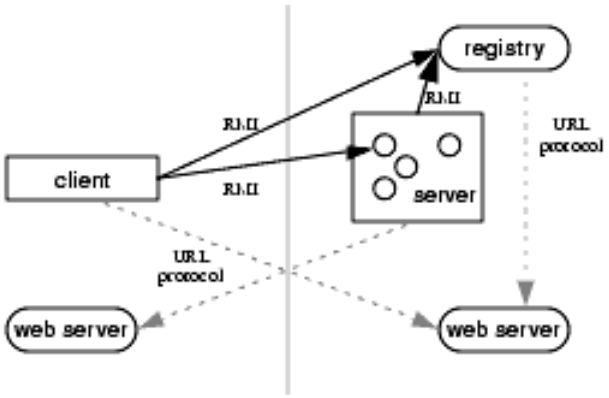

Figure 2.1: Overview of a basic RMI system. (Sun Microsystems 2002)

> things we traditionally think of as computers but also most of the things we think of as peripherals, such as printers, storage devices, and specialised hardware.

One of Jini's key strengths is its ability to adapt to a dynamic network environment. It assumes a changing network, and supports the "incremental upgrading of network components" (Waldo 1999, p. 76). Adding and removing services to and from a Jini federation is very straightforward, and does not effect the operation of the rest of the network.

Jini is built on top of RMI, and as such there are some similar concepts in both systems. Jini has a *lookup service,* similar to RMI's naming service, which allows entities to register a service that they are offering and to find services that other entities are offering. Jini's lookup service stores *service objects*, and these serve a similar purpose to the stub objects in RMI. However, Jini also provides features not found in RMI, such as transactions and distributed events.

The central part of a Jini system are three protocols; *discovery*, *join* and *lookup*. A brief overview of these protocols is given below; for more detail refer to the Jini Architecture Specification (Sun Microsystems 2001, pp. 12-15).

When an entity wishes to join a Jini network, or federation, it first needs to find the lookup service(s) in the network using the discovery protocol. This is basically a broadcast request on a local network, to which any existing lookup services should respond and inform the requester of their existence.

If the entity joining the federation wishes to offer a service, it will use the join protocol to register its service with the lookup service. To offer a service, a

service object must be sent to the lookup service. A copy of the service object will be sent to any other entity in the federation who requests the service. The service object acts as a proxy to the actual service, just like a stub does for a remote object in RMI.

Finally, the lookup protocol is used by entities in a Jini federation to find services, via the lookup service, being offered by other entities. The lookup is done by matching a service with its Java type, along with specific attributes. For example, if we wanted to find a printer on a Jini network that has at least one hundred pages left, we might use the lookup protocol to find an object of type `PrinterService` with an attribute `pages >= 100`. If a match is found, the `PrinterService` object will be transferred to the requester. This object can be used to invoke the actual service, in this case a printer.

2.3.3 JavaSpaces

JavaSpaces is a distributed computing technology that provides a persistent shared memory and simple yet powerful object exchange mechanisms. It is very closely related to Jini in that a JavaSpace is a specific Jini service. Hence, many similar concepts can be found between the two technologies. This allows JavaSpaces to take advantage of all of the features provided by Jini, and therefore RMI, such as object serialisation, code mobility, distributed events and transactions.

Freeman et al (1999, p. 4-5) states that

> (JavaSpaces) is a departure from conventional distributed tools, which rely on passing messages between processes or invoking methods on remote objects. JavaSpaces technology provides a fundamentally different programming model that views an application as a collection of processes cooperating via the flow of objects into and out of one or more spaces.

The design of JavaSpaces was heavily influenced by a previous space-based distributed computing technology called Linda. The Linda coordination language was developed by David Gelernter at Yale University in the early 1980s. It provided a tuple-space where processes could store and retrieve tuples. A characteristic of these systems was the loose coupling of processes, whereby processes communicate through a persistent store rather than through direct messages (Freeman et al 1999, p. xv). For other similarities and difference between Linda

and JavaSpaces refer to the JavaSpaces Specification (Sun Microsystems 2002, p. 6-8).

The JavaSpaces shared memory is provided by a service called a *space*. A space is used to hold *entries*, that is, a Java object that implements the `net.jini.core.entry.Entry` interface. A space has some useful properties, namely persistence, and the ability to be associatively searched. Entries stored in a space remain there until they are explicitly removed or their lease expires, even if the process that placed the object there ceases to exist. Furthermore, it is possible to locate objects stored within a space by performing a search based on an objects type and values of its attributes (Freeman et al 1999, p. 31-32). This method of object location is the same as for the Jini lookup service detailed in the previous section.

There are four fundamental operations that can be performed on a JavaSpaces service; *read*, *write, take* and *notify*.These operations are very simple, and yet expressive enough to facilitate the development of complex distributed systems. (Sun Microsystems 2002, p. 11) states that write is a store operation, read and take are combination search and fetch operations, and notify sets up repeated search operations as entries are written to the space.

Multiple operations can be performed across multiple JavaSpace services atomically, a feature based on the transactions model provided by Jini. The use of transactions results in either all or none of the operations being applied, even if the operations modify multiple spaces. This feature ensures that the system remains in a consistent state and can help to protect against certain problems caused by partial failures, as discussed in Section 2.1.1.

The standard Java programming language facilitates an event-based style of programming. Events may occur when certain designated objects, called *event sources*, react to some internal state change, and it is possible to listen for these events, using *event listeners,* and handle them accordingly. Events are assumed to be delivered reliably and instantaneously in a single-JVM system, however this is not always the case in a distributed system. (Sun Microsystems 2001, pp. 55-56) suggests that notification of events may arrive to clients out of order or not at all, or notification may take an indeterminate amount of time due to latency. These issues are dealt with by the distributed events features of JavaSpaces.

The distributed event model in JavaSpaces allows object to register interest in entries in a space that match a given template. This is achieved using the *notify* operation mentioned previously. If an object matching the description of

the template is written into the space, an event is "fired" to notify the object that registered interest (Freeman et al 1999, p. 219).

JavaSpaces is described by (Sun Microsystems 2001, p. 4) as a tool for building distributed protocols, and for distributed applications that can be modelled as flows of objects through one or more servers. Its use can greatly decrease the complexity inherently encountered when developing a distributed system.

2.3.4 Servlets

Java has great potential for use as a server-side development platform, and this is supported by the Java 2 Enterprise Edition (J2EE). Servlets are one of the central technologies in the J2EE platform, and they provide a powerful and flexible platform for building server applications or producing dynamic web content.

Hunter & Crawford (2001, p. 1) define a Servlet as "a small, pluggable extension to a server that enhances the server's functionality." Servlets are, when used as an add-on to a web server, an alternative to the Common Gateway Interface (CGI) web technology. They offer many advantages over CGI, including performance and portability. However, Servlets can be used for a wide variety of other purposes, including client/server applications.

A *servlet container* is required to execute a servlet. There are three types of servlet container: *standalone*, *add-on* and *embedded*. These are described in more detail below:

- *Standalone* - A standalone container is a server with built-in servlet support (Hunter & Crawford 2001, p. 7). An example is the Apache Tomcat server, an official reference implementation for servlet containers.

- *Add-on* - An add-on servlet container acts as a pluggable module for an existing server, often a web server. These containers add servlet support to servers with a poor, outdated or non-existent servlet implementation (Hunter & Crawford 2001, p. 9). There are add-on containers for most web servers, including Apache and Microsoft's Internet Information Server.

- *Embedded* - Servlet containers can be embedded into an application, ultimately making the application a container in its own right. The Apache Tomcat's open source license has made it popular as a solution as an embed-

ded container, as it can be modified without restriction and thus adapted for many purposes.

There are a number of features which make servlets suited to client/server applications, including portability, efficiency, simplicity, integration with the server, along with all the capabilities provided by the core Java APIs.

Servlets are portable between servlet containers, as long as the containers supports the required version of the Servlet API. Like any Java program, servlets can execute on any JVM without recompilation, and are thus truly portable. Also like any Java program, servlets can take advantage of the capabilities provided by the Java platform API, such as networking, object serialisation, and database connectivity.

Multithreading is an inherent property of servlets; each servlet request is handled by a separate thread, as opposed to CGI where are new process has to be spawned to handle each request. Furthermore, once a servlet has been loaded into memory, it remains there are an object instance (Hunter & Crawford 2001, p. 12). Hence, a servlet maintains its state between requests, and therefore hold onto resources which might otherwise takes time to obtain. These features make servlets highly efficient and scalable.

Servlets are highly extensible and flexible, and can be easily adapted for specific tasks. They are relatively simple, as the Servlet API handles many of the routine tasks of development. Servlets are therefore a very powerful solution for a wide variety of client/server applications.

2.3.5 Applets

Applets are small Java programs that are embedded into a web page. When the web page is loaded into a web browser, the Java byte-code is downloaded and executed within the web browser. The Java Runtime Environment plugin is required for a web browser to run applet code.

There is little difference in capability between an applet and a standalone Java application. However, applets are generally restricted to the *sandbox* security model found in versions of Java previous to version 1.2. This prevents malicious applets from causing any damage to computers they are downloaded and executed on, however it also restricts the capabilities of trusted applets.

Applets are a useful technology for use in distributed systems, as they offer a simple method to download and execute code on on worker computers without any software installation or configuration overhead. This is especially true considering that Java-enabled web browsers are practically ubiquitous on general-purpose workstations.

2.3.6 Web Start

Java Web Start is a technology that allows standalone Java applications to be deployed over the world wide web. A Web Start applications can be launched simply by clicking on a hyperlink embedded in a web page, and spares the user from having to install and configure the software manually (Sun Microsystems 2003).

Web Start technology is enabled by the Java Network Launching Protocol (JNLP). This protocol allows Java applications to be packaged for deployment on a web server, and specifies how the application should be downloaded and launched (Sun Microsystems 2001, p. 8). The JNLP also allows incremental updates and local caching of an application. The core component of a JNLP application is its descriptor file; this is an XML-based file that is used to specify all of an application's attributes, such as the location of the Java class code, and the security model employed.

The Web Start software is installed as a plugin to a web browser, and is launched when the user click on a hyperlink to a JNLP descriptor file. Web Start acts mainly as a JNLP client, and takes care of the downloading and launching of the application.

Web Start has many advantages over Applets, most notably that it is completely independent of the web browser once the application has been launched, and also that an application's class code is cached locally once it has been downloaded. Therefore, the class code will only be downloaded the first time that an application runs, and only ever again is the class code is modified. This also allows the application to be launched again when the computer is not connected to the network.

2.4 Research in Distributed Computing

Distributed computing has been the focus of much research in the past from those in the fields of parallel and high-performance computing. The widespread adoption of the Internet has sparked interest in so-called massively distributed systems, which could harness the power of potentially huge numbers of computers to form a supercomputer with unprecedented processing capabilities. On a slightly smaller scale, many are recognising the potential for exploiting the wasted processing power of idle workstations within an organisation.

This section will examine some of the research that has been done in these areas.

2.4.1 Utilising Idle Workstations

Many organisations have large numbers of computers that, for the vast majority of the time, are performing very little computation. This is the case when users are only running a word processor or web browser on their machine, and especially at night when the computers might be doing nothing at all. The amount of wasted processing power could potentially equal that of very expensive high performance computers. Phillips (1997) identified this as the *idle-workstations problem*, and put forth several ideas about how the problem could be addressed.

A situation such as this is a prime candidate to run some form of distributed system to perform computations when workstation load is low. Shoch & Hupp (1982) developed one of the first such systems, calling them "worm" programs. They were named as such due to their behaviour, being described as "a program or computation that can move from machine to machine, harnessing resources as needed, and replicating itself when necessary" (Shoch & Hupp 1982, p. 172). Each *segment,* or computational section, of a worm is located on an individual computer, and communicates with each other segment to form a complete system. Shoch & Hupp (1982, p. 173) observed that "as segments (machines) join and leave the computation, the worm itself seems to move through the network." They also highlighted the need for control algorithms that monitor how large the worm grows and ensures that system resources are released to users when needed.

Several applications were built using this system, ranging from the simple billboard worm through to a complex real-time multi-machine animation program. The development of this system clearly illustrated the potential of this kind of

system, but also some of the issues such as releasing resources back to users and controlling how and where the worms can spread.

Phillips (1997, pp. 3-8) identifies several other models which may be applied to the idle-workstations problem. Four of these model are detailed briefly below, along with some example systems, to give an idea of some of the available approaches.

- *Processor pool* - Avoid idle workstation problem by logically decoupling the processor from the desktop platform. Examples include the Amoeba, Plan 9 and Clouds systems.

- *Supervisor-worker model* - Divides tasks into a supervisor process and multiple worker processes. Workers are given subproblems to complete, and results are merged by the supervisor. Example include the SETI@Home and distributed.net systems.

- *Location-transparent execution* - Processes may appear to be executing on a single machine but may in fact be distributed, however their behaviour remains unchanged whether executing locally or remotely. For example, the Sprite operating system.

- *Process migration* - Allows processes to migrate between machines; more general than remote execution mentioned above. Examples include the V, Locus and Accent operating systems.

Organisations could reap huge benefits from harnessing their wasted computing resources, and one feels that if a general purpose platform were available to enable this then it would be a much more widespread practice.

2.4.2 Networks of Web Browsers

Fletcher (2002) and Tan (2002), each Honours and Masters students respectively at the School of Computing, University of Tasmania, successfully demonstrated the feasibility of a network of web browsers as a way to build a distributed system. This was made possible by a web browser's ability to download and run Java Applets; the near-ubiquity of these browsers on personal computers hints that this may be a possible approach to solve the idle-workstations problem.

Fletcher and Tan used Java Servlets and Java RMI as the underlying technology in their respective systems to facilitate communications between an Applet in a web browser and the server. Both systems conformed to the supervisor-worker model described in Section 4.1; the Applets were workers who request jobs from the server, complete the jobs and return the results. Fletcher successfully developed a crossword program to run on his system, and achieved an almost linear performance increase up to three computers. Tan achieved similar results for a sorting program. Neither system scaled well with more than five computers, however Fletcher & Malhotra (2003, p. 4) suggest that some of the scalability problems are due to server saturation and could be addressed using well known strategies. They also suggest that robustness, fault tolerance and a suitable interface for submitting tasks to the system would be needed to make the systems more general purpose.

Networking web browsers to form a distributed system is an innovative idea, with a lot of potential when one considers the massive numbers of computers connected to the Internet. In fact, Fletcher & Malhotra (2003, p. 1) describe the system as "a network of browsers that combines available free computers into a huge multi-processor supercomputer."

2.4.3 Adaptive Cluster Computing using JavaSpaces

Batheja & Parashar (2001) propose a framework for an opportunistic, adaptive distributed system using the JavaSpaces technology. Their aim was the exploit the idle processing resources on a cluster of workstations in a non-intrusive manner. The homogeneity provided by a purely Java system assisted them in meeting their goals, however issues such as intrusiveness, system management and adaptability were still encountered.

This system is based on a model similar to the supervisor-worker model described by (Phillips 1997, p.3), whereby tasks are divided into smaller subtasks by the master and distributed among the workers for computation, before the results are returned to the master to be merged. However, the distribution of subtasks to workers is not quite as explicit as this; the subtasks are actually place into the space provided by the JavaSpace service. The workers themselves query the space for available tasks, and when one is found they remove it, complete the computation and return the results to the space. Batheja & Parashar (2001, p.

8) refer to this as the *bag-of-tasks* model. They assert that this model provides natural load balancing, in that workers will be kept busy as long as there are tasks to be completed, however each workers will only do as much work as it is capable of. They also note that this model is naturally scalable, and consequently additional workers will provide improved performance. The bag-of-tasks model is suitable for coarse-grained applications that can be partitioned into relatively independent subtasks.

One of the key characteristics of this system was its ability to be both opportunistic and non-intrusive at the same time. The system was opportunistic so far as it took advantage of any available processing capabilities of idle workstations on the network. However, it did not intrude on users' access the these resources when they were needed. Batheja & Parashar (2001, p. 2) suggest that "a local user should not be able to perceive that local resources are being stolen for foreign computations." The system achieved its goal of non-intrusiveness by using the Simple Network Management Protocol (SNMP) along with the Java Native Interface (JNI) to monitor the state of the network and of the worker machines.

Several "real-world" applications were used to test this system, including a financial simulation program, a scientific ray tracing application and a web page pre-fetching optimisation application. Each application had different characteristics, such as varying memory, task dependency and scalability requirements. The results of these tests revealed good scalability, at least for up to five workers, and performance increase for coarse-grained tasks, along with minimal intrusiveness upon the cluster. The performance gains were most apparent for extremely parallel applications such as the ray tracing program.

This system demonstrates that the idle resources of a network of workstations can be exploited to form a powerful distributed system in a dynamic and non-intrusive manner. It also demonstrates that the space-based paradigm can be used for applications with varied requirements.

2.4.4 Distributed Middle Tier

Chuang & Cheng (1999) identify the three tiers of a web based application as the *presentation* and *computation* of results, and the *data source*. They propose an extended version of this programming model, called a *distributed middle tier*, where there are a pool of worker machines at the computation tier. Tasks are

partitioned and distributed to workers in this pool for computation.

A straightforward way to partition tasks is also suggested by Chuang & Cheng. A simple interface, *Split-and-Merge*, can be implemented by a program to specify how it should be partitioned, executed and finally merged. This method of task partitioning separates the policy and the method of partitioning (Chuang & Cheng 1999, p. 1): the application programmer specifies how a task should be divided, and the system does the actual partitioning accordingly.

The distributed middle tier system was developed using standard Java technologies, including Applets and Servlets, similar to the system described in Section 4.2. An example application was developed for the system which distributed the computation of a Mandelbrot fractal set over several workers (Chuang & Cheng 1999, p. 5). These workers were Java Applets running on a number of computers. Test runs of this program showed results of almost linear speedup for up to four machines.

Chuang & Cheng successfully demonstrated the feasibility of web-based distributed systems, along with a simple method for a programmer to specify how a task should be divided. Their system did not allow, however, for any relationship between sub-tasks, and did not consider issues such as fault tolerance and robustness.

2.4.5 Distributed JVMs

An alternative approach to distributed computing which is currently experiencing considerable research effort are distributed Java Virtual Machines (JVMs). Distributed JVMs aim to make the distribution of an application implicit by creating a JVM that spans multiple computers, whilst making it appear to the application that it is running on a single system, in essence a multi-processor computer.

Remote object systems such as Java RMI provide an effective way to transfer flow of control between JVMs and provides mechanisms for objects to migrate between JVMs. However, RMI does not provide any means for threads to migrate between JVMs. A solution to this problem was proposed by Haumacher et al (2003), who developed a replacement for Java RMI called KaRMI. Their system not only offered superior performance than traditional RMI, but also provides threads with a global identity in a distributed system, and preserves the normal Java thread semantics so that synchronisation still works in a distributed setting.

Their KaRMI system forms the basis of the JavaParty platform.

Various approaches have been taken to developing a distributed JVM. Some systems introduce new keywords to the Java language and have developed their own preprocessors or compilers, such as the JavaParty system. Other system, such as JavaNOW (Thiruvathukal et al, 2000), used freely available Java libraries to construct a framework for parallel applications. Unfortunately, such systems require the application programmer to explicitly tailor the application for these systems, thus limiting the transparency provided.

However, there is a system developed by IBM called the *Cluster Java Virtual Machine* (cJVM) that can potentially provide total transparency (Aridor et al 1999). The cJVM provides a *single system image* to an application, allowing it to execute unmodified using the entire resources of a cluster of computers. The objects and threads of an applications are automatically managed by the cJVM, being distributed around the cluster to achieve scalability and performance benefits. A Java application containing over ten-thousand lines of code was successfully executed on a cluster using the cJVM without any code modifications whatsoever, obtaining promising scalability and performance results (Aridor et al 2000). Currently the cJVM is not very flexible in terms of network configuration or hardware platforms, however if these issues could be addressed then this style of distributed computing could prove extremely effective.

2.5 Summary

This review has explored the fundamental concepts of distributed computing, and looked at various tools which can be used when developing a distributed system. We have detailed some of the past and current research in the field, focusing particularly on efforts that utilise idle computers or systems that are deployed on the web using the Java platform.

The wasted processing potential of idle workstations has long been known to be a problem. Carriero & Gelernter (1990, p. 6) stated that

> The typical modern office or lab environment - many workstations or personal computers, networked together - is another promising environment for parallelism, arguable the most promising of all. If you sum up total computing power over all nodes of a typical local area network, you often wind up with a significant power tool.

If this reasoning is extended to include the total computing power of all nodes on the Internet, obviously the potential processing power is massive; even a tiny portion of it would suffice for a powerful supercomputer. There have been many attempts to harness this wasted processing capability for a worthwhile cause, with varying levels of success. The ability to access a large untapped pool of processing power for an extremely small cost is ample justification for further research into the area.

In Chapter 3, we will look at the system that was developed in an attempt to further explore ways of utilising idle workstations.

Chapter 3

Methodology

This chapter presents the design and implementation of a dynamic distributed system to which users may easily donate their computer; the unused CPU cycles of each computer are aggregated to produce a potentially very powerful distributed processing system.

The first part of this chapter presents such a system, which will allow a large, computationally intensive task to be divided and distributed among whichever computers have joined the system. This system provides a complete framework for the development and deployment of a wide variety of distributed applications.

The second part of this chapter details two sample applications that were developed for the system for testing purposes; namely the n-Queens problem and a parallel sorting (shearsort) application.

3.1 System Overview

There are two main parts to the system: a distributed environment to which users can submit jobs for execution, and a development framework which can be used to build distributed applications suitable for deployment on the system.

The distributed environment enables a user to donate their computer to the system, effectively adding their computer's processing capability to a pool of other worker computers. These workers are used to collectively process large, computationally intensive applications.

3.1.1 Goals

The goal of this research, as alluded to in Section 1.1, is to develop a distributed system that attempts to utilise the wasted CPU cycles of workstations in an organisation. As such, the system should be as efficient as possible, in order to maximise performance and scalability and thus realise the full benefit of adding extra computers to the system.

Furthermore, the system should be robust and reliable, and be able to handle partial failure in the case of one or more computers unexpectedly becoming unavailable to the system. For example, there may be a program crash or loss of network connectivity. In these instances, the system should be able to recover and still produce the correct result of the task.

The process of joining the system should be as simple as possible for a user, and ideally not require any software to install or configure. The system must be dynamic; it should be possible to add computers to the system at any time and at any stage of the execution of a task. Also, a user should be able to end their computer's participation in the system at any time they wish.

3.1.2 Application Development

A development framework is provided to facilitate the creation of applications suitable for execution using the distributed environment. Whilst the software running on the client workstations is generic, the process of preparing jobs for submission to the system for execution will be unique for each application. At the same time, there will be many function that will be common to all applications.

Several classes are provided from which a programmer should inherit, namely `GenericMaster`, `TaskEntry` and `ResultEntry`. An implementation must be provided by every application for each of these classes. The purpose of each of these classes will be explained in subsequent sections. The full source code for these classes can be found in the appendices.

3.1.3 Architecture

The design of the system follows the *master/worker*, or *agenda parallelism* design suggested by Carriero & Gelernter (1991) and similar to that used by Batheja & Parashar (2001).

In such a system, a master process will divide a large task into multiple smaller sub-tasks. These sub-tasks are distributed out to however many worker processes are available. The worker processes will execute the sub-task, and return the results of the computation back to the master. Once all of the sub-tasks have been executed and the results returned, the master will merge the results into a meaningful result of the original large task. An arrangement such as this is illustrated in Figure 3.1.

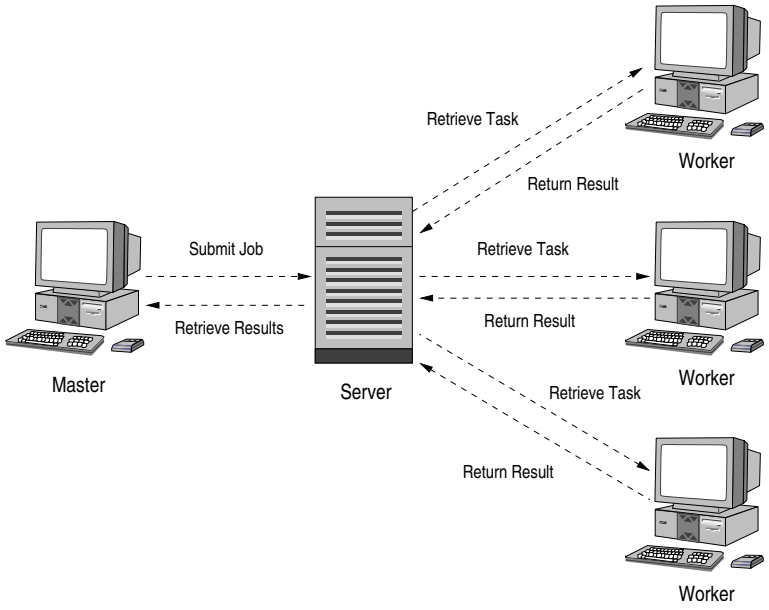

Figure 3.1: An example master/worker system.

The distinguishing characteristics of master/worker systems are discussed in Section 2.2.4. The most notable advantages are natural load balancing, a loose coupling of processes, and a generative communication model which allows temporal as well as spatial distribution.

It was decided to build the system using Sun Microsystem's Jini distributed software architecture, specifically the 'Outrigger' JavaSpaces shared memory. Also used are the 'Mahalo' Transaction Manager and the 'Reggie' Lookup Service. These technologies provide an ideal platform for building a distributed system to meet the goals stated above. Consequently, all development work was carried out with the Java programming language. A more detailed discussion of Jini and JavaSpaces was given in Sections 2.3.2 and 2.3.3.

To allow users to easily join the system and donate their unused CPU cycles, the client-side software was deployed using Java Web Start. This allows a user to launch the client-side application, which acts as a worker in the system, by simply clicking a hyperlink in a web browser. It is assumed that a Java Runtime Environment (JRE) is installed on the majority of computers, thus sparing the user from installing any software and making the system open to a large number of computers. Web Start applications offer many advantages over Java Applets, as they are not tied to the browser after being launched, and don't have be repeatedly downloaded each time the user runs the program. More details on Web Start were given in Section 2.3.6.

3.2 Core Components

This section will discuss the core components of the system, including the master and worker programs and the coordination mechanisms that are employed to control their operation.

3.2.1 Master

It is the role of the master process to split a large task into multiple smaller sub-tasks and distribute these sub-tasks to whatever worker processes are available. The sub-tasks are not distributed directly to individual workers, but rather simply written into the JavaSpace, and workers will take sub-tasks themselves. Once the master has written the subtask objects into the JavaSpace, it will wait until all of the sub-tasks have been successfully executed, at which time it will take all of the resulting result objects from the JavaSpace and assemble them into some meaningful result. Further details of the task and result objects are given in Section 3.2.3.1.

The basic operation of the master process is illustrated in Figure 3.2.

Obviously, the process of dividing a task and merging the results of sub-tasks is going to be different for every application. However, there is a class `core.master.GenericMaster` defined that contains all of the functions that will be common to all applications. A programmer should inherit this class and implement its abstract methods when developing a master program for a specific application.

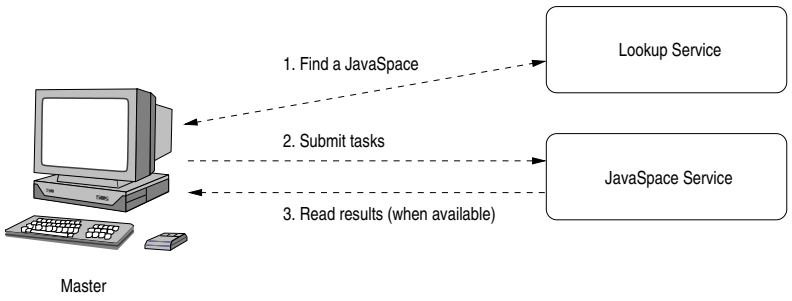

Figure 3.2: Basic operation of the master process.

The `GenericMaster` class is represented, along with its most notable methods, by the class diagram in Figure 3.3.

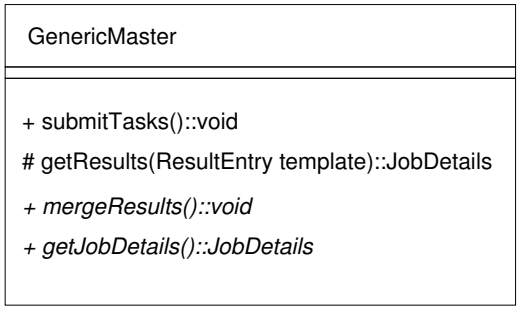

Figure 3.3: The `core.master.GenericMaster` class.

The `submitTasks()` and `getResults()` methods are used to write tasks and take results from the JavaSpace respectively. The `submitTasks()` method calls the abstract methods `getJobDetails()` to obtain the set of sub-tasks for a particular application, as well as other required objects such as those discussed in Section 3.2.3. The other abstract method, `mergeResults()`, should also be implemented by the programmer to combine the set of result objects obtained from `getResults()` into a meaningful result for that particular application.

The source code for `GenericMaster` can be found in Appendix A.

3.2.2 Worker

The worker process' sole purpose is to retrieve a task from the JavaSpace, execute the task, and return any results that are produced to the JavaSpace. It is also

possible that the execution of the task will produce additional further tasks.

The worker program in this system is a Java Web Start application, the main class of which is `core.worker.Worker`. The core functionality of Worker is to obtain the service objects for the JavaSpace and a Transaction Manager services, and then to initialise its graphical interface and also create a new thread which will carry out the actual execution of tasks. These are defined in `core.worker.WorkerGUI` and `core.worker.TaskExecutor` respectively, as illustrated in Figure 3.4.

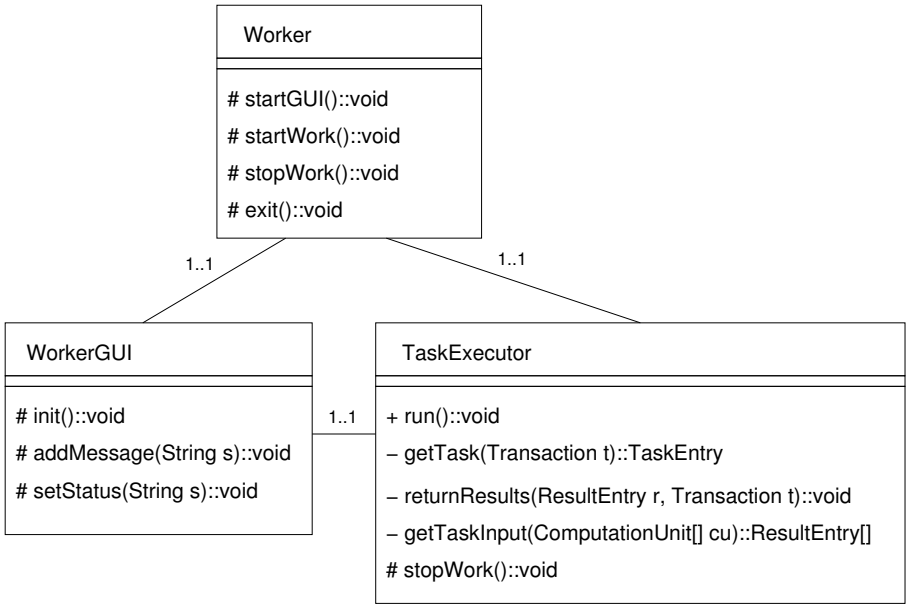

Figure 3.4: Classes from the `core.worker` package.

The `Worker` is launched by clicking a hyperlink in a web browser to a Java Network Launching Protocol (JNLP) file. This file is used to specify various properties of the application, and will be used to start the `Worker` using the Web Start software. The application is packaged in a signed JAR file and made available on a public HTTP server, located at a URL specified in the JNLP file. The same `Worker` program is used for every applications that is built for the system. It would not be modified or reimplemented for individual applications.

3.2.2.1 Task Executor

The `TaskExecutor` runs as a separate thread, after being started by the `Worker`, and it is responsible for carrying out almost all of the work completed by the application. It is completely general-purpose, and can be used to execute any component of the application as long as it conforms to the coordination interface discussed in Section 3.2.3.

Fault Tolerance

The Transaction Manager is utilised by the `TaskExecutor` to maintain the consistency of the system in the event of partial failure. All objects taken from or written to the JavaSpace are done so under a transaction. All transactions are *leased* from the transaction manager for a given period of time; this lease time is specific when the transaction is created. If a transaction's lease expires, the transaction is automatically aborted, thus cancelling all operations performed under it.

The use of transactions becomes especially important in cases where a worker process fails for some reason after having after having acquired some object from the JavaSpace. Instead of these objects being lost indefinitely, they are instead only rendered unavailable until the expiry of the transaction under which they were taken from the JavaSpace, at which time they will be made available once more to other worker processes. The use of transactions also gives the system the added advantage of preventing potential deadlocks from occurring, as objects are only acquired for a finite amount of time. For more details on the Jini transaction model or distributed leasing refer to (Sun Microsystems 2003).

Execution

The `TaskExecutor`'s cycle of obtaining a task, executing it, and then return the results that was discussed previously is, in fact, slightly over simplified. The first operation actually performed is to obtain a *dependency graph* from the JavaSpace. For now, a dependency graph can be thought of simply as an object specifying the order in which tasks could be executed; more detail will be provided in Section 3.2.3.

So, the TaskExecutor uses the dependency graph to determine which task should be executed next, and then obtains and executes the specified task, af-

ter returning the dependency graph to the JavaSpace for any other workers in the system to use. Furthermore, not all tasks will produce results, hence the `TaskExecutor` will only return results to the JavaSpace when they are produced.

The operation of the `TaskExecutor` is illustrated in Figure 3.5.

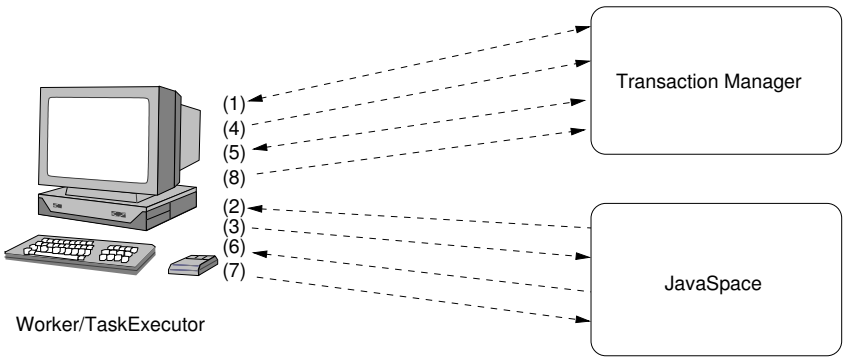

Figure 3.5: Operation of `TaskExecutor`.

A description of the operations performed in Figure 3.5 are as follows:

1. Create a new transaction, which we will call transaction A.
2. Take dependency graph using transaction A.
3. Determine which task can be next executed by invoking the dependency graph's `next()` method; mark the corresponding computation unit in the graph as IN PROGRESS.
4. Write the dependency graph back into the JavaSpace, and commit transaction A.
5. Create a new transaction, which we will call transaction B.
6. Take the task specified by the dependency graph using transaction B.
7. If this task has data dependencies, obtain the relevant result entries to fulfil these requirements using `getTaskInput()`.
8. Execute the task.
9. Create a new transaction, which we will call transaction C.

10. Take the dependency graph from the JavaSpace using transaction C.

11. Update the dependency graph by marking the computation unit corresponding to the current task as EXECUTED.

12. Write the dependency graph back into the JavaSpace using transaction C.

13. Write the result object that was produced into the JavaSpace using transaction B.

14. Finally, commit both transaction B and transaction C.

Note that steps 2 and 10 will block if the dependency graph is currently held by another worker. Also, step 6 involves satisfying the data dependencies of a task; this function is discussed further in Section 3.2.3.

3.2.2.2 Interface

A simple, user-friendly graphical user interface was designed for the `Worker` program, as shown in Figure 3.6. The interface is defined in the `core.worker.WorkerGUI` class, and uses Java Swing GUI components.

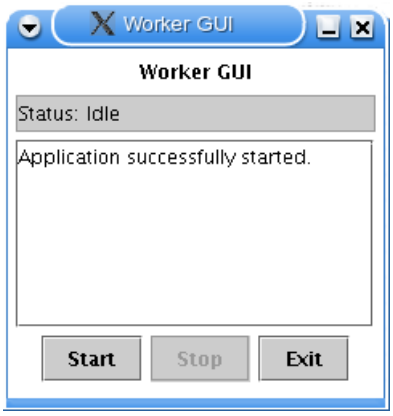

Figure 3.6: The `Worker` graphical interface.

This interface allows the user to join and leave the system at any time, via the **Start** and **Stop** buttons. More specifically, the **Start** button causes the Worker to obtain the service objects for the JavaSpace and the Transaction Manager, and then to create and start running a new `TaskExecutor` thread. The **Stop** button

terminates the `TaskExecutor` thread, as soon as all current operations have completed. The **Exit** button terminates the `Worker` application. The current status of the application (eg. IDLE, RUNNING etc) is displayed in the status bar, and messages can be printed to the text area below.

3.2.3 Coordination

We have seen the details of how the master and worker each operate. However, for any meaningful computation to be carried out, there must be a way to coordinate the activities of each independent worker in the system, and to ensure that the computation will produce a correct result. Many applications designed for execution on a master/worker system will have dependencies between its subtasks, either requiring that some particular tasks are executed sequentially, or the results of some tasks may be needed as input for other tasks.

This section explores the components provided in the `core.coordination` package which can be used to coordinate applications with a reasonable level of complexity.

3.2.3.1 Entries

Entries are the name given to objects that can be stored in a JavaSpace, and accessed by other processes through the JavaSpace's `write`, `read` and `take` operations. The two fundamental units of communication in the system are *task entries*, which are used to represent a computation to be performed, and *result entries*, which are used to store the resultant data produced by the computation.

There are two classes provided, `core.entry.TaskEntry` and `core.entry.ResultEntry`, which can be used to implement entries suitable for an individual application. In particular, `TaskEntry` contains an `execute()` method which defines the function to be carried out by each individual task. When a `Worker` obtains a `TaskEntry` object, it will invoke this method, which will in turn return a `ResultEntry` object that will be written back into the JavaSpace. A `ResultEntry` will simply act as a wrapper class for some data.

Both type of entries are assigned unique identification numbers; each `ResultEntry` will have the same identification value as the `TaskEntry` that produced it. Some `TaskEntry` objects will depend on other `ResultEntry` objects as input, and so each `TaskEntry` object has a vector which can be loaded with the re-

quired `ResultEntry` objects. These features are discussed further in the next section.

Although both `TaskEntry` and `ResultEntry` will be inherited from and be implemented differently for each individual application, a `Worker` process makes no distinctions between the entries of each application, and will simply invoke the `execute()` method and write the resultant `ResultEntry` back to the JavaSpace, if there is one produced.

3.2.3.2 Execution Control & Data Dependencies

One of the major difficulties of distributed systems is the lack of a global state. For applications with even a low level of complexity, it is imperative that each worker process knows the current status of execution. The system provides a solution to this problem in the form of a *dependency graph*.

Dependency Graph

The dependency graph is used to coordinate the execution of, and specify the relationships between the many sub-tasks that make up an application. By using a dependency graph, it is possible to specify the order in which sub-tasks should be executed, along with the data dependencies between these sub-tasks.

The dependency graph is represented by the `core.coordination.DependencyGraph` class, and the graph itself is made up of *computation units*, which are defined in the `core.coordination.ComputationUnit` class. There is exactly one computation unit in the graph for each sub-task in the application; they are associated via a shared identification number.

The graph itself is a directed acyclic graph, with vertices represented by computation units, and edges denoting the data dependencies between them. Each computation unit contains a vector which is used to hold references to other computation units denoting the sub-tasks on which it is depending for input data.

An example dependency graph for a small application consisting of four sub-tasks is shown in Figure 3.7.

A computation unit may have four different states: UNSCHEDULED, SCHEDULED, IN-PROGRESS and EXECUTED. Generally a computation unit will progress through these states in the order listed, however this is not always the case. The initial state of a computation unit will always be UNSCHEDULED, and will be

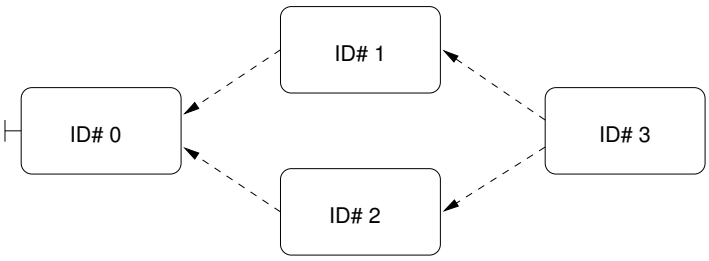

Figure 3.7: An example dependency graph.

changed to SCHEDULED only when all of its data dependencies are satisfied. These dependencies are known to be satisfied when all of the computation unit nodes contained in a given computation unit's preceding tasks vector are marked as EXECUTED. The `ComputationUnit` class provides a method `canSchedule()` which will determine if this is the case, and the `DependencyGraph`'s `scheduleUnits()` method is called at regular intervals to set the status of schedulable units to SCHEDULED.

The numerous computation units that form the dependency graph are stored in an array; the `next()` method iterates through this array and return the first SCHEDULED unit that it finds. Therefore, if there are multiple scheduled computation units, it is the one that appears first in the array that will be executed next.

After a worker process retrieves the next computation unit from the dependency graph, the unit will be marked as IN-PROGRESS. Similarly, after a result entry is produced by the execution of a task entry and returned to the JavaSpace, the corresponding computation unit will be marked as EXECUTED. It is important to note that this sequence (UNSCHEDULED/SCHEDULED/IN-PROGRESS/EXECUTED) is not strict, and if a task does not complete its execution then it's associated computation unit should be reset to UNSCHEDULED. It is also entirely possible for the dependency graph to be modified directly by an application if required, as we will see later.

Usage

Before a worker process takes a task entry from the JavaSpace for execution, it will first take the dependency graph, and invoke the `next()` method to obtain the

next scheduled computation unit. Once the dependency graph has been returned to the JavaSpace, the worker uses the computation unit to retrieve the associated task entry.

On completion of a task's execution, the worker will again retrieve the dependency graph from the JavaSpace. Assuming that the task produced a result entry when executed, the task's corresponding computation unit will be marked as EXECUTED. Following this, the dependency graph and result entry are both written back into the JavaSpace.

3.3 Applications

Two applications were implemented using the framework presented in Section 3.2 to test the suitability and effectiveness of the design that was adopted. Each application has contrasting characteristics in terms of data and communication requirements. The details of their implementation are discussed below.

3.3.1 The n-Queens Problem

3.3.1.1 Overview

The n-queens problem is a generalisation of the well known eight-queens problem. The basic premise of the problem is this: find the number of ways that n queens can be arranged on an $n \times n$ chess board so that no two queens can attack each other, according to the rules of the game. One of the ninety-two possible solutions to the eight-queens problem is shown in Figure 3.8.

.	.	.	.	.	Q	.	.
.	.	.	Q	.	.	.	.
.	.	.	.	.	.	Q	.
Q	.	.	.	.	.	.	.
.	.	.	.	.	.	.	Q
.	Q	.	.	.	.	.	.
.	.	.	.	Q	.	.	.
.	.	Q	.	.	.	.	.

Figure 3.8: A solution to the eight-queens problem.

This particular implementation of this combinatorial problem uses a simple backtracking algorithm. This algorithm has an exponential execution time, which makes it one of the least efficient methods of solving the problem. However it is also one of the most straightforward algorithms to parallelise. A linear-time algorithm can be found in (Sosic, R. & Gu, J., 1994), and an almost constant-time parallel algorithm is discussed in (Sosic, R., 1994).

The problem is parallelised by dividing the entire space that needs to be searched, in this case all possible combinations of queens placings on the chess board, into smaller sub-tasks that will each search a subset of the space. This is achieved by placing a queen in each of the first two columns of the board, in positions where they cannot attack each other. This produces $(n-1) \times (n-2)$ sub-tasks of approximately equal size, given a board of size n, with each sub-task involving a search of n^{n-2} possible combinations. The execution of the sub-task will result in every possible combination of queens placings being searched for the remaining columns, and testing if each combination is a valid solution to the problem, given the position of the queens in the first two columns.

3.3.1.2 Implementation

To make use of the framework presented earlier in this chapter, this applications provides an implementation of the abstract methods in the `GenericMaster` class, and extends the `TaskEntry` and `ResultEntry` classes. The classes related to the n-queens problem are located in the `app.nQueens` package, and include `nQueensMaster`, `QueensTask` and `QueensResult`.

The application is relatively simple, in that there are no data dependencies between the sub-tasks, and no need for the sub-tasks to be executed in any particular order. Thus, given enough workers, this application can be completely parallelised. The `nQueensMaster` will still produce a dependency graph, however all of the computation units will be completely unrelated, and will be scheduled for execution simply in the order in which they are placed in the array.

Every `QueensTask` object will contain a board of size n, with a queen placed in each of the first two columns. The execution of this task will produce a `QueensResult` object, which will contain a single value denoting the number of solutions that were found. Once the dependency graph becomes exhausted, the `QueensMaster` will take all of the `QueensResult` objects from the JavaSpace, and sum all of their values to produce the final result.

A full listing of the program source code for the n-queens application can be found in Appendix D.

3.3.2 Shear Sort

3.3.2.1 Overview

Shear sort is a parallel sorting algorithm that we have adapted for a distributed environment. The shear sort algorithm takes a list of n numbers that are to be sorted, and arranges them into a two dimensional mesh of size $\sqrt{n} \times \sqrt{n}$. Once this has been done, exactly $\sqrt{n}$ phases of execution will be completed. If an odd-numbered phase of execution is being completed, each odd- and even-numbered row will be sorted in ascending and descending order respectively. If an even-numbered phase is being completed, then all columns are sorted in ascending order. After all phases have been completed, the mesh will be sorted. Pseudo-code for the shear sort algorithm is shown below.

```
for step = 1 to sqrt(N) do
  if odd(step)
    if odd(row)
      sort_left_to_right(row);
    else
      sort_right_to_left(row);
  else
    sort_top_to_bottom(col);
```

The shear sort algorithm has an average execution time of $n^{1/2}$ when executed on $\sqrt{n}$ processors. It can be parallelised by sorting each row or column in parallel, depending on the phase of execution.

3.3.2.2 Implementation

There are several classes that have been implemented which allow the shearsort algorithm to run within the system framework. The classes, all located within the `app.shearSort` package, include `ShearSortMaster`, `ShearSortTask`, `Cell`, and `StepCounter`.

The shear sort program has some significant differences to the n-queens program, most notably the use of a shared data structure and dynamic updating of

the dependency graph. This is in contrast to the relatively static nature of the n-queens application. These different characteristics uncovered some design considerations that required a slightly modified approach to the relationship between task and result entries. Specifically, `ShearSortTask` objects, when executed, do not create a new result object. Rather, they will read in some existing result objects, in this case `Cells`, apply the shear sort algorithm to these objects, and then write them back into the JavaSpace. This will be expanded upon below.

Initialisation & Data

The `ShearSortMaster` class inherits the `GenericMaster` class and provides an implementation of its abstract methods. The application is initialised by the `getJobDetails()` method; this method takes the data that is to be sorted, and creates an equivalent number of `Cell` objects to store each value. These cells are arranged into a mesh using relevant row and column indexes. Figure 3.9 illustrates how the data is rearranged.

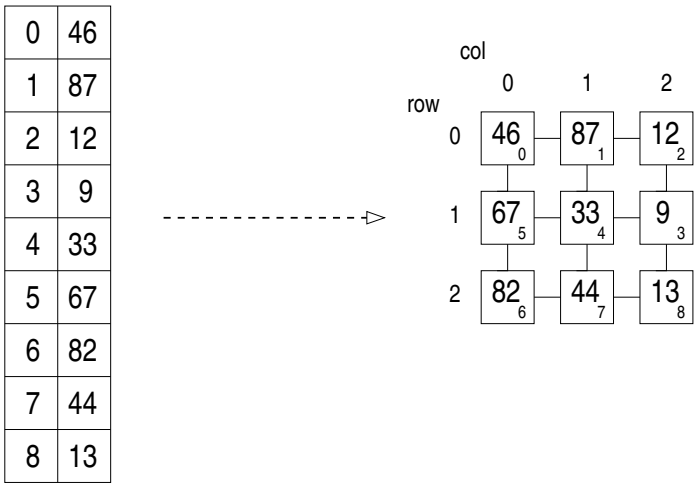

Figure 3.9: Initialisation of list of data into mesh.

After the data has been arranged into a mesh, a `ShearSortTask` object is created for each row in the mesh, where each object has an identification number equal to the corresponding row's identification number. A `ComputationUnit` object is also created for each task object, and used to create a `DependencyGraph` for the application. Finally, a `StepCounter` object is created, and all of these objects used to initialise a new `JobDetails` object.

Execution

As mentioned previously, there an equivalent number of `ShearSortTask` entries as there are rows or columns in the mesh of data that is being sorted. Each task will sort the row/column with the same index as its identification number. There is also a dependency graph which contains a computation unit for every task entry.

There are no explicit data dependencies between tasks in this application; however, each task's data dependencies are implied by its identification number. The dependency graph in this case is used to enforce some ordering in the execution of the application's tasks. More specifically, it ensures that all tasks in a particular sorting phase are complete before moving onto the next phase.

Another distinction with the n-queens application is that this application interacts directly with the JavaSpace and Transaction Manager services. The interaction includes dynamically updating the dependency graph and generating new task entries. Note that all operations performed on the JavaSpace are done so under a transaction created by the transaction manager. The details of the steps involved in the execution of a `ShearSortTask` entry follow.

1. The task will first of all attempt to take the `StepCounter` entry from the JavaSpace.

2. If the step counter's value is equal to $\sqrt{n}$ then all of the phases of sorting have been completed. In this case, the dependency graph will be obtained and updated by marking the computation unit corresponding to the current task as executed. The task will then terminate, as there is no data left to sort. Otherwise, it will continue.

3. If the current task will be sorting the last row or column of the mesh (ie. the task's identification number is equal to the row/column index), then the step counter is incremented.

4. The step counter is then written back into the JavaSpace regardless of its value or which row or column is being sorted. Note that a local copy of the step counter is retained, as it is needed for the next step.

5. If the step counter has an odd value, then the row of the mesh corresponding to the current task is retrieved from the JavaSpace. In the other possible

case where the step counter is even, the corresponding column is retrieved.

6. The next step is to sort the data that has been retrieved. If a column of the mesh was retrieved in the previous step, it is sorted in ascending order[1], or from top-to-bottom. If an even numbered row was retrieved, it is also sorted in ascending order, or from left-to-right. However if an odd numbered row was retrieved, it is sorted in descending order, or from right-to-left. The index values of each cell must be updated to reflect their new ordering.

7. Now that the row/column has been sorted, the dependency graph must be updated via the `updateDepGraph()` method. This method attempts to retrieve both the dependency graph and the step counter from the JavaSpace.

8. If the previous step is successful, the computation unit in the dependency graph that is associated with the current task has its status updated. If the step counter indicates that sorting is complete, then the computation unit will be set to EXECUTED; otherwise, it is set to UNSCHEDULED. In the latter case, a new `ShearSortTask` object is created and written into the JavaSpace to facilitate further sorting. In either case, the step counter and dependency graph are both written back to the JavaSpace.

9. The cells are then written back into the JavaSpace. Execution of the current task is now complete.

10. Once all sorting phases are complete, the dependency graph will be marked EXHAUSTED, at which time the `ShearSortMaster` will retrieve all of the `Cell` objects from the JavaSpace. Once all of the cells have been fetched they are reassembled into a correctly sorted list based on their index numbers.

It is plain to see that the complexity of this application is significantly greater than that of the n-queens application, due the the fact that the dependency graph is dynamically modified during task execution. Also, additional coordination mechanisms are required as the application's sub-tasks are performing operations on a shared data structure.

A full listing of the program source code for the shearsort application can be found in Appendix E.

[1] All rows and columns are sorted using a modified mergesort provided by the `Arrays.sort()` method. This algorithm offers a guaranteed $n \log_2(n)$ performance.

3.4 Summary

This chapter has explored the distributed system framework that was developed, and also two example applications that were built to make use of this system. The distributed system framework can greatly simplify the process of developing, deploying and coordinating distributed applications. The example applications successfully demonstrate the flexibility of the proposed framework, and that it is extensible enough to allow applications with a variety of requirements to be deployed.

In Chapter 4, we will present some performance results gathered from the execution of these applications.

Chapter 4

Results & Discussion

This chapter presents the performance results obtained from various testing methods based on the n-queens and shearsort applications that were described in Chapter 3. These results will then be used to justify the suitability of the system for utilising idle workstations.

4.1 Overview

One of the primary motivations for distributing an application is to allow it to be parallelised; the application may then benefit for the performance gains that result. There is, unfortunately, always an overhead associated with the distribution of an application, usually as a result of slow network communications. Therefore it is important that the potential gains of distributing an application outweigh the overhead involved, to make the distribution worthwhile. Generally speaking, an application should be reasonably computationally intensive to make the most of these potential performance gains.

As previously stated, the motivation for this research was to devise a way to utilise the wasted CPU cycles of idle workstations. In Chapter 3, we presented software that allows a user to simply and easily donate their computer to the pool of worker machines. This chapter will determine whether the system uses these donated computers efficiently, and is able to produce a significant speedup for large applications.

Testing was carried out using the two applications presented in the previous chapter. The different characteristics of each application will be used to test whether the system is capable of supporting applications with a variety of

requirements.

4.2 Testing Environment

All testing was conducted using the computer labs in the School of Computing, Launceston. All required Jini services were set to run on an 800 MHz Pentium 3 machine with 256 MB of RAM, and running Slackware GNU/Linux 9.0. Also running on this machine was the master process of an application, along with the Apache web server, which was used to service requests for the client Web Start application.

The rest of the system consisted of the client (worker) software running on numerous independent computers. Unless otherwise stated, these worker machines were 400 MHz G4 Apple Mac machines with 512 MB of RAM, and running Mac OS X. All computers used for testing were connected via a 100 Mb switched fast-ethernet network. Also, all computers were using the Java SDK 1.4.2.

4.3 Performance Measurement

The usual measures of the performance of a parallel system, suggested by Carriero & Gelernter (1990, p. 74), are speedup and efficiency. Following their definitions, *speedup* is "the ratio of sequential run time to parallel run time", and *efficiency* is "the ratio of speedup to number of processors". Both of these measures give a good indication of the effectiveness of a parallel system in using available processors to their maximum capacity. In particular, the efficiency of a system is an excellent indicator of a system's scalability. Parallel applications should therefore strive to maximise efficiency in order to achieve the highest possible speedup.

In our case, we will define the *sequential run time* of an application as the time it takes to execute when there is a single worker computer present in the system. The speedup and efficiency for all other number of workers will be calculated based on this value.

4.4 The n-Queens Problem

Testing was carried out for the n-queens problem, with a value of n equal to sixteen (ie. we wish to place sixteen queens onto a chess board of dimension

sixteen). This is a non-trivial problem, which requires the search of 16^{16} possible states. The process of dividing this job into sub-tasks will yield two-hundred and ten sub-tasks, each of which will search 16^{14} possible board states.

The performance results obtained from executions of the n-queens program, where n is equal to sixteen, are given in Table 4.1. The average and optimal run times are presented in graphical form in Figure 4.1. The speedup ratios are presented in Figure 4.2, and a graph of the efficiency of the application can be found in Figure 4.3.

Workers	Average Time (secs)	Optimal Time (secs)	Speedup	Efficiency
1	6392.6	6392.6	1.00	100.00%
2	3190.8	3196.3	2.00	100.00%
3	2133.9	2130.9	2.99	99.67%
4	1620.4	1598.2	3.95	98.75%
6	1083.7	1065.4	5.90	98.33%
8	810.8	799.1	7.88	98.50%
12	552.5	532.7	11.57	96.42%
16	427.5	399.5	14.95	93.44%

Table 4.1: Execution time of n-queens problem on varying number of computers.

These graphs clearly show that the n-queens application achieves excellent speedup, made possible by its high level of efficiency on up to sixteen machines. The problem took almost two hours to compute using a single computer, however on sixteen machines the time was reduced dramatically, to around seven minutes. These results are especially pleasing, as they indicate that doubling the amount of computers working on the problem will go very close to halving the execution time. This level of speedup was consistent for up to sixteen computers, made possible by the high level of efficiency.

These positive results are mostly due to the problem being relatively coarse-grained, in that each sub-task requires the worker to do a significantly large amount of work. Also, the subtasks can be executed in parallel due to the absence of any data dependencies. Further discussion of these results can be found in Section 4.6.

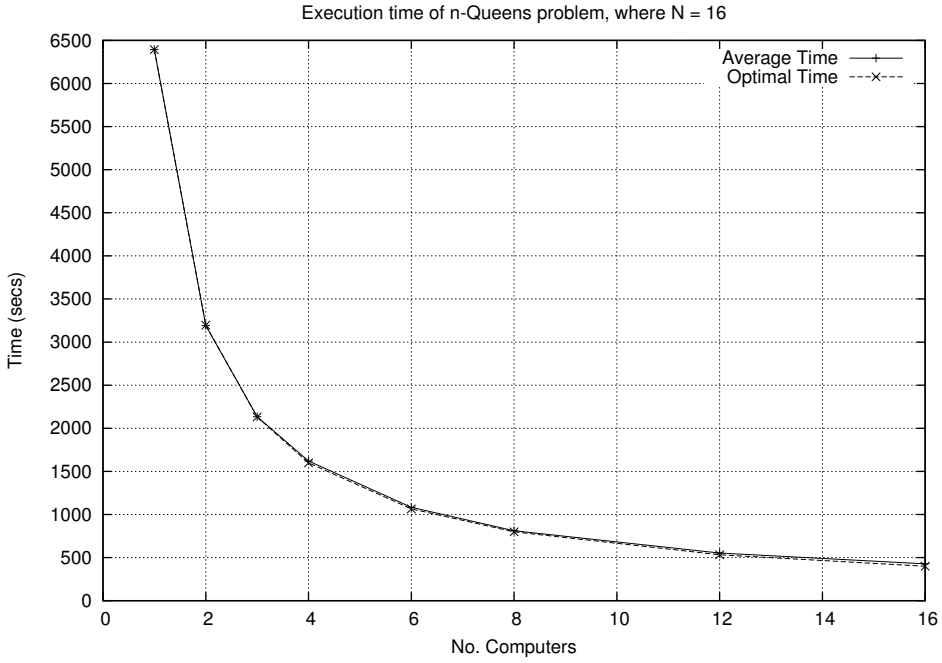

Figure 4.1: Graph of average and optimal run times of n-queens problem.

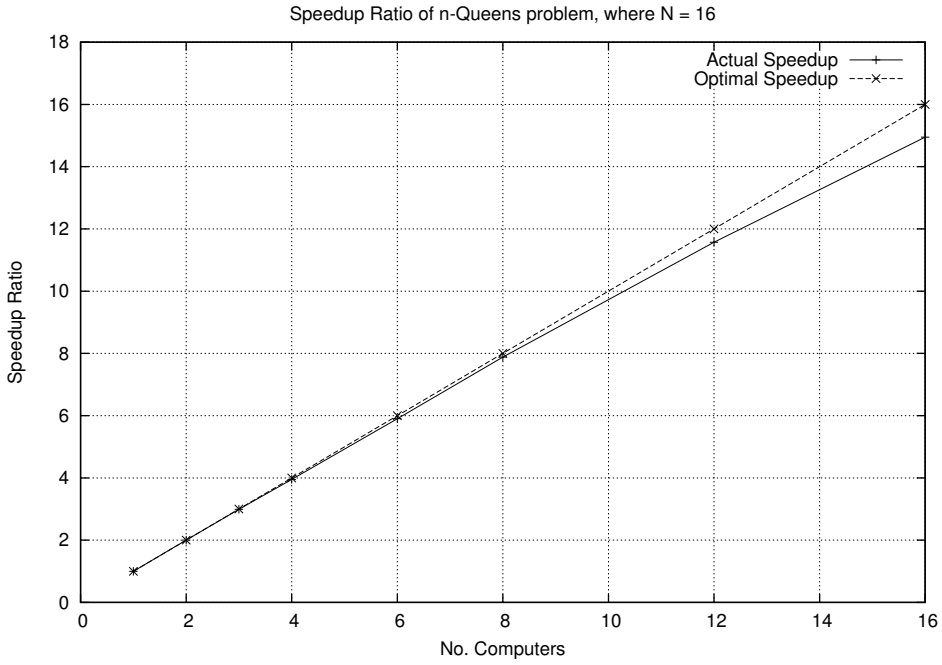

Figure 4.2: Speedup ratios of n-queens problem.

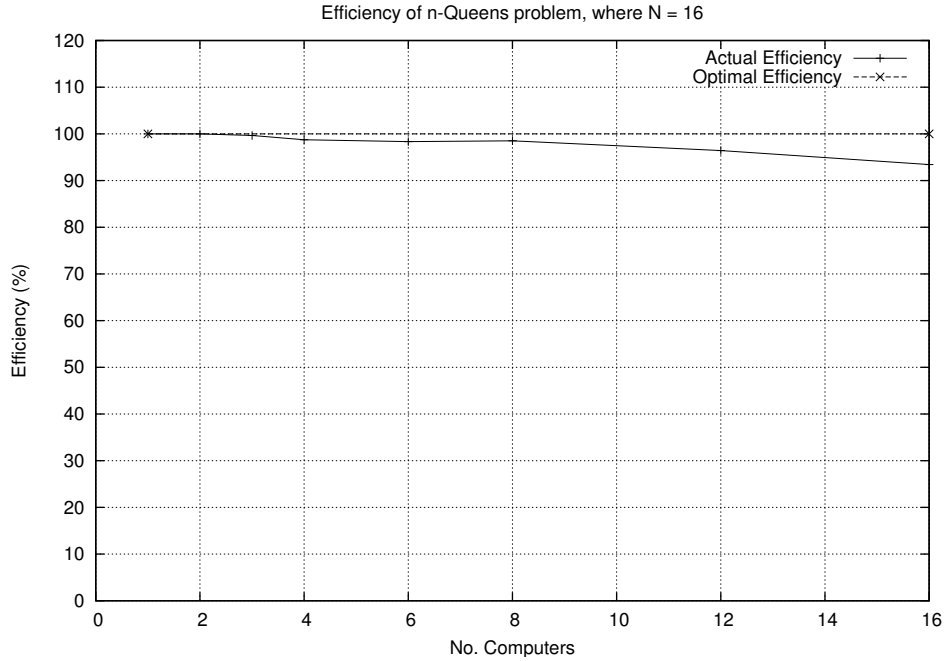

Figure 4.3: Efficiency of n-Queens application.

4.5 Shearsort

The shearsort application was firstly tested on a list of nine-hundred randomly-generated long integers. Initialisation by the master process arranged these numbers into a thirty-by-thirty mesh, and therefore program involved exactly thirty sorting phases. The results obtained from executions of the application are shown in Table 4.2. The application's performance is represented graphically in Figure 4.4, and the speedup is shown in Figure 4.5.

Workers	Average Time (secs)	Optimal Time (secs)	Speedup	Efficiency
1	2673.9	2673.9	1.00	100.00%
2	1334.2	1336.9	2.00	100.00%
4	1305.4	668.5	2.05	51.25%
8	1229.3	334.2	2.18	27.25%
12	1184.4	222.8	2.26	18.83%
16	1090.0	167.1	2.45	15.31%

Table 4.2: Results of Shearsort on nine-hundred numbers.

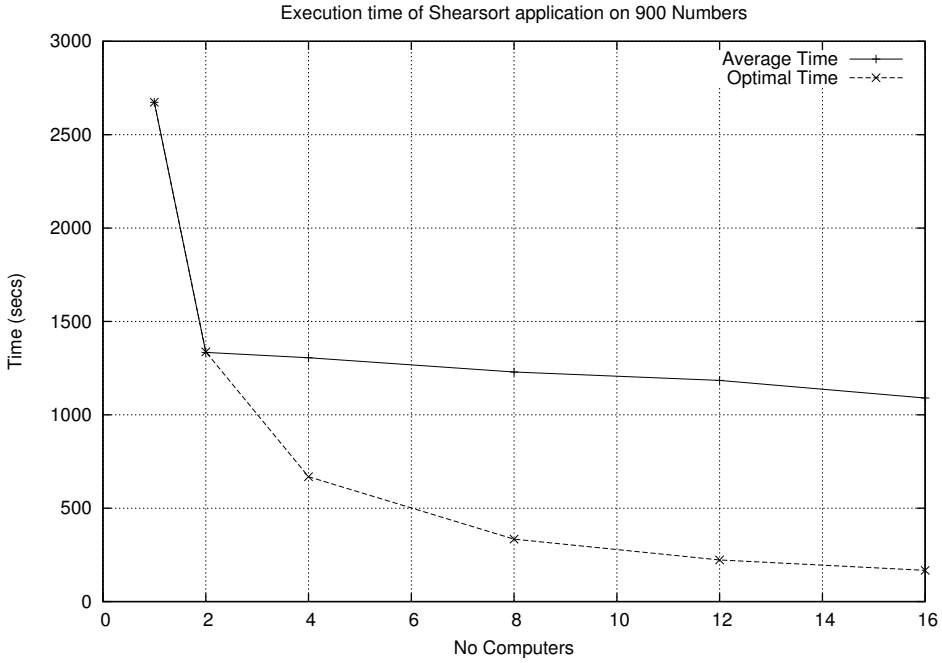

Figure 4.4: Graph of average and optimal run times of the shearsort application.

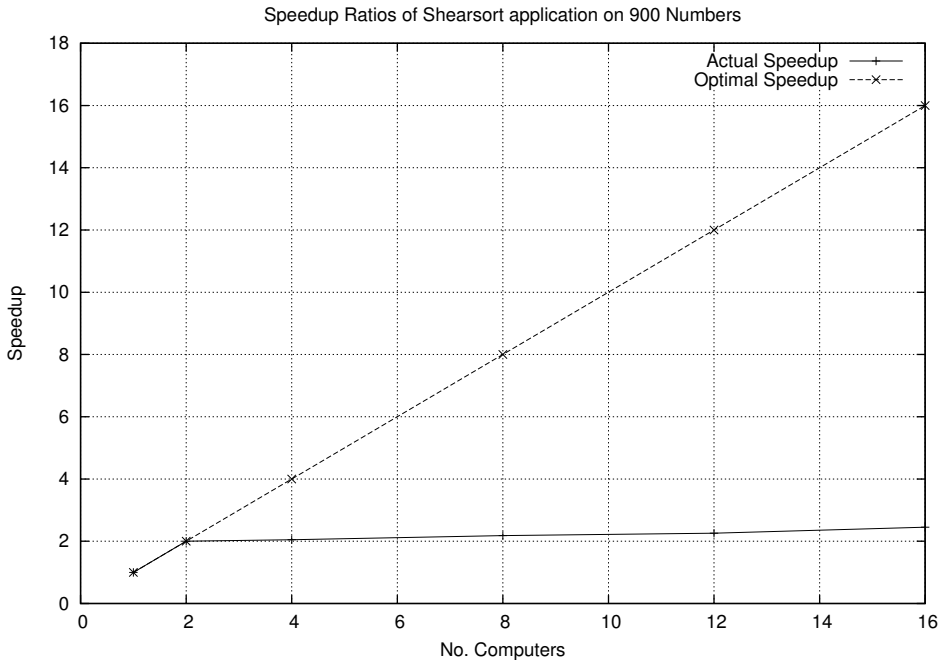

Figure 4.5: Speedup of shearsort application on nine-hundred numbers.

These results show good speedup on two machines, followed by a dramatic levelling off on any additional machines. This coincides with a steep fall in the efficiency of the system, as shown in Figure 4.6. The efficiency of this application indicates that is does a very poor job of utilising the workstations in the system.

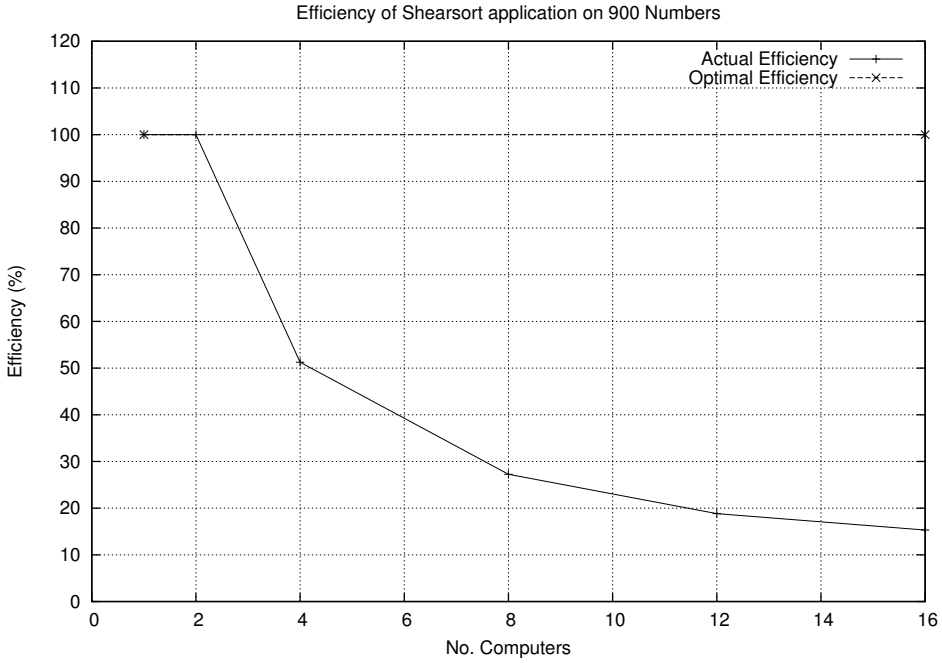

Figure 4.6: Efficiency of shearsort application.

The main contributing factor to this poor scalability is the very fine-grained parallelism of the application. A relatively large amount of JavaSpace operations must be carried out for each task; however for all of these communications, only a relatively small amount of actual work is performed by a task.

The CPU usage of the server machine during these tests was observed to be consistently between 80%-85% when there are two worker machines, and 95%-100% when there are four or more workers in the system. This would suggest that the server is not able to service the large amount of JavaSpace operation in a timely manner, thus explaining the sudden levelling off of performance.

Coarse-grained Shearsort

To determine whether the fine-grained nature of shearsort is indeed the cause of its poor performance, each task was programmed to sleep for five seconds during

execution. This modified approach was then tested on a list of four hundred numbers. This had the overall effect of producing fewer tasks that will take a greater amount of time to execute, effectively make the application more coarse-grained, and thereby decrease the communications and server CPU load. Note that this test was performed using 700 MHz G4 Apple iMacs, with 384 MB of RAM and running Mac OS X. The performance results of this modified shearsort are shown in Table 4.3 and Figure 4.7. A graph of the application's speedup can be found in Figure 4.8.

Workers	Average Time (secs)	Optimal Time (secs)	Speedup	Efficiency
1	670.2	670.2	1.00	100.00%
2	337.0	335.1	1.99	99.50%
4	240.9	167.6	2.78	69.50%
6	176.4	111.7	3.80	63.33%
8	151.4	83.8	4.43	55.38%

Table 4.3: Results of Shearsort on four hundred numbers with a five second delay.

These results show a marked improvement on those previously presented in Table 4.2, suggesting that the fine-grained tasks of the shearsort application are indeed the cause of the poor speedup. Figure 4.9 clearly shows that this coarser-grained approach is much more efficient than when a delay is not used.

4.6 Discussion

The contrasting characteristics of each application tested successfully illustrates the different properties and limitations of the system. The results show that the n-queens application achieved a much greater performance gain and experienced a higher level of scalability than the shearsort application. Some of the reasons for this are discussed below.

Task Granularity

The most obvious contributing factor to the difference in scalability lies in the granularity of the sub-tasks that make up an application. In this case, the granularity should be thought of as the amount of work done compared to the amount

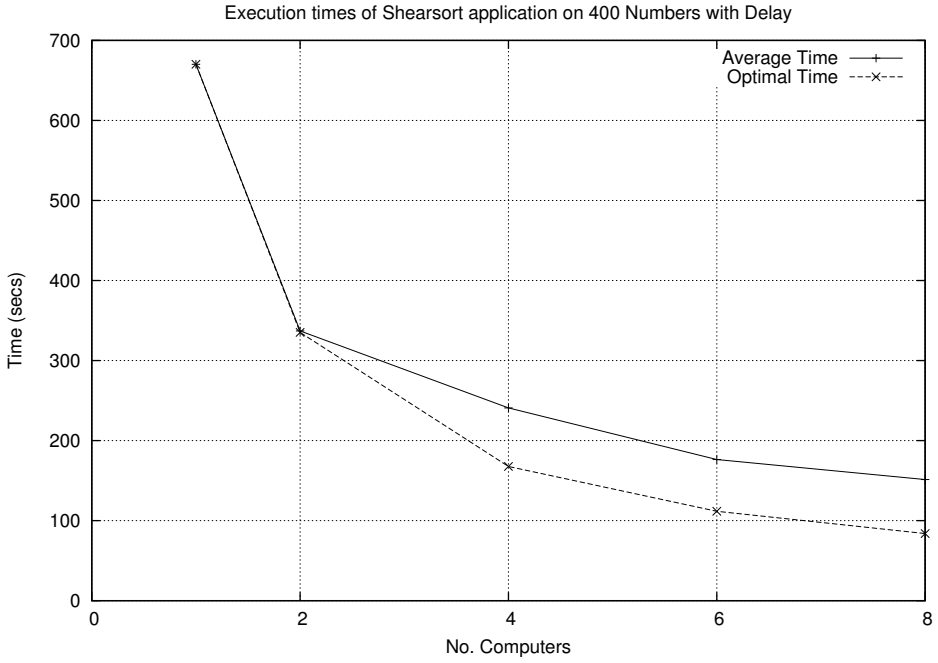

Figure 4.7: Execution times of shearsort application, with introduced delay.

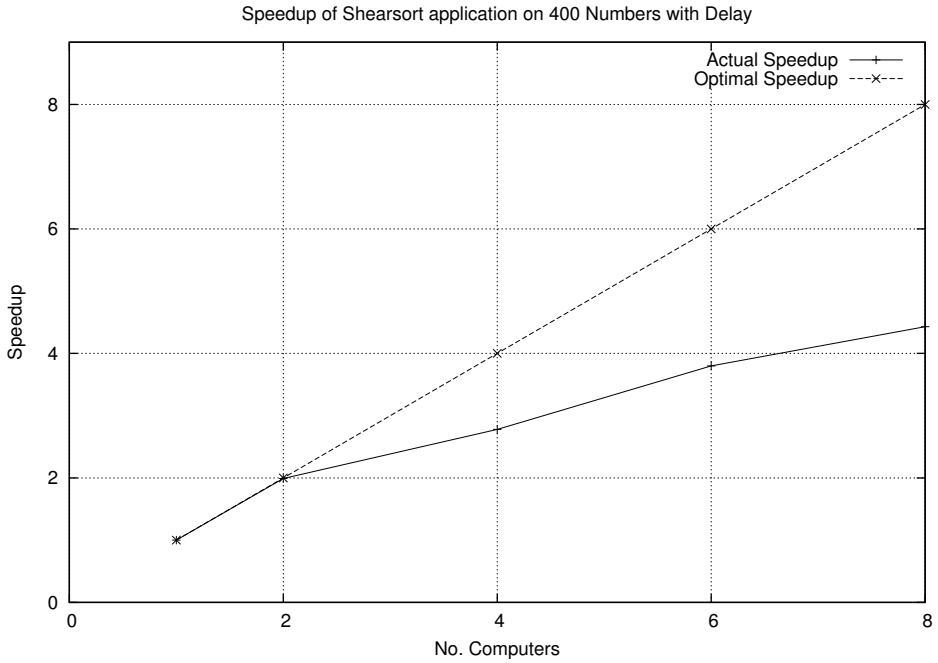

Figure 4.8: Speedup of shearsort application, with introduced delay.

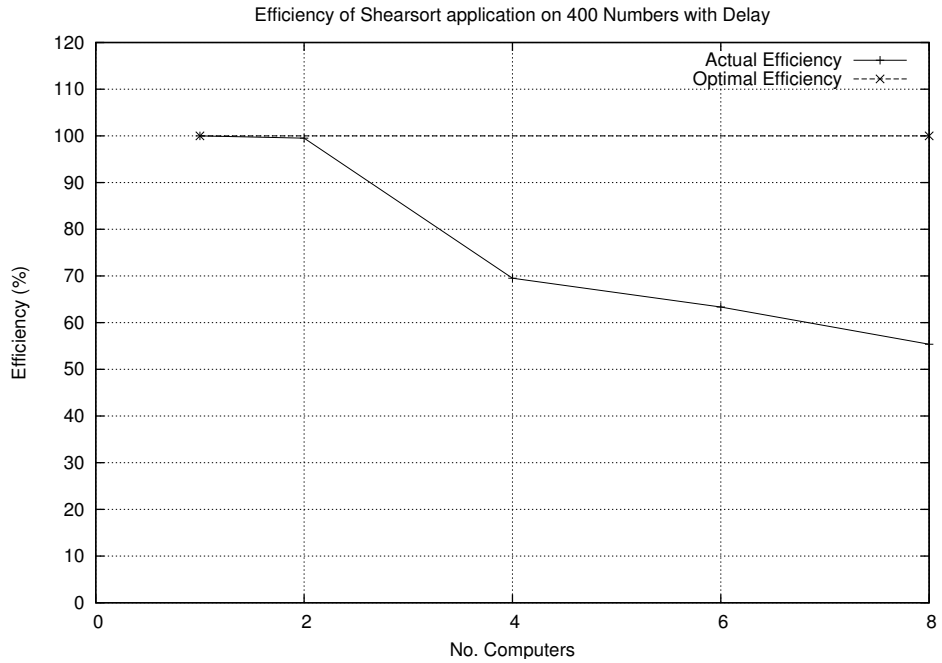

Figure 4.9: Efficiency of shearsort application, with introduced delay.

of communication overhead (ie. JavaSpace operations) incurred during the execution of each task.

The shearsort application performs many times the amount of operations of the n-queens application, as it must read each cell of each row or column of the data mesh individually, and also the step counter entry, in addition to the standard dependency graph and task entry. JavaSpace operations are relatively expensive, both in terms of time and server CPU usage. For all of this communications overhead, only a relatively small amount of work is actually performed. For example, if a shearsort is used to sort nine-hundred numbers, each task will sort only thirty numbers, but will perform over thirty JavaSpace operations. As the results show, this extra cost prevents the shearsort application from scaling effectively.

The n-queens application is vastly different to shearsort, in that each sub-task does not need to fetch any data objects in order to execute. All data is encapsulated in the task entry itself, and the task execution involves heavy computation, which takes a reasonably significant period to complete. This coarse grained parallelism results in excellent speedup and scalability.

Resource Contention

The advantages provided by the dependency graph have been discussed in Chapter 3. However, there is also a significant drawback involved in its use; it must be obtained by each worker before a new task can be taken from the JavaSpace for execution. If the graph cannot be obtained, the worker will block until it is available. Depending on how many workers are present in the system, there may be a high amount of contention for this single object, and as a result a lot of time may be wasted by each worker waiting for the graph to become available.

This problem is further compounded in the shearsort application, which performs dynamic updating of the dependency graph. This requires the dependency graph to be obtained an once more per task, thus thus further increasing the contention for this object. Furthermore, shearsort also involves a shared step counter object which will pose the same contention problems as the dependency graph.

This bottleneck problem is related to the granularity problem mentioned above. A more coarse grained approach will reduce the contention for shared objects such as the dependency graph, thus increasing efficiency by reducing the amount of time that is wasted waiting for an object to become available. An ideal solution to this problem may involve splitting the dependency graph into multiple parts, each of which controls a separate part of the application. However this would only be a viable option for applications that have several relatively independent parts.

Sequential Execution

The level of performance gain is inevitably associated with the level of parallelism of an application; it is unlikely that a purely sequential program will achieve any speedup at all, most likely the opposite would prove to be true.

The n-queens application can be fully parallelised; every task entry can be executed in parallel if there are enough workers available. However the shearsort application must execute sequentially in part, due to its different sorting phases which alternate between sorting rows and columns. This means that every row or column must be finished being sorted before all other workers can continue on in the subsequent phase of execution. This problem would not be pronounced in a system where the workers each have approximately equal processing capability.

However, in a scenario where there is one worker that is particularly slow, the performance could be seriously degraded as all of the faster workers would be continually waiting on this slow worker.

4.7 Summary

The results presented in this chapter clearly show that the system that was developed is capable of using the aggregated processing power of worker machines to execute computationally intensive applications. However, the contrast in the speedup and efficiency of the n-queens and shearsort applications indicated that the amount of performance gain actually realised is dependent on the granularity of the application; a coarse-grained application will generally achieve a greater performance gain than a fine-grained application.

Granularity is a design issue which will pose different considerations for each individual application. If the problems that were encountered in relation to the shearsort application are avoided, then it will be much easier for an application to achieve performance gains that are scalable up to a reasonable number of worker machines.

Chapter 5

Conclusion & Further Work

In Chapters 3, we detailed the design and operation of the distributed system that was produced, along with two applications that were implemented in order to perform performance testing on the system. The results of this testing were presented in Chapter 4, and further analysis of these results highlighted several important characteristics of the system.

This chapter will firstly discuss some possible directions that further research could explore in this area, to address some of the weaknesses that were underlined by the testing that was conducted. Finally, the findings of this dissertation will be summarised.

5.1 Further Research

The development and testing of the system highlighted some areas of further research that would be worthwhile undertaking. They are discussed below.

5.1.1 Non-Intrusiveness

The client (worker) software attempts to use the wasted processor cycles of the machine on which it is running. However it does not in any way monitor the processor usage by other applications that the user may also have loaded on the machine. As a result, the user may experience a performance degradation of their computer as a direct result of their participation in the system. Obviously, this is a highly undesirable situation.

There is no clear solution on how to remedy this situation, as the platform

independence of the Java language makes interaction with the operating system very difficult. It may be possible with the help of the Java Native Interface (JNI), however this has the disadvantage of losing platform independence.

It is also possible for the user to manually change the priority of the worker process on their machine, assuming that their operating system allows this to be done. However it would be more desirable for the worker application to automatically monitor the computer's processor usage, and perhaps also other performance effecting factors such as the amount of free memory and network usage.

5.1.2 Scalability

The shearsort application highlighted some of the factors, namely fine-grained sub-tasks, that can cause an application to scale poorly. This produces an increase in the server's CPU load because of the rate at which task requests must be handled. However it is important to note that, even for coarse-grained applications, there will come a point where the application will scale no further, and the addition of extra workers will not produce any further performance gain.

A possible way to remove these limits to scalability would be to allow the use of multiple JavaSpace services, running on separate machines, for a single application. The server CPU usage bottleneck would not exist in this situation, as additional JavaSpace services could be added to the system as they were required.

5.1.3 Transparency

An application must be explicitly tailored and designed to make use of the system, and the benefits provided. Alternatively, an existing non-distributed application must be modified to enable it to run on the system. Either way, a programmer must inherit and implement various Java classes, and specify how an application should be divided and executed. Furthermore, any data dependencies that exist between subtasks must be explicitly defined.

The existence of these requirements prevents the system from achieving a high level of transparency, as defined previously in Section 2.1.1.4. If a system such as this one was to be widely deployed, and used for a large variety of applications, then further work would be worthwhile to try to minimise requirements imposed on the programmer, thereby increasing the level of transparency. Projects such as those discussed in Section 2.4.5 are good examples; the cJVM system can exe-

cute standard Java programs without any modification. These systems typically use customised preprocessors or compilers, or require highly sophisticated run-time management features. Such features are beyond the scope of this particular research.

5.2 Summary

This thesis has presented a dynamic distributed system which aims to make use of the wasted processing capacity of idle workstations for the execution of computationally intensive tasks. The system is capable of operating in a heterogeneous computing environment, and allows workstations to dynamically join and leave the system at any time, even during the execution of an application. The use of transactions adds robustness to the system, and keeps the it in a consistent state in the event of partial failure.

A development framework was developed to allow parallel applications to be built that are suitable for execution on the system. Suitable coordination mechanisms are provided to allow a programmer to specify the data dependencies between an application's sub-tasks, and the order in which these sub-tasks should be executed. These coordination mechanisms are general enough to be used by a diverse range of applications.

This thesis also illustrates how the Jini and Java Web Start technologies can be combined to successfully produce a highly dynamic and flexible distributed computing environment. The dynamic nature of the system allows a user to add or remove their machine from the system at any time, without affecting the correct operation of the system. The use of Web Start also avoids the need for a user to manually install and configure the client software. It is hoped that this flexibility and ease of use would encourage users to donate their workstation to the system were it ever put to use.

The performance testing that was conducted indicated that parallel applications executed using the system should be able to achieve good speedup. However, a comparison of the n-queens and shearsort results clearly illustrated that an application's subtasks should be relatively coarse grained in order to gain optimal results. Also, the number of data dependencies that exist between subtasks should be minimised; the existence of too many dependencies will result in greater sequential execution, which in turn will reduce the performance gain realised by

an application.

The positive results, especially those of the n-queens application, suggest that the system that was produced is capable of effectively coalescing idle workstations into a powerful multiprocessor system.

References

Aridor, Y., Factor, M. & Teperman, A., 1999, 'cJVM: a Single System Image of a JVM on a Cluster', *1999 IEEE International Conference on Parallel Processing*, September 1999.

Aridor, Y., Factor, M. & Teperman, A., 2000, 'Transparently Obtaining Scalability for Java Applications on a Cluster', *Journal of Parallel and Distributed Computing*, issue. 60, pp. 1159-1193.

Batheja, J. & Parashar, M., 2001, 'A Framework for Opportunistic Cluster Computing using JavaSpaces', *Lecture Notes in Computer Science*, vol. 2110, pp. 647-674.

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

Brose, G., et al. 2001, *Java Programming with CORBA*, John Wiley & Sons, Inc., Canada.

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

Chuang, T. & Chen, D., 1999, 'Distributed Middle Tier: A Programming Model for Web-based Scalable Computing', *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999, pp. 843-849.

Colouris, G., et al. 1994, *Distributed Systems Concepts and Design*, Addison-Wesley, United Kingdom.

Farley, J., 1998, *Java Distributed Computing*, O'Reilly & Associates, Inc., Sebastopol.

Fletcher, L., 2002, 'A Dynamic Networked Browser Environment for Distributed Computing', Honours thesis, School of Computing, University of Tasmania.

Fletcher, L. & Malhotra, V., 2003, 'Network of Browsers: A Supercomputer', private communications.

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

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

Haumacher, B. et al, 2003, 'Transparent Distributed Threads for Java', *5th International Workshop on Java for Parallel and Distributed Computing*, April 2003.

Hunter, J. & Crawford, W., 2001, *Java Servlet Programming*, O'Reilly & Associates, Inc., Sebastopol.

Java Object Serialization Specification, 2001, Sun Microsystems, California.

Java RMI Specification, 2001, Sun Microsystems, California.

JavaSpaces Specification, 2002, Sun Microsystems, California.

Jini Architecture Specification, 2001, Sun Microsystems, California.

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

McGraw, G. & Felten, E., 1999, *Securing Java*, John Wiley & Sons, Inc., Canada.

Öberg, R., 2001, *Mastering RMI*, John Wiley & Sons, Inc. Canada.

Phillips, G., 1997, 'Utilizing Idle Workstations', site viewed 5/11/2003, URL - http://www.geocities.com/grahamgrebe/publications/idle_workstations.ps.gz

Schmidt, R., 2001, *Java Network Launching Protocol & API Specification*, Sun Microsystems Inc, California.

Shoch, J. & Hupp, J., 1982, 'The Worm Programs - Early Experience with a Distributed Computation', *Communications of the ACM*, vol. 25, no. 3, pp. 172-180.

Sosic, R., 1994, 'A Parallel Search Algorithm for the N-Queens Problem', *Parallel Computing and Transputer Conference Proceedings*, Woolongong, pp. 162-172.

Sosic, R. & Gu, J., 1994, 'Efficient Local Search with Conflict Minimisation: A Case Study of the N-Queens Problem', *IEEE Transaction of Knowledge and Data Engineering*, vol. 6, issue. 5, pp. 661-668.

Stallings, W., 2000, *Network Security Essentials Applications and Standards*, Prentice Hall, New Jersey.

Sun Microsystems Inc., 2003, *Java Web Start 1.4.2 Developer Guide*, site viewed 15/10/2003, URL - http://java.sun.com/j2se/1.4.2/docs/guide/jws/developersguide/contents.html

Tan, T., 2002, 'A Distributed Computer: Networking Web-Browsers Using Java RMI', Masters thesis, School of Computing, University of Tasmania.

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

Tel, G., 1994, *Introduction to Distributed Algorithms*, Cambridge University Press, Cambridge.

Thiruvathukal, G., Dickens, P. & Shazad, B., 2000, 'Java on Networks of Workstations (JavaNOW): A Parallel Computing Framework Inspired by Linda and the Message Passing Interface (MPI)', *Concurrency: Practice and Experience*, vol. 12, issue. 13, pp. 1093-1116.

Waldo, J., et al. 1994, 'A Note on Distributed Computing', Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc.

Waldo, J., 1999, 'The Jini Architecture for Network-centric Computing', *Communications of the ACM*, vol. 42, no. 7, pp. 76-82.

Wollrath, A., et al. 1996, 'A Distributed Object Model for the Java System', *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies*, pp. 219-232.

Appendix A

Generic Master & Worker

This section contains a full listing of the source code for the following classes:

- `GenericMaster`
- `Worker`
- `TaskExecutor`
- `WorkerGUI`

Appendix B

Entries

This section contains a full listing of the source code for the following classes:

- `TaskEntry`
- `ResultEntry`
- `Command`

Appendix C

Coordination

This section contains a full listing of the source code for the following classes:

- `JobDetails`
- `DependencyGraph`
- `ComputationUnit`
- `ServiceFinder`

Appendix D

n-Queens Application

This section contains a full listing of the source code that comprises the n-queens application, including:

- `nQueensMaster`
- `QueensTask`
- `QueensResult`

Appendix E

Shearsort Application

This sections contains a full listing of the source code that comprises the shearsort application, including:

- `ShearSortMaster`
- `ShearSortTask`
- `Cell`
- `StepCounter`